PharmaSUG2010 - Paper AD02

The new SAS 9.2 FCMP Procedure, what functions are in your future?

John H. Adams, Boehringer Ingelheim Pharmaceutical , Inc., Ridgefield, CT

ABSTRACT

Our company recently decided to upgrade from SAS 8.2 to SAS 9.2. In doing so, we had to roll up the cumulative changes over 5 intermediate SAS versions (9.0 → 9.1→ 9.1.2 → 9.1.3 → ◊ 9.2), no easy task. We then tested most of the changes and new features and presented the results of our testing and observations to our global SAS user community.

We discovered that there are so many new features in SAS 9.2 to get exited about. So many new options to control the environment, over a hundred new functions, dozens of new formats, exiting new features in Procs and ODS and so much more.

This paper explores one area of the enhancements – functions. To be sure, SAS 9.2 has a very rich repertoire of functions. There are handy functions for almost everything, e.g. character , math, statistics and even perl functions. But SAS did us one better! They have now given us the capability to define our own functions with Proc FCMP. In this paper we explore the possibility of defining (and using) user defined functions and sub-routines on a project level.

INTRODUCTION

This paper explores one area of significant enhancements an SAS 9.2 – functions. To be sure, SAS 9.2 already has a very rich repertoire of functions. There are hundreds of handy functions for almost everything, e.g. character , math, statistics and even perl functions. But SAS did us one better! They have now given us the capability to define our own functions with Proc FCMP. In this paper we explore the possibility of defining (and using) your own functions, call routines and sub-routines on a project level.

There are many times when we have code for deriving some aspects of the data in each program. For example, we might have to impute a date when we only have a partial date. We do have a project standard for imputations. So, the necessary code for imputing a date would have to repeated within every project program in a that needed that capability.

Of course, we could make an imputation macro out of it and call it, when need. That solution has some drawbacks, however. The macro source code has to be included (at least once) to compile it and generally can only be called within a data step. Variable scopes can also be a problem with macros. Why not make a function out of this code and then I could be used with any SAS procedure or in any project program? Another advantage is that user defined functions can also used with %sysfunc in a %let statements in macros and programs.

The new proc FCMP gives us this capability. We can easily define (and store) our own project level functions, call routines and subroutines. The simple example for date imputation is just one possibility. This paper will explore other possibilities for using this new capability, more suggested functions, and the process of creating them.

THE FCMP PROCEDURE – AN OVERVIEW

This new SAS 9.2 FCMP procedure allows you to create new Functions or Call routines. Call Routines are also referred to as subroutines, at times. Proc FCMP compiles the user defined functions and call routines and stores them in a sharable library. From there, they can be used in DATA steps, Proc SQL (no array args), Proc REPORT (for compute blocks) and some other procedures that allow DATA-step programming, i.e. GENMOD, MODEL, NLIN, NLMIXED, NLP, PHREG, etc.

This paper is should give you a good head start in using and understanding Proc FCMP. You can get further technical details about this procedure in the Base SAS 9.2 Procedure guide.

So what are the differences between Functions and Call routines? The next few paragraphs should answer this question and leave you with a better understanding about the new procedure.

FUNCTIONS

Functions always return an explicit result. In other words, you can take the result of a function and assign it to any variable. Functions can return either a numeric or a character type of result. Input variables can be numeric or character type . We'll see later how to define the types for input variables and result. Variables inside the function are local and are not accessible to the program that uses the function.

Let's take a look at a very simple example, a function to return the difference between two variables in a data step, i.e. A=delta(var1,var2). The code for this function (var1-var2) would simply subtract the values and return the result. So if b=5; and c=2; , the statement A=delta(b,c) would result in A = 3. Notice that var1 and var2 arguments of the function definition are merely placeholders for input variable values being passed into the function.

You can define a new function using code that is essentially the same as code used in data steps and you use most features of the SAS programming language in functions and call routines. See the Base SAS 9.2 Procedure guide [1] for differences between FCMP and data step code.

