PharmaSUG China 2024 - Paper CC-10049

Data-Driven Programming With CALL EXECUTE

Lin Jiang, BeiGene

ABSTRACT

The CALL EXECUTE routine in SAS is a powerful tool that allows for the dynamic generation and execution of SAS code within a DATA step. It is commonly used when programmers need to develop dynamic data-driven SAS applications: key information can be stored in a metadata dataset, making it easy to call and review; the data steps and processes can be predesigned in one or several macros, preparing for reuse. CALL EXECUTE functions as a highly efficient assembly line, grabbing metadata observations and sending them to the appropriate branches. All we need to do is prepare the proper metadata and processing steps, and then the data can drive the flow. However, warnings and errors can disrupt the assembly line, potentially producing unexpected outputs. In this paper, we will introduce the syntax of CALL EXECUTE, explains how it works with several scenarios, demonstrate some pitfalls and how to trouble shooting them.

INTRODUCTION

To enable interaction between different portions of the SAS language and the macro facility during execution, SAS provides macro facility interfaces. One of the DATA step interfaces that allows a program to interact with the macro facility during DATA step execution is CALL EXECUTE. As described in the SAS 9.4 Macro Language Reference, Fifth Edition, CALL EXECUTE "resolves its argument and executes the resolved value at the next step boundary (if the value is a SAS statement) or immediately (if the value is a macro language element)." It is perfectly suitable for data-driven programming.

When you want to generate data steps and processes based on a dataset, execute a macro conditionally, or call a macro with several parameters but different values repeatedly, think about using CALL EXECUTE first. However, you may encounter warnings and errors when using this routine, or even produce unexpected outputs. To ensure it works correctly, it is essential to understand the process of CALL EXECUTE and improve your troubleshooting skills.

SYNTAX OF CALL EXECUTE

The syntax of CALL EXECUTE is simple:

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

It is a DATA step call routine. Only one argument is required, and it will be resolved and then issued for execution at the next step boundary. This argument could be:

- A character string enclosed in quotation marks.
- The name of a DATA step character variable whose value is a string.
- Or their concentrations.

HOW CALL EXECUTE WORKS

To understand the working process of SAS completely, it is recommended to read the paper by Russell Lavery, Saad Anbari, and Musa Nsereko. It covers the map of the SAS system (not just the Macro Facility) and provides a clear mental framework for SAS macro processing. In this paper, we will highlight the process of CALL EXECUTE, and show some scenarios of using CALL EXECUTE.

Generally, during the execution of data step, CALL EXECUTE resolves the argument to text by PDV (Program Data Vector), then simply writes the text (the value resolved from argument) to the Input Stack, and the text will be executed at the next step boundary, as shown in Figure 1.

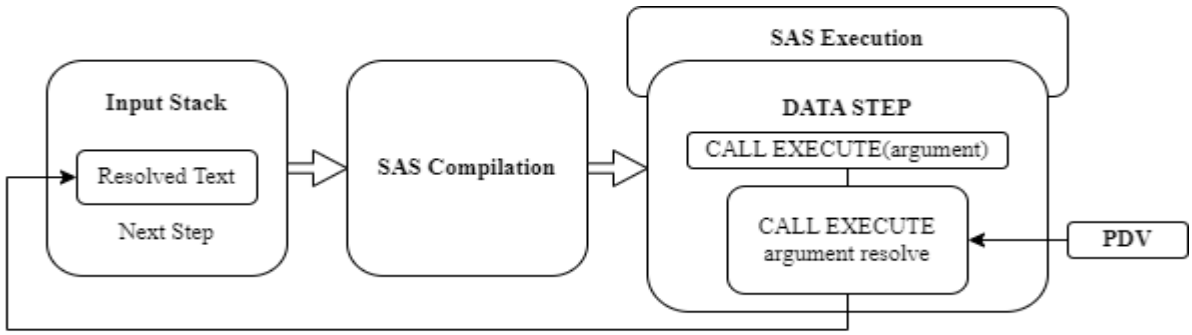

Figure 1

The reality is more complicated. Even the quotation marks and the exact value resolved from the argument will affect the working process. Below we will discuss how CALL EXECUTE works in different scenarios with examples.

