

FCMP: A Powerful SAS® Procedure You Should Be Using

Bill McNeill, Andrew Henrick, Mike Whitcher, and Aaron Mays, SAS Institute Inc.

ABSTRACT

The FCMP procedure is the SAS® Function Compiler procedure. As the name suggests, it enables you to create and save functions (and subroutines) that can be of varying size and complexity. The functions and subroutines can then be integrated with other built-in SAS functions in your SAS programs. This integration simplifies your SAS programming and facilitates code reuse that can be shared with your fellow SAS programmers. You should be using this procedure. The modularity of your code will increase. The readability of your code will increase. The maintainability of your code will increase. Your productivity will increase. Come learn how this powerful procedure will make your programming life simpler.

INTRODUCTION

Keeping the software easy to read and understand is an important part of making the software maintainable. Just about all software needs to be maintained if not enhanced. The “KISS” (Keep It Simple Stupid) methodology is an important part of making software easy to maintain and enhance. Simpler software is easier to understand and modify. Functions and subroutines, along with modularity, are proven methods to make software easier to understand. They allow for breaking complicated tasks into smaller, simpler blocks of code that are more easily understood.

In writing software using SAS, you probably have knowledge of SAS functions and CALL routines along with including SAS files (with “%INCLUDE”) for modularity, and named custom SAS macros for isolating repetitive tasks. In SAS 9.2, the FCMP procedure was added to the programmer’s tool chest. With PROC FCMP you can write custom functions and subroutines, save them to special data sets called packages, and call them from practically anywhere in your SAS programs. And, if you make the packages available, the functions and subroutines in the packages can be shared with other SAS programmers. The FCMP procedure basically allows you to add to the extensive list of SAS functions and CALL routines available within SAS. As you will see, functions and subroutines that are created using the FCMP procedure can be quite powerful and can be customized to your specific needs. Let us start by demonstrating how to create these custom functions and subroutines that can make your SAS programs easier to read, understand, and maintain - thereby making your programming life easier.

Note that for the remainder of this paper, the term “function” refers to both functions and subroutines being created using PROC FCMP. The term “subroutine” specifically refers to information about the FCMP SUBROUTINE statement.

DEFINING CUSTOM FUNCTIONS

Depending on the programming language, functions and subroutines have various meanings. For PROC FCMP, the main difference is that a function usually passes back one value while a subroutine returns zero or more values.

FUNCTIONS

To define a function, use this syntax within PROC FCMP:

```
FUNCTION function-name(argument-1 <, argument-2, ...) <VARARGS> <$>
    <length>;
... more-program-statements ...
RETURN (expression);
ENDSUB;
```

Here is an example of a function to compute the cost of a purchase including tax:

```
LIBNAME billsLib 'C:/SAS_Global_Forum/2018/sasuser/';

PROC FCMP OUTLIB=billsLib.paper.tax;
  FUNCTION priceWithTax_func(price);
    STATIC taxRate;
    IF (MISSING(taxRate))
      THEN taxRate = (7.25 / 100);
    taxedPrice = (price * (1 + taxRate));
    RETURN (taxedPrice);
  ENDSUB;
QUIT;
```

This example created a function called “priceWithTax_func” that takes a parameter, “price”. A new price that includes the sales tax, 7.25% in this case, is computed. By declaring the “taxRate” variable as STATIC, the tax rate needs to be computed only the first time the function is called. This is done by setting “taxRate” only when it has been initialized to a SAS numeric missing value. On repeated function calls, the variable “taxRate” retains the constant value previously computed. With the tax rate set, the variable “taxedPrice” can then be computed for the price passed into the function. The “taxedPrice” is then passed back to the caller. The PROC FCMP OUTLIB= option saves the “priceWithTax” function to the “tax” package of the “paper” member of the “billsLib” SAS library (with the library being set with the LIBNAME statement).

Expensive Computation Efficiency

A STATIC variable comes in quite handy in cases where it is expensive to compute a constant value. Computing the value over and over as the function is called repeatedly will be a performance hit. Declaring the variable STATIC means that the expensive computation only needs to occur once.

