PharmaSUG 2018 - Paper BB-01

User-defined Functions for Processing Lab Data

Richard Read Allen, Peak Statistical Services

ABSTRACT

Lab data is often messy with multiple units given for the same parameter. Converting these to standard units can be challenging. Usually macros are used for these conversions. This presentation shows a method for converting labs using a user-defined function written using PROC FCMP. The procedure builds functions using DATA step code that can be stored and reused for multiple studies. A method for grading the toxicity of labs using a function will also be presented.

INTRODUCTION

The purpose of this paper is to introduce the reader to the FCMP procedure in SAS® for creating user-defined functions. It can be used to do repetitive tasks in the processing of clinical lab data. You will learn how to use DATA step code to create functions to convert labs to SI (standard international) units and to grade labs using a toxicity scale such as the CTCAE (Common Terminology Criteria for Adverse Events). Intermediate programming skills and a basic understanding of lab data in a clinical trial are beneficial.

BASICS OF PROC FCMP

The FCMP procedure (SAS Function Compiler) is part of Base SAS software beginning with version 9.2 and was originally developed as a programming language for several SAS/STAT, SAS/ETS, and SAS/OR procedures.

Using DATA step syntax PROC FCMP enables you to create, test, and store user-defined

- SAS functions
- CALL routines
- Subroutines

These are stored in a special type of data set and can be called in a DATA step and many SAS procedures. You can use most of the SAS programming statements and SAS functions that you can use in a DATA step to define FCMP functions and subroutines. The final code for your functions will be much easier to understand and maintain than the typical macro code usually used for such tasks.

In order to use functions defined by PROC FCMP, we need to store them in a special dataset (function library) with a four-level name as follows:

LIBREF . DATASET . PACKAGE . FUNCTION

where

LIBREF = library where dataset containing functions will be stored
DATASET = dataset within the library defined above containing the package of functions
PACKAGE = name of package containing functions
FUNCTION = name of function performing a specific task

It's possible to have multiple packages within a dataset and multiple functions within a package. For example, one could have a package called "conversions" and within this have a function for converting temperature from Fahrenheit to/from Celsius (LIBREF.DATASET.conversions.ConvertTemp) and another function for converting lab values to SI units (LIBREF.DATASET.conversions.ConvertLabs).

For SAS to find these functions we need to add the dataset to the search path using the CMPLIB system option:

```
options cmplib = (LIBREF . DATASET);
```

This option is used when creating the functions to tell SAS where to store them and in the code when calling the functions to tell SAS where to find them. It's the same syntax in both situations.

When creating the functions in the PROC FCMP statement we then use the OUTLIB option to point to this dataset and the package within the dataset where we wish to store the functions.

```
PROC FCMP outlib = LIBREF . DATASET . PACKAGE;
```

To create a basic function in PROC FCMP, the following syntax and structure are used:

FUNCTION *function-name(argument-1, ..., argument-n)*;

  ... programming-statements ...

  RETURN (*expression*);

ENDSUB;

In the programming statements above, most of the code you normally use in normal DATA step coding can be used to create your function. There are some exceptions which are explained in the PROC FCMP documentation. None of these affect what we plan to use in these simple examples. There are many other options and more detail available in PROC FCMP, but we will be concentrating on functions of this simple type in the examples that follow.

CONVERTING LAB UNITS

To illustrate the usefulness of user-defined functions in clinical programming, we're going to look at some typical lab data from a clinical trial. Table 1 below is a subset of variables from a standard SDTM LB data set. These are the minimal set of variables needed to illustrate this technique. The typical LB dataset will contain many more variables. Note that LBORRESN is not a standard SDTM variable. The original result (LBORRES) is a character variable, LBORRESN would be the numerical representation of this used for the conversions.