CALL ROUTINES (SUBROUTINES)

Subroutines are computational blocks of code that can be called from any statement in a program. After completing the call, control returns to the next statement of the program that called it. Subroutines can update the values of one or more of it's arguments. Unlike functions call routine do not return a explicit result. That means that the variables you wish to modify with the call must be part of the arguments.

As with functions, you can define simple subroutines using code that is essentially the same as code used in data steps and you use most features of the SAS programming language in functions and call routines. See the Base SAS 9.2 Procedure guide [1] for differences between FCMP and regular data step code. Subroutines, however, can harness much more power than functions, i.e. they are able to use more data and have some special powerful call routines available.

Large amount of data can be passed to a subroutine via arrays (up to 6 dimensions). Subroutines have a number of special 'array call routines' available that can be used to operate on those input arrays, i.e. matrix transpose, zero matrix, element-wise addition or subtraction of 2 matrices, element-wise multiplication of 2 matrices, raising a matrix to a power, inverse of a matrix, etc.

Subroutines also support the use of C language structures and calls to some C helper functions

In general, variables from outside of the subroutine are not accessible inside of it (except the arguments). Variables inside the subroutine are local and are not accessible to the program that calls it.

Subroutines can be recursive. A recursive subroutine is a subroutine can call itself over and over again. Using recursion is a very powerful technique to reduce a potentially large program applications to a much smaller solution. No matter how many times a subroutine calls itself, when all the processing is done,

control returns to the next statement in the program that made the initial call.

Consider the following example. You have to collect all of the dataset names in a directory structure (including all of the sub-directories). Each time you read a directory folder , you may find SAS datasets and sub-directories. If you look into each sub-directory, you could find more datasets and sub-directories and so on and so on. The number of directories and their paths are undefined when the program starts, so traversing the directory hierarchy is almost like a random walk. The process, however, is quite simple and quite repetitive, i.e. 1) go into the directory and gather the dataset and directory names , 2) if there are directories, repeat step 1 for each one. Please see my paper in the SUGI 28 conference proceedings for this example by using recursive macros. A recursive subroutine, however, would now be a good candidate for doing this type of recursive process. The Base SAS 9.2 Procedure guide [1] also has a good discussion on using recursive routines for transversing a directory..

Let's take a look at a very simple example, a call routine that would replace a missing date value with a default date, call date_replace(in_date,default_date). If the in_date argument had a value, that value would remain the same. So if AEendDT=. ; and DeathDT='11Dec2008'd; , then the statement call date_replace(AEendDT, DeathDT) would result in AEendDT = 11Dec2008 (as numeric SAS data) .

THE FCMP PROCEDURE – EXAMPLES FOR MAKING YOUR OWN FUNCTION

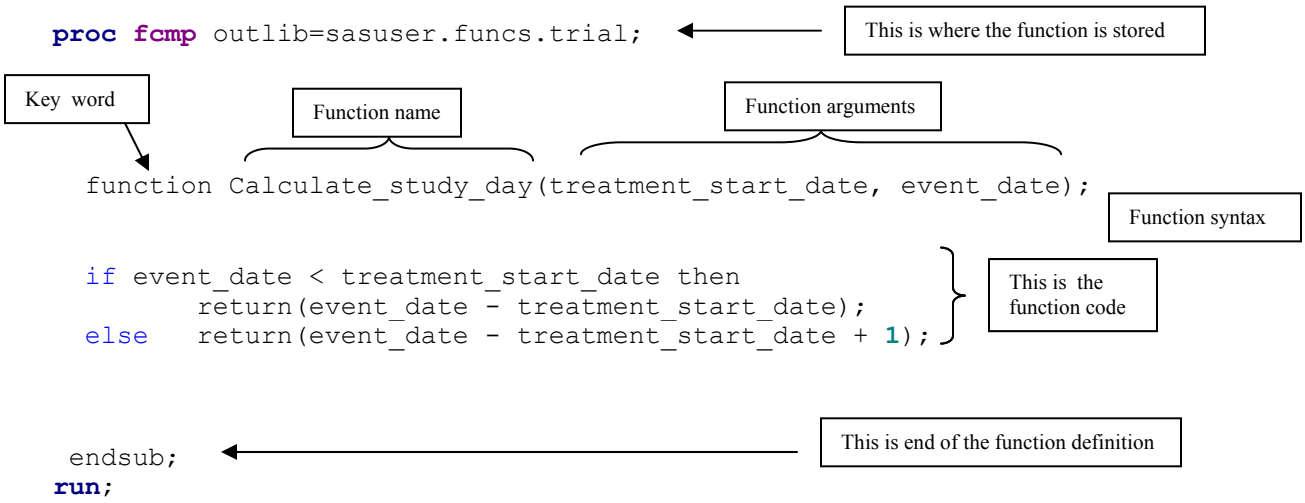
The following examples use long function and arguments in order to make the example more readable. Usually, these names should be shorter in order to reduce the amount of typing.

