PharmaSUG 2015 – Paper BB15

Creating Data-Driven SAS® Code with CALL EXECUTE

Hui Wang, Biogen, Cambridge, MA

ABSTRACT

In SAS programming language, the basic function of CALL EXECUTE is to resolve its argument and then execute the resolved value. It is one of the call routines which can interact with the macro facility within a data step. Therefore, in data-driven programming practice, it is widely used on such occasions as running macros conditionally or passing data step values to macros in a data step.

This paper will discuss how to take advantage of the features of CALL EXECUTE to apply data-driven programming strategies. Such examples include: 1) Create a data set frame based on metadata provided; 2) Evaluate assessment values based on external criteria; 3) Merge comments from EXCEL file back to corresponding data records regardless of variable attributes.

INTRODUCTION

In data-driven programming practice, the properties of data itself directly or indirectly drive the flow of the program. As a result, the program seems to be smart and needs relatively low maintenance by the programmer. The SAS programming language provides many tools for this strategy. The call routine EXECUTE is one of them since it can create subsequent dynamic SAS code during data step execution. Its syntax is quite simple:

```
CALL EXECUTE(argument);
```

The argument usually seen is a character string or a character expression that can be resolved to elements of a macro or regular SAS statements.

In this paper, we will discuss how CALL EXECUTE works after SAS code with it is submitted and we will also discuss three scenarios to learn what role CALL EXECUTE can play in data-driven programming. As you may have found out, there have been a number of papers regarding this topic on the Internet. Hopefully, this paper can add something new to this trend.

HOW DOES CALL EXECUTE WORK

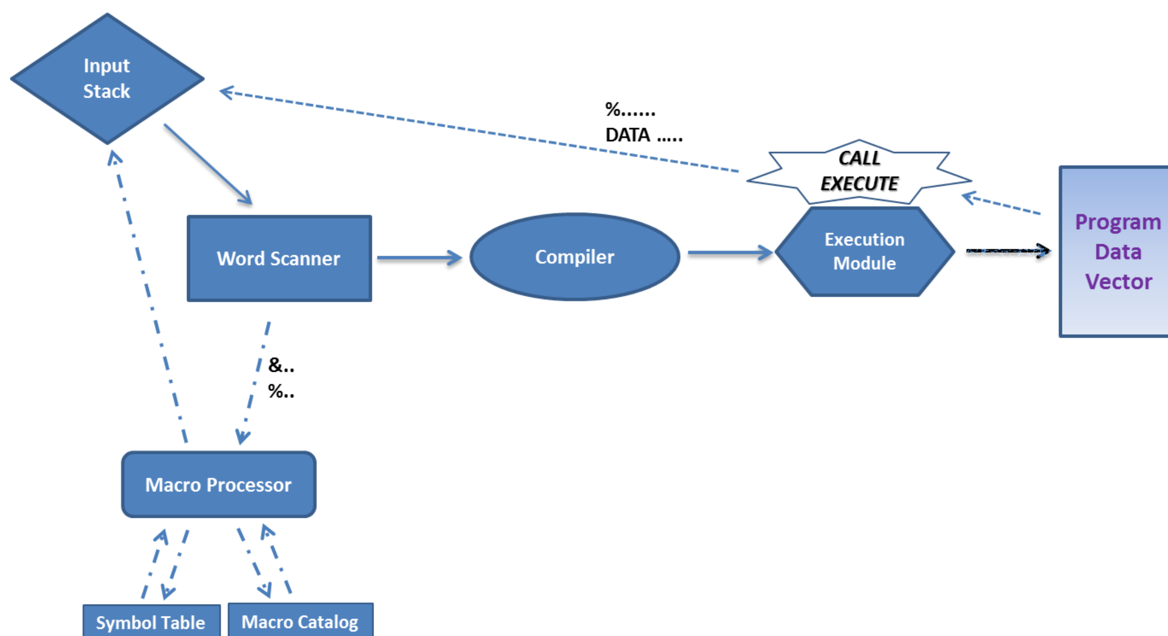
To fully answer this question, it is necessary to understand how code is processed by the SAS system (hereafter referred to as the system) as described in Display 1. Right after being submitted, the code goes to the input stack, which is a virtual component holding the code until the system is ready to push it into the next component called the word scanner. In the word scanner, the so-called tokenization happens, which assembles the SAS code characters into tokens based on certain rules. Generally, it is in here that the special character '&' and '%' can trigger activation of the macro facility component macro processor and then leads the tokens following them to get into it. With the pre-stored information in the symbol table or the macro catalog, the macro processor handles macro expression (macro variables or macro instructions) by repeatedly interacting with the input stack and the word scanner.

