Paper 033-2013

RUN_MACRO Run! With PROC FCMP and the RUN_MACRO Function from SAS® 9.2, Your SAS® Programs Are All Grown Up

Dylan Ellis, Mathematica Policy Research, Washington, DC

ABSTRACT

When SAS® first came into our life, it comprised but a DATA step and a few procedures. Then we trained our fledgling programs using %MACRO and %MEND statements, and they were able to follow scripted instructions. But with SAS 9.2 and 9.3, your macros are now wearing the clothes of a PROC FCMP function; you no longer need to feed every parameter with a spoon. These functions are independent programming units, and this paper shows how they can be put to use for handy calculations, standardizing and simplifying code, and adding dynamic new capabilities that may change the way you program.

INTRODUCTION

Imagine this scenario: you are faced with a set of undocumented spreadsheets in response to a data request, and you must sort through them to find the variables you need. This will demand a long, frustrating manual investigation.

1. **Categorize the variables on each file.** While SAS dictionary tables can give us variable names and type, they cannot tell us whether a variable is an indicator variable, a date, or – most importantly – a likely ID variable.
2. **Determine which files can be merged.** If the information we need is on separate files, we will have to find a common ID variable (or variables) by which to merge them. Of course, some of the files we may not need at all.
3. **Identify the record-level of each file.** In order to merge files, we must know whether each file is long or wide – i.e. will our merge be one-to-one, one-to-many, or many-to-many? This may depend on the variable of interest.

Once we have compiled all of this information, we can begin cleaning variables and merging the files to produce the desired data set for analysis. These initial investigative steps would be difficult to automate with a macro approach. Macros need parameters, and the parameters for each step depend on information gathered in the previous step.

That said, the steps involved are straightforward; it would be easy to write 'specs' for a junior programmer to perform the work. The programmer would have to open each file and examine variable names, use the FREQ procedure to compare values, and perhaps attempt a preliminary merge to test whether variables represent a common ID.

This paper will demonstrate how, by employing PROC FCMP and the RUN_MACRO function, SAS can be that junior programmer for us. RUN_MACRO enables iterative, dynamic routines, using a function to extract values and metrics from the execution of a macro, which can then be used as input for further RUN_MACRO routines which generate information on even higher orders. It is almost like SAS is thinking on its own.

FUNCTIONS IN SAS

What role do SAS functions really perform for us? We use them so often we might take their operation for granted. Functions accept numeric or character values as parameters and produce a single return value. The return value can represent the result of a calculation, lookup-operation, or string transformation. Below are a few familiar examples.

- `STD(1,2,3,4,5)` – What is the standard deviation of these five values?
- `SUBSTR('name',1,3)` – What are the first three characters in the string 'name'?
- `DATE()` – What is today's date as a SAS date value?

I like to think of functions as answering a question. When we use a function in a DATA step it is like we have an expert at our side, and the function call is our Phone-a-Friend. Rather than have to spell out calculations manually, such as for the standard deviation of a set of variables, we simply call the STD() function. The DATA step pauses, passes the parameters to the function, and it calculates the result. Functions are often visualized as a 'black box'.

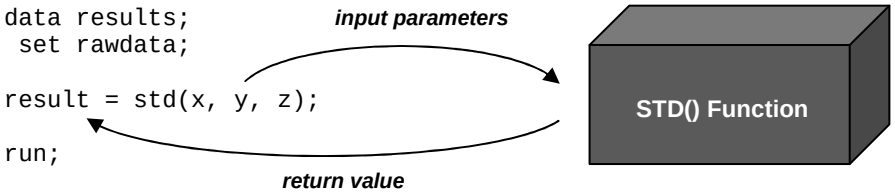

Figure 1. Diagram of SAS function during program execution

The diagram in Figure 1 illustrates two important features of functions:

DATA Step Context: Functions typically operate in context of a DATA step (or, similarly, in SELECT statements in PROC SQL or the WHERE clause of SAS procedures). Functions take values *from* the DATA step as parameters and return values *to* the DATA step as the result. Granted, using the macro function %SYSFUNC() we can employ DATA step functions in a macro context, taking macro variables as parameters and returning values to the macro facility. But apart from SYMGET() and CALL SYMPUT(), input parameters and return value are at the same level.