This next example is run from a new SAS session. To work in a new SAS session, the libref “billsLib” must be set. The option CMPLIB is needed to identify the package to search for the saved FCMP function, “priceWithTax”. The “priceWithTax” function is used in the DATA step just like any other SAS function or CALL routine. In this example, the function is used with the sashelp.cars data set to compute the after-tax cost of the vehicles:

```
LIBNAME billsLib 'C:/SAS_Global_Forum/2018/sasuser/';
OPTION CMPLIB=billsLib.paper;

/* TAXES (function) */
DATA work.carPriceWithTaxFun;
  SET sashelp.cars;
  FORMAT msrpWithTax DOLLAR8.;
  msrpWithTax = priceWithTax_func(msrp);
  OUTPUT;
RUN;

PROC PRINT DATA=work.carPriceWithTaxFun(
  KEEP=make model invoice msrp msrpWithTax
  OBS=5);
RUN;
```

The price of the vehicle is passed to the function and the after-tax price is passed back from the function. Here are the first 5 observations from printing only the key fields and the price data in the resulting data set:

Obs	Make	Model	MSRP	Invoice	msrpWithTax
1	Acura	MDX	\$36,945	\$33,337	\$39,624
2	Acura	RSX Type S 2dr	\$23,820	\$21,761	\$25,547
3	Acura	TSX 4dr	\$26,990	\$24,647	\$28,947
4	Acura	TL 4dr	\$33,195	\$30,299	\$35,602
5	Acura	3.5 RL 4dr	\$43,755	\$39,014	\$46,927

Figure 1: PROC PRINT Output of the carPriceWithTax Data Set (First 5 Observations)

SUBROUTINES

To define a subroutine, use this syntax within PROC FCMP:

```
SUBROUTINE subroutine-name (argument-1 <, argument-2, ...>) <VARARGS>;
OUTARGS out-argument-1 <, out-argument-2, ...>;
... more-program-statements ...
ENDSUB;
```

In this example, the same tax calculation is done with the subroutine instead of the function. This gives the chance to see the differences in defining and using functions and subroutines:

```
PROC FCMP OUTLIB=billsLib.paper.tax;
  SUBROUTINE priceWithTax_sub(price, taxedPrice);
    OUTARGS taxedPrice;
    STATIC taxRate;
    IF (MISSING(taxRate))
      THEN taxRate = (7.25 / 100);
    taxedPrice = (price * (1 + taxRate));
  ENDSUB;
QUIT;
```

In the SUBROUTINE statement, the “taxedPrice” variable is a new parameter to the subroutine. The OUTARGS statement is needed to update the “taxedPrice” value that is computed within the subroutine. Also, the RETURN statement has been removed.

As before with the function example, “priceWithTax_func”, this next example can run from a new SAS session:

```
LIBNAME billsLib 'C:/SAS_Global_Forum/2018/sasuser/';
OPTION CMPLIB=billsLib.paper;

DATA work.carPriceWithTaxSub;
  SET sashelp.cars;
  FORMAT msrpWithTax DOLLAR8.;
  msrpWithTax = 0;
  CALL priceWithTax_sub(msrp, msrpWithTax);
  OUTPUT;
RUN;
```

In calling the subroutine from the DATA step, the assignment statement is replaced by a CALL statement. The data set variable "msrpWithTax" is added to the subroutine call. This is the value that the subroutine updates and pass back with the calculated taxed price.

FUNCTION ARGUMENTS

In FCMP functions, all the arguments are, by default, passed by value. This means that you cannot retain the changed value of an argument outside of the function. All of the arguments are local to the function. The usual way to get a value from the FUNCTION statement is via the RETURN statement, which is assigned to the variable that is invoking the function call. In the FUNCTION statement example above, that variable is "msrpWithTax". This default behavior of the FUNCTION statement can be changed by using the OUTARGS statement (as explained below).

While the SUBROUTINE statement has the same defaults, those defaults are usually overridden. Notice in the SUBROUTINE statement example that a second argument is passed in and that argument name is included in the OUTARGS statement. The OUTARGS statement lists the variables whose values are updated and accessible from the SUBROUTINE statement caller. In this case, the price with the tax value is updated and passed back to the caller in the "msrpWithTax" variable.

CHARACTER ARGUMENTS AND VARIABLES

To specify passing a character argument to an FCMP function, put a dollar sign symbol "\$" after the argument (in this case, "string"). Here is an example that counts the vowels in a character string:

```
PROC FCMP OUTLIB=billsLib.paper.vowels;
  FUNCTION getVowelCount (string $);
    ARRAY vowel[5] $ ('a', 'e', 'i', 'o', 'u');

    vowelCount = 0;
    DO stringIndex=1 to LENGTH(string);
      stringChar = SUBSTR(string, stringIndex, 1);
      DO vowelIndex=1 TO dim(vowel);
        IF ( (stringChar EQ vowel[vowelIndex]) OR
            (stringChar EQ UPCASE(vowel[vowelIndex])) )
          THEN DO
            vowelCount = vowelCount + 1;
            LEAVE;
          END;
        END;
      END;
    RETURN(vowelCount);
  ENDSUB;
QUIT;
```