If no such special characters are found by the word scanner, tokens move to the compiler, which attempts to compile the code after it receives a data step boundary token (e.g. RUN, QUIT). During this compilation phase, the compiler checks code syntax, sets up the PDV (program data vector) and executes certain statements like KEEP and DROP. Meanwhile, other system components are inactive. If everything is fine, the compiled code keeps moving and eventually reaches the execution module, where it directs the system to do more to the PDV in order to create new data sets or modifies old ones.

The magic of CALL EXECUTE is its ability to pull data values from the PDV, build up new SAS code with them and then send the new code back to the input stack (Display 1), when the "parental" code, which contains this call routine, is still in the execution module. The newly constructed code can be either macro expressions or data step statements. In the input stack, they are in the queue per the time they were born. Right after execution of the "parental" code, they start to move along this "assembly line" following the same rules as regular SAS code. In this way, the data values from a "control" data set can be utilized to produce data-dependent subsequent code to fulfill data-driven programming designs.

A quotation mark is usually required to separate the "static" part (character string) and the "dynamic" part (character expression) in an argument. Different quotation marks cause CALL EXECUTE to resolve arguments in different ways. Per the messages from SAS website, with single quotation marks, argument is resolved during program execution.

And with double quotation marks, argument is resolved during data step construction. But in real programming practice, both quotation marks give the same expected subsequent code if the subsequent code is only relevant to data step statements. However, if the subsequent code is related to a macro, double quotation marks may lead to unexpected results, which will be discussed in details later.



Display 1. Flow of SAS Processing and CALL EXECUTE

No matter what quotation mark is applied, the argument execution by CALL EXECUTE follows the same rule: the resolved macro elements are executed immediately while the resolved SAS statements have to wait until the execution of the control data set is complete. This rule is quite easy to understand. When the execution module is still in control to deal with the "parental" code, the word scanner will not deliver tokens from the new code, which is resolved from the CALL EXECUTE argument, to the complier. However, the macro processor is not busy meanwhile and thus can take care of macro elements without any problem.

This is a short illustration for how CALL EXECUTE gets involved in the system processing. The next sections will discuss some cases/examples.

CREATE A DATA SET FRAME PER PRE-DEFINED METADATA

Generally speaking, creating variables and setting up variable attributes are not fun jobs. But what if the variable-level metadata could create an empty data set (data frame) first and then variable values could be derived from a pre-defined algorithm? This method could save work and reduce human errors. Display 2 is such a simple example.

Display 2A shows the metadata data set METADATA, which will be used as the control data set for CALL EXECUTE. Display 2B is about the "parental" code, showing that CALL EXECUTE pulls values of the control data set variables, which describe variable attributes expected for the data frame (highlighted with dark blue), and combines them with fixed elements in the argument (highlighted with red) into new SAS code (Display 2C). In this case, the control data set has two observations and thus its internal loop iterates twice. During each iteration, CALL EXECUTE generates one row of code, which is shown in Display 2C as the rows with blue pieces:

```
3   SUBJID label="Subject ID" length=$7.
4   AGE label="Age (Years)" length=8.
```

Obviously, the data values from the control data set now serve as parts of SAS code in this new program. This code-generated code will get into the SAS processing flow as described in Display 1 in the same way as regular code does. And it will end up with a data frame with all variable-level attributes defined in the metadata. Moreover, if any updates to the metadata are needed afterwards, re-running the "parental" code is more than enough to deliver the changes to the data frame.

A. Metadata:

Name	Label	Type	Length	Format
SUBJID	Subject Number	Char	7	$7.
AGE	Age (Years)	Num	8	8.

