

User-Written DATA Step Functions

Jason Secosky, SAS Institute Inc., Cary, NC

ABSTRACT

For years, users have asked for the ability to write functions in DATA step syntax. The day has come when this is possible. In SAS[®] 9, the FCMP procedure provides the ability to write functions using DATA step syntax. In SAS[®] 9.2, these functions can be called from a DATA step. This paper explains how to write functions with PROC FCMP and how these functions are invoked from a DATA step. It also contains caveats when calling PROC FCMP functions from a DATA step.

INTRODUCTION

Solving a difficult programming problem is made possible by combining smaller program units. The DATA step provides the ability to specify small program units with the LINK and RETURN statements or with textural substitution using SAS macros. While the LINK and RETURN statements enable you to use small program units, the program units cannot have parameters and are difficult to reuse in other programs. Macros are easier to reuse; however, they are not independent from the main program, they involve non-DATA step syntax, and substantial use of macros can result in illegible code.

To ease writing complex DATA step programs, the DATA step needs a less cumbersome way to code reusable, independent program units written in DATA step syntax. PROC FCMP provides the ability to write true functions and CALL routines in DATA step syntax that are stored in a data set. In SAS 9.2, FCMP routines can be called from the DATA step like any other SAS function. This enables programmers to more easily read, write, and maintain complex code with independent and reusable subroutines.

The first section of this paper presents an example of creating an FCMP function and calling it from a DATA step. The next section describes the syntax and features of PROC FCMP. The third section shows how to write a recursive directory walker with FCMP and call it from DATA step code. The final section lists several differences between FCMP and DATA step syntax. Throughout the paper, the term *FCMP routines* refers to both functions and CALL routines that are created using PROC FCMP.

WRITING YOUR OWN FUNCTIONS

PROC FCMP is a Base SAS[®] procedure that enables users to write functions and CALL routines using DATA step syntax. FCMP routines are stored in a data set and can be called from several SAS/STAT[®], SAS/ETS[®], and SAS/OR[®] software procedures, like the NLIN, MODEL, and NLP procedures. In SAS 9.2, FCMP routines can be called from a DATA step. The following program shows the syntax to create and call an FCMP function that computes the study day during a drug trial:

```
proc fcmp outlib=sasuser.funcs.trial;
  function study_day(intervention_date, event_date);
    if event_date < intervention_date then
      return(event_date - intervention_date);
    else
      return(event_date - intervention_date + 1);
  endsub;

options cmplib=sasuser.funcs;
data _null_;
  start = '15Feb2006'd;
  today = '27Mar2006'd;
  sd = study_day(start, today);
  put sd=;
run;
```

This code creates a function named STUDY_DAY in a package named Trial. The package is stored in the data set Sasuser.Funcs. A *package* is a collection of routines that have unique names. STUDY_DAY takes two numeric parameters, INTERVENTION_DATE and EVENT_DATE. The body of the routine uses DATA step syntax to compute the difference between the two dates, where days before INTERVENTION_DATE begin at -1 and get smaller and days after and including INTERVENTION_DATE begin at 1 and get larger. This function never returns 0 for a study day.

STUDY_DAY is called from DATA step code like any other function. When the DATA step encounters a call to STUDY_DAY, it does not find this function in its traditional library of functions. When this occurs, it searches each of the data sets specified in the CMPLIB system option for a package that contains STUDY_DAY. In this case, it finds STUDY_DAY in Sasuser.Funcs.Trial and calls the function, passing in the variable values for Start and Today. The result of STUDY_DAY is assigned to the variable SD.

This example demonstrates how writing FCMP routines produce:

- A simpler program—Abstracting common operations makes the program easier to read, write, and modify.
- An independent routine—A program that calls a routine is not affected by the routine's implementation.
- A reusable routine—Any program that has access to the data set where the routine is stored can call the routine.

Macros and the LINK and RETURN statements address simplicity and reusability, yet these abstraction mechanisms are cumbersome and often require programming tricks in order to implement maintainable code (Chung 2004). FCMP routines provide a simpler syntax, truly independent routines, and an easy way to store and share routines.