CALCULATE_STUDY_DAY FUNCTION

Often, trial analysis requires the calculation of a study day from start of treatment. Instead of repeating the code for this calculation in every program, it would be simpler to have a function to use.

The function below uses two numeric input arguments and will return a numeric result , i.e. the number of days that the event happened after the start of treatment. As you can see in the function code, a zero day will never be returned.

The following annotated code shows you how to define the function:



.

Let's use a simple data step to demonstrate how the function may be used:

```
options cmplib=sasuser.funcs;

data _null_;
  TRT_start_date = '15Feb2006'd;
  AE_Event_date  = '27Mar2006'd;

  studyday = Calculate_study_day(TRT_start_date, AE_Event_date);

  put studyday= ' Days';
run;
```

Point to where the function is stored

This is the function call

Write to the Log

LOG OUTPUT:

```
STUDYDAY=41  Days
```

DATE_IMPUTE FUNCTION

It's is quite common to have missing or partial dates for events in clinical trials. A common practice is to substitute day 15 if the day is missing and month 6 if the month is missing. The statistical analysis plan (SAP), however, usually provide specific rules for handling missing data, including partial dates, that may differ between projects. Instead of repeating the code for this imputation in every program, it would be simpler to have a function to use.

The Date_Impute function below will return a standard numeric SAS date value. Note the $ at the end of input arguments YYMMDD_date$ and logmsg$. That means they will be treated as character type input variables. There is no imputation if YYMMDD_date$ has a full value, i.e. length =8.

