Paper 298-2013

FCMP – Why?

Lisa Eckler, Lisa Eckler Consulting Inc., Toronto, ON

ABSTRACT

PROC FCMP allows a SAS® programmer the opportunity to create user-defined functions in SAS. Prior to the availability of FCMP in SAS 9, SAS macros or linked routines were often used to achieve a similar – but less elegant – effect. This paper examines the advantages of FCMP over the earlier alternatives and why it is therefore so valuable to the programmer.

BACKGROUND

What is FCMP? FCMP is an acronym for Function Compiler. What does FCMP do? "The SAS Function Compiler (FCMP) procedure enables you to create, test, and store SAS functions, CALL routines, and subroutines before you use them in other SAS procedures or DATA steps." (Source: SAS Institute Inc. 2012. Base SAS® 9.3 Procedures Guide.) PROC FCMP is the procedure used to invoke the SAS Function Compiler. It puts the power to create user-defined functions, which behave the same way native SAS functions do, into the hands of the programmer. PROC FCMP was introduced for limited procedures with SAS 9 and became available to the DATA step with 9.2.

What is a function? In the mathematical sense, a function is a transformation from a set of one or more values to a singular result, by means of some calculation. While we think of the inputs to a function as values explicitly passed in, it is possible to map from implicit values such as the system timestamp. It's important to understand that a SAS *function* doesn't alter any values in the environment other than the *result,* and the result requires an *assignment* statement. The statement to invoke a function takes the form

```
        result_var = function_name(data_step_var<,another_data_step_var>);
```

A *routine* is similar to a *function* except without the emphasis on returning a singular value. A routine doesn't require an assignment statement; it may take an action without returning a result to a variable, or may return multiple result variables.

There have been several users group papers which address the syntax and *how to* use PROC FCMP, with examples. See Secosky (2007) and Eberhardt (2009) for details on how to implement user-written functions. There are also several new papers to be delivered at SAS Global Forum 2013, which will showcase the use of PROC FCMP. Here we will focus on *why to* use PROC FCMP in its simplest form, and particularly the advantages of FCMP over other methods of coding to achieve the same result.

FEATURES OF COMPILED FUNCTIONS

- Because a function is independent of the step it's called from, there is no risk of inadvertently altering an existing variable in the data set being processed or leaving stray interim variables. The result is the only variable that can be altered by invoking a function. In this way, functions may be considered self-cleaning. For code that may be widely shared and used across applications, this is a great advantage.
- Like a macro or INCLUDE module, a user-defined function may be stored in a central library and shared across programs or applications.
- Syntax to call a function is more familiar than macro language to many SAS programmers and we don't have to be concerned about quotes or ampersands.

So, should I just start turning my data step code into functions now? No! First, it pays to be familiar with existing functions. There's no elegance in creating an unnecessary or redundant function. Also, the name of a user-defined function cannot be the same as the name of a SAS-supplied function. There's no advantage in turning DATA step statements into a function unless those statements appear repeatedly, whether in the same program or across multiple programs. There's real benefit in using a function when the same conditions are applied repeatedly but with assignment to a different result variable.

EXAMPLES: DATA STEP CODE VERSUS COMPILED FUNCTION

Let's look at a simple program and the advantages of using a compiled function. There are several different ways to accomplish the same thing in SAS. Often, during development, a SAS program will begin with a structure that's quickest and easiest to write and evolve into something that's easier to read and maintain, and then further on to something that is more robust and supports and encourages reusable code. Here's a structural *example* <note: not actual code> for a program that starts out looking much the same as the way we might describe the logical requirement in English:

```
** Example 1:  In-stream code embedded in the DATA step **;

data CALC_RESULTS;
     set MY_DATA;
     if <major_condition_1 is true> then do;
        if <minor_condition_1 is true> then result_1 = 1;
           else if <minor_condition_2 is true> then result_1 = 2;
           else if <minor_condition_3 is true> then result_1 = 3;
        end;
        else if <major_condition_2 is true> then do;
                if <minor_condition_1 is true> then result_2 = 1;
                   else if <minor_condition_2 is true> then result_2 = 2;
                   else if <minor_condition_3 is true> then result_2 = 3;
                end;
run;
```

The code above was easy to write and would work fine, but it is repetitive and for a lengthy set of conditions it might yield a very lengthy DATA step to accomplish something fairly simple. A first step toward improving the readability of the program might be to isolate the repetitive statements and use a LINK/RETURN block of source code:

```
** Example 2:  LINK/RETURN block **;

      data CALC_RESULTS;
           drop result;   ←
           set MY_DATA;
           if <major_condition_1 is true> then do;
              LINK BLOCK_OF_CODE;       ←
              result_1 = result;
             end;
             else if <major_condition_2 is true> then do;
                    LINK BLOCK_OF_CODE;       ←
                    result_2 = result;
                   end;
     return;

     BLOCK_OF_CODE:        ←
          if <minor_condition_1 is true> then result = 1;
             else if <minor_condition_2 is true> then result = 2;
             else if <minor_condition_3 is true> then result = 3;
     return;
     run;
```