Encapsulation: The black box represents encapsulation – the fact that we do not know or care how the function is being implemented by SAS, we only care that it returns the correct result. In the example of standard deviation, the calculations involved are relatively long and cumbersome. Using a function ensures the calculation is standardized, and increases readability of the code. It is easy to see that `STD_DEV = STD(of vars[*]);` stores the standard deviation of an array of variables. Whereas if we had written out the intermediary calculations it would be less obvious what is happening. Moreover, for some functions such as substr(), we do not have a feasible hard-coded alternative.

LIMITATIONS OF DATA STEP FUNCTIONS

While functions are a powerful construct for answering questions in the context of a DATA step, the questions usually concern the *values* of DATA step variables – not data sets and variables themselves. For example, `SQRT(x)` will tell us the square root of variable x. The only information the function needs to answer this question is the value of x and an algorithm to produce the square root. In contrast, consider the function `is_this_an_ID_variable(dsn,var)`, where dsn is a data set name and var a variable name. In this function, data set name and variable name would not represent strings to be manipulated, as in `catt(dsn, var)`, rather instructions regarding the data set and variable in question. There is no standard algorithm, calculation, transformation, or lookup table that can provide the answer. The function would have to open the data set and 'look' for itself.

Before we see how a function would answer this question, we should consider the approach a human programmer might take to categorize a variable. As Bost discusses in "To FREQ, Perchance to MEANS" (paper 093-2011), one key metric for judging a variable's purpose is the number of unique values observed in the data. Variables with few unique values, such as 0 and 1, or 'M' and 'F', are probably indicators or categorical variables. ID variables, on the other hand, will have many unique values relative to the number of observations in the data set. Therefore one of a programmer's first steps might be to obtain the number of unique values observed for the variable in question.

Nlevels Metric

There are many ways to obtain the number of unique values taken by a given variable in a given data set. Perhaps the most straightforward is to use a PROC FREQ with the NLEVELS option, as described in SAS Usage Note 30867.

```
ods output nlevels=NLevels_out;
    proc freq data=sashelp.class nlevels;
       tables age / noprint;
    run;
```

TableVar	NLevels
Age	6

Output 1. NLEVELS output object

The NLEVELS option tells the procedure to generate an output dataset containing the variable name and 'NLevels', which is a count of the number of distinct levels of that classification variable. The ODS statement names this output object as the data set NLevels_out. From the output we see variable Age in SASHELP.CLASS has 6 unique values. If Age had contained missing values, the output data set would also contain 'NMissLevels' and 'NNonMissLevels'.

Since our programmer will need to determine NLEVELS for a large number of variables and data sets, a macro would probably be used, accepting data set name and variable name as parameters. Ultimately the goal would be a SAS data set containing at least these three variables: data set name, variable name, and the corresponding NLevels.

```
%macro NLevels(dsn, var, out);

      ods output nlevels=NLevels_out;
          proc freq data=&dsn nlevels;
             tables &var / noprint;
          run;

      data &out;
       set NLevels_out;
       dsn = "&dsn"; *add data set name to output;
      run;

      proc datasets library = work nolist;
                   delete NLevels_out; quit;

%mend;
```

This will work for one data set and variable, but how will the output data sets, NLevels_out, be combined across macro calls? The programmer could implement a new parameter for output data set name so that the output data sets do not overwrite one another. Still, this results in a large number of intermediary data sets being generated which would then have to either be appended together afterwards or appended to a pre-existing shell data set.

It would be great if we could circumvent the entire issue of managing output data sets, as all we really want is one number, NLevels, for each variable and data set combination. Let us restate the problem in the form of a question: *What is the NLevels number for this variable in this data set?* Alternatively, expressed in the form of a function:

```
NLevels = NLevels(dsn, var); *the function approach;
```

We cannot simply put the macro on the right of the equal sign: `NLevels = %NLevels(dsn, var);`. Because the macro contains multiple statements, it will generate syntax errors when it resolves. Macros are *not* functions.