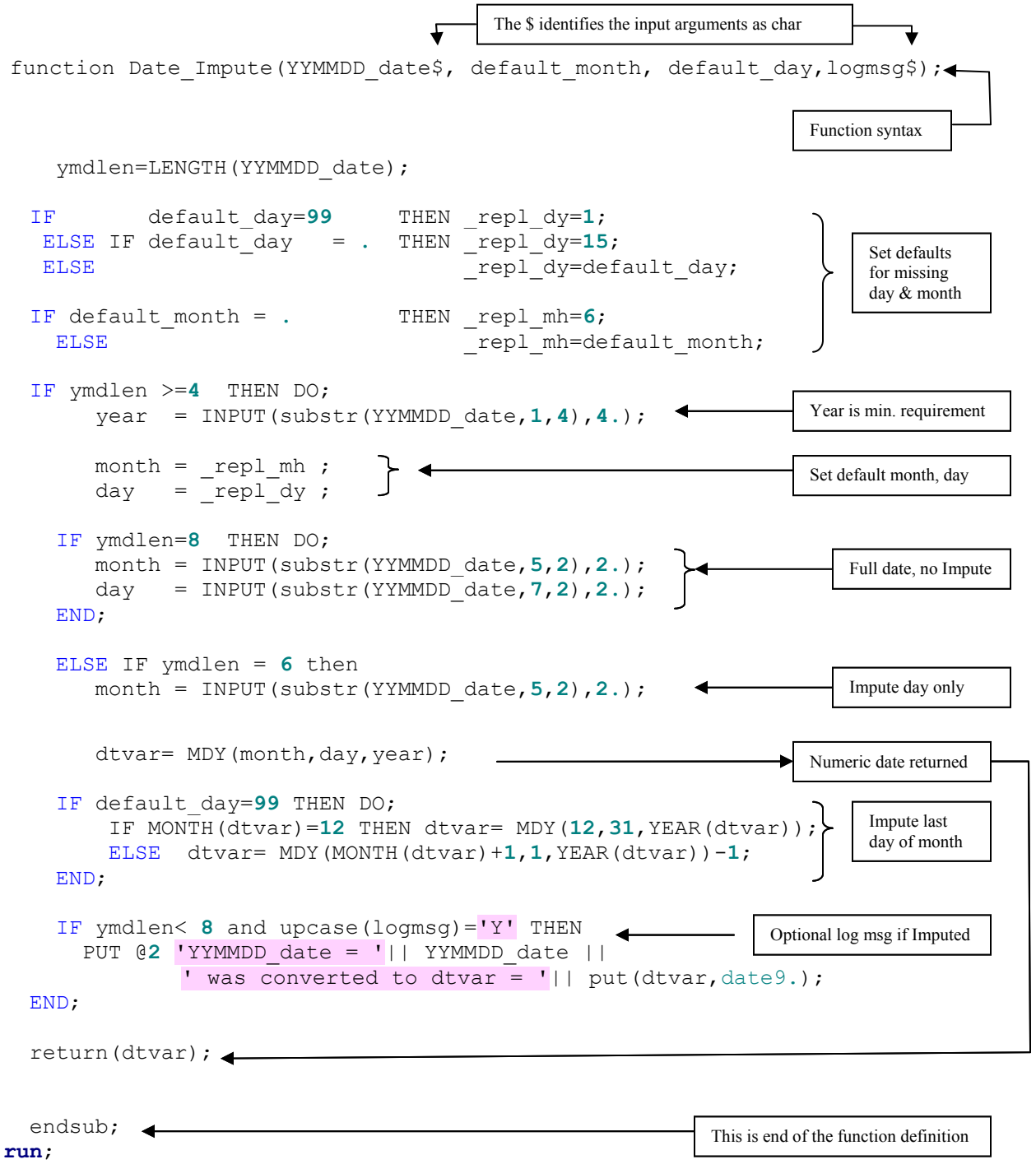
However, if the YYMMDD_date value represents a partial date, i.e. a missing month and/or day, those missing date components will be imputed. If no default values for the default_month and default_day arguments in the function call, the function will use it's own internal defaults (month=6, day=15) for imputations. If a default_day=99, the function will default the day (if it's missing) to the last day of the month. Optionally, Log messages may be issued when there is an imputation.

The Date_Impute function below will return a standard numeric SAS date value. Note the $ at the end of input arguments YYMMDD_date$ and logmsg$. That means they will be treated as character type input variables. There is no imputation if YYMMDD_date$ has a full value, i.e. length =8. However, if the YYMMDD_date value represents a partial date, i.e. a missing month and/or day, those missing date components will be imputed. If no default values for the default_month and default_day arguments in the function call, the function will use it's own internal defaults (month=6, day=15) for imputations. If a default_day=99, the function will default the day (if it's missing) to the last day of the month. Optionally, Log messages may be issued when there is an imputation.

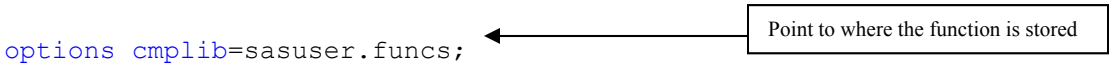
The following annotated code shows you how to define such a function:

```
proc fcmp outlib=sasuser.funcs.trial;
```

This is where the function is stored



Let's use a few simple data steps to demonstrate how the function may be used:



```
data _null_;
 format dtvar date9.;
  ymd='200207';
  dtvar= Date_Impute(ymd,.,.,'Y');
run;

LOG MSG:
YYMMDD_date = 200207 was converted to dtvar = 15JUL2002
```