#	Variable	Type	Len	Label
1	usubjid	Num	8	Unique Subject ID
2	lbdtc	Char	16	Date of Collection
3	lbtestcd	Char	8	Laboratory Test Short Name
4	lborresu	Char	8	Original Units
5	lborresn	Num	8	Result or Finding in Original Units (N)
6	lbornrlo	Char	8	Original Normal Range Lower Limit
7	lbornrhi	Char	8	Original Normal Range Upper Limit

Table 1. CONTENTS of example LB dataset

Below is the dataset that will be used for the examples. This sample of data has six different parameters, each with multiple original units (LBORRESU). The object is to convert LBORRESN, LBORNRLO and LBORNRHI to a standard international (SI) units. These units will be stored in LBSTRESU and are the following for the six lab parameters in this example data.

ALB	→	g/L
PROT	→	g/L
BILI	→	umol/L
CREAT	→	umol/L
PLAT	→	x10E9/L
WBC	→	x10E9/L

usubjid	lbdtc	lbtestcd	lborresu	lborresn	lbornrlo	lbornrhi
1	03JAN2017	ALB	g/L	32	34	48
2	18AUG2017	ALB	g/dL	4.39	3.4	4.8
1	03JAN2017	BILI	umol/L	11	0	25
2	18AUG2017	BILI	mg/dL	0.35	0	1
1	03JAN2017	CREAT	umol/L	83	50	90
2	18AUG2017	CREAT	mg/dL	0.56	0.51	0.95
1	03JAN2017	PLAT	x10E9/L	233	145	483
2	18AUG2017	PLAT	/mmE3	329000	150000	450000
4	26Aug2017	PLAT	/uL	0.314	0.146	0.367
1	03JAN2017	PROT	g/L	65	61	79
2	18AUG2017	PROT	g/dL	7.51	6.61	8.01
1	03JAN2017	WBC	10*6/uL	6600	3500	11000
2	18AUG2017	WBC	x10E9/L	4.11	3.5	11
3	28JUN2017	WBC	/mmE3	5600	3500	11000
4	26AUG2017	WBC	/uL	0.0078	0.0035	0.011

Table 2. Example dataset

CREATING A FUNCTION USING DATA STEP CODE

Using the sample dataset described in Tables 1 and 2, we will take you through the steps of building a function to convert LBORRESN, LBORNRLO and LBORNRHI to standard international (SI) units from the units that they were collected in. In some cases (hopefully many in your data), the collected units will already be the same as the SI units. We will still need to include these cases into the function so that all records can be run through the same function for conversions,

Using PROC FCMP to Create the Function

First we will need to assign the library and data set name where we will store our package of functions:

```
libname funclib ‘c:\functions’;
options cmplib=(funclib.functions);
```

Here we are defining the library FUNCLIB as the area where we will be storing the dataset (FUNCTIONS) that will contain the package of functions we define.

Within this dataset we will define a package called conversions in the PROC FCMP statement:

```
proc fcmp outlib=funclib.functions.conversions;
```

Next we will define a function called ConvertLabs with three arguments:

```
function ConvertLabs(old_units $, new_units $, old_value);
```

- **old_units** for original units collected for labs (mapped to LBORRESU)
- **new_units** for the SI units for each parameter (mapped to LBSTRESU)
- **old_value** for the original result (mapped to LBORRESN)

Note that when an argument in the function is character, it is followed by a $ sign in the function definition.

Now we are ready to start creating the conversions. Most of the conversions are independent of the parameter (LBTESTCD). We also have the cases where no conversions are needed. These can be added to the function using ‘programming statements’ as below:.

```
 /*** g/dL --> g/L ***/
  if old_units='g/dL' & new_units='g/L' then new_value=old_value*10;

 /*** /mmE3 --> 10^9/L ***/
  if old_units='/mmE3' & new_units='x10E9/L' then new_value=old_value/1000;

 /*** 10*6/uL --> 10^9/L ***/
  if old_units='10*6/uL' & new_units='x10E9/L' then new_value=old_value/1000;

 /*** /uL --> 10^9/L ***/
  if old_units='/uL' & new_units='x10E9/L' then new_value=old_value*1000;

 /*** Same units ***/
  if old_units=new_units then new_value=old_value;
```