PROC FCMP

PROC FCMP is invoked to create functions and CALL routines. The syntax for the procedure is:

```
PROC FCMP OUTLIB=libname.dataset.package INLIB=in-libraries ENCRYPT;  
  routine-declarations;
```

The OUTLIB= option is required and specifies the package where routines declared in the *routine-declarations* section are stored. The routines declared in *routine-declarations* may call FCMP routines that exist in other packages. In order to find these routines to check the validity of the call, the data sets specified on the INLIB= option are searched, where INLIB=*in-libraries* can be:

```
inlib=library.dataset  
inlib=(library1.dataset1 library2.dataset2 ... libraryN.datasetN)  
inlib=library.datasetM - library.datasetN
```

If the routines being declared do not call FCMP routines in other packages, you do not need to specify the INLIB= option. The ENCRYPT option specifies that the routines are encrypted when stored.

DECLARING FUNCTIONS

Routine-declarations are where one or more functions or CALL routines are declared. A routine consists of four parts: a name, parameters, a body of code, and a RETURN statement. These four parts are specified between the FUNCTION or SUBROUTINE keyword and an ENDSUB keyword. For functions, the syntax is:

```
FUNCTION name (parameter-1, ..., parameter-N);  
  program-statements;  
  RETURN (expression);  
ENDSUB;
```

After the FUNCTION keyword, the name of the function and its parameters are specified. Parameters in the function declaration are termed *formal parameters* and can be used within the body of the function. To specify a string parameter, a dollar sign (\$) is placed after the parameter name. For functions, all parameters are passed by value. This means the value of the actual parameter, the variable or value passed to the function from the calling environment, is copied before being used by the function. This ensures any modification of the formal parameter by the function doesn't change the original value.

DATA step _temporary_ arrays can also be passed to FCMP routines. Arrays are passed by value. That is, the array is copied before calling the routine and the copy is used by the routine. Copying large arrays can be costly. The next section, "Declaring CALL Routines", describes how to pass arrays without copying by using the OUTARGS statement. The syntax for specifying a formal array parameter is:

```
FUNCTION name (numeric-array-parameter[*], character-array-parameter[*] $);
```

Program-statements are a series of DATA step statements that describe the work to be done by the function. In general, most DATA step statements and functions are accessible from FCMP routines. The DATA step file and data set I/O statements, like INPUT, INFILE, SET, and MERGE, are not available from FCMP routines. However, the PUT

statement is supported to a certain extent. For a more complete list of differences, see the section titled "FCMP and DATA Step Differences" later in this paper.

All functions must return a value. This is done with the RETURN statement. The RETURN statement accepts a parenthesized expression, which is the value to be returned to the calling environment. The function declaration ends with an ENDSUB statement.

DECLARING CALL ROUTINES

CALL routines are also declared within *routine-declarations* by using the SUBROUTINE keyword instead of the FUNCTION keyword. Functions and CALL routines have the same form, except CALL routines do not return a value and can modify their parameters. The parameters to be modified are specified in an OUTARGS statement. The syntax of a CALL routine declaration is:

```
SUBROUTINE name (parameter-1, ..., parameter-N);  
  OUTARGS out-parameter-1, ..., out-parameter-N;  
  program-statements;  
  RETURN;  
ENDSUB;
```

The formal parameters listed in the OUTARGS statement are passed by reference instead of by value. This means that any modification of the formal parameter by the CALL routine will modify the original variable that was passed. It also means the value is not copied when the CALL routine is invoked. Reducing the number of copies can improve performance when passing large amounts of data or arrays between a CALL routine and the calling environment.

A RETURN statement is optional within the definition of a CALL routine. When a RETURN statement executes, execution is immediately returned to the caller. A RETURN statement within a CALL routine does not take a value to be returned as CALL routines do not return values like functions do.

VARIABLE SCOPE

A critical part of routines and programs being independent of each other is variable scope. A variable's *scope* is the section of code where a variable's value can be used. In the case of FCMP routines, variables declared outside of a routine are not accessible inside a routine. Also, variables declared inside a routine are not accessible outside of the routine. Variables declared within a routine are called *local variables* because their scope is "local" to the routine.