Internal defaults will be used

Issue Impute log messages? Y or N

```
data _null_;
 format dtvar date9.;
  ymd='2002';
  dtvar= Date_Impute(ymd,1,1,'Y');
run;

LOG MSG:
YYMMDD_date = 2002 was converted to dtvar = 01JAN2002
```

change defaults to Month=1, Day=1

```
data _null_;
 format dtvar date9.;
  ymd='200207';
  dtvar= Date_Impute(ymd,1,14,'Y'); run;

LOG MSG:
YYMMDD_date = 200207 was converted to dtvar = 14JUL2002
```

change defaults to Month=1, Day=14

```
data _null_;
 format dtvar date9.;
  ymd='200207';
                                  /* override defaults: month(6) and day(LDoM)*/
  dtvar= Date_Impute(ymd,6,99,'Y');
run;

LOG MSG:
YYMMDD_date = 200207 was converted to dtvar = 31JUL2002
```

Impute last day of month

You can learn a number of things from this function example. As you can see , the code for this example is much longer than that of the first example. So it would make good sense to make a function out of this code and not to repeat the full code in a program over and over. This example also demonstrates how to declare the input arguments to be of character type by ending it's name with a $. In the code, you don't need to use the ending $ for the argument/variable names. Note there is no declaration of type for the function's result. We'll cover that in another example.

NON_MISS_CHAR FUNCTION

It is quite common to have missing values in character string variable. For reporting, however, we may want to put something in those variables that have missing values, i.e. 'N/A' or 'No results', etc.. A function that works with character strings and returns a character type result would be more convenient to use instead of having 'IF … THEN…ELSE… ' statements all over the program. There is an optional put statement to generate a log message. The following annotated code shows you how to define such a function:

```
proc fcmp outlib=sasuser.funcs.trial ;

  function Non_miss_char(Char_value$, default_value$, logmsg$)   $ ;

    Return_value = Char_value ;

    If  Char_value =' ' then do;
      Return_value = default_value ;
     IF upcase(logmsg)='Y' THEN
       PUT @2 'Missing character value was set to '||Return_value  ;
  end;

    return(Return_value);

  endsub;
run;
```

This is where the function is stored

The $ identifies the input arguments as char

This $ specifies that the return value will be Char type (see ***Note**)

Character string is returned

This is end of the function definition

***Note**: If you follow this $ with a space and number, i.e. $ 20, the returned character will be set to the length specified by that number. However, by specifying no number, the length of the string will be variable. In the code, you don't need to use the ending $ on the argument/variable name.

Now let's use a couple simple data steps to demonstrate how the function may be used:

```
options cmplib=sasuser.funcs;
```

Point to where the function is stored

```
data _null_;
  varlabel='Sample variable label';

  varlabel= Non_miss_char(varlabel,'Missing value','Y');
  put varlabel=;
run;
```

No substitution needed

LOG MSG:

```
VARLABEL=Sample variable label
```

```
data _null_;
  varlabel=' ';

  varlabel= Non_miss_char(varlabel,'''Empty string''' ,'Y');
run;
```

substitution needed

LOG MSG:

```
Missing character value was set to 'Empty string'
```

THE FCMP PROCEDURE – EXAMPLES FOR MAKING YOUR CALL ROUTINES

The following examples use long names for the subroutine and the arguments in order to make the example more readable. Usually, these names should be shorter in order to reduce the amount of typing.

CALC_YEARS CALL ROUTINE (SUBROUTINE)

This is a very simple example from the financial world. The subroutine calculates the number of years to maturity for a loan. Since this subroutine only /creates/modifies one variable (years) it could have been done just as easy with a function. However, it suffices for the purpose of showing how subroutines are defined and how the work.

The subroutine below takes the first two numeric input arguments and returns a calculated numeric value to the third argument. *Please note the Outargs statement must be before the code that computes the return variable. I realize that this is a very simple example, but it serves will as a basis for understanding how subroutines are defined.

The subroutine below takes the first two numeric input arguments and returns a calculated numeric value to the third argument. *Please note the Outargs statement must be before the code that computes the return variable. I realize that this is a very simple example, but it serves will as a basis for understanding how subroutines are defined.