SCENARIO 1 – REGULAR SAS STATEMENTS

The simplest scenario is to generate regular SAS statement through CALL EXECUTE. These SAS statements created by CALL EXECUTE will be sent to the Input Stack, waiting in the queue to start execution at the next step boundary. Please specify CALL EXECUTE argument correctly so that it can resolve to the desired SAS statement.

Let's try to solve the following task: copy datasets whose name containing "class" in library SASHELP to WORK. We add flags to put messages in log, which can show the sequence of execution clearly.

```
proc sql noprint;
    create table sample as
    select strip(memname) as memname
    from dictionary.tables
    where index(lowcase(memname), 'class') and upcase(libname)='SASHELP'
    ;
quit;

data _null_;
    set sample;
    call execute('data '||memname||";");
    call execute('set sashelp.'||memname||';');
  ❶ call execute('if _n_=1 then put "EXECUTE: create dataset
'||upcase(memname)||'";'); /*Flag for data step created by CALL EXECUTE*/
    call execute('run;');
  ❷ put "PDV" _n_; /*Flag for PDV*/
run;
```

Example 1

A dataset SAMPLE is created and contains 2 observations:

obs	MEMNAME
1	class
2	classfit

Dataset 1

The statements for potential log lines are marked with numbers in the sequence of programming lines. They will be print to log in the following sequence:

```
1 PDV1 ❷
2 PDV2 ❷
```

❶

```
3 EXECUTE: create dataset CLASS
4 EXECUTE: create dataset CLASSFIT ❶
```

Log 1

Let's call the DATA step containing CALL EXECUTE as Outer DATA Step, and the DATA steps created by CALL EXECUTE the Inner DATA Step. Log line 1 and 2 indicate the execution of statements in the Outer Data Step, and log line 3 and 4 indicate the execution of statements in the Inner Data Steps. When SAS executes the Outer Data Step, CALL EXECUTE statements resolve their arguments to text and insert the text into the Input Stack, which are the Inner Data Steps below.

```
data CLASS;
    set sashelp.CLASS;
    if _n_=1 then put "EXECUTE: create dataset CLASS"; ❶
run;

data CLASSFIT;
    set sashelp.CLASSFIT;
    if _n_=1 then put "EXECUTE: create dataset CLASSFIT"; ❶
run;
```

Inner Data Steps wait in the Input Stack until the current SAS execution (Outer Data Step) finishes, then continue the process of compilation and execution.