Last we will issue the RETURN statement and designate the value to be returned when the function is evaluated as **new_value** and the ENDSUB statement to close the function.

```
    return (new_value);
  endsub;
```

Putting this all together, we have the basic code for most of the conversions we need for our example:

```
proc fcmp outlib=funclib.functions.conversions;
 function ConvertLabs(old_units $, new_units $, old_value);
  if old_units='g/dL' & new_units='g/L' then new_value=old_value*10;
  if old_units='/mmE3' & new_units='x10E9/L' then new_value=old_value/1000;
  if old_units='10*6/uL' & new_units='x10E9/L' then new_value=old_value/1000;
  if old_units='/uL' & new_units='x10E9/L' then new_value=old_value*1000;
  if old_units=new_units then new_value=old_value;
  return (new_value);
 endsub;
run;
```

Calling the function created by PROC FCMP in a data step

The function created above is usually created in a different program than the program used to convert the labs, but it can be done in the same program. If we are using a separate program, we will need to issue the libname and options statements like above so that SAS knows where to find the functions we created:

```
libname funclib ‘c:\functions’;
options cmplib=(funclib.functions);
```

In our example, we will need to start by creating the variable LBSTRESU in our dataset. The easiest way to do this is by using a format.

```
proc format;
  value $siunit
    'PROT'='g/L'
    'ALB'='g/L'
    'CREAT'='umol/L'
    'BILI'='umol/L'
    'WBC'='x10E9/L'
    'PLAT'='x10E9/L'
  ;
run;
```

We will also need to convert LBORNRLO and LBORNRHI to numeric values to use in the conversion to standard ranges. Then we’ll call the function to translate the following:

- LBORRESN into LBSTRESN – original observed value into standard observed value
- LBORNRLO to LBSTNRLO – original to standard lower normal range
- LBORNRHI to LBSTNRHI – original to standard upper normal range
-

This can all be done in one DATA step as follows:

```
data Example(drop=ornrlo ornrhi);
  set Sample;
  lbstresu=put(lbtestcd,$siunit.);
  ornrlo=input(lbornrlo,best.);
  ornrhi=input(lbornrhi,best.);

 *-- Call custom ConvertLabs function to convert values into SI units;

  lbstresn=ConvertLabs(lborresu,lbstresu,lborresn);
  lbstnrlo=ConvertLabs(lborresu,lbstresu,ornrlo);
  lbstnrhi=ConvertLabs(lborresu,lbstresu,ornrhi);

 *-----------------------------------------------------------------;
run;
```

When this DATA step is run using the sample data in table 2, what we get for the new variables is summarized in table 3 below:

usubjid	**lbdtc**	**lbtestcd**	**lborresu**	**lborresn**	**lbornrlo**	**Lbornrhi**
1	03JAN2017	ALB	g/L	32	34	48
2	18AUG2017	ALB	g/dL	4.39	3.4	4.8
1	03JAN2017	BILI	umol/L	11	0	25
2	18AUG2017	BILI	mg/dL	0.35	0	1
1	03JAN2017	CREAT	umol/L	83	50	90
2	18AUG2017	CREAT	mg/dL	0.56	0.51	0.95
1	03JAN2017	PLAT	x10E9/L	233	145	483
2	18AUG2017	PLAT	/mmE3	329000	150000	450000
4	26Aug2017	PLAT	/uL	0.314	0.146	0.367
1	03JAN2017	PROT	g/L	65	61	79
2	18AUG2017	PROT	g/dL	7.51	6.61	8.01
1	03JAN2017	WBC	10*6/uL	6600	3500	11000
2	18AUG2017	WBC	x10E9/L	4.11	3.5	11
3	28JUN2017	WBC	/mmE3	5600	3500	11000
4	26AUG2017	WBC	/uL	0.0078	0.0035	0.011

			lbstresu	**lbstresn**	**lbstnrlo**	**lbstnrhi**
			g/L	32	34	48
			g/L	43.9	34	48
			umol/L	11	0	25
			umol/L			
			umol/L	83	50	90
			umol/L			
			x10E9/L	233	145	483
			x10E9/L	329	150	450
			x10E9/L	314	146	367
			g/L	65	61	79
			g/L	75.1	66.1	80.1
			x10E9/L	6.6	3.5	11
			x10E9/L	4.11	3.5	11
			x10E9/L	5.6	3.5	11
			x10E9/L	7.8	3.5	11

Table 3. Example dataset converted except for mg/dL→ umol/L conversions

Expanding the function

All of the above have been converted to SI units except the mg/dL to umol/L conversions for BILI and CREAT. These are a bit more difficult since these conversion factors are parameter dependent.