B. Code produced by programmer:

```
1  data _null_;
2    set METADATA end=eof;
3    if _N_ = 1 then do;
4      call execute('data FRAME;');
5      call execute('  attrib');
6    end;
7    call execute('    ' || strip(NAME) || ' label="'|| strip(LABEL) ||'"
                  length='|| strip(FORMAT) || ' ');
8    if eof then do;
9      call execute('  ;');
10     call execute('  call missing (of _all_);');
11     call execute('run;');
12   end;
13 run;
```

C. Code produced by CALL EXECUTE:

```
1  data FRAME;
2    attrib
3      SUBJID label="Subject ID" length=$7.
4      AGE label="Age (Years)" length=8.
5    ;
6    call missing (of _all_);
7  run;
```

Display 2. Create a Data Set Frame Based on Metadata

EVALUATE ASSESSMENT RESULTS WITH EXTERNAL CRITERIA

In this case, certain assessment results must be evaluated with criteria from an external source. For example, the safety department may provide a customized file for potentially clinically significant criteria (Display 3B) and then request an evaluation for some laboratory data (Display 3A) based on it. The intuitive way might be to apply multiple IF – THEN – ELSE statements, which undoubtedly involves a lot of typing or copying-pasting if there are many parameters in the original data set. One of the alternative ways, of course, is to let CALL EXECUTE do the tedious and repetitive work.

A. Original Dataset:

Parameter Code	Analysis Value (N)
ABC	30
ABC	60
DEF	0.5
DEF	0.01

B. Normal range:

Parameter Code	Criteria
ABC	< 50
DEF	> 0.2

C. Code produced by programmer:

```
1  data _null_;
2    set NORMAL_RANGE end=eof;
3    if _N_ = 1 then do;
4      call execute('data ADLB;');
5      call execute('  set ADLB;');
6      call execute('  attrib FLAG label="Normal Range Indicator" length=$1
                           CRITERIA label="Criteria" length=$50;');
7    end;
8    call execute('  if strip(PARAMCD) = "'|| strip(PARAMCD) ||'" then do;');
9    call execute('    if AVAL '|| strip(CRITERIA) ||' then FLAG="Y";');
10   call execute('    else FLAG="N"; ');
11   call execute('    CRITERIA="'|| strip(CRITERIA) ||'";');
12   call execute('  end;');
13   if eof then call execute('run;');
14 run;
```

D. Code produced by CALL EXECUTE:

```
1  data ADLB;
2    set ADLB;
3    attrib FLAG label="Normal Range Indicator" length=$1
            CRITERIA label="Criteria" length=$50;
4    if strip(PARAMCD) = "ABC" then do;
5      if AVAL < 50 then FLAG="Y";
6      else FLAG="N";
7      CRITERIA="< 50";
8    end;
9    if strip(PARAMCD) = "DEF" then do;
10     if AVAL > 0.2 then FLAG="Y";
11     else FLAG="N";
12     CRITERIA="> 0.2";
13   end;
14 run;
```

E. Final Dataset after Evaluation:

Parameter Code	Analysis Value (N)	Normal Range Indicator	Criteria
ABC	30	Y	< 50
ABC	60	N	< 50
DEF	0.5	Y	> 0.2
DEF	0.01	N	> 0.2

Display 3. Evaluate Assessment Results with External Criteria

Similar to the example from Display 2, the criteria file (Display 3B) can function as a control data set (Display 3C). In Display 3, blue color suggests variables or values originated from the control data set while the red color belongs to the newly born code as well. The key piece from the "parental" code is:

```
9   call execute('        if AVAL '|| strip(CRITERIA) ||' then FLAG="Y";');
```

When it is executed, CALL EXECUTE resolves "strip(CRITERIA)" by taking the value of the variable CRITERIA from the PDV of the control data set NORMAL_RANGE (e.g. "<50"), and integrating it with the rest part of the argument. Consequently, in Display 3D, we have the code like

```
5   if AVAL < 50 then FLAG="Y";
```

which is exactly the IF – THEN statement we want. Above this level, the "parental" code

```
8  call execute('    if strip(PARAMCD) = "'|| strip(PARAMCD) ||'" then do;');
```

produces the subsequent code:

```
4  if strip(PARAMCD) = "ABC" then do;
```

which assures that both the assessment value and the criterion value are assessed for the same parameter.

Again, each internal loop of the control data set leads to a new IF – THEN block in the subsequent code (Display 3D). After the subsequent code is ready, it will create the data set as shown in Display 3E, where the red-highlighted values address the request we received. In addition, the variable CRITERIA values are also pulled to the new data set as references.

MERGE COMMENTS FROM EXCEL FILE BACK TO CORRESPONDING DATA RECORDS

This example is about data cleaning for an ongoing study. Display 4A represents a data issue found during periodic AE data review. To make the study team aware of it, the record was exported to an EXCEL file and then sent out with a comment. After a little while, the data manager delivered a new AE data set, as represented by Display 4B. In order to determine if the comment was addressed, the comment file needs to be merged back to the current AE data set. Since the SAS data set-EXCEL file-SAS data set conversion definitely changes variable attributes, direct merging is almost impossible. The challenge is how to put previous data back to the right location of the new AE data set regardless of variable name, type, length, etc.