The following annotated code shows you how to define such a subroutine:

```
proc fcmp outlib=sasuser.funcs.trial;

  subroutine calc_years(maturity, current_date, years);

  outargs years;

  years = round((maturity - current_date) / 365.25,0.1);

  endsub;
run;
```

Key word

subroutine name

Subroutine arguments

Subroutine syntax

Returned variable*

This is end of the subroutine definition

Now let's use a simple data step to demonstrate how the subroutine may be used:

```
options cmplib=sasuser.funcs;
```

Point to where the function is stored

```
data _null_;
  LoanEnd_date = '15Feb2016'd;
  today        = '7Oct2008'd;
  today        = today();
  call  calc_years(LoanEnd_date, today, years);
  put years=;
run;
```

LOG MSG:

```
YEARS=6.1
```

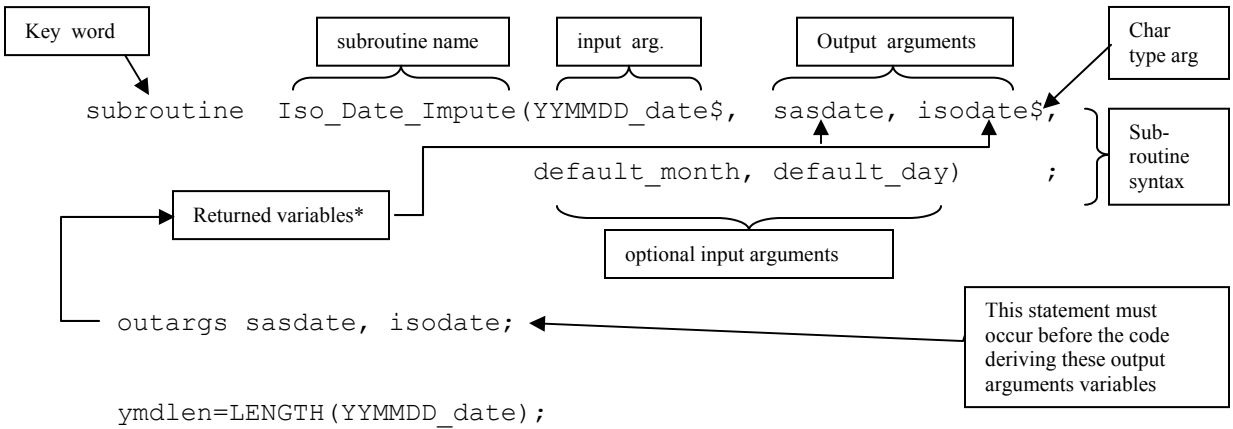
ISO_DATE_IMPUTE CALL ROUTINE (SUBROUTINE)

If you recall, the Date_Impute Function example took a partial ymmdd partial date, imputed the missing month and/or day and returned an explicit result (i.e. an numeric SAS date). Let's say that we also want to create and return an ISO 8601 compatible date after the imputation. Functions can only return one (explicit) result, but a call routine can create / modify multiple (implicit) results. Remember – arguments for a subroutine must include both the input and output arguments.

In this example we take the code from the Date_Impute Function and add a few lines to create a subroutine. The subroutine imputes the date, if necessary, creates an ISOdate and returns both the SASdate (numeric) and the ISOdate(char) .

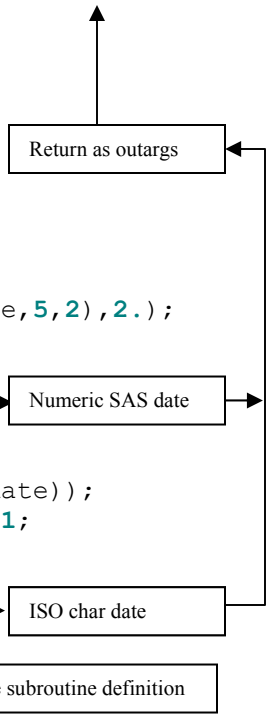
We wont discuss the imputation code here again as it was already done in the Function example. You can see that there is a simple statement added to create the ISOdate and the 'return' statement was replaced with and 'outargs' statement. The declaration for the arguments (i.e. Num or char) is done the same as for functions. The following annotated code shows you how to define such a subroutine:

```
proc fcmp outlib=sasuser.funcs.trial;
```



