PharmaSUG2011 - Paper TU07

A Cup of Coffee and Proc FCMP: I Cannot Function Without Them

Peter Eberhardt, Fernwood Consulting Group Inc, Toronto ON

ABSTRACT

How many times have you uttered "If only I could call this DATA Step as a function"?

If this statement describes you, then PROC FCMP are for you. This tutorial will get you started with everything you need to write, test, and distribute your own "data step" functions with PROC FCMP.

INTRODUCTION

SAS® programmers inherently understand the concept of breaking a complex problem into manageable pieces; a quick look at any SAS programme with the rich variety of DATA and PROC steps will demonstrate this. However, within the main programming environment, the DATA step, the ability to break down problem complexity into manageable pieces has been limited to two basic methods: Link/Return blocks and Macros.

Link/Return blocks allow the SAS programmer to isolate (encapsulate) business logic using familiar DATA step code; although useful, there are some drawbacks to this method. First, the SAS code is embedded into the DATA step. This means the Link/Return block needs to be copied to each DATA step where it is to be used. As with all cut-and-paste operations this is both error prone and maintenance heavy. Second, since the Link/Return block is part of the DATA step, it uses DATA step variables not parameters. This can lead to the need to a series of variable assignments before and after the Link statement so the DATA step variables that need processed are appropriately renamed to and from the Link/Return block variable names. Once again, this can lead to errors.

SAS Macros help to eliminate some of the re-use and parameter issues at the expense of introducing non-DATA step syntax. In addition, the overuse of macros can lead to code that is more difficult to read and maintain.

To overcome the limitations of these two alternatives, SAS 9.2 has made PROC FCMP available to the DATA step programmer; prior to SAS 9.2 PROC FCMP functions could be used in several SAS/STAT®, SAS/ETS® and SAS/OR® procedures. PROC FCMP allows the SAS programmer to create and reuse parameterized functions using familiar DATA step code. This paper will step you through the creation, testing, and deployment of your own functions. Although PROC FCMP uses DATA step like syntax, there are a few differences between its syntax and the DATA step syntax; these differences will be highlighted.

Before we look at how to create a function with PROC FCMP, let’s examine the definition of a function.

WHAT IS A FUNCTION

Every year the local ballet company holds a black-tie function to raise money; in this case, a function can be defined as a social gathering. This is not the type of function to be created by PROC FCMP. The function of a soda bottle is to hold the soda; in this case a function can be defined as the purpose for which the object exists. This is not the type of function to be created by PROC FCMP. What then is the thing we SAS programmers call a function?

A general definition of a function is:

A function is a rule for transforming zero or more values called arguments; the result of the transformation is called the value or result of the function. The transformation can also have side effect; that is permanent changes in the values of arguments.

The SAS 9.2 online help has a similar definition:

A SAS function performs a computation or system manipulation on arguments, and returns a value that can be used in an assignment statement or elsewhere in expressions.
What is not explicit in either the general description above or in the definition supplied in the SAS help is the fact that functions have names. Not only does the function have a name but also that name should be meaningful. So although the statement:

```plaintext
rc = foo(2500, 3);
```

may be a valid way to call a function, a function called `foo` does not convey very much meaning to most people reading the code. On the other hand the statement:

```plaintext
interestPaid = monthlyInterest(2500, 3);
```

would make us guess that function is probably calculating a monthly interest on $2,500 at 3% interest. This would be reinforced if the function were called with variables instead of constants:

```plaintext
interestPaid = monthlyInterest(balance, iRate);
```

The SAS 9.2 online help has a definition for a second type of function, called a CALL Routine:

A **CALL routine** alters variable values or performs other system functions. CALL routines are similar to functions, but differ from functions in that you cannot use them in assignment statements or expressions.

All SAS CALL routines are invoked with CALL statements. That is, the name of the routine must appear after the keyword CALL on the CALL statement.

In our general definition of a function we introduced side effects; that is, changes in the values of the arguments. In the case of a SAS CALL Routine these side effects are the method for returning values. If our function `monthlyInterest` were written as a CALL Routine, it would look something like:

```plaintext
call monthlyInterest(balance, iRate, interestPaid);
```

In this case the variable `interestPaid` is passed in as an argument and its value is changed in the CALL Routine.

In this paper I will be using the term function to include both functions and CALL Routines as defined in the SAS help; in cases where there could be ambiguity I will explicitly refer to SAS functions or SAS CALL Routines.