Local variables store intermediate results of a computation and cannot be accessed after a routine returns. When a routine is called, memory for local variables is pushed on the call stack (Wikipedia 2007). When the routine returns, the memory used by local variables is popped off the call stack.

Variable scope can get confusing when local variables in different routines have the same name. When this occurs, each local variable is distinct. In the following program, the DATA step and CALL routines SUBA and SUBB have a local variable named x. Each x is distinct from the other x variables. When this program executes, the DATA step calls SUBA and SUBA calls SUBB. Each environment writes the value of x to the log. The log output shows how each x is distinct from the others.

```
proc fcmp outlib=sasuser.funcs.math;  
  subroutine subA();  
    x = 5;  
    call subB();  
    put 'In subA: ' x=;  
  endsub;  
  
  subroutine subB();  
    x = 'subB';  
    put 'In subB: ' x=;  
  endsub;  
run;  
  
options cmplib=sasuser.funcs;  
data _null_;  
  x = 99;  
  call subA();  
  put 'In DATA step: ' x=;  
run;
```

Output:

```
In subB: x=subB
In subA: x=5
In DATA step: x=99
```

RECURSION

Recursion is a problem-solving technique that reduces a problem to a smaller one that is simpler to solve and then combines the results of the simpler solution(s) to form a complete solution. A *recursive function* is a function that calls itself, either directly or indirectly. FCMP routines can be recursive.

Each time a routine is called, whether it is recursive or not, memory for local variables is pushed on the call stack. The memory on the call stack ensures independence of local variables for each call. This can be confusing when a routine calls itself. When a routine calls itself, both the caller and callee must have their own set of local variables for intermediate results. If the callee were able to modify the caller's local variables, it would not be easy to program a recursive solution. A call stack ensures the independence of local variables for each call.

In the next example, the FCMP routine ALLPERMK takes two parameters, n and k and prints all $P(n, k) = n! / (n-k)!$ permutations that contain exactly k out of the n elements. The elements are represented as binary (0/1) values. The function ALLPERMK calls the recursive function PERMK to traverse the entire solution space and only output items that match a particular filter.

```
proc fcmp outlib=sasuser.funcs.math;
  subroutine allpermk(n, k);
    array scratch[1]/nosymbols;
    call dynamic_array(scratch, n);
    call permk(n, k, scratch, 1, 0);
  endsub;

  subroutine permk(n, k, scratch[*], m, i);
    outargs scratch;
    if m-1 = n then do;
      if i = k then
        put scratch[*];
      end;
    else do;
      scratch[m] = 1;
      call permk(n, k, scratch, m+1, i+1);
      scratch[m] = 0;
      call permk(n, k, scratch, m+1, i);
    end;
  endsub;
run; quit;

options cmplib=sasuser.funcs;
data _null_;
  call allpermk(5,3);
run;
```

Output:

```
1 1 1 0 0
1 1 0 1 0
1 1 0 0 1
1 0 1 1 0
1 0 1 0 1
1 0 0 1 1
0 1 1 1 0
0 1 1 0 1
0 1 0 1 1
0 0 1 1 1
```

This program uses the */NOSYMBOLS* option on the *ARRAY* statement to create an array without a variable for each array element. A */NOSYMBOLS* array can only be accessed with an array reference, *scratch[m]*, and is equivalent to a *DATA* step *_temporary_* array. A */NOSYMBOLS* array takes less memory than a regular array because no space is allocated for variables. *ALLPERMK* also uses *FCMP* dynamic arrays, which are discussed in the next section.

DYNAMIC ARRAYS

In *FCMP* routines, arrays can be resized. This is done by calling the built-in *CALL* routine *DYNAMIC_ARRAY*. The syntax is:

```
CALL DYNAMIC_ARRAY (array, new-dim1-size, ..., new-dimN-size);
```

The *DYNAMIC_ARRAY* *CALL* routine is passed the array to be resized, and a new size for each dimension of the array. In *ALLPERMK*, a *scratch* array that is the size of the number of elements being permuted is needed. When the function is created, this value is not known since it is passed in as parameter *N*. A dynamic array enables the routine to allocate the amount of memory that is needed, instead of having to create an array that is large enough to handle all possible cases.