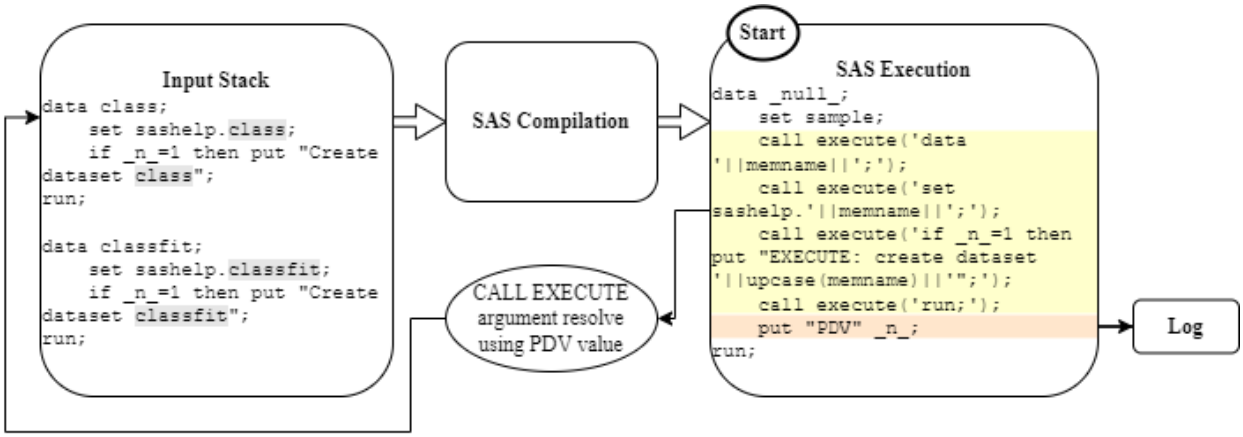

Figure 2

SCENARIO 2 – MACRO STATEMENT AND MACRO VARIABLE

Macros and macro variables make it more flexible to generate SAS code. They can also be used as arguments in CALL EXECUTE, but they guide the execution differently compared to regular SAS statements. Let's see the code below with two more flags for macro statement and macro variables compared to the code in scenario 1. We just add two CALL EXECUTE routines using arguments with the macro statement `%put` and one of them contains a macro variable.

```
%let name=%str(Macro Statement put);
%let class = %str(MVAR class);
%let classfit = %str(MVAR classfit);

data _null_;
    set sample;
    call execute('data '||memname||';');
    call execute('set sashelp.'||memname||';');
  ❶ call execute('if _n_=1 then put "EXECUTE: create dataset
'||upcase(memname)||'";'); /*Flag for data step created by CALL EXECUTE*/
```

```
  ❷ call execute('%put EXECUTE: macro statement of observation
'||cat(_n_)||';'); /*Flag for macro statement*/
    call execute('run;');
  ❸ put "PDV" _n_; /*Flag for PDV*/
  ❹ call execute('%put EXECUTE: &name. &'||memname||';');
run;
```

Example 2

The flags will be print to log in below sequence:

```
1 EXECUTE: macro statement of observation 1 ❷
2 PDV1
3 EXECUTE: Macro Statement put MVAR class ❹
4 EXECUTE: macro statement of observation 2 ❷
5 PDV2 ❸
6 EXECUTE: Macro Statement put MVAR classfit ❹
7 EXECUTE: create dataset CLASS ❶
8 EXECUTE: create dataset CLASSFIT ❶
```

Log 2

As we can see in log lines, CALL EXECUTE statement ❷ and ❹ create macro statements with `%put`, and the flags are written to log immediately during the execution of Outer Data Step. Below is our explanation:

The processor for macro facilities is different from regular SAS statements. To distinguish them, we call the modules for compilation and execution of regular SAS statements the SAS compilation module and SAS execution module, respectively. For macro facilities, we call these the macro compilation module and macro execution module. When CALL EXECUTE generates a regular SAS statement, it will be sent to Input Stack, waiting for further compilation and execution. This further process, in our example, is the Inner DATA Step, which will not start until the Outer DATA Step finishes its execution, as the SAS execution module is occupied by Outer DATA Step. When CALL EXECUTE generates a macro statement, it will be sent to macro processor and start its journey from macro compilation module to macro execution module immediately, regardless of what is happening in SAS execution module.