To keep the output of reasonable size, the next DATA step that calls the "getVowelCount" function uses only data from the first quarter of 1999 for the children product line in the children sports product group:

```
DATA work.vowels(KEEP=product_group vowelCount);
  SET sashelp.orsales(KEEP=quarter product_line
                    product_category product_group);
  WHERE quarter="1999Q1" AND
        product_line="Children" AND
        product_category="Children Sports";
  vowelCount=getVowelCount(product_group);
RUN;
```

```
PROC PRINT DATA=work.vowels;
  RUN;
```

Obs	Product_Group	vowelCount
1	A-Team, Kids	4
2	Bathing Suits, Kids	5
3	Eclipse, Kid's Clothes	6
4	Eclipse, Kid's Shoes	6
5	Lucky Guy, Kids	3
6	N.D. Gear, Kids	3
7	Olssons, Kids	3
8	Orion Kid's Clothes	6
9	Osprey, Kids	3
10	Tracker Kid's Clothes	5
11	Ypsilon, Kids	3

Figure 2: Vowel Count Output

This program also demonstrates using a character array within PROC FCMP. The vowel array is created and set in the ARRAY statement. The "\$" following the array variable name indicates that the array contains character data. The elements of the array are used via subscript (the "vowelIndex" variable).

In this next example, the "getVowelCount" function is called from an FCMP function that produces a sentence that indicates the number of vowels in the product group. This new function shows the usage of character variables within PROC FCMP:

```
PROC FCMP OUTLIB=billsLib.paper.vowels;
  FUNCTION getVowelCountStmt (product $) $;
    LENGTH vowelStmt $60;
    vowelClause1 = "There are ";
    vowelCount   = getVowelCount(product);
    vowelClause2 = " vowels in the product. ";
    vowelStmt    = CAT(vowelClause1, vowelCount, vowelClause2, product);
    RETURN(vowelStmt);
  ENDSUB;
QUIT;
```

The "\$" at the end of the FUNCTION statement notes that the function is passing back a character value. The LENGTH statement declares the "vowelStmt" variable as a character variable with sufficient space (60 characters) to contain the longest expected character value being passed back to the caller. In the following "getVowelCountStmt" function DATA step example, the same data that is used in the "getVowelCount" function example is used:

```
DATA work.vowelsStmt(KEEP=product_group vowelCountStmt);
  SET sashelp.orsales(KEEP=quarter product_line
                     product_category product_group);
  WHERE quarter="1999Q1" AND
         product_line="Children" AND
         product_category="Children Sports";
```

```

vowelCountStmt=getVowelCountStmt(product_group);
RUN;

PROC PRINT DATA=work.vowelsStmt(OBS=5);
RUN;

```

Obs	Product_Group	vowelCountStmt
1	A-Team, Kids	There are 4 vowels in the product A-Team, Kids
2	Bathing Suits, Kids	There are 5 vowels in the product Bathing Suits, Kids
3	Eclipse, Kid's Clothes	There are 6 vowels in the product Eclipse, Kid's Clothes
4	Eclipse, Kid's Shoes	There are 6 vowels in the product Eclipse, Kid's Shoes
5	Lucky Guy, Kids	There are 3 vowels in the product Lucky Guy, Kids

Figure 3: Vowel Statements in the Data Set (First 5 Observations)

DETERMINING AVAILABLE FUNCTIONS

To determine what functions are available to the current SAS session, use the LISTFUNCS option in the FCMP procedure statement.

```

PROC FCMP INLIB=billsLib.paper LISTFUNCS;
RUN;

```

Here is the type of information you see in the Output window:

The FCMP Procedure			
FCMP Function/Subroutine Listing			
Name	Type	Returns	Prototype
isPossibleCar	FCMP Function	Num	isPossibleCar(model \$ 32);
factorial	FCMP Function	Num	factorial(currentValue);
writeJsonFile	FCMP Function	Num	writeJsonFile(dsName \$ 32, age);
getArrayRowSum	FCMP Function	Num	getArrayRowSum(arrayToSum[], rowIndex);
getAverageTemp	FCMP Function	Num	getAverageTemp(monthlyTimeTemp[], timeIndex);
averageTempAtTime	FCMP Function	Num	averageTempAtTime(dsName \$ 32, timeIndex);
priceWithTax_test	FCMP Subroutine	Void	priceWithTax_test(price, taxedPrice);
			OUTARGS taxedPrice;

Figure 4: Partial Listing of Available Functions and Subroutines

DELETING A SAVED FUNCTION OR SUBROUTINE

If you need to delete a saved function or subroutine, use the DELETEDFUNC or DELETESUB statement respectively. Each statement takes the name of the function or subroutine to delete. The deletion is done from the named package specified in the OUTLIB option.

```
PROC FCMP INLIB=billsLib.paper OUTLIB=billsLib.paper.delete;
  DELETESUBR priceWithTax_test;
RUN;
```