MACROS VS. FUNCTIONS

SAS macros are designed to generate output and data sets, not values themselves. In the past, programmers went to great lengths to create a return value from a macro with so-called 'function-style macros'. These macros could only contain macro statements, as they had to resolve to their 'return' value. Thanks to RUN_MACRO, in PROC FCMP, this is no longer necessary. Why imitate a function when we can now write our own functions to suit our purpose?

RUN_MACRO routines give us a way to retrieve a metric from macro execution, by wrapping the macro call inside an FCMP function. The function executes the macro, and returns a single value to the DATA step. But in order to write our own functions we first need to take a look 'inside the box', and get a crash course in PROC FCMP.

USER-WRITTEN FUNCTIONS WITH PROC FCMP

With SAS 9.2 and 9.3 we gained the ability to write our own functions using the FCMP procedure. Functions defined using PROC FCMP can perform the same roles as standard SAS functions – calculations, lookup operations, string transformations, and comparisons. Below are two simple examples of user-written functions: 'my_sum' returns the sum of non-missing values among two numeric arguments, and 'my_cat' concatenates two character arguments.

```
PROC FCMP OUTLIB=work.functions.demo;

    function my_sum(var1, var2) ;
        if missing(var1) then v1= 0; else v1= var1;
        if missing(var2) then v2= 0; else v2= var2;
        if missing(var1) and missing(var2)
        then sum = .;  else sum = v1 + v2;
    return(sum);
    endsub;

    function my_cat(str1 $, str2 $) $;
        length cat $32767;
        cat = trim(str1) || trim(str2);
    return(trim(cat));
    endsub;

RUN;
OPTIONS CMPLIB=work.functions; *two-part name in program where used;
```

Like formats and macros, function definitions are stored in a library. The storage location is specified on the `OUTLIB=` statement by a three-part name: libname, data set name, and package name (function definitions are stored in a SAS data set). To use the function in a program we need a corresponding options statement, `CMPLIB=work.functions`. Once the proper function library has been referenced in the options statement, FCMP functions may be used in all the same places as other SAS functions. For two parameters x and y, `sum()` and `my_sum()` give the same result.

The `PROC FCMP` and `RUN;` statements define the workspace for the individual functions to be defined. Each function definition begins with the `function` statement. Here we name the function and list the parameters. Any character parameters are followed by a $ sign, and if the function returns a character value the function statement is followed by a $ sign. Each function definition ends with an `endsub` statement. The return value of the function will be the value of the variable or expression inside the `return()` statement. The `return()` can be part of a conditional statement.

The syntax to construct your function is similar to a DATA _NULL_ step without any I/O statements like set or output. Essentially you 'set' the values from the parameters, and 'output' the result with the return statement. All variables created inside the function definition are *local* to the function, so we retain the important property of encapsulation. There are some limitations, particularly when dealing with arrays (see "PROC FCMP and DATA Step Differences" in the Base SAS 9.2 Procedures Guide). But for the most part, the inside of a function's 'black box' is not too scary.

…

This crude example, while enough for the purposes of RUN_MACRO, does not do justice to the extreme versatility of PROC FCMP. Typically FCMP functions are written to *expand* upon existing SAS functions, or to provide capabilities not currently available. For example, FCMP functions can be written to "substring" by word, returning the third through fifth words from a string. Or a function can be made to return the number of unique values in a DATA step array.

Even if one never uses the RUN_MACRO capability in PROC FCMP, user-written functions should become part of every programmer's repertoire. FCMP functions can do much to help standardize and increase readability of code, as well as increasing the reusability of calculations common across programs.