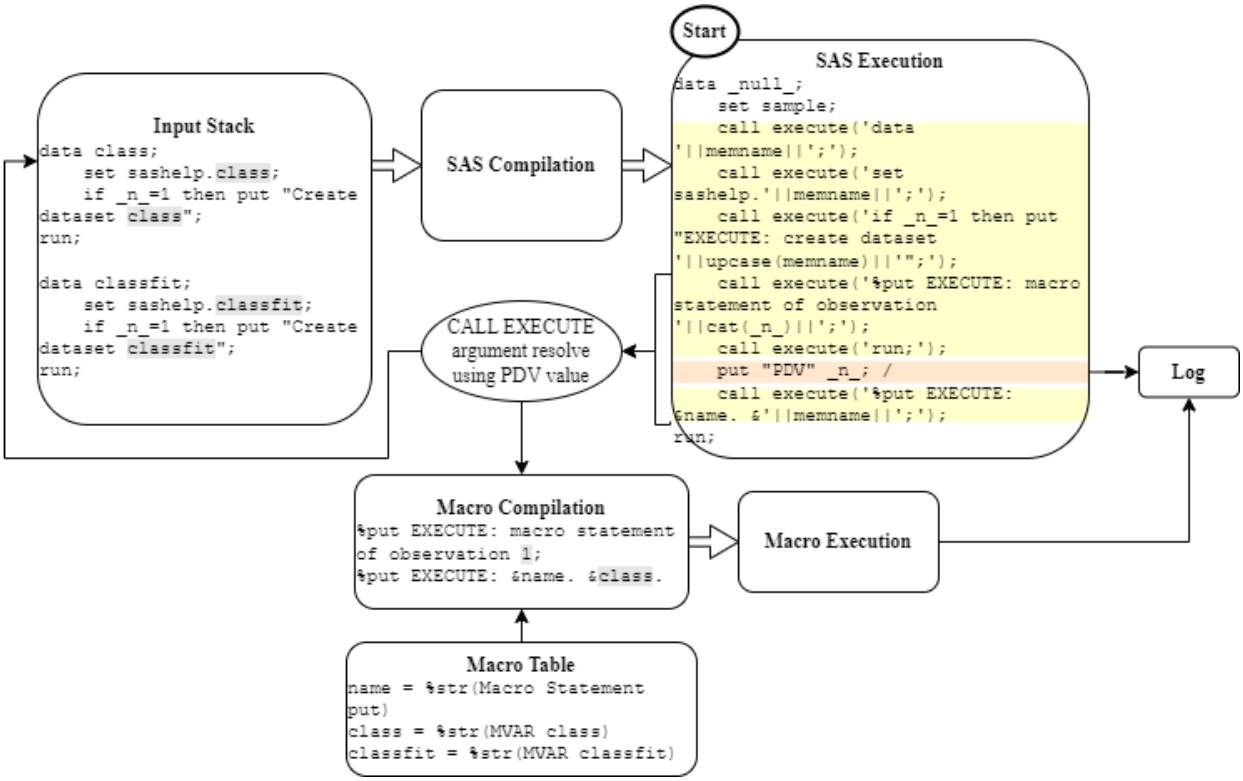

Figure 3

SCENARIO 3 – MACRO

Macros enable you to substitute text in a program and to do many other things. A SAS program can contain any number of macros, and you can invoke a macro any number of times in a single program. When there is requirement of call a macro multiple times, my preferred practice is to prepare the metadata of macro parameter values first and use CALL EXECUTE to pass PDV values to the macro. There are also many points to take care of. Let's create a sample macro `%flag` and call it using CALL EXECUTE.

```
%macro flag(ds=);
    data _null_;
        set sashelp.&ds.(obs=1);
      ❶put "Macro: data step";
    run;
    proc sql noprint;
        select count(*) into: num trimmed
        from sashelp.&ds.
        ;
    quit;
  ❷ %put Macro: macro statement;
  ❸ %put Macro: Number of obs of dataset &ds. is &num.;
%mend flag;

data _null_;
    set sample;
    call execute('%flag(ds='||memname||');');
  ❹ put "PDV" _n_;
run;
```

Example 3

The flags will be print to log in below sequence, and warnings appear in log:

```
1 Macro: macro statement ❷
2 WARNING: Apparent symbolic reference NUM not resolved.
3 Number of obs of dataset CLASS is &num.❸
4 PDV1❹
5 Macro: macro statement ❷
6 WARNING: Apparent symbolic reference NUM not resolved.
7 Number of obs of dataset CLASS is &num. ❸
8 PDV2❹
9 Macro data step ❶
10 Macro data step ❶
```

Log 3

This feature of CALL EXECUTE is mentioned in SAS 9.4 Macro Language Reference, Fifth Edition: If an EXECUTE routine argument is a macro invocation or resolves to one, the macro executes immediately. Execution of SAS statements generated by the execution of the macro will be delayed until after a step boundary. SAS macro statements, including macro variable references, will execute immediately.