A. Old AE data with comment:

Investigator Term	Preferred Term	Start Date	End Date	Duration	Comment
POUNDING HEADACHE	Headache	2001-11-14	2001-11-12	1	Issue: AE start date is after end date.

B. New AE data:

Reported Term for the Adverse Event	Dictionary-Derived Term	Start Date/Time of Adverse Event	End Date/Time of Adverse Event	Duration of Adverse Event
NAUSEA	Nausea	2001-11-12	2001-11-13	2
POUNDING HEADACHE	Headache	2001-11-12	2001-11-12	1
WOMITING	Vomiting	2001-11-13	2001-11-13	1

C. Code produced by programmer:

```
1  data _null_;
2    set AE_COMMENT end=eof;
3    if _N_ = 1 then do;
4      call execute('data AE;');
5      call execute('  set AE;');
6      call execute('  attrib COMMENT label="Comment" length=$200
                           FLAG label="Indicator" length=8;');
7      call execute('  array varlist AETERM AEDECOD AESTDTC AEENDTC AEDUR;');
8    end;
9    call execute('  FLAG=0;');
10   call execute('  do I=1 to dim(varlist);');
11   call execute('    if index("'||cats(of _ALL_) ||'", strip(varlist[iii]))
                          then FLAG=FLAG + 1;');
12   call execute('  end;');
13   call execute('  if FLAG >= dim(varlist) - 1
                       then COMMENT="'|| strip(COMMENT) ||'";');
14   if eof then do;
15     call execute('  drop I FLAG;');
16     call execute('run;');
17   end;
18 run;
```

D. Code produced by CALL EXECUTE:

```
1  data AE;
2    set AE;
3    attrib COMMENT label="Comment" length=$200 FLAG label="Indicator" length=8;
4    array varlist AETERM AEDECOD AESTDTC AEENDTC AEDUR;
5    FLAG=0;
6    do I=1 to dim(varlist);
7      if index("POUNDING HEADACHEHeadache2001-11-142001-11-121Issue: AE start date is
                after end date.", strip(varlist[iii]))
8      then FLAG=FLAG + 1;
9    end;
10   if FLAG >= dim(varlist) - 1 then COMMENT="Issue: AE start date is after end date.";
11   drop I FLAG;
12 run;
```

E. Final dataset after merging:

Reported Term for the Adverse Event	Dictionary-Derived Term	Start Date/Time of Adverse Event	End Date/Time of Adverse Event	Duration of Adverse Event	Comment
NAUSEA	Nausea	2001-11-12	2001-11-13	2	
POUNDING HEADACHE	Headache	2001-11-12	2001-11-12	1	Issue: AE start date is after end date.
WOMITING	Vomiting	2001-11-13	2001-11-13	1	

Display 4. Merge Comments Back to Corresponding Data Records

Again, CALL EXCUTE could provide a solution. Basically, the idea is to examine, in the new AE data set, how well the values of the variables, which were used for the previous export, match the concatenation of all variable values from the comment file. In this case, those variables for export are AETERM, AEDECOD, AESTDTC, AEENDTC and AEDUR. With the comment data set as the control, the automatic variable _ALL_ can be used to concatenate values from the same row to make the code more flexible. The INDEX function examines if the value of each variable that is represented by the array VARLIST from the new AE data set can be found in the comment data set:

```
11 call execute('     if index("'||cats(of _ALL_) ||'", strip(varlist[iii]))
                      then FLAG=FLAG + 1;');
```

If yes, the counter flag will be incremented by one. Since a mismatch could occur due to data update, the threshold to determine row-to-row match (highlighted in green) is set as the total number of variables compared minus 1. If the final counter flag value reaches the threshold, CALL EXECUTE transfers the value of COMMENT from the control data set to the new AE data set:

```
13 call execute('     if FLAG >= dim(varlist) - 1
                        then COMMENT="'|| strip(COMMENT) ||'";');
```

The subsequent code produced by CALL EXECUTE is shown in Display 4D. Similarly, each internal loop of the control data set results in one DO – END block. Based on the color, the subsequent code can be easily traced back to the "parental" code. Display 4E indicates that the comment is placed at the right location and the error found previously has been corrected.