**WHAT IS A RULE**

In our general definition of a function we called it “a rule for transforming zero or more values”. The rule can be a simple formula such as:

- Square(x) == x * x
- Product(x, y) == x * y

The rule may be more complex such as:

1. Discount(quantity) == if quantity > 100 then discount = 0.15 else discount = 0.0

In addition, some of the rules we will implement will end up being more complex than originally conceived. Consider a simple rule of division:

2. Divide(dividend, divisor) == dividend / divisor

This is fine, as long as divisor is a number and not equal to zero. Our simple rule can be expanded to something more complex, but also more robust:
3. Divide(dividend, divisor) == if divisor is missing then missing else if divisor is 0 then 0 else dividend / divisor

Such a divide function in a data step where we expect a number of missing or 0 values in the divisor field would simplify our code and clean up the Division by Zero and Missing Value NOTES in our log.

With this introduction to functions, let’s turn to creating our own functions.

CREATING FUNCTIONS

Starting in SAS 9.2 DATA step programmers can start to take advantage of PROC FCMP to create and use DATA step language functions. PROC FCMP was available before SAS 9.2, but the functions could only be used in selected SAS/STAT and SAS/ETS procs. We will start by looking at how we could create 2 simple functions, one to return an integer constant and the other to return a character constant. In order to create these functions we will have to learn some of the PROC FCMP syntax.

PROC FCMP SYNTAX

The following is the syntax for PROC FCMP:

PROC FCMP option(s);
ARRAY array-name[dimensions] </NOSYMBOLS | variable(s) | constant(s) | (initial-value(s))>
ATTRIB variable(s) <FORMAT=format-name LABEL='label' LENGTH=length>
FORMAT variable(s) <format> <DEFAULT=default-format>
FUNCTION function-name(argument-1, argument-2, ..., argument-n) <$> <length>
LABEL variable='label'
LENGTH variable(s)<$> length <DEFAULT=n>
STRUCT structure-name variable;
SUBROUTINE subroutine-name (argument-1, argument-2, ..., argument-n);
OUTARGS out-argument-1, out- argument-2, ..., out-argument-n;

As you can see, there are some familiar elements in this syntax. We will be concentrating on two of these elements, function and subroutine, as well as some of the proc options.

```sas
proc fcmp outlib=work.funcs.Test; /* where will the functions be saved */
function whatAmI(); /* declare a function returning a number */
  return(42); /* return the number */
endsub;

function whereAmI($) ; /* declare a function returning a string */
  return('In Test'); /* return the string */
endsub;
quit;
```

Let’s examine the code above.

1. **outlib=work.funcs.Test**: this option tells SAS where to store the function. If no outlib= is entered then none of the functions are saved.

2. **function whatAmI()**

    function is the keyword to start the definition. Functions return a value, either numeric or character; since there is not character option in the definition, this function will return a number. NOTE: the function name, like all SAS names, is not case sensitive.
whatAmI is the name of the function.

() indicates there are no arguments to the function.

3. return(42);
   the return statement is used to return the value of the function to the DATA step. In this case we are
   returning a constant (42); however, it could be, and usually will be, a variable calculated in the function.

4. endsub
   this ends the definition of the function.

5. function whereAmI() $
   the $ modifier tells SAS that this function will return a character string. Since there is no length specified
   the function can return a string of any valid SAS string length. NOTE: the function name, like all SAS names, is
   not case sensitive.

6. return('In Test')
   the return statement returns the constant string In Test; however, it could be, and usually will be, a variable
   calculated in the function.

7. endsub
   this ends the definition of the function.

After we submit this code, we check the log. The log for this code is:

```sas
1   proc fcmp
2      outlib=work.funcs.test; /* where will the functions be saved */
3   function whatAmI(); /* declare a function returning a number */
4      return(42);       /* return the number */
5   endsub;
6
7   function whereAmI() $; /* declare a function returning a string */
8      return('In Test');   /* return the string */
9   endsub;
10
11 quit;
```

NOTE: Function whereAmI saved to work.funcs.test.
NOTE: Function whatAmI saved to work.funcs.test.
NOTE: PROCEDURE FCMP used (Total process time):
   real time           1.48 seconds
   cpu time            0.14 seconds

We see the function appears to have compiled correctly, so let's try using them in a simple DATA _NULL_ step:

```sas
data _null_;
   rci = whatAmI();
   put rci=; /* should be 42 */
   rcc = whereAmI();
   put rcc=; /* should be In Test */
run;
```

If we look at the log, there appears to be a problem, our new functions could not be found:

```sas
12 13 data _null_;
```
In order for SAS to recognize our functions we have to use the SAS option `CMPLIB=`; this option can be set in the configuration file, in the autoexec.sas file, or in your program; in these examples the `CMPLIB=` option will be set in the program.