DEBUGGING INFORMATION

When you need to see what is being executed by the function, you can trace the code execution with the TRACE option. As the code is executed, the steps taken by the code, along with the actual values used, are displayed in the output window. Note that the FCMP defined function needs to be called from within the FCMP procedure where the TRACE option is set.

```
PROC FCMP INLIB=billsLib.paper TRACE;
  ATTRIB itemCost FORMAT=DOLLAR8.2
        taxPrice FORMAT=DOLLAR8.2;
  itemCost=6.99;
  taxPrice=priceWithTax_func(itemCost);
  PUT itemCost "will be " taxPrice " with tax";
QUIT;
```

Here is what is displayed in the Output window:

```

                                     The FCMP Procedure

--- Program Execution Starting.
1   1 (68:6)   Executing Stmt           : ASSIGN itemCost =
1   (68:14)   itemCost = 6.99
1   2 (69:6)   Executing Stmt           : ASSIGN taxPrice =

--- Subroutine priceWithTax_func Execution Starting.
1   1 (0:8)    Executing Stmt           : FUNCTION
1   2 (6:11)   Executing Stmt           : IF
1   (0:22)    _temp1 = MISSING( taxRate=. )
1   3 (7:18)   Executing Stmt           : ASSIGN taxRate =
1   (0:34)    taxRate = 7.25 / 100 = 0.0725
1   4 (8:11)   Executing Stmt           : ASSIGN taxedPrice =
1   (0:36)    _temp1 = 1 + (taxRate=0.0725) = 1.0725
1   (0:31)    taxedPrice = (price=6.99) * (_temp1=1.0725) = 7.496775
1   5 (9:11)   Executing Stmt           : RETURN _priceWithTax_func_ =
1   (0:19)    _priceWithTax_func_ = (taxedPrice=7.496775) = 7.496775
1   6 (10:11) Executing Stmt           : ENDSUB

--- Subroutine priceWithTax_func Execution Finished.

1   (69:32)   taxPrice = priceWithTax_func( itemCost=6.99 ) = 7.496775
1   3 (70:6)   Executing Stmt           : PUT
$6.99 will be $7.50 with tax
1   (70:6)    PUT (4 items)

--- Program Execution Finished.
```

Figure 5: Trace Option Output

Note that even the values of sub-computations are shown (like temporary variable “_temp1”), making it easier to find problems should the results be incorrect.

ARRAY USAGE IN FUNCTIONS

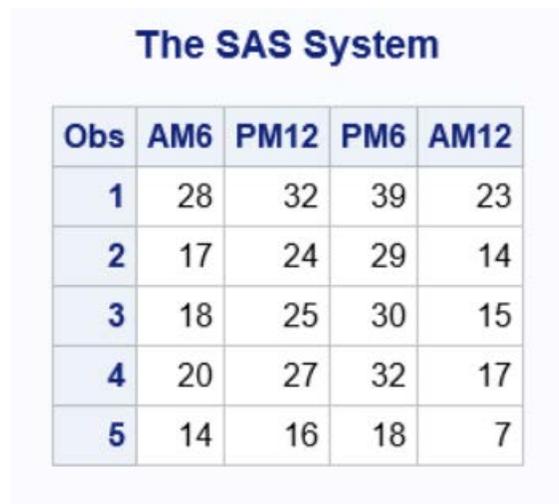
The FCMP procedure allows for creating and using numeric and character arrays. The arrays are similar to DATA step arrays with a few exceptions:

- The array dimensions are limited to 6.
- Lower bounds are not supported.
- References require explicit indexing via subscripts (that is, letter[4], not letter4) to any array variable passed into an FCMP function. This restriction does not apply to any array variable declared in the function.
- The subscript must be within square brackets or braces, not parentheses (that is, letter[4] or letter{4}; letter(4) is a call to the “letter” function with the value 4).

As was demonstrated in the “getVowelCount” function, arrays allow for enumerating values (especially useful in loops). Arrays take on even more importance with functions that need access to SAS data sets.

The FCMP procedure allows for reading data sets into an array and writing array values into a SAS data set. This is done with specialized functions and CALL routines (in this case, the READ_ARRAY and WRITE_ARRAY functions) built into FCMP and available only from within FCMP (they cannot be called from DATA step).