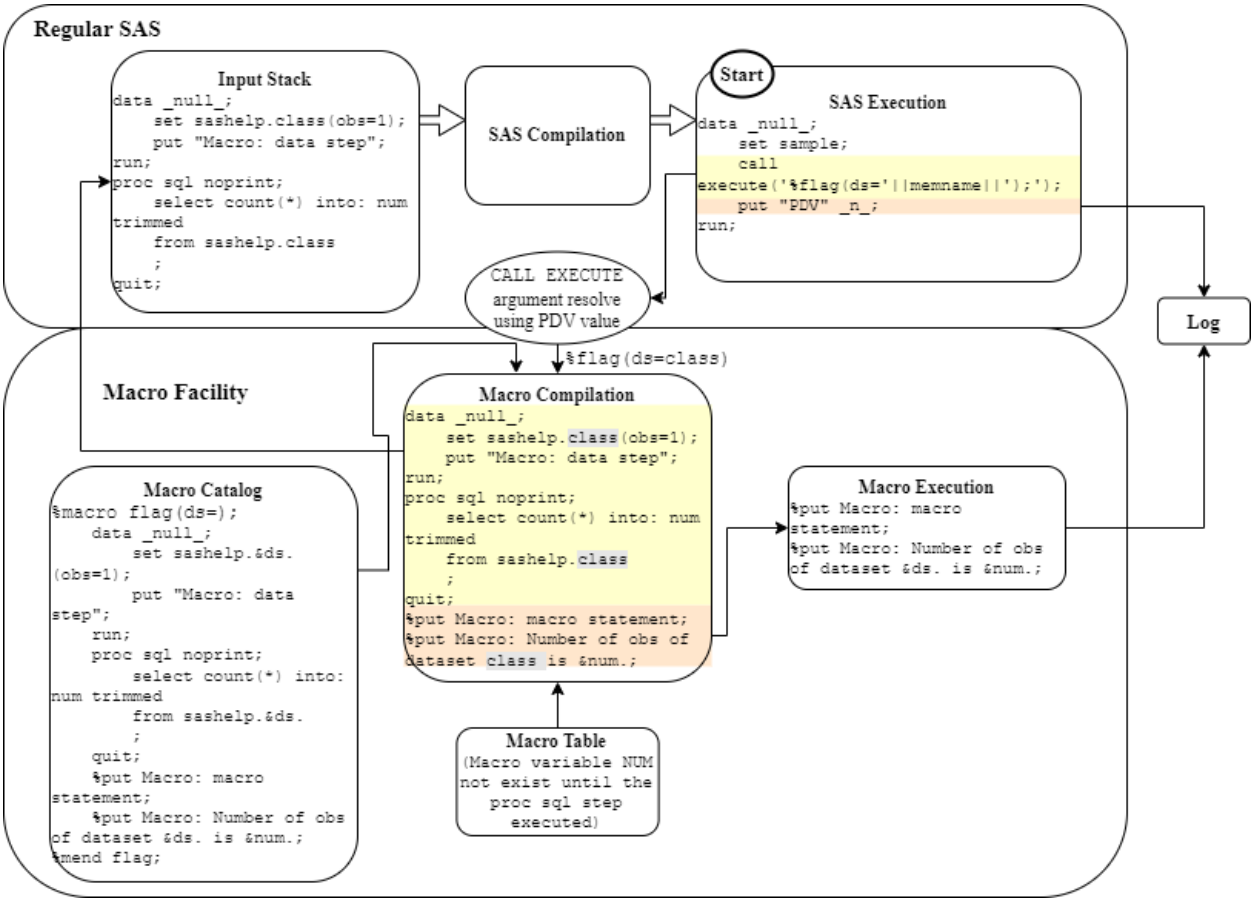

Figure 4

In practice, we often generate complex macro that contains macro logic and regular SAS statements. What can we do to just let CALL EXECUTE pass PDV values to macro parameter and let the macro executes in a whole? Use `%nrstr`! It can delay the macro processes, letting them execute after the Outer Data Step finish its execution. Let's run below code and check out what will happen in the log:

```
data _null_;
    set sample;
    call execute('%nrstr(%flag(ds='||memname||'));');
```

```
  ❹ put "PDV" _n_;
run;
```

Example 4

Warning disappeared! Macro FLAG compiled and executed in the order we want, and the flags will be print to log in below sequence:

```
1 PDV1 ❹
2 PDV2 ❹
3 Macro: data step ❶
4 Macro: macro statement ❷
5 Macro: Number of obs of dataset CLASS is 19
6 Macro: data step ❶
7 Macro: macro statement ❷
8 Macro: Number of obs of dataset CLASSFIT is 19 ❸
```

Log 4

You should be aware of the pair of parentheses. One practice is just including the macro name in `%nrstr`, like below, but sometimes it may cause different output, it depends on your macro. An example is provided in Pitfall 3.

```
      call execute('%nrstr(%flag)(ds='||memname||');');
```

PITFALLS OF CALL EXECUTE AND TROUBLESHOOTING

Even though CALL EXECUTE is a powerful tool, programmers need to be very careful while using it. In this section, we demonstrate some pitfalls of CALL EXECUTE and how to troubleshooting them.