CREAT	mg/dL → mmol/L	multiply by 88.4
BILI	mg/dL → mmol/L	multiply by 17.1

In order to apply these conversions we will need to add another character argument to the function as a placeholder for the LBTESTCD:

```
proc fcmp outlib=funclib.functions.conversions;
  function ConvertUnits(code $, old_units $, new_units $, old_value);
```

We now have four arguments in our function:

- **code** for lab parameters (mapped to LBTESTCD)
- **old_units** for original units collected for labs (mapped to LBORRESU)
- **new_units** for the SI units for each parameter (mapped to LBSTRESU)
- **old_value** for the original result (mapped to LBORRESN)

The conversions for CREAT and BILI can then be accomplished using the following programming statements in the function definition in PROC FCMP:

```
   /*---------------------------------------------------------------*/
   /* Conversions dependent on lab analyte (code)
   /*---------------------------------------------------------------*/
   /*** mg/dL <--> umol/L ***/
    if old_units='mg/dL' & new_units='umol/L' then do;
      if code='CREAT' then new_value=old_value*88.4;
      else if code='BILI' then new_value=old_value*17.1;
    end;
```

Combining all of this together, we will have defined a function to convert all of the records in our sample data. This is just a small snapshot of the big picture. There are many more lab parameters to consider and lots of possible conversions. These can be added to the code as needed relatively easily.

The complete function to convert our sample dataset is as follows:

```
proc fcmp outlib=funclib.functions.conversions;
 function ConvertUnits(code $, old_units $, new_units $, old_value);
  if old_units='g/dL' & new_units='g/L' then new_value=old_value*10;
  if old_units='/mmE3' & new_units='x10E9/L' then new_value=old_value/1000;
  if old_units='10*6/uL' & new_units='x10E9/L' then new_value=old_value/1000;
  if old_units='/uL' & new_units='x10E9/L' then new_value=old_value*1000;
  if old_units=new_units then new_value=old_value;

 /*---------------------------------------------------------------*/
  if old_units='mg/dL' & new_units='umol/L' then do;
    if code='CREAT' then new_value=old_value*88.4;
    else if code='BILI' then new_value=old_value*17.1;
  end;

  return (new_value);
 endsub;
run;
```

CREATING A FUNCTION USING AN EXCEL SPREADSHEET

Another method of getting conversions into a function is to store all of the various types of units and their conversion factors to SI units in a central excel file. You could have one master excel conversion file to use on all studies or create a separate one for each study or group of studies. These excel files should have columns for LBTESTCD, LBORRESU, LBSTRESU and FACTOR – the conversion factor from original to SI units – at a minimum.

For our example this excel spreadsheet would contain the information in table 4 below:

lbtestcd	lborresu	lbstresu	factor
ALB	g/L	g/L	1
ALB	g/dL	g/L	10
BILI	umol/L	umol/L	1
BILI	mg/dL	umol/L	17.1
CREAT	umol/L	umol/L	1
CREAT	mg/dL	umol/L	88.4
PLAT	x10E9/L	x10E9/L	1
PLAT	/mmE3	x10E9/L	0.001
PLAT	/uL	x10E9/L	1000
PROT	g/L	g/L	1
PROT	g/dL	g/L	10
WBC	10*6/uL	x10E9/L	0.001
WBC	x10E9/L	x10E9/L	1
WBC	/mmE3	x10E9/L	0.001
WBC	/uL	x10E9/L	1000