Consider the task of getting a data set each month that contains outdoor temperatures taken in 6-hour increments. Here are the first 5 observations in the data set:



Obs	AM6	PM12	PM6	AM12
1	28	32	39	23
2	17	24	29	14
3	18	25	30	15
4	20	27	32	17
5	14	16	18	7

Figure 6: Daily Temperature Data (First 5 Observations)

The task is to produce a data set that has the average temperature at each time interval over that month. The next example demonstrates a way to accomplish this with READ_ARRAY.

```

PROC FCMP OUTLIB=billsLib.paper.averages;

/* Determine the average temperature for the same time index
   over a month of specified daily temperature data. */
FUNCTION averageTempAtTime(dsName $, timeIndex);
  ARRAY monthlyTimeTemp[31,4] / NOSYMBOLS;
  /* Read the data from the named SAS data set. */
  rc = READ_ARRAY(dsName, monthlyTimeTemp);
  IF (rc EQ 0)
    THEN DO;
      averageTemp = getAverageTemp(monthlyTimeTemp, timeIndex);
    END;
  ELSE DO;
    /* Problem reading the data set. */
    ATTRIB missingValue FORMAT=8.;
    missingValue = .;
    averageTemp = missingValue;
  END;
RETURN(averageTemp);
ENDSUB;

/* Compute the average temperature for the specified time index. */
FUNCTION getAverageTemp(monthlyTimeTemp[*,*], timeIndex);
  ARRAY tempByTime[1,1] / NOSYMBOLS;
  rowDimension = 1;
  columnDimension = 2;
  daysInMonth = DIM(monthlyTimeTemp, rowDimension);
  readingsPerDay = DIM(monthlyTimeTemp, columnDimension);

  CALL DYNAMIC_ARRAY(tempByTime, readingsPerDay, daysInMonth);
  CALL TRANSPOSE(monthlyTimeTemp, tempByTime);
  timeSum = getArrayRowSum(tempByTime, timeIndex);
  averageTimeTemp = (timeSum / daysInMonth);
RETURN(averageTimeTemp);
ENDSUB;

/* Get the sum of the numbers in a specified row of an array. */
FUNCTION getArrayRowSum(arrayToSum[*,*], rowIndex);
  columnDimension = 2;
  columns = DIM(arrayToSum, columnDimension);
  sum = 0;
  DO columnIndex=1 to columns;
    sum = (sum + arrayToSum[rowIndex, columnIndex]);
  END;
RETURN(sum);
ENDSUB;

QUIT;

DATA work.monthAvgs;
  AM6Avg = averageTempAtTime("work.monthTemps", 1);
  PM12Avg = averageTempAtTime("work.monthTemps", 2);
  PM6Avg = averageTempAtTime("work.monthTemps", 3);
  AM12Avg = averageTempAtTime("work.monthTemps", 4);
RUN;

PROC PRINT DATA=work.monthAvgs;
  RUN;

```

Here are the results of running the above SAS program:

The SAS System				
Obs	AM6Avg	PM12Avg	PM6Avg	AM12Avg
1	25.5667	30.5667	34.9	23.6667

Figure 7: Average Temperatures

In the “getAverageTemp” function, two more special FCMP CALL routines are used: DYNAMIC_ARRAY and TRANSPOSE. The DYNAMIC_ARRAY CALL routine allows for resizing an array that is declared within a function. In this simplistic case, since the month data can be 28–31 rows depending on the month, the array to be transposed is sized to the current monthly data array. This special CALL routine can be very handy when the possible size of the array varies drastically. For example, an array could have 10 – 100,000 elements. Without the DYNAMIC_ARRAY function, the array would have to be declared for the maximum amount, 100,000, even though most of the time the data contains only a few hundred elements. This becomes even more problematic when the function that declares the array is called recursively.

The TRANSPOSE CALL routine basically turns the array on its side. It takes the rows of data for the days and transpose them into columns. Likewise, the columns of times are transposed into rows. The resulting array has the dimensions of [4,31]. With the 4 rows now containing the time interval data, summing up the temperatures for a given time interval becomes much easier.

Another thing to note in the “averageTempAtTime” and “getAverageTemp” functions is declaring the ARRAY with /NOSYMBOLS. This means that the array does not include symbols - variable names for each array element (that is array element variables “monthlyTimeTemp1”, “monthlyTimeTemp2”, ..., “monthlyTimeTemp124”). With the array declared with /NOSYMBOLS, the SAS program can access the array elements only via an index. This is the same as declaring a DATA step array as _TEMPORARY_.

In the PROC FCMP code for the Average Temperatures example, notice how decomposition was used to break the function into small, more easily understood functions while abstracting their functionality. Also, if

DYNAMIC_ARRAY Efficiencies

Does your SAS program initialize or set the array value immediately after calling DYNAMIC_ARRAY? The DYNAMIC_ARRAY special function automatically initializes the created array with “.”, a SAS missing numeric value. If the SAS program sets all the values in the array to a SAS missing numeric value immediately after calling DYNAMIC_ARRAY, the SAS program is doing an unnecessary (and possibly costly) action.