In the program segment above, in order to use the same LINKed code, we have introduced a new interim variable. It would therefore be necessary to make sure that variable name wasn't already used in the data set and, for good housekeeping, to drop that interim variable from the new data set being created. While the DATA step may now be simpler to read, we have done nothing to improve the integrity of the code or make it reusable. As the program evolves, the next stage might be to move the code from the LINK/RETURN block out of the DATA step and into a simple macro or INCLUDE module:

```
      ** Example 3:  Defining and calling a macro to substitute code **;

      %macro BLOCK_OF_CODE;      ←
           if <minor_condition_1 is true> then result = 1;
              else if  <minor_condition_2 is true> then result = 2;
              else if  <minor_condition_3 is true> then result = 3;
      %mend BLOCK_OF_CODE;
```

The macro above could be called by the step below:

```
      data CALC_RESULTS;
           drop result;
           set MY_DATA;
           if <major_condition_1 is true> then do;
              %BLOCK_OF_CODE;    ←
              result_1 = result;
             end;
             else if <major_condition_2 is true> then do;
                   %BLOCK_OF_CODE;    ←
                   result_2 = result;
                  end;
      run;
```

The BLOCK_OF_CODE macro may be stored in a separate file in a macro library or the statements inside the BLOCK_OF_CODE macro may be moved to a separate file in an INCLUDE library and invoked with

`%INCLUDE MY_LIB(BLOCK_OF_CODE);` instead of `%BLOCK_OF_CODE;`

to achieve the same effect. Now, we have a module that can be easily shared by multiple programs.

Still, with any of the above approaches, even if the code is housed outside of the DATA step, it gets substituted into the DATA step for execution. This limits the flexibility and reusability of the code because it is fully dependent on variable names within the DATA step and there is a risk of adding or modifying variables in the data set.

The code below will compile and store a function to calculate the same result as the examples above. In this example, MY_LIB is the library reference, FUNCS is the data set name and TEST is the package name.

```
** Example 4a:  Defining a compiled function **;

proc fcmp outlib=MY_LIB.FUNCS.TEST;     ←
     function BLOCK_OF_CODE( minor_condition_1,
                             minor_condition_2,
                             minor_condition_3);
     if <minor_condition_1 is true> then result = 1;
        else if <minor_condition_2 is true> then result = 2;
        else if <minor_condition_3 is true> then result = 3;
     return(result);
endsub;
```

Once the function has been compiled to a permanent library, it can be used in programs by supplying the library information and referring to the function by name (in this case, BLOCK_OF_CODE) with the appropriate arguments:

```
** Example 4b:  Calling a compiled function **;

options CMPLIB=MY_LIB.FUNCS;      ←

data CALC_RESULTS;
       set MY_DATA;
      if <major_condition_1 is true>
         then result_1 = BLOCK_OF_CODE( minor_condition_1,  ←
                                        minor_condition_2,
                                        minor_condition_3);
         else if <major_condition_2 is true>
                then result_2 = BLOCK_OF_CODE( minor_condition_1,  ←
                                               minor_condition_2,
                                               minor_condition_3);
run;
```

Note that we are not introducing the variable named *result* into the DATA step. The result of the function is assigned to a DATA step variable. In this example, the result is returned to a different variable name from each of the places it's called. While the same effect could be achieved using a macro function with parameters, that would require some familiarity with macro language syntax, whereas all of this has been accomplished with DATA step syntax.

ADVANCED APPLICATION

Beginning with SAS 9.2, we have the ability to *invoke a PROC from within a user-defined function* inside a DATA step. This means not just queuing the request to run a PROC after the DATA step ends, but actually suspending the DATA step until after the PROC has run and completed. (See Secosky, 2012 for discussion and examples.) This is an advanced application of PROC FCMP that goes beyond the packaging of DATA step statements into functions. It will allow manipulation of multiple data sets, for example, in order to derive and return a single result value from a function assignment, all during the execution of a DATA step. This opens a new realm of possibilities but requires the coding of both a function and a macro to implement, so it won't make our programs simpler or more readable.

CONCLUSIONS

Compiled functions remove the possibility of inadvertently altering the value of a variable in the data being operated on or adding stray variables. Functions can be stored separately from the program, which reduces program length, improves readability and facilitates safe and efficient sharing of functions across programs, applications and users. When used under the right circumstances and for the right reasons, PROC FCMP can be a wonderful tool for sharing DATA step code across programs while protecting the integrity of data within the data set being processed.

REFERENCES

Eberhardt, Peter. "A Cup of Coffee and Proc FCMP: I Cannot Function Without Them. "*Proceedings of the SAS Global Forum 2009 Conference.* http://support.sas.com/resources/papers/proceedings09/147-2009.pdf

SAS Institute Inc. 2009. SAS® 9.2 Macro Language: Reference. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2012. Base SAS® 9.3 Procedures Guide, Second Edition. Cary, NC: SAS Institute Inc.

Secosky, Jason. 2007 "User-Written DATA Step Functions. "*Proceedings of the SAS Global Forum 2007 Conference.* http://www2.sas.com/proceedings/forum2007/008-2007.pdf

Secosky, Jason. 2012 "Executing a PROC for a DATA Step." *Proceedings of the SAS Global Forum 2012 Conference*. http://support.sas.com/resources/papers/proceedings12/227-2012.pdf

ACKNOWLEDGEMENTS

This paper was inspired by a Super Demo delivered by Jason Secosky at SAS Global Forum 20102, "Top 3 Reasons for Using DATA Step Functions over Macros".

CONTACT INFORMATION

Your comments are welcome.

Lisa Eckler
lisa.eckler@sympatico.ca