When using dynamic arrays, keep in mind that support is limited to *FCMP* routines. When an array is resized, the resized array is only available within the routine that resized it. It is not possible to resize a *DATA* step array or to return an *FCMP* dynamic array to a *DATA* step.

DIRECTORY TRAVERSAL EXAMPLE

Several users have requested functions that traverse a directory hierarchy. Implementing this functionality with *DATA* step and macro code is difficult because recursion or pseudo-recursion is not easy to code. In this section, we develop a routine, named *DIR_ENTRIES*, that fills an array with the full pathname of all the files in a directory hierarchy. This exercise shows the similarity between *FCMP* and *DATA* step syntax and underscores how *FCMP* routines simplify a program and produce independent, reusable code.

DIR_ENTRIES takes as input a starting directory, a result array to fill with pathnames, an output parameter that is the number of pathnames placed in the result array, and another output parameter that indicates if the complete result set was truncated because the result array was not big enough. The flow of control for *DIR_ENTRIES* is:

1. Open the starting directory.
2. For each entry in the directory,
 - If the entry is a directory, call *DIR_ENTRIES* to fill the result array with the subdirectory's pathnames.
 - Otherwise, the entry is a file, so add the file's path to the result array.
3. Close the starting directory.

OPENING AND CLOSING A DIRECTORY

We begin by abstracting two common operations into their own routines. Opening and closing a directory are handled by the *CALL* routines *DIROPEN* and *DIRCLOSE*. *DIROPEN* takes a directory path and has the following flow of control:

1. Create a fileref for the path using the filename function.
2. If the filename function fails, output an error message to the log and return.
3. Otherwise, use the *DOPEN* function to open the directory and get a directory ID.
4. Clear the directory fileref.
5. Return the directory ID.

The *DIRCLOSE* *CALL* routine is passed a directory ID, which is passed to *DCLOSE*. *DIRCLOSE* sets the passed directory ID to missing so that an error occurs if a program tries to use the directory ID after the directory has been closed. The following code implements the *DIROPEN* and *DIRCLOSE* call routines:

```
proc fcmp outlib=sasuser.funcs.dir;
  function diropen(dir $);
    length dir $ 256 fref $ 8;

    rc = filename(fref, dir);
    if rc = 0 then do;
      did = dopen(fref);
      rc = filename(fref);
    end;
    else do;
```

```

        msg = sysmsg();
        put msg '(DIROPEN(' dir= '));
        did = .;
    end;
    return(did);
endsub;

subroutine dirclose(did);
    outargs did;
    rc = dclose(did);
    did = .;
endsub;

```

GATHERING FILENAMES

File paths are collected by the DIR_ENTRIES CALL routine. DIR_ENTRIES takes:

- a starting directory
- a result array to fill
- an output parameter to fill with the number of entries in the result array
- another output parameter to set to 0 if all pathnames fit in the result array or 1 if some of the pathnames do not fit into the array.

You do not have to specify the result array in the OUTARGS statement because all arrays are output parameters.

The body of DIR_ENTRIES is almost identical to the code to implement this in a DATA step, yet DIR_ENTRIES is a real CALL routine that is easily reused in several different programs.

DIR_ENTRIES calls DIROPEN to open a directory and get a directory ID. The routine then calls DNUM to get the number of entries in the directory. For each entry in the directory, DREAD is called to get the name of the entry. Now that we've got the entry name, the routine calls MOPEN to determine if the entry is a file or a directory.

If the entry is a file, MOPEN returns a positive value. In this case, the full path to the file is added to the result array. If the result array is full, the truncation output argument is set to 1.