If the SAS program requires immediately filling the whole array with values other than a SAS numeric missing value, the DYNAMIC_ARRAY function can be instructed to not initialize the elements in the array. Again, this could be a significant time savings for a large array. But, it is up to the SAS program to initialize the array since running with uninitialized values can have unexpected results.

To create an array without initialization, set the last parameter in the DYNAMIC_ARRAY function to “.”, the period character. For example, to create a 5 by 3 element array that is not initialized:

```
CALL DYNAMIC_ARRAY (arrayName, 5, 3, .);
```

others can use these functions, the functions could be made available to other programs – thereby keeping from writing duplicate functions.

Anyone that has coded in SAS for a while is probably thinking, “I can do the same thing with SAS macros.” And that statement would be true. But the SAS macro language has limitations. The following are two obvious limitations:

1. Returning a macro value in the classic function style (x=SIN(y)) is not possible.
2. Macros treat all variables as character values making numeric computations tedious.

You do not have these limitations with functions created with PROC FCMP.

ADVANCED FEATURES

RUN_MACRO/RUN_SASFILE SPECIAL FUNCTIONS

With the basics covered, it is time to get into the more advanced features of using PROC FCMP. Have you been coding a DATA step and thought, “I wish I could nest DATA steps” or “I wish I could call a PROC from within a DATA step”? With an FCMP function, your wishes can come true by using two other FCMP special functions, RUN_MACRO and RUN_SASFILE. As their names imply, these functions can run a SAS macro or a SAS file respectively. Since the concepts of these 2 special functions are similar, the example shows only the RUN_MACRO function:

```
PROC FCMP OUTLIB=billsLib.paper.runMacro;
  FUNCTION writeJsonFile(dsName $, age);
    procRc = 0;
    rc = run_macro("jsonFileCreate", dsName, age, procRc);
    IF (rc NE 0)
      THEN DO;
        PUT "Problems with submitting JSON file macro (" rc ")";
      END;
    ELSE DO;
      IF (procRc NE 0)
        THEN DO;
          PUT "Problems with writing JSON file (" procRc ")";
          rc = procRc;
        END;
      END;
    RETURN(rc);
  ENDSUB;
QUIT;

%MACRO jsonFileCreate();
  %LOCAL/READONLY inDsName = %SYSFUNC(DEQUOTE(&dsName));
  %LOCAL/READONLY inAge = %SYSFUNC(DEQUOTE(&age));
  %LOCAL/READONLY outputFile = "./age_&inAge..json";

  PROC JSON OUT=&outputFile NOSASTAGS;
    EXPORT &inDsName (WHERE=(Age EQ &inAge));
  QUIT;

  %LET procRc = &syserr;
  %IF (&procRc EQ 0)
    %THEN %PUT Created file &outputFile;
```

```

    %MEND jsonFileCreate;

%LET sortedAgeDataSetName=WORK.sortedAges;

PROC SORT DATA=sashelp.class
    OUT=&sortedAgeDataSetName;
    BY age;
    RUN;

DATA _NULL_;
    SET &sortedAgeDataSetName;
    BY age;
    IF (first.age)
        THEN DO;
            rc = writeJsonFile("&sortedAgeDataSetName", age);
            IF (rc NE 0)
                THEN ABORT;
            END;
    RUN;

```

In the previous example, the project requires writing data that is based on the “age” variable from the sashelp.class data set to age-specific JSON files for consumption by another process. The DATA step iterates over the sorted data by “age”. When the “age” variable value changes, the FCMP function is called to write all of the observations that match the current “age” value to a specific JSON file. Note that all of this processing is done from the middle of the DATA step. The same idea can be applied to nesting DATA steps when the need arises.

An in-depth look at using RUN_MACRO is available in (Rhodes 2012).

RECURSION

Recursion can be a powerful tool in your development toolbox. A web search found a few techniques for implementing recursion using SAS macro. While accomplishing the task, they were cumbersome and unwieldy (as noted in the limitations listed above). The FCMP procedure supports recursion in a clean and straightforward way. One of the classic examples used in teaching recursion is computing a factorial of a nonnegative number. Here is the FCMP implementation of that classic example:

```

PROC FCMP OUTLIB=billsLib.paper.factorial;
    FUNCTION factorial(currentValue);
        IF (currentValue EQ 0)
            THEN RETURN(1);
        ELSE RETURN(currentValue * factorial(currentValue - 1));
    ENDSUB;
QUIT;

```

Here is the SAS program to exercise the function:

```

DATA work.facts;
    DO number=5 TO 9;
        factorial=factorial(number);
        OUTPUT;
    END;
RUN;

PROC PRINT DATA=work.facts;
RUN;

```

Obs	number	factorial
1	5	120
2	6	720
3	7	5040
4	8	40320
5	9	362880

Figure 8: Output of Factorial Function

For a more involved usage and explanation of recursion, reference (Secosky 2007) for a directory tree traversal example.