THE RUN_MACRO FUNCTION

We only need one line of code inside our PROC FCMP function in order to call a macro – the special RUN_MACRO function (hereafter, 'command'). In a RUN_MACRO routine, the FCMP function often serves as a wrapper.

```
PROC FCMP outlib=work.functions.wrapper;

   function nlevels(dsn $, var $);
      rc = run_macro('get_nlevels', dsn, var, nlevels);
      return(nlevels);
   endsub;

RUN;

options cmplib=work.functions;
```

```
/* Think:

%Local dsn = dsn;
%Local var = var;
%Local nlevels = nlevels;

%get_NLevels();

dsn = symget('dsn');
var = symget('var');
nlevels = symget('nlevels');

*/
```

During execution, when the command is reached, the function takes each variable on the RUN_MACRO parameter list and creates a local macro variable with the same name and value. Then the macro is executed, and the values of each macro variable at the end of the macro execution are written back to the corresponding variables in the function. Parameters on the RUN_MACRO command represent two-way communication between the function and the macro. The rc variable returns 0 if the macro was submitted successfully, otherwise it returns an error code.

There are some slight modifications that need to be made to our %NLevels() macro for it to be 'function-ready'.

```
%macro NLevels(dsn, var);

ods output nlevels = NLevels_out;

  proc freq data = &dsn nlevels;
     tables &var / noprint;
  run;

%mend NLevels;
```

```
%macro get_NLevels(); ①
   %let dsn = %sysfunc(dequote(&dsn.)); ②
   %let var = %sysfunc(dequote(&var.));

   ods output nlevels = NLevels_out;

   proc freq data = &dsn nlevels;
      tables &var / noprint;
   run;

   data _null_;
    set NLevels_out;
   call symput('nlevels',NLevels); ③
   run;

   proc datasets library = work nolist;
                delete NLevels_out; quit;
%mend get_NLevels;
```

Process to modify a SAS macro for use in a RUN_MACRO function:

1. Move the list of parameters from the %macro statement to the RUN_MACRO command. Generally these parameters will also be parameters to the function itself, as in the case of data set name and variable name.
2. Add a %sysfunc(dequote()) statement for any parameters that are character. Character values get quoted when they are passed from the function to the macro, so we use dequote to restore the original values.
3. Use CALL SYMPUT() to return values to the function from the macro. Make sure the macro variables used for the return values are included in the list of parameters in the RUN_MACRO command.

Note: While all of the macro variables are local to the macro, and all of the function variables are local to the function, there is no special WORK library for RUN_MACRO to use. Therefore, just like with other reusable macros, we must perform data cleanup using PROC DATASETS to avoid temporary files carrying over into another session.

THE BIG PICTURE

We can visualize a RUN_MACRO routine in terms of three components: the calling data set, the wrapper function, and the underlying macro. The function is called once for each observation in the calling data set, which means the macro is executed once for each observation in the calling data set. So far this is looking a lot like CALL EXECUTE. The advantage of RUN_MACRO functions is that we have a built-in channel to *report back* to the calling data set.

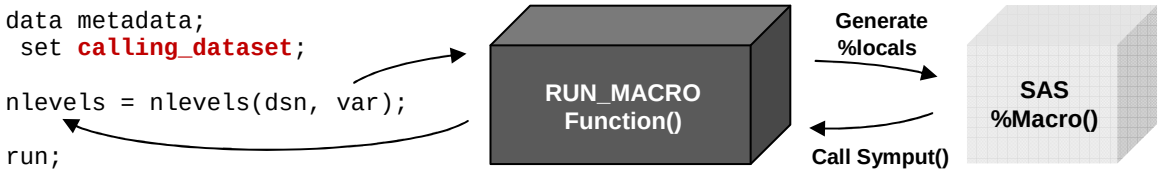

Figure 2. Diagram of a RUN_MACRO function during program execution

If the DATA step represents the executive who is giving the instructions, then the underlying macro represents the programmer that actually performs the task. The RUN_MACRO function serves as a middleman, facilitating two-way communication between the calling data set and the macro. Unlike typical macros which generate data sets or output, macros used in RUN_MACRO functions generate *information*. By extracting values and metrics from the execution of a macro, RUN_MACRO routines give us the capability to answer high-level questions about data sets, variables, and entire analytical processes. Rather than Phone-a-Friend, think of RUN_MACRO routines as Phone-a-Programmer.