PITFALL 1 – EFFECT OF QUOTATION MARKS

Quotation marks also affect the execute process when the argument contains macro/macro statement/macro variable. Below code and log will explain what will happen.

```
%let name=%str(Macro Statement put);
%let class = %str(MVAR class);
%let classfit = %str(MVAR classfit);
data _null_;
    set sample;
  ❶ call execute('%put &name. &'||memname||';');
  ❷ call execute("%put &name. &"||memname||';');
run;
```

Example 5

If we comment line ❷, the program can run successfully and output lines to log; but it encounters an ERROR with line ❷:

```
1 Macro Statement put &"||memname||';') ❷
2 ERROR 79-322: Expecting a ).
3 ERROR 180-322: Statement is not valid or it is used out of proper order.
```

Log 5

If macro statements in double quotation marks is assigned to the argument of CALL EXECUTE routine, they will be sent to macro processors much earlier, while the Outer DATA step is being constructed (in SAS compilation module), and executes (using macro execution module) immediately.

What will happen if we use both single and double quotation marks in one CALL EXECUTE routine?

```
data _null_;
    set sample;
    call execute(❶'%put &name. in single quote;'||❷"%put &name. in double
quote;");
    call execute(catx(' ',❸'%put &name. in catx single quote;',❹"%put &name.
in catx double quote;"));
  ❺put "PDV" _n_;
run;
```

Example 6

And logs will be:

```
1 Macro Statement put in double quote ❷
2 Macro Statement put in catx double quote ❹
3 Macro Statement put in single quote ❶
4 Macro Statement put in catx single quote ❸
5 PDV1 ❺
6 Macro Statement put in single quote ❶
7 Macro Statement put in catx single quote ❸
8 PDV2 ❺
```

Log 6

Even in the same routine, regardless of the concentrate method (|| or catx function), a macro statement with double quotation marks still resolves while the Outer DATA Step is being constructed and executes immediately. Its execution does not relate to execution of the Outer DATA Step and the number of observations in the dataset, but occurs only once during the construction of Outer DATA Step.

In conclusion, an argument within single quotation marks (without `%nrstr`) resolves during the execution of Outer DATA Step, while an argument within double quotation marks resolves while the Outer DATA Step is being constructed. It is better to avoid using double quotation marks in argument.

PITFALL 2 – INVOKE MACRO VARIABLES CREATED IN SAME DATA STEP

Understanding the timing of CALL EXECUTE can help you generate code and use it more effectively. If you want to invoke a macro variable in a CALL EXECUTE argument, please confirm if it exists and is the exact one you want when CALL EXECUTE is in charge. If you want to use a macro variable created by a CALL EXECUTE routine, please confirm if it exists and is the exact one you want at the time of invoking. One of the bad practices is shown in Scenario 3 Example 3, and you may find more in practice.

PITFALL 3 – SPECIAL SYMBOLS IN VARIABLE VALUE AND USING OF %NRSTR

There might be special symbols in variable value that you want to pass to a CALL EXECUTE argument. Below, we create a variable with percent sign and a macro that simply puts the parameter to log:

```
data pctmark;
    length pctmark $200.;
    set sample;
    pctmark = strip(memname)||'(%)';
run;
```

obs	MEMNAME	PCTMARK
1	class	class(%)
2	classfit	classfit(%)

Dataset 2

```
%macro indicator(pctmark=);
    %put &pctmark.;
%mend indicator;
```

```
data _null_;
    set pctmark;
  ❶ call execute('%indicator(pctmark='||pctmark||');');
  ❷ call execute('%nrstr(%indicator)(pctmark='||pctmark||');');
  ❸ call execute('%nrstr(%indicator(pctmark='||pctmark||'));');
run;
```

Example 7

Line ❶ and ❷ can output the value of variable PCTMARK to log successfully, but line ❸ encounters to an ERROR:

```
ERROR: Expected close parenthesis after macro function invocation not
found.
```

Log 7

Check related parts in log, we can see line ❸ resolved to below text, which the percent mark is recognized as a mask to right parenthesis, and cause the error of expected close parenthesis lost.

```
%indicator(pctmark=CLASS()
        !+           ));
```

Log 8

This situation occurs because of the features of macro quoting, which is another important part of macro facility.