When we invoked PROC FCMP we used the `outlib=` option to tell SAS where to store the function; in our case we used `work.funcs.test`. We will look at the library structure later, for now we will take on faith that the `outlib=` option is a three level reference (1 – work, 2 – funcs, 3 – test) but the `CMPLIB=` options takes only a two lever reference `work.funcs` (1 – work, 2 – funcs). The following is the SAS log when we add the options statement and re-execute our code:

```
19   options cmplib=work.funcs;
20   data _null_;
21      rci = whatAmI();
22      put rci=; /* should be 42 */
23      rcc = whereAmI();
24      put rcc=; /* should be In Test */
25   run;

rci=42
rcc=In Test
NOTE: DATA statement used (Total process time):
       real time           0.14 seconds
       cpu time            0.07 seconds
```

Now we see the results we expected.

To recap our progress, here are the highlights:

1. we used the PROC FCMP `outlib=` option to tell SAS where to store the compiled functions.
2. we create two functions, one returning a number and one returning a string, using the `function` statement.
3. we used the `cmplib=` system option to tell SAS where to find our functions.
4. we invoked the two new functions in a DATA step and confirmed the results.

The next example will demonstrate the use of function arguments and building in some decision rules to our functions. In this example the function will take one numeric argument and return a value based on the argument. If you had a series of variables (eg survey questions) that needed to be recoded the same way, you could create a similar function. The following is the SAS log showing the call to PROC FCMP:
Let's examine the code above.

1. **Function whatAmI(startValue)**
   
   we are creating a function that takes one numeric argument (`startValue`) and will return a numeric value.

2. **WARNING: Function whatAmI**
   
   a SAS warning telling us we are redefining the function `whatAmI`.

3. **If ... then ... else... rc=**
   
   a series of if/then/else statements where we set a value of the variable `rc`. For more complex logic we could have `do... end` blocks.

4. **return(rc)**
   
   return the variable `rc` to the DATA step.

A check of the log of a DATA _NULL_ step shows the new function working.

```
40 options cmplib=work.funcs;
41 data _null_
42   iAm = .;
43   rci = whatAmI(iAm);
44   put iAm= rci=; /* should be .S */
45   iAm = -1;
46   rci = whatAmI(iAm);
47   put iAm= rci=; /* should be .Z */
48   iAm = 10;
49   rci = whatAmI(iAm);
50   put iAm= rci=; /* should be 0 */
```
The second type of function that can be defined is a subroutine, or in SAS terms, a CALL Routine. We can redefine our function as a subroutine as follows:

```
proc fcmp outlib=work.funcs.test; /* where will the functions be saved */
subroutine whatAmI(startValue, rc); /* declare a subroutine */
   if missing(startValue) then rc=.S;
   else if startValue <  0  then rc=.Z;
   else if startValue < 20  then rc=0;
   else if startValue < 50  then rc=20;
   else if startValue < 100 then rc=50;
   else                          rc=100;
   return;
endsub;
quit;
```

The important change here is

1. **subroutine whatAmI(startValue, rc)**
   
   we are using **subroutine** instead of **function**. In addition, we have two arguments instead of one. We will use the **rc** argument to return the value to the DATA step.

2. **return**
   
   the return statement takes no arguments. In this example we could have omitted the **return** statement since it is implicit with the **endsub** statement.
When we examine a portion of the SAS log for this code we see some warning messages:

```
62 proc fcmp
63   outlib=work.funcs.test; /* where will the functions be saved */
64 subroutine whatAmI(startValue, rc); /* declare a subroutine */
65 WARNING: Function whatAmI is already defined in packet test. Function whatAmI as defined in the current program will be used as default when packet is not specified.
66     if missing(startValue) then rc=.S;
67 WARNING: The variable rc should not be the result of the '=' operation because it is a read-only argument to subroutine whatAmI. Any changes to this argument will not be returned from the subroutine whatAmI. Use the OUTARGS statement to allow return values from a subroutine.
```

1. **WARNING: Function whatAmI**
   
   the WARNING telling us we are redefining the definition.

2. **WARNING: The variable rc should not...**
   
   We wanted to use rc to return a value to the DATA step, however SAS is telling us that any changes to the variable will not be returned to the DATA step. It also reminds us we missed the `OUTARGS` statement.