Execution of a macro within the function does not affect the logic of the calling DATA step. Our function executes as normal and retains the property of encapsulation. Consider the alternative: a nested macro approach. If we use a macro do loop to fire off calls to a submacro, we have to make sure our macro variables are of the proper scope and resolve at the appropriate time for the whole process to work. Whereas RUN_MACRO routines have distinct layers – DATA step and macro – which operate independently, communicating via the function. The advantages are clear. And we have seen that it takes only a few lines of code to create FCMP functions to wrap our macros.

The real question is, where are we going to find a data set with the list of parameters for our RUN_MACRO function?

CALLING DATA SET

Our NLevels() RUN_MACRO function takes data set name and variable name as parameters. Therefore, the calling data set must contain a list of data set names and variable names. Happily, a data set already exists that meets just these criteria – the SASHELP view of the dictionary table COLUMNS. The dictionary table contains the name, type, length, label, format, and informat for every variable in every data set in every library known to the SAS session.

While dictionary tables have many uses, they seem to be tailor-made for driving RUN_MACRO functions. If we have already imported our files into SAS and they are stored in the same library, all we need to do is set sashelp.vcolumn where libname="our library" and we can call our nlevels function on every variable in every one of our files.

```
data metadata;
  set sashelp.vcolumn;
  where libname="SASHELP";

  nlevels = nlevels(memname, name);

run;
```

Memname	Name	NLevels
SASHELP.Class	Name	19
SASHELP.Class	Sex	2
SASHELP.Class	Age	6

Output 2. Output data set with NLEVELS values

The underlying macro cannot open a data set if the calling DATA step already has it open, so the calling data set is not generally available as a parameter to a RUN_MACRO function. For instance if sashelp.vcolumn is used as the calling data set, then the function call `nlevels= nlevels("sashelp.vcolumn",name);` would not work. In this way, dictionary tables provide a natural segmentation. Calling data sets represents metadata; the function takes that metadata and passes it to the underlying macro, which operates on the actual data set and generates new metadata, which is then returned to the DATA step to be combined with the existing metadata. This underscores the advantages over a pure macro approach. With RUN_MACRO routines the metadata evolves through a sequence of DATA steps.

*As a side note, directory listings are another great way to drive RUN_MACRO functions, such as for a file-import routine. SAS has a macro (%DIRLISTWIN) on support.sas.com that reads a directory listing into a SAS data set.

PROGRAMMING WITH RUN_MACRO ROUTINES

In the introduction I outlined the general problem of categorizing variables on an undocumented file. NLevels will not tell us everything we want to know about a given variable in a given data set. However now that we have seen how to retrieve one metric from a macro, it is not hard to extend the idea to more sophisticated measures. Since the macros are too long to include in the paper, I present only pseudo-code. We already know how to write macros. The goal is to see how macros can make educated guesses about our data and spare us a great deal of analytical legwork.

Metrics: First we generate a data set of frequencies for the desired variable using PROC FREQ and the OUT= option. There is a lot we can determine just by looking at this data set, which contains variable value and count.

1. Get NOBS from the original data set, using the NOBS= option on the Set statement.
2. Get NMISSING from the frequency data set – the frequency count where missing(variable).
3. Get NLEVELS, the number of unique values, from the frequency data set or ODS output object.
4. Attempt to generate DATE field on frequency data set by using input() and ANYDTDTE. informat.
5. ID fields often have a consistent length. We can test using LENGTH() on the frequencies data set.
6. Single-word alphanumeric fields are often IDs. Test using ANYALPHA(), ANYDIGIT(), and COUNTW().
7. We can look for clues in the variable name itself. Check INDEX(varname, 'ID' or 'NAME').
8. ID variables tend to have more unique values than any other variable in the data set. Most variables in large data sets have so few unique values compared to the number of observations that the threshold does not need to be set very high for this check. I used 40%, or IF NLEVELS > .4 * (NOBS)
9. Any (good) ID field should have few missing values, so NMISSING should be small relative to NOBS.