CALLING C FUNCTIONS

Another code reuse, and therefore efficiency scenario, is to leverage existing C functions. Say that your company has a vast array of business rules that are coded in C functions and you would like to access them from SAS. This can be accomplished by first using another Base SAS procedure, PROC PROTO. The PROTO procedure is used to define within SAS the interface for accessing the C function. Once the interface is defined, use PROC FCMP to create a function that uses the interface to call the C function from within SAS. Here is an example that registers an external C function to compute a factorial of a number:

```
PROC PROTO PACKAGE=billsLib.proto.factFunc;
  LINK "factorial";
  int factorial(int currentValue);
RUN;

PROC FCMP INLIB=billsLib.proto
  OUTLIB=billsLib.paper.factorial;
  FUNCTION factorialProto(currentValue);
    RETURN(factorial(currentValue));
  ENDSUB;
QUIT;
```

Here is the slightly modified version of the SAS program that is used to compute factorial values. Since the PROC PROTO PACKAGE statement uses a different data set to store the factorial function interface information, this path is added to the CMPLIB option value. The only modification to the DATA step is to call the newly defined FCMP function that, in turn, calls the external C function to compute the factorial values of some numbers:

```
OPTION CMPLIB=(billsLib.paper billsLib.proto);

DATA work.factsProto;
  DO number=5 TO 9;
    factorial=factorialProto(number);
    OUTPUT;
  END;
RUN;
```

The output of this SAS program is the same as the previous recursion example.

Be sure to read (Henrick 2015) for detailed information about the PROTO procedure.

HASH OBJECTS

SAS 9.3 saw the introduction of hashing support in PROC FCMP. The HASH object allows for creating a hash table that can store and retrieve data using a hashed index value. This can be an efficient retrieval scheme based on the data, size, and composition of the data. In this next example, the task is to obtain information from the sashelp.cars data set, specifically for a list of cars that are stored in the SAS data set work.possCars. As noted in (Henrick 2013), a large number of values in a WHERE statement can become unwieldy. Using an FCMP function called from the WHERE statement greatly cleans up the code and allows for easily updating a dynamic list of WHERE statement values. This technique makes the code easier to maintain.

```
PROC FCMP OUTLIB=billsLib.paper.cars;
  FUNCTION isPossibleCar(model $);
    DECLARE HASH hashObj(DATASET="work.possCars");
    rc = hashObj.definekey("model");
    rc = hashObj.definedone();
    rc = hashObj.check();
    checkRc = (not rc);

    RETURN(checkRc);
  ENDSUB;
QUIT;

DATA work.possCars;
  INPUT Model $40.;
  DATALINES;
    Element LX
    Malibu Maxx LS
    Rio Cinco
    Outback
    Rodeo S
    Matrix XR
    Xterra XE V6
    Murano SL
    Jetta GL
    Passat GLS 1.8T
    4Runner SR5 V6
    V40
    ; ; ;
RUN;

PROC PRINT DATA=work.possCars;
RUN;

DATA work.carDetails;
  SET sashelp.cars;
  WHERE isPossibleCar(strip(model));
RUN;

PROC PRINT DATA=work.carDetails(KEEP=make model type origin
                                drivetrain msrp invoice);
RUN;
```

The first 3 lines of the function create and load the hash table. The HASH object CHECK method determines if the car's model name, which is used as the hash key, is in the hash table. When the model

name matches a key, an “rc” value of zero is set. In order to indicate to the WHERE statement that the current model should be included in the work.carDetails data set, the value returned from the “isPossibleCar” function is set to the value 1 via the negation of “rc” in setting the “checkRc” return value. Note that a DATA step hash object cannot be used in FCMP functions, but using FCMP hashing has an advantage. In the example code, the DEFINEKEY and DEFINEDONE method calls are executed only once no matter how many times the function is called. The “strip” function accomplished the needed modification so that the matching key was found in the search of the hash table.