Table 4. Excel conversion file for example data

We can use DATA _NULL_ coding to generate the 'programming statements' to use in our function to convert labs based on the spreadsheet in table 4. These statements are stored in an external file that is read back into PROC FCMP. It also serves as a record of the conversions that are programmed into the function. The DATA _NULL_ code to generate these statements would look like this:

```
filename convert 'c:\Your Data Folder\conversions.sas';

data _null_;
  file convert;
  set ConversionFactors end=EOF;
  code=cats('"',lbtestcd,'"');
  unit1=cats('"',lborresu,'"');
  unit2=cats('"',lbstresu,'"');

 /* unit1 --> unit2 */
  if _n_=1 then
      put "if lbtestcd=" code " & old_units=" unit1 " & new_units=" unit2
          " then lbtestcd=old_value*" factor ";";
  else
    put "else if lbtestcd=" code " & old_units=" unit1 " & new_units=" unit2
        " then new_value=old_value*" factor ";";
  output;
run;
```

This DATA _NULL_ creates a file with a bunch of if/then/else statements that define the conversions. These are similar to the statements we programmed into the previous function where we created them based on the conversions we needed. In this case, there are more lines since we include all cases for conversions of g/dL to g/L, for example, and include the LBTESTCD in the logic for all conversions where we did not before.

The big advantage here though is that we have code to derive these statements rather than having to type them all in individually. Here is what the derived statements created by the DATA _NULL_ step above and placed into the file conversions,sas will look like:

```
 if lbtestcd="ALB" & old_units="g/L" & new_units="g/L" then lbtestcd=old_value*1 ;
 else if lbtestcd="ALB" & old_units="g/dL" & new_units="g/L" then new_value=old_value*10 ;
 else if lbtestcd="BILI" & old_units="umol/L" & new_units="umol/L" then new_value=old_value*1 ;
 else if lbtestcd="BILI" & old_units="mg/dL" & new_units="umol/L" then new_value=old_value*17.1 ;
 else if lbtestcd="CREAT" & old_units="umol/L" & new_units="umol/L" then new_value=old_value*1 ;
 else if lbtestcd="CREAT" & old_units="mg/dL" & new_units="umol/L" then new_value=old_value*88.4 ;
 else if lbtestcd="PLAT" & old_units="x10E9/L" & new_units="x10E9/L" then new_value=old_value*1 ;
 else if lbtestcd="PLAT" & old_units="/mmE3" & new_units="x10E9/L" then new_value=old_value*0.001 ;
 else if lbtestcd="PLAT" & old_units="/uL" & new_units="x10E9/L" then new_value=old_value*1000 ;
 else if lbtestcd="PROT" & old_units="g/L" & new_units="g/L" then new_value=old_value*1 ;
 else if lbtestcd="PROT" & old_units="g/dL" & new_units="g/L" then new_value=old_value*10 ;
 else if lbtestcd="WBC" & old_units="10*6/uL" & new_units="x10E9/L" then new_value=old_value*0.001 ;
 else if lbtestcd="WBC" & old_units="x10E9/L" & new_units="x10E9/L" then new_value=old_value*1 ;
 else if lbtestcd="WBC" & old_units="/mmE3" & new_units="x10E9/L" then new_value=old_value*0.001 ;
 else if lbtestcd="WBC" & old_units="/uL" & new_units="x10E9/L" then new_value=old_value*1000 ;
```

Now all we have to do is to %include this file into PROC FCMP where the ‘programming statements’ belong and the function to convert labs is built and ready to use.

```
options cmplib=(funclib.functions);
proc fcmp outlib=funclib.functions.conversions;
  function ConvertLabs(code $,old_units $, new_units $, old_value);
    %include convert;
    return (new_value);
  endsub;
run;
```

TOXICITY GRADING OF LABS USING A FUNCTION

One other great use of functions in clinical lab programming is to create a routine to derive the toxicity scores based on a standard toxicity grading system such as the CTCAE (Common Terminology Criteria for Adverse Events). It is easiest to apply this function after the conversions are done so that we only need to have grading equations for one type of units – SI units – for each lab parameter (LBTESTCD). It is also helpful to have identified the baseline observations for each parameter since some of the toxicity rules depend on the value of the baseline observation.