These checks will still miss certain cases. Consider a numeric measurement with many digits of precision. This may be mistakenly be classified as an ID variable. One approach might be to look at the distribution of values, since for ID variables the distribution should be relatively uniform.

COMPILING THE RESULTS

Whatever the metrics we use, we have seen that all it takes is a DATA _NULL_ step and a few CALL SYMPUTs to return these values from the macro to the RUN_MACRO function. Decision rules can be implemented either inside the macro or inside the function in order to determine a best guess category for the selected variable. Alternatively, the metrics could be combined into a scalar ID-likelihood score for post-processing in the calling DATA step.

Here are some of the heuristics I implemented. First I address the categories that are relatively easy to define:

- IF NOBS = NMISSING then varcategory = "EMPTY"
- IF NLEVELS = 2 and values IN ('0','1','2', 'Y','N') then varcategory = "INDICATOR"
- IF input (values, ANYDTDTE) field is never missing, or the format or informat look like a date (since this is being called from a dictionary table) then varcategory = "DATE"
- IF NLEVELS is small relative to NOBS then varcategory = "CATEGORICAL"

After eliminating the low-hanging fruit, we are left with variables that have many unique values and are not dates. The main goal is to distinguish ID variables from other many-valued fields such as numeric measures or character strings. I assign points based on metrics 5-9, and if the variable scores above a certain cutoff it is classified as an ID variable. Ultimately we end up with a function that looks like this: `varcategory = categorize_variable(dsn, var);`

BUILDING OFF PRIOR INFORMATION

Now that we have identified potential ID variables on each file, we are ready to attack the next question: do any of these variables represent common keys? Expressed as a function for the single case, the call might look like this:

```
Similarity_Index = Compare_Variables(dsn1, var1, dsn2, var2);
```

Before we examine how to produce such a metric we should consider the calling data set we will use for this function. We need pairs of variables and data sets, so we can use PROC SQL to join our dictionary table to itself. But if we run the function on *all* possible pairings of data sets and variables, the number of combinations involved quickly grows prohibitive even for SAS. Since we have already identified likely ID variables, we are able to produce pairings of the top two or three most likely ID variables in each data set – reducing our problem by several orders of magnitude.

Determining Similarity:

There are many metrics we might like to know about how these two variables compare. Do they share the same range of values? How many of the observations in each dataset have values in common? Last but not least, are the names the same or similar? Since we are only considering two variables at a time, we do not need to merge the full data sets for this question. We can merge the frequency data sets and still retrieve all of the necessary information.

If we perform a full join of the two frequency data sets we can quickly tabulate the following metrics:

- The number of unique values in each data set that did and did not merge
- The sum of frequencies for values in each data set that did and did not merge
- Compare the names of each variable, perhaps by using COMPLEV()

From these metrics we can get a sense of whether we feel two variables represent the same quantity. If so, we can then determine whether a merge on these two variables would be 1-to-many, many-to-1, 1-to-1, or many-to-many. Unlike our first routine, where our metrics fed through a set of decision rules to return a variable 'category' as a result, it would be difficult to combine the above metrics into a single meaningful return value. Luckily we do not have to.

USER-WRITTEN CALL ROUTINES WITH PROC FCMP

While this may seem a bit of a bait-and-switch, in reality many if not most RUN_MACRO 'functions' are call routines. This means that rather than simply return one value, a routine updates several arguments in the calling data set.

Here is an example of the call routine that I use to return multiple metrics at once from the comparison of variables.

```
proc fcmp outlib=work.functions.wrapper;

  ① subroutine Merge_Metrics(dsn1 $, var1 $, dsn2 $, var2 $,
                    NOBS1, missing1, numobsmerged1, nlevels1, numvalmerged1,
                    NOBS2, missing2, numobsmerged2, nlevels2, numvalmerged2);


          ② outargs NOBS1, missing1, numobsmerged1, nlevels1, numvalmerged1,
                    NOBS2, missing2, numobsmerged2, nlevels2, numvalmerged2;


    ③ rc = run_macro('get_mergemetrics', dsn1, var1, dsn2, var2,
                    NOBS1, missing1, numobsmerged1, nlevels1, numvalmerged1,
                    NOBS2, missing2, numobsmerged2, nlevels2, numvalmerged2);
  endsub;

quit;

options cmplib=work.functions;
```