If the entry is a directory, MOPEN returns a value less than or equal to 0. In this case, DIR_ENTRIES needs to gather the pathnames for the entries in this subdirectory. This is done by recursively calling DIR_ENTRIES and passing the subdirectory's path as the starting path. When DIR_ENTRIES returns, the result array contains the paths of the subdirectory's entries.

```

subroutine dir_entries(dir $, files[*] $, n, trunc);
    outargs files, n, trunc;
    length dir entry $ 256;

    if trunc then return;

    did = diropen(dir);
    if did <= 0 then return;

    dnum = dnum(did);
    do i = 1 to dnum;
        entry = dread(did, i);

        /* If this entry is a file, then add to array */
        /* Else entry is a directory, recurse.          */
        fid = mopen(did, entry);
        entry = trim(dir) || '\' || entry;
        if fid > 0 then do;
            rc = fclose(fid);
            if n < dim(files) then do;
                trunc = 0;
                n = n + 1;
                files[n] = entry;
            end;
        else do;

```

```

        trunc = 1;
        call dirclose(did);
        return;
    end;
end;
else
    call dir_entries(entry, files, n, trunc);
end;

    call dirclose(did);
    return;
endsub;

```

CALLING DIR_ENTRIES FROM A DATA STEP

Invoke DIR_ENTRIES like any other DATA step CALL routine. An array is declared with enough entries to hold all the files that may be found. Then, the routine is called. Upon return from the routine, the result array is looped over and each entry in the array is output to the log.

```

options cmplib=sasuser.funcs;
data _null_;
    array files[1000] $ 256 _temporary_;
    dnum = 0;
    trunc = 0;
    call dir_entries("c:\logs", files, dnum, trunc);
    if trunc then put 'ERROR: Not enough result array entries.  Increase array size.';
    do i = 1 to dnum;
        put files[i];
    end;
run;

```

Output:

```

c:\logs\2004\qtr1.log
c:\logs\2004\qtr2.log
c:\logs\2004\qtr3.log
c:\logs\2004\qtr4.log
c:\logs\2005\qtr1.log
c:\logs\2005\qtr2.log
c:\logs\2005\qtr3.log
c:\logs\2005\qtr4.log
c:\logs\2006\qtr1.log
c:\logs\2006\qtr2.log

```

This example shows the similarity between FCMP syntax and the DATA step. For instance, numeric expressions and flow of control statements are identical. The abstraction of DIROPEN into a FCMP function simplifies DIR_ENTRIES. All of the FCMP routines created can be reused by other DATA steps without any need to modify the routines to work in a new context.

FCMP AND DATA STEP DIFFERENCES

PROC FCMP was originally developed as a programming language for several SAS/STAT, SAS/ETS, and SAS/OR procedures. Because the implementation isn't identical to the DATA step, there are differences between the two languages. In this section, we highlight some of the differences between the two languages. For a complete list of differences, please refer to *The FCMP Procedure* (SAS Institute Inc. 2003).

PUT STATEMENT

The syntax for the PUT statement is similar in FCMP and the DATA step, yet their operation can be different. The FCMP PUT statement is typically used as a debugging tool, not as a report or file creation tool as in the DATA step. Therefore, FCMP supports some, though not all, features of the DATA step PUT statement and adds features that are useful for debugging.

The PUT statement in FCMP follows the output of each item with a space. This is similar to list mode output in the DATA step. Detailed control over column and line position is supported to a lesser extent than in the DATA step.

FCMP's PUT statement evaluates an expression and outputs the result by placing the expression in parentheses. The DATA step does not have the ability to evaluate expressions in the PUT statement. In the following example, the expressions $x/100$ and $\sqrt{y}/2$ are evaluated and output.

```
put (x/100) (sqrt(y)/2);
```

Because parentheses are used for expression evaluation, they cannot be used for variable or format lists as in the DATA step. Other DATA step features not available in FCMP are line pointers, format modifiers, column output, and features provided by FILE statement options, like DLM= and DSD.