When encountering errors, warnings, or unexpected outputs, what can we do to locate the cause? You can follow the steps below:

1. Avoid using double quotation marks, as they may cause macro execution during the compilation of the Outer DATA Step.
2. Check if its argument can be resolved to the text you expect. It is more intuitive to save the argument as another character variable in dataset. Then, you can review the potential text that will call execute.
3. Check if the parentheses are paired correctly. There might be several parts in the concentration of the argument, and forgetting to close parentheses is common. Again, it is more intuitive to save the argument as another character variable in dataset.
4. If the macro invoked by CALL EXECUTE contains any macro facility, do not forget `%nrstr`.
5. A great option for troubleshooting is the system option SOURCE. It controls whether the code passed to the CALL EXECUTE routine is written to the SAS log. When SOURCE is in effect, the code is written to the log, and you can review the code for execution.

```
proc options option=source; run; /* Print current setting */
options source; /* Write source to the SAS log */
```

CONCLUSION

In this paper, we explored the intricacies of using CALL EXECUTE in SAS programming, focusing on its capabilities, nuances, and best practices. CALL EXECUTE serves as a powerful tool for generating and executing SAS code dynamically, particularly useful in scenarios requiring repetitive or conditional code generation.

Throughout our discussion, we emphasized several key points:

- Understanding Execution Timing: The timing of when arguments are resolved—whether during the construction of the Outer DATA Step or its execution—significantly impacts how CALL EXECUTE operates. It's crucial to avoid using double quotation marks to prevent unintended macro execution during compilation.

- Best Practices for Argument Handling: Saving arguments as character variables in datasets allows for intuitive review and verification of the generated code before execution. Checking for correctly paired parentheses and ensuring the use of `%nrstr` when invoking macros with macro facilities are essential for avoiding errors and unexpected outputs.
- Troubleshooting Tools: Leveraging the system option SOURCE provides valuable insights by logging the code passed to CALL EXECUTE, aiding in diagnosing issues and refining code execution.

In conclusion, mastering CALL EXECUTE empowers SAS programmers to streamline processes, automate tasks, and enhance code flexibility. By adhering to best practices and understanding its operational subtleties, practitioners can harness its full potential to meet diverse programming challenges effectively.

REFERENCES

SAS Institute Inc. 2016. *SAS® 9.4 Macro Language: Reference, Fifth Edition*. Cary, NC: SAS Institute Inc.

Russell Lavery, Saad Anbari , Musa Nsereko, 2008, “An Animated Guide: The Map of the SAS Macro Facility”, http://www.lexjansen.com/nesug/nesug02/bt/bt005.pdf

Hui Wang, 2015, “Creating Data-Driven SAS® Code with CALL EXECUTE”, https://www.pharmasug.org/proceedings/2015/BB/PharmaSUG-2015-BB15.pdf

Kirk Paul Lafler, 2019, “Introduction to Data-driven Programming Using SAS®”, https://support.sas.com/resources/papers/proceedings19/3186-2019.pdf

ACKNOWLEDGMENTS

I would like to express my very great appreciation to Yan Qiao for her valuable and constructive suggestions during the planning and generating of this paper.

RECOMMENDED READING

- *SAS® 9.4 Macro Language: Reference, Fifth Edition*
- *Russell Lavery, Saad Anbari , Musa Nsereko, 2008, “An Animated Guide: The Map of the SAS Macro Facility”, http://www.lexjansen.com/nesug/nesug02/bt/bt005.pdf*

CONTACT INFORMATION

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

Lin Jiang
BeiGene, Inc.
13051616639
lin.jiang@beigene.com