Below are the CTCAE toxicity grading scales for some of the parameters in our example:

Lab Parameter	**Grade 1**	**Grade 2**	**Grade 3**	**Grade 4**
Albumin (g/L)	<LLN - 30	<30 - 20	<20	Not applicable
Bilirubin (umol/L)	>ULN - 1.5 x ULN	>1.5 - 3.0 x ULN	>3.0 - 10.0 x ULN	>10.0 x ULN
Platelets (10^9/L)	<LLN -75.0	<75.0- 50.0	<50.0- 25.0	<25.0
White Cells Count (10^9/L)	<LLN-3	<3-2	<2-1	<1

Table 5. CTCAE Toxicity grading for selected lab parameters

When creating functions for grading toxicity, I prefer to use arithmetic expressions rather than a bunch of nested if/then/else logic. The programming statement to grade Albumin (ALB) would look like this:

```
if lbcd='ALB' then grade=3*( . <  value <  20)
                        +2*(20 <= value <  30)
                        +1*(30 <= value < uln);
```

This single statement returns the same result as the following 4 nested if/then/else statements

```
if lbcd='ALB' then do;
  if . <value <20 then grade=3;
  else if 20<=value<30 then grade=2;
  else if 30<=value<uln then grade=1;
  else grade=0;
end;
```

When designing the programming statements using arithmetic statements as above, one must be careful of missing limits in your data. The following code would be the full function to grade the toxicity levels for the four parameters above. Note that at the beginning flags NML and NMU are defined based on the existence of the limits in the data set. We used the NMU flag in the programming statement for BILI, otherwise if ULN is missing, all values would be graded 4.

```
proc fcmp outlib=funclib.functions.conversions;
  function GradeTox(lbcd $, lln, uln, value);
    nml=(missing(lln)=0);  *<== Non-missing normal range lower limit;
    nmu=(missing(uln)=0);  *<== Non-missing normal range upper limit;
   /*-----------------------------------------------------------*/
   /* Tox Grades
   /*-----------------------------------------------------------*/
   /*HEMATOLOGY */
     if lbcd='WBC' then grade=4*(       value < 1.0)
                             +3*(1.0 <= value < 2.0)
                             +2*(2.0 <= value < 3.0)
                             +1*(3.0 <= value < lln);
     else if lbcd='PLAT' then grade=4*( . <  value <  25)
                                   +3*(25 <= value <  50)
                                   +2*(50 <= value <  75)
                                   +1*(75 <= value < lln);

  /*CHEMISTRY */
     else if lbcd='ALB' then grade=3*( . <  value <  20)
                                  +2*(20 <= value <  30)
                                  +1*(30 <= value < lln);

     else if lbcd='BILI' then grade=4*(nmu &            value >  10.0*uln)
                                   +3*(nmu & 3.0*uln <  value <= 10.0*uln)
                                   +2*(nmu & 1.5*uln <  value <=  3.0*uln)
                                   +1*(nmu &     uln <  value <=  1.5*uln);

   return (grade);
  endsub;
run;
```

If we apply the above function to our example data we would do it as follows:

```
data Example(drop=ornrlo ornrhi);
  set Sample;
  lbstresu=put(lbtestcd,$siunit.);
  ornrlo=input(lbornrlo,best.);
  ornrhi=input(lbornrhi,best.);
 *-- Call custom ConvertLabs function to convert values into SI units;
  lbstresn=ConvertLabs(lborresu,lbstresu,lborresn);
  lbstnrlo=ConvertLabs(lborresu,lbstresu,ornrlo);
  lbstnrhi=ConvertLabs(lborresu,lbstresu,ornrhi);
 *-- Call custom GradeTox function derive toxicity grades;
  lbtoxgrn=GradeTox(lbtestcd,lbstnrlo,lbstnrhi,lbstresn);
run;
```

For the first record for ALB, lbtoxgrn=1. For all other ALB, BILI, PLAT, and WBC lbtoxgrn=0. For CREAT and BILI lbtoxgrn will be missing.