The subroutine did compile and was saved. We could try accessing it in a DATA step; this SAS log shows what the result will be:

```
79 data _null_
80 length rci 8.;
81 rci = -1;
82 iAm = .;
83 call whatAmI(iAm, rci);
84 put iAm= rci=; /* should be .S */
85 iAm = -1;
86 call whatAmI(iAm, rci);
87 put iAm= rci=; /* should be .Z */
88 iAm = 10;
89 call whatAmI(iAm, rci);
90 put iAm= rci=; /* should be 0 */
91 iAm = 30;
92 call whatAmI(iAm, rci);
93 put iAm= rci=; /* should be 20 */
94 iAm = 80;
95 call whatAmI(iAm, rci);
96 put iAm= rci=; /* should be 50 */
97 iAm = 20000;
98 call whatAmI(iAm, rci);
99 put iAm= rci=; /* should be 100 */
100 run;
```
From the log we see:

1. parameters \textit{iAm} and \textit{rci} are initialized
2. the subroutine is called with the two arguments
   Note there are no \textsc{Warning}s or \textsc{Error}s, so the subroutine was called.
3. \texttt{iAm=. rci=.1}
   Note that the value of rci has not been changed in any of the subroutine calls

The fix for this is simple; add the \texttt{outargs} statement as the following SAS log shows

```sas
103 proc fcmp
104   outlib=work.funcs.test; /* where will the functions be saved */
105 subroutine whatAmI(startValue, rc); /* declare a subroutine */
106 WARNING: Function whatAmI is already defined in packet test. Function whatAmI as defined in the current program will be used as default when packet is not specified.
107       outargs rc; /* argument rc will return a value */
108 if missing(startValue) then rc=.S;
109 else if startValue < 0 then rc=.Z;
110 else if startValue < 20 then rc=0;
111 else if startValue < 50 then rc=20;
112 else if startValue < 100 then rc=50;
113 else rc=100;
114 endsub;
115 quit;
116
NOTE: Function whatAmI saved to work.funcs.test.
NOTE: PROCEDURE FCMP used (Total process time):
   real time 0.14 seconds
   cpu time 0.06 seconds
```

```sas
118 options cmplib=work.funcs;
119 data _null_;```
Let's examine the code above.

1. *outargs rc*;
   - this tells SAS the value of *rc* can be altered and the altered value returned

2. *iAm=. rci=S*...
   - our CALL Routine now returns the correct values.

After looking at these examples of a *function* and a *subroutine*, you might ask yourself “Why would I use a subroutine when a function calling a function is clearer?” One main reason is that a subroutine can return multiple values whereas a function can only return one value. The following scenario demonstrates the use of a subroutine to return multiple values.

One data set has the following columns related to payments:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Unique entity identifier</td>
</tr>
<tr>
<td>GROUP</td>
<td>Group in which ID is a member</td>
</tr>
<tr>
<td>PAYRULE</td>
<td>Payment rule</td>
</tr>
</tbody>
</table>
There are several hundred thousand of these records covering three years.

A second dataset has the following columns relating to payments for specific services.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Unique entity identifier.</td>
</tr>
<tr>
<td>GROUP</td>
<td>Group in which ID is a member</td>
</tr>
<tr>
<td>SERVICECODE</td>
<td>A code identifying the service</td>
</tr>
<tr>
<td>SERVICEDATE</td>
<td>Service Date. These are the actual date of service</td>
</tr>
<tr>
<td>SERVICEAMOUNT</td>
<td>Amount of the Payment</td>
</tr>
</tbody>
</table>

There are about 140 million of these records per year covering the same three year period. The goal is to combine the payments by ID, GROUP and MONTH. In order to get a common 'month' value it was decided that the PAYDATE column would be the appropriate value to use. In order to do that, the SERVICEDATE had to be converted to the equivalent PAYDATE. In addition, since this process would be repeated on a monthly basis, a repeatable and automated process was required. Essentially a simple lookup that would convert any date to the appropriate PAYDATE was needed. To minimize the number of times the larger dataset had to be accessed a format lookup to convert the SERVICEDATE to the PAYDATE was chosen. The problem then is how to automate this process.

The first step was to create a list of distinct PAYDATES; remember, each month has only one PAYDATE. With only one date per month, we would need just two other values to create the format – the first day of the month and the last day of the month. The following subroutine shows how to do this:

```plaintext
proc fcmp outlib=work.FunCs.Dates;
subroutine datesInMonth(inDate, startDate, endDate, days);
outargs startDate, endDate, days;
   nextMonth = intnx('month', inDate, 1);
   startDate = intnx('month', nextMonth, -1);
   endDate = nextMonth - 1;
   days = nextMonth - startDate;
   return;
endsub;
```

Let's review this code:

1. `outlib=work.myFuncs.Dates`  
   The compiled functions will be going into a different 3rd level location.
2. `subroutine datesInMonth(inDate, startDate, endDate, days)`  
   An attempt to give both a meaning routine name, and meaningful argument names.
3. `outargs startDate, endDate, days`  
   Three of the arguments will return values. In this way we need only one call to get three new values.  
   `startDate` will be the first day of the month for `inDate`, `endDate` will be the last day of the month for `inDate`,

```plaintext
1 proc fcmp outlib=work.FunCs.Dates;
2 subroutine datesInMonth(inDate, startDate, endDate, days);
3   outargs startDate, endDate, days;
4       nextMonth = intnx('month', inDate, 1);
       startDate = intnx('month', nextMonth, -1);
       endDate = nextMonth - 1;
       days = nextMonth - startDate;
       return;
endsub;
```
and days will be the number of days in the month. For the problem as described above this is not needed, but for the real problem it was also required.

4. nextMonth = intnx(....

using the built-in SAS function intnx() and some simple date arithmetic we can calculate the values we want to return.

With this subroutine in place, the following code shows how to create and use the format:

```sas
data cntlinPaydates;
    retain fmtName 'payDates';
    set payDates;
    call missing(start, end, days);
    call datesInMonth(payDate, start, end, days);
    put start= end= days= paydate=;
    label = paydate;
    format payDate start end yymmdd10.;
run;

proc format cntlin=cntlinPaydates;
run;

data test;
    set allVisits;
    paydate = input(put(servicedate, paydates.), 7.);
    if _N_ = 1000 then stop;
    format paydate yymmdd10.;
run;
```

The main parts of this code are:

1. create the cntlin dataset using the first day in the month as the start value, the last day of the month as the end value and the PAYDATE as the label.
2. create the format using the cntlin dataset just created.
3. Apply the format to servicedate to lookup the appropriate paydate. Our two datasets now have a common date value we can use in a join.

In this section we looked at how you create functions and subroutines. The example functions were very simple since the purpose here is to show the basic mechanics of creating a function. Needless to say, complex rules can be encapsulated in functions; to see examples of a complex functions look at Secosky (2007).

Now that we know how to create functions, let’s look at how to test them before compiling them into a permanent library.

TESTING/DEBUGGING FUNCTIONS

In the previous section we looked at creating and storing functions, then using them in a DATA _NULL_ step. For simple functions this works well, however for more complex functions we need to look at alternatives. We do not have access to the DATA step debugger so we will have to make judicious use of the PUT statement within the function to help test and debug new functions. The results of the PUT statement can go to the PRINT (default) and LOG destinations.

Let’s revisit our daysInMonth routine and add some PUT statements to see the results as we execute:

```sas
  proc fcmp outlib=work.Funcs.Dates;;
```
subroutine datesInMonth(inDate, startDate, endDate, days);
WARNING: Function datesInMonth is already defined in packet Dates. Function
datesInMonth as defined in the current program will be used as default when
packet is not specified.
outargs startDate, endDate, days;
FILE log;
put inDate yymmdd10. ;
nextMonth = intnx('month', inDate, 1);
put nextMonth yymmdd10. ;
startDate = intnx('month', nextMonth, -1);
put startDate yymmdd10. ;
endDate = nextMonth - 1;
put endDate yymmdd10. ;
days = nextMonth - startDate;
put days;
return;
endsub;
QUIT;

NOTE: Function datesInMonth saved to work.Funcs.Dates.
NOTE: PROCEDURE FCMP used (Total process time):
  real time           0.10 seconds
  cpu time            0.04 seconds

data _null_;
format firstDay lastDay yymmdd10.
  numdays 2. ;
call missing(firstDay, lastDay, numdays);
PUT 'calling routine:';
call datesInMonth('15jAN2009'd, firstDay, lastDay, numdays);
PUT 'end of routine:';
put _all_;
call datesInMonth('15FEB2009'd, firstDay, lastDay, numdays);
PUT 'end of routine:';
put _all_;
run;
calling routine:
2009-01-15
2009-02-01
2009-01-01
2009-01-31
31
end of routine:
firstDay=2009-01-01 lastDay=2009-01-31 numdays=31 _ERROR_=0 _N_=1
calling routine:
2009-02-15
2009-03-01
The main points here are

1. **FILE log**
   direct the results of the PUT statements to the log.

2. **PUT inDate yymmdd10.** (and others)
   Put the contents of inDate to the log using the specified date format. Similar statements are used for the other variables.

3. **call datesInMonth...**
   call the routine with a constant value in the **inDate** parameter. Note that messages are also written before and after the call.

4. The message from the DATA step before the call to the routine.

5. The values of the variable within the routine.

Earlier, when we first saw **outlib=** it was mentioned that if not library is given then the function is not saved. At first glance this seems odd. That is, why would you create a function if you were not going to save it? One answer is “So you can test it”. You can call functions within PROC FCMP to test them. For example:

```plaintext
   184  proc fcmp ;
   185  subroutine datesInMonth(inDate, startDate, endDate, days);
   WARNING: Function datesInMonth is already defined in packet Dates. Function datesInMonth as defined in the current program will be used as default when packet is not specified.
   186       outargs startDate, endDate, days;
   187       FILE log;
   188       put inDate yymmdd10. ;
   189       nextMonth = intnx('month', inDate, 1);
   190       put nextMonth yymmdd10. ;
   191       startDate = intnx('month', nextMonth, -1);
   192       put startDate yymmdd10. ;
   193       endDate = nextMonth - 1;
   194       put endDate yymmdd10. ;
   195       days = nextMonth - startDate;
   196       put days;
   197       return;
   198 endsub;
   199       format testDate1 testDate2 yymmdd10.;
   200       format days 3. ;
   201       call datesInMonth('15jan2009'd, testDate1, testDate2, days);
   202       put 'date1: ' testDate1 ' date2: ' testDate2 ' days: ' days;
```
We can see from this SAS log:

1. PROC FCMP was called with no outlib= statement
2. Variables for the call are given formats
3. The routine datesInMonth is called
4. The variables that were returned will be displayed in the log

Although it is possible to test new functions within PROC FCMP, using a DATA step does provide more flexibility for testing and debugging.

STORING AND SHARING FUNCTIONS

Looking back at the outlib= option for PROC FCMP we see there are three levels in the name; in our datesInMonth example the library was work.funcs.Dates. What does this mean? All of the functions defined in this call to PROC FCMP will be saved in a package called Dates; the package is stored in the data set work.funcs. A package is simply the collection of functions and subroutines, each with a unique name. Although it is possible to have the same function name in two different packages on the same data set there is no way within a DATA step to specify which package to use.

Although it is not possible to be able to access the two versions of a function if they are in different packages in the same data set, it is possible to access different versions of the same function if they are in different data sets. For example, the Risk department may have a larger set of parameters for a particular function (or functions) than the rest of the company. The subset of risk related functions could be in the package allfuncs.risk.dates while the common set (including the general set of functions intended for Risk) could be in the package allfunc.company.dates. Then, to make sure each gets the appropriate versions the cmplib= options could be set as follows:

1. Risk: cmplib=( allfunc.company.dates. allfuncs.risk.dates)
2. Everyone else: cmplib= allfunc.company.dates.

The cmplib= option can set a “search path” of libraries. Note that the search path is left to right (allfuncs.risk.dates is search first, then allfunc.company.date); this behaviour is not the same as other SAS “search paths” such as fmtsearch=, so be careful when using it.

In order to share functions they simply need to be compiled into packages that are saved in a public folder. The SAS autoexec and/or configuration files can then appropriate cmplib= and libname statements.

ADVANCED

We have looked at the basics of creating and using user defined functions. There is much more available that we will not look at here, however if you check the SAS online documentation for PROC FCMP and PROC PROTO you will see the richness that is available.
CONCLUSION

SAS DATA step code has often been maligned as being ‘spaghetti code’ and ‘wall paper’ code because there was no simple way to encapsulate complex logic into the DATA step. The introduction of user defined functions for DATA step programmers in SAS 9.2 has changed that. This paper has provided a glimpse into the workings of user defined functions.

REFERENCES


CONTACT INFORMATION

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

Name: Peter Eberhardt
Enterprise: Fernwood Consulting Group Inc.
Address: 288 Laird Dr
City, State ZIP: Toronto, ON, Canada M4G 3X5
Work Phone: (416)429-705
Fax:
E-mail: peter@fernwood.ca
Web: www.fernwood.ca

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.