Differences between an FCMP call routine and an FCMP function:

1. Call routines begin with a 'subroutine' statement instead of a function statement. Like functions, we put a $ sign after character parameters, however unlike functions there is never a $ sign at the end of the statement.
2. Instead of a return() statement, a list of arguments appears on the OUTARGS statement. These variables must exist in the calling data set and also be listed as parameters to the subroutine. The values of each of these arguments will be copied to the corresponding DATA step variable upon completion of the subroutine.
3. The RUN_MACRO function, if used, operates the same in subroutines and functions. Local macro variables are created with the same name and value as each of the parameters, the macro is executed and the macro variables are copied back to the function variables - which may or may not be returned to the DATA step.

Just as parameters on the RUN_MACRO command represent two-way communication between the function and the macro, the variables on the OUTARGS statement represent two-way communication between the calling data set and the function. While subroutines can quickly become cluttered with many OUTARGS arguments, the power of retrieving numerous high-level metrics at once may outweigh the cost in readability.

In the Merge_Metrics() subroutine, dsn1, var1, dsn2, and var2 are the real input parameters. The remaining ten arguments are empty variables that are populated by the %get_mergemetrics() macro called by the RUN_MACRO command. These ten variables are then returned to the calling DATA step, replacing the previously missing values.

RESULTS

The full code of my RUN_MACRO routines is available online at SAScommunity.org. In looking at my code, you may disagree with certain aspects of my implementation. As with any machine-implemented heuristic, more sophisticated methods could be developed depending on the need. However the point is not that the metrics are perfect, the point is that such metrics are possible. Even the crude metrics so far devised will eliminate a lot of manual investigation.

Imagine, instead of staring at a mess of undocumented data, you could dispatch SAS to investigate for you. Simply by supplying a library name where the data sets have been imported, SAS can add 'category' to your dictionary table:

LIBNAME	MEMNAME	NAME	CATEGORY
MY_LIBRARY	Attendance2012	FRPL_status	Indicator
MY_LIBRARY	Attendance2012	Stdnt_Key1	Likely ID Variable
MY_LIBRARY	Enrollment2013	Enrolled	Date

Output 3. Output data set from first RUN_MACRO routine

This is qualitatively different information than we had before. It represents the type of high-level insight that we might typically ask a programmer to develop, but now it can be generated automatically via RUN_MACRO. Next, we can use this information as input to develop even higher-level insights such as which ID variables appear to be the same. We do this by pairing ID variables identified by the first routine and calling a second routine to compare the variables.

Finally, once we have identified common ID variables, we can report metrics as to how the files will potentially merge.

Top ID Variable	Data Set	Top Match	Data Set	Match Type
Stdnt_Key1	Attendance2012	OffenderID	DisciplineFile_2013	m:m
		Stuid_VNSD	RosterCollection2012	1:m
		COL4	StaffAssignments2011	m:1
SSID	Background_2012	SSID	Background_ 2013	1:1
		DistrictLink	Enrollment 2013	1:m

Output 4. Report using metrics generated by second RUN_MACRO routine

Granted, once SAS finally reports the results, sometimes the insights seem a bit obvious. We probably could have figured out for ourselves that the 'SSID' on Background_2012 is the same variable as 'SSID' on Background_2013. However we would not have immediately known that the variable represented a unique key on each file. By replacing costly and slow manual investigations and human decision-making processes with efficient SAS approximations, we are able to better focus our time and energy on where it will provide the most benefit.

CONCLUSION

Hopefully this paper has made clear the capacity to develop a dynamic programming structure via the RUN_MACRO function from PROC FCMP. Instead of using nested macro loops, we now have the ability to explicitly program at the meta-level using the familiar logic of the DATA step. Macros can return values to the DATA step, which can then compile these values as inputs for further processing. My routines demonstrate how the values returned by macros parallel the types of generalizations and classifications programmers typically develop through manual investigation.