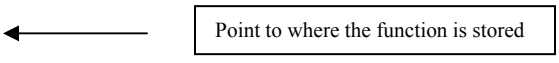

```
  ymdlen=LENGTH(YYMMDD_date);

IF       default_day=99       THEN _repl_dy=1;
 ELSE IF default_day    = .    THEN _repl_dy=15;
 ELSE                              _repl_dy=default_day;


IF default_month = .           THEN  _repl_mh=6;
  ELSE                               _repl_mh=default_month;
```

```
  IF ymdlen >=4  THEN DO;
       year  = INPUT(substr(YYMMDD_date,1,4),4.);
       month = _repl_mh ;
       day   = _repl_dy ;
    IF ymdlen=8  THEN DO;
       month = INPUT(substr(YYMMDD_date,5,2),2.);
       day   = INPUT(substr(YYMMDD_date,7,2),2.);
    END;
    ELSE IF ymdlen = 6 then  month = INPUT(substr(YYMMDD_date,5,2),2.);


       sasdate= MDY(month,day,year);

    IF default_day=99 THEN DO;
      IF MONTH(sasdate)=12 THEN sasdate= MDY(12,31,YEAR(sasdate));
       ELSE  sasdate= MDY(MONTH(sasdate)+1,1,YEAR(sasdate))-1;
    END;
  END;
    isodate = put(sasdate,E8601da.);


  endsub;
run;
```

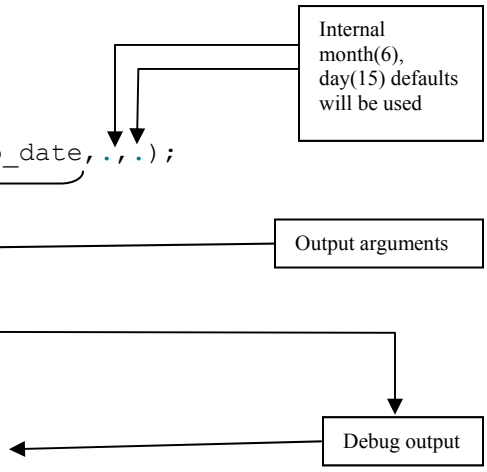


Now let's use a simple data step to demonstrate how the subroutine may be used.
I purposely used different variable names in the data step then those used in the subroutine definition. This is to demonstrate that the subroutine arguments are mere placeholders for passing values, i.e. the arguments and internal variables are all local to the subroutine and are not known outside:

```
options cmplib=sasuser.funcs;
```



```
data _null_;
 format sas_date date9. iso_date $20.;

  sas_date=.; iso_date=' ';

  AEonymd='200207';

  call Iso_Date_Impute(AEonymd,sas_date,iso_date,.,.);


  put AEonymd= sas_date= iso_date=;

run;


LOG MSG:
AEONYMD=200207 SAS_DATE=15JUL2002 ISO_DATE=2002-07-15
```



CONCLUSIONS

The new SAS 9.2 FCMP procedure is a powerful new tool for SAS users. You can now take the code that you used over and over again and make functions or subroutines out of it. These functions and subroutines are compiled once and then easily shared between programs and users. Having standard functions and subroutines in a project can also help standardize the implementation of imputation rules, computational algorithms , analysis models , etc. across all study programs in a project.

ACKNOWLEDGEMENT

I want like to thank Michael Pannucci of Boehringer Ingelheim for his timely efforts in reviewing and providing valuable feedback on this paper.

REFERENCES

[1] SAS Institute - Product documentation > SAS 9.2 Documentation > Base SAS Procedures Guide

CONTACT INFORMATION

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

John H Adams
900 Ridgebury Road
Ridgefield, CT, 06877-0368
Work Phone: 203-778-7820
Fax: 203-837-4413
Email: john.adams@boehringer-ingelheim.com
adamsjh@mindspring.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.