IF EXPRESSIONS

An IF expression allows IF...THEN ... ELSE conditions to be evaluated within an expression. IF expressions are supported by FCMP and not by the DATA step. Some expressions can be simplified with IF expressions by not splitting the expression between IF...THEN ... ELSE statements. For instance, the following two samples of code are equivalent, yet the IF expression is more succinct:

```
x = if y < 100 then 1 else 0;

if y < 100 then
  x = 1;
else
  x = 0;
```

The consequent and alternative of an IF expression are expressions. This means that parentheses are used to group operations instead of DO ... END blocks.

DATA SET INPUT AND OUTPUT

FCMP does not support the DATA and OUTPUT statements for creating and writing to an output data set. Nor does it support the SET, MERGE, UPDATE, or MODIFY statements for data set input. Data are typically transferred into and out of FCMP routines with parameters. If a large amount of data needs to be transferred, one suggestion is to pass arrays to an FCMP routine.

FILE INPUT AND OUTPUT

FCMP does have PUT and FILE statements, yet the FILE statement is limited to the LOG and PRINT destinations. There are no INFILE or INPUT statements in FCMP.

ARRAYS

FCMP uses parentheses after a name to represent a function call. When referencing an array, square braces [] or curly braces { } must be used. For an array named ARR, this would look like ARR[i] or ARR{i}. FCMP limits the number of dimensions for an array to 6.

DO STATEMENT

The DO statement in FCMP doesn't support character loop control variables. While the following code will work in a DATA step, it will not work in FCMP:

```
do i = 'a', 'b', 'c';
```

DATA STEP DEBUGGER

When using the DATA step debugger, FCMP routines are like any other routine, it is not possible to debug the code in the routine. One way to debug FCMP routines is to use the PUT statement within the routine.

ADDITIONAL FEATURES

There are several noteworthy features not mentioned in this paper that are summarized here. Details of these features can be found in *The FCMP Procedure* (SAS Institute Inc. 2003).

- PROC REPORT uses the DATA step to evaluate compute blocks. With the DATA step being able to call FCMP routines, they are also callable from PROC REPORT compute blocks.
- The FCMP Package Viewer is an application for traversing packages of functions that is located on the Solutions menu of an interactive SAS session.
- FCMP has the ability to call C and C++ functions that have been registered with PROC PROTO. In addition, there is support for C structures within FCMP.

- FCMP has a SOLVE function for computing implicit values of a function.
- Many Microsoft Excel functions, not typically available in SAS, have been implemented in FCMP. These can be found in the SAShelp.Slkwxl data set. These functions were written for PROC SYLK, a Base SAS procedure for importing and executing SYLK spreadsheets.

CONCLUSION

DATA step LINK and RETURN statements and macros provide code abstraction and sharing, yet programming with these mechanisms can be awkward and cumbersome. PROC FCMP is a solution for this awkwardness. With FCMP routines, a library of routines can be built using DATA step syntax. FCMP routines make complex programs simpler by abstracting common computations into named program units. Also, FCMP routines are independent from their use and can be reused in any DATA step program that has access to their storage location. FCMP routines enable programmers to more easily read, write, and maintain complex code.

REFERENCES

Chung, C.Y. 2004. "Recursive Subroutine Macros." *Proceedings of the Seventeenth Annual Northeast SAS Users Group Meeting*. Cary, NC: SAS Institute Inc. Available at www.nesug.org/html/Proceedings/nesug04/po/po02.pdf.

SAS Institute Inc. 2003. *The FCMP Procedure*. Cary, NC: SAS Institute Inc. Available at support.sas.com/documentation/onlinedoc/base/91/fcmp.pdf.

Wikipedia contributors. 2006. Call Stack. *Wikipedia, The Free Encyclopedia*. (Accessed January 15, 2007) Available at en.wikipedia.org/w/index.php?title=Call_stack&oldid=94363650.

ACKNOWLEDGMENTS

Stacey Christian implemented and supports PROC FCMP, and Al Kulik extended the DATA step to be able to call FCMP routines. The author appreciates their help reviewing this paper and describing several implementation details. The author also appreciates Kevin Hobbs and Amber Elam for reviewing this paper.

CONTACT INFORMATION

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

Jason Secosky
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-677-8000
Jason.Secosky@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.