Beyond Heuristics

PROC FCMP was not designed merely to produce heuristics regarding variables and data sets. Even before FCMP became available to the DATA step in SAS 9.2, it was already available for use in statistical procedures. The iterative capabilities that FCMP supports are even more powerful in that context. For some examples, see Stacey Christian's paper (326-2010) which shows how to implement an iteratively reweighted least squares algorithm. Another ability that FCMP supports is recursion. Jason Secosky, also of SAS, demonstrates how to use FCMP for directory traversal in his 2007 paper "User-Written DATA Step Functions". Lastly, Mike Rhoads 2012 paper demonstrates how to create a 'Macro-Function-Sandwich' – using %SYSFUNC and RUN_MACRO – to effectively change the syntax of SAS.

REFERENCES

The full RUN_MACRO routines mentioned in this paper are posted on SAScommunity.org.
Available at http://www.sascommunity.org/wiki/RUN_MACRO_Run!_SGF_Paper_033-2013

SAS Institute Inc. 2009. "The FCMP Procedure." *Base SAS 9.2 Procedures Guide*. Cary, NC: SAS Institute Inc.
Available at http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/a002890483.htm.

SAS Institute Inc. 2008. "Modernizing Your SAS Code: PROC FREQ Applications." *SAS Knowledge Base: Usage Note 30867*. Cary, NC: SAS Institute Inc.
Available at http://support.sas.com/kb/30/867.html.

ACKNOWLEDGMENTS

I first saw RUN_MACRO in a presentation by Erin Lynch of SAS Institute at our local SAS users group. Erin walked us through how she used RUN_MACRO functions to automate a reporting process for school districts. Her example inspired me to further investigate the technique, and thanks to her support I was encouraged to develop a paper for SAS Global Forum. Hopefully this paper will perform a similar role, spurring ideas as to what RUN_MACRO can do.

RECOMMENDED READING

- Christian, Stacey M., and Jacques Rioux. 2010. "Adding Statistical Functionality to the DATA Step with PROC FCMP." *Proceedings of the SAS Global Forum 2010 Conference.* Cary, NC: SAS Institute Inc.
  Available at http://support.sas.com/resources/papers/proceedings10/326-2010.pdf.
  Sample programs are available from *SAS Presents*: http://support.sas.com/rnd/papers/index.html.
- Lynch, Erin, Daniel O'Connor and Himesh Patel. 2011. "My Reporting Requires a Full Staff—Help!" *Proceedings of the SAS Global Forum 2011 Conference.* Cary, NC: SAS Institute Inc.
  Available at http://support.sas.com/resources/papers/proceedings11/291-2011.pdf.
- Bost, Christopher J. 2011. "To FREQ, Perchance to MEANS" *Proceedings of the SAS Global Forum 2011 Conference.* Cary, NC: SAS Institute Inc.
  Available at http://support.sas.com/resources/papers/proceedings11/093-2011.pdf.
- Secosky, Jason. 2007. "User-Written DATA Step Functions." *Proceedings of the SAS Global Forum 2007 Conference.* Cary, NC: SAS Institute Inc.
  Available at http://www2.sas.com/proceedings/forum2007/008-2007.pdf.
- Secosky, Jason. 2012. "Executing a PROC from a DATA Step." *Proceedings of the SAS Global Forum 2012 Conference.* Cary, NC: SAS Institute Inc.
  Available at http://support.sas.com/resources/papers/proceedings12/227-2012.pdf.
- Rhoads, Mike. 2012. "Use the Full Power of SAS® in Your Function-Style Macros." *Proceedings of the SAS Global Forum 2012 Conference.* Cary, NC: SAS Institute Inc.
  Available at http://support.sas.com/resources/papers/proceedings12/004-2012.pdf.

CONTACT INFORMATION

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

Dylan Ellis
Mathematica Policy Research
1100 1st Street NE, 12th Floor
Washington, DC 20002-4221
(202) 554-7542
djellis@mathematica-mpr.com
www.linkedin.com/in/dylanellis