In the real world, it may be necessary to trace hundreds of comments back to thousands of records and there may be much more complicated situations such as missing values. But this strategy will still work and save time and effort.

CAVEATS WHEN DEALING WITH CALL EXECUTE

CALL EXECUTE is a powerful tool that can reduce work. However, it functions in a much more intricate way than other SAS data step statements. To maximize the benefit and minimize the risk, several things must be kept in mind:

1. There are warning messages in a lot of places including the SAS website, showing that macro variables that are created by CALL SYMPUT (or CALL SYMPUTX, INTO of PROC SQL) inside a macro cannot be resolved in the same macro if this macro is invoked by CALL EXECUTE. As stated from the SAS website, the macro mask function %NRSTR can help solve this issue. The reason behind it has been explained in the early part of this paper: when the execution module is working, all SAS statements, including CALL SYMPUT, CALL SYMPUTX

and INTO of PRROC SQL, can do nothing but wait in the input stack or the word scanner while macro expressions are free to get into the macro processor for resolution or execution.

2. As mentioned earlier, both single and double quotation marks work fine when the argument is only relevant to data step statements. However, when the argument is a macro expression, double quotation marks could prompt the argument to switch to the macro processor but not the compiler when the "parental" code is being tokenized. The consequence is that the macro in the argument is executed much earlier than CALL EXECUTE is effective. To avoid this unexpected situation, %NRSTR, likewise, can be used to prevent the early involvement of the macro processor. Additionally, that does not mean that we stay away from double quotation if possible. In real programming practice, if we want the values pulled from PDV by CALL EXECUTE work as character string instead of parts of SAS statement in next steps, we still need to use double quotation marks (e.g. Display 4C, row 13).

3. If the CALL EXECUTE argument is very long, a warning message may appear in SAS log, saying "WARNING: The quoted string currently being processed has become more than 262 characters long. You might have unbalanced quotation marks." To mute it, the system option NOQUOTEMAXLEN should be set before the program.

4. CALL EXECUTE may need more system resource. For example, in Display 4, to correctly locate each comment row, each observation from the control data set must interact with every single row of the new AE data set. If both data sets are large, it may take a long time to run. In this case, manually adjusting all variable attributes and then doing data step merging may be an alternative option if compute resources are very limited.

5. In the real world, clinical data is complicated and so multiple rounds of trial and error may be necessary for successful implementation of CALL EXECUTE.

CONCLUSION

Based on the scenarios discussed above, CALL EXECUTE is quite useful for data-driven programming. With this call routine, a correctly defined-control file enables SAS programs to be more adaptive and less costly for maintenance. In the long run, it will make programmers work more efficiently.

REFERENCES

SAS(R) 9.3 Macro Language: Reference, http://support.sas.com/documentation/

Vinod Kaippillil Mukundaprasad, Can I CALL EXECUTE here, www.phusewiki.org/docs/2009%20PAPERS/CC01.pdf

Artur Usov, Call Execute: Let Your Program Run Your Macro, www.lexjansen.com/nesug/nesug96/NESUG96029.pdf

H. Ian Whitlock, CALL EXECUTE: How and Why, www.lexjansen.com/pharmasug/2008/po/PO11.pdf

Bob Virgile, MAGIC WITH CALL EXECUTE, www2.sas.com/proceedings/sugi22/.../PAPER86.PDF

Robert Musterer and Owen Jiang, Use of the call execute routine to pass a list of variables from a data step to a proc step, http://www.lexjansen.com/nesug/nesug95/NESUG95103.pdf

Daniel Boisvert and Shafi Chowdhury, CALL EXECUTE for everyone! Examples for programmer, statistician and data manager, http://www.lexjansen.com/pharmasug/2006/TechnicalTechniques/TT05.pdf

Russell Lavery, An Animated Guide: The Map of the SAS Macro Facility, http://www.lexjansen.com/nesug/nesug02/bt/bt005.pdf

ACKNOWLEDGEMENTS

The author really appreciates Arthur Collins, Sanjiv Ramalingam and Cecilia Mauldin reviewing the manuscript and providing valuable comments.

CONTACT INFORMATION

Any comments or questions are highly encouraged. Please contact the author at:

Name: Hui Wang
Enterprise: Biogen
Address: 300 Binney Street
Cambridge, MA 02142
Email: hui.wang@biogen.com
bio.email@gmail.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 registered trademarks or trademarks of their respective companies.