Other toxicity grading equations are a bit more challenging to program. In the example above, we have limited the equations to some of the simpler ones. However, there are more complicated grading rules some of which include comparisons to baseline values or other ‘special’ rules. The rule for CREAT is

Creatinine (umol/L)	>1 - 1.5 x baseline; >ULN -1.5 x ULN	>1.5 - 3.0 x baseline; >1.5 -3.0 x ULN	>3.0 baseline; >3.0 - 6.0 xULN	>6.0 x ULN

The programming statement for CREAT would require an extra argument be added to the function for the baseline value (BSLN). This statement would look something like this:

```
  nmb=(missing(bsln)=0);  *<== Non-missing baseline value;

  if lbcd='CREAT' then  grade=max(4*(   (nmu & value>6.0*uln)
                                     or (nmb & value >  6.0*bsln)
                                     )
                                 ,3*(   (nmu & 3.0*uln<value<= 6.0*uln)
                                     or (nmb & 3.0*bsln<value<=6.0*bsln)
                                     )
                                 ,2*(   (nmu & 1.5*uln<value<= 3.0*uln)
                                     or (nmb & 1.5*bsln<value<=3.0*bsln)
                                    )
                                 ,1*(   (nmu &     uln<value<=1.5*uln)
                                     or (nmb &     bsln<value<= 1.5*bsln)
                                    )
                                 );
```

The reason to use the maximum rather than the sum is that it is possible for the value to satisfy one condition compared baseline and a different condition based on the value and ULN. Note also the NMB is a check for non-missing baseline value similar to the check for NML and NMU is the original function.

Other lab parameters also have separate rules for values out of lower range (HYPO) and out of upper range (HYPER). One such parameter is Glucose

Glucose (mmol/L)	HYPO	<LLN - 3.0	<3.0 - 2.2	<2.2 - 1.7	<1.7
Glucose (mmol/L)	HYPER	>ULN - 8.9	>8.9 - 13.9	>13.9 -27.8	>27.8

Programming statements to grade GLUC would look like this and does both.

```
                            /**  #Hypoglycemia            #Hyperglycemia **/
   if lbcd='GLUC' then grade=4*(( . < value<1.7) or (      27.8<value      ))
                            +3*((1.7<=value<2.2) or (      13.9<value<=27.8))
                            +2*((2.2<=value<3  ) or (       8.9<value<=13.9))
                            +1*((  3<=value<lln) or (nmu & uln<value<=8.9));
```

These can be flagged as HYPO or HYPER depending whether LBFLAG is L or H, respectively.

CONCLUSION

The features of PROC FCMP enable clinical programmers to more easily read, write and maintain complex code with independent and reusable subroutines. It is especially useful in processing lab data.

Functions can be expanded and modified easily. They use regular DATA step code so they are easier to understand and maintain than the equivalent MACRO code.

You can reuse the PROC FCMP routines in any DATA step or SAS procedure that has access to their storage location.

REFERENCES

Using PROC FCMP to the Fullest: Getting Started and Doing More, Arthur L. Carpenter, California Occidental Consultants, Anchorage, AK – HOW presented at WUSS 2013.

Base SAS 9.4 Procedures Guide, Second Edition, SAS Institute, Cary, NC

CTCAE vs Laboratory Parameters, PhUSE Wiki, http://www.phusewiki.org/wiki/index.php?title=CTCAE_criteria_vs_laboratory_parameters

ACKNOWLEDGMENTS

Many thanks to Art Carpenter whose HOW at WUSS in 2013 started my thinking of uses of PROC FCMP in my daily work.

RECOMMENDED READING

- *Hashing in PROC FCMP to Enhance Your Productivity, Andrew Henrick, Donald Erdman and Stacey Christian, SAS Institute, Cary, NC – Presented at WUSS 2014*

CONTACT INFORMATION

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

Richard Read Allen
Peak Statistical Services
303-670-5386
rrallen@peakstat.com
www.peakstat.com