Here is the abbreviated output of the example. To make the output fit on the page, a number of data columns were excluded:

Obs	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice
1	Chevrolet	Malibu Maxx LS	Wagon	USA	Front	\$22,225	\$20,394
2	Honda	Element LX	SUV	Asia	All	\$18,690	\$17,334
3	Isuzu	Rodeo S	SUV	Asia	Front	\$20,449	\$19,261
4	Kia	Rio Cinco	Wagon	Asia	Front	\$11,905	\$11,410
5	Nissan	Xterra XE V6	SUV	Asia	Front	\$20,939	\$19,512
6	Nissan	Murano SL	Wagon	Asia	Rear	\$28,739	\$27,300
7	Subaru	Outback	Wagon	Asia	All	\$23,895	\$21,773
8	Toyota	4Runner SR5 V6	SUV	Asia	Front	\$27,710	\$24,801
9	Toyota	Matrix XR	Wagon	Asia	Front	\$16,695	\$15,156
10	Volkswagen	Jetta GL	Wagon	Europe	Front	\$19,005	\$17,427
11	Volkswagen	Passat GLS 1.8T	Wagon	Europe	Front	\$24,955	\$22,801
12	Volvo	V40	Wagon	Europe	Front	\$26,135	\$24,641

Figure 9: Details of Possible Cars

Note that DATA step hash object cannot be used in FCMP functions, but using FCMP hashing has an advantage. In the example code, the DEFINEKEY and DEFINEDONE method calls are only executed only once no matter how many times the function is called. Once the hash object is defined with the DEFINEDONE method, these method calls are no longer executed. This also applies to the DEFINEDATA method (which is not used in the example as the data was assumed to equal the hash key). If a DATA step hash object had been used, the DATA step would have had to have been coded to only call those methods on the first loop through the data.

WHAT'S NEW IN SAS 9.4 MAINTENANCE 5

DICTIONARY OBJECTS

Dictionary objects are similar to the hash object. As such, the data is accessed via a key-value pair. Like the hash object, a dictionary object supports access to numeric and character data, but also in-memory storage of arrays, hash objects, and other dictionary objects. Also, a dictionary is not static in definition like a hash object. If the need arises to later store a hash object at a key position that contains character data, the dictionary object can gladly make the replacement. The dictionary object is available from the FCMP procedure when running the SAS client. For more information about dictionaries, be sure to read (Henrick 2017) as listed in the references section.

ASTORES

Astores are an advanced and specialized addition to the FCMP procedure. Astores are an analytic store for scoring models. For information about Astores, refer to the FCMP Procedure in the Base SAS 9.4 Procedures Guide.

CONCLUSION

This paper demonstrates many of the basic FCMP statements and options. There are also demonstrations of some of the more advanced features plus a preview of what is new in the latest release of SAS 9.4 maintenance 5. The demonstrations have included techniques such as encapsulation, decomposition, and abstraction to help understanding and maintenance of SAS programs that use PROC FCMP. This paper only scratches the surface in the FCMP functionality (especially for analytics programmers, as a number of the special functions are geared to analytic usage). Start coding with PROC FCMP. The code you write with the demonstrated techniques will be appreciated by the maintenance programmer (or maybe even yourself) who might need to modify the code in years to come.

All the programming example used in this paper are available for download from https://support.sas.com/resources/sgf/507642_SAS2125-2018_Examples.zip

REFERENCES

Henrick, Andrew, Whitcher, Mike. and Croft, Karen. "Dictionaries: Referencing a New PROC FCMP Data Type" *Proceedings of the SAS Global 2017 Conference*, 2017. Available at <http://support.sas.com/resources/papers/proceedings17/SAS0418-2017.pdf>

Henrick, Andrew, Erdman, Donald. and Croft, Karen. "Helping You C What You Can Do with SAS" *Proceedings of the SAS Global 2015 Conference*, 2015. Available at <https://support.sas.com/resources/papers/proceedings15/SAS1747-2015.pdf>

Henrick, Andrew, Erdman, Donald. and Christian, Stacey. "Hashing in PROC FCMP to Enhance Your Productivity" *Proceedings of the SAS Global 2013 Conference*, 2013. Available at <http://support.sas.com/resources/papers/proceedings13/129-2013.pdf>

Rhoads, Mike. "Use the Full Power of SAS in Your Function-Style Macros" *Proceedings of the SAS Global 2012 Conference*, 2012. Available at <https://support.sas.com/resources/papers/proceedings12/004-2012.pdf>

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

ACKNOWLEDGMENTS

The authors would like to thank all the people that helped in putting this paper together: Stacey Christian, Shameka Coleman, Diane Olson, Chris Johns, Elizabeth Downes, and Sandy McNeill.

RECOMMENDED READING

- *Base SAS® Procedures Guide* available at <http://documentation.sas.com/?docsetId=proc&docsetTarget=titlepage.htm&docsetVersion=9.4>

CONTACT INFORMATION

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

Bill McNeill
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Bill.McNeill@sas.com
<http://www.sas.com>

Andrew Henrick
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Andrew.Henrick@sas.com
<http://www.sas.com>

Mike Whitcher
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Mike.Whitcher@sas.com
<http://www.sas.com>

Aaron Mays
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Aaron.Mays@sas.com
<http://www.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.