Paper 4679-2020

Using SAS® Macro Variable Lists to Create Dynamic Data-Driven Programs

Joshua M. Horstman, Nested Loop Consulting

ABSTRACT

The SAS® macro facility is an amazing tool for creating dynamic, flexible, reusable programs that can automatically adapt to change. In this hands-on workshop, you'll learn how to create and use macro variable lists, a simple but powerful mechanism for creating data-driven programming logic. Don't hardcode data values into your programs. Eliminate data dependencies forever and let the macro facility write your SAS code for you!

INTRODUCTION

This paper gives an overview of a programming construct known as a macro variable list. Macro variable lists are powerful tools that can be used to eliminate hard-coded data dependencies and build dynamic logic controlled by the data or the computing environment. Macro variable lists are not something pre-defined by SAS, but rather a specific way of utilizing certain features of the SAS macro facility.

We'll start with a review of some macro language fundamentals, including how to define and use macro variables. Next, we'll discuss the concept of a macro variable list, including how to create and use one. Finally, we'll look at several examples that demonstrate how macro variable lists can make programs more flexible and adaptable by dynamically generating SAS code based on the data.

MACRO LANGUAGE BASICS

At its core, the SAS macro language is a tool for generating SAS code. It has some syntactical similarities with the base SAS programming language, but it's actually an entirely separate language. While a complete tutorial on the macro language is beyond the scope of this paper, we will review some fundamental concepts regarding macro processing and macro variables. Slaughter and Delwiche (2004) provide a more comprehensive tutorial for those new to the macro language, while Carpenter (2016) has produced the most exhaustive and definitive reference on the topic.

MACRO PROCESSING

We begin with a quick overview of macro processing. What follows is a gross simplification of the process but will provide some necessary context for the material in this paper. For a more through treatment of this process, the reader is referred to Li (2010) or Lyons (2004).

When a program is submitted, the SAS word scanner parses the statements into tokens. These tokens are sent to the compiler for syntax checking and execution. Execution does not occur until a step boundary is reached (typically the end of a DATA or PROC step).

If the word scanner detects a macro language trigger (typically & or %), it routes any macro language elements to the macro processor for handling. This includes executing macro statements and resolving references to macro variables.

The output from the macro processor must be run back through the parser once more to check for additional macro language elements that may have been generated during the

process. If none are detected, then the tokens, which now hopefully consist of valid SAS code, are passed along to the compiler as before.

CREATING MACRO VARIABLES USING %LET

There are multiple ways to create macro variables in SAS. Carpenter (2017) catalogs several. One of the most common methods is using the %LET macro statement.

```
%let output_path = C:\temp;
```

This statement simply takes the character string "C:\temp" (without the quotation marks) and assigns it to a macro variable named OUTPUT_PATH. Any subsequent references to the OUTPUT_PATH macro variable will be replaced with the corresponding value by the macro processor before the code is compiled and executed. In the following statement, the OUTPUT_PATH macro variable is referenced by preceding the name of the macro variable with a single ampersand:

```
filename myfile "&output_path\myfile.txt";
```

The macro processor will resolve the macro variable reference by substituting the assigned value. The following statement is what will be sent to the SAS engine for compilation and execution:

```
filename myfile "C:\temp\myfile.txt";
```

This approach works well when the value to be assigned is known in advance and can be hard-coded within the program. In the above example, a file path is assigned to a macro variable and can be referenced repeatedly throughout the program wherever that path is used. Should the location change in the future, the program need be edited in only one place.

LIMITATIONS OF %LET

One limitation of creating macro variables using the %LET macro statement is that the value of the macro variable is assigned by the macro processor before any SAS code is executed, or even compiled. For example, the following code will not produce the desired result.

```
data _null_;
  set sashelp.class;
  where name='Alfred';
  %let alfred_age = age;   /* This will not have the desired result. */
run;
```

When this code is submitted, the macro processor will step in to take care of any macro language code. In this case, the only macro language element is the %LET statement. Everything else is ordinary SAS code, which is not handled by the macro processor. The macro processor will assign a value of "age" (literally, the string consisting of those three letters) to a macro variable called ALFRED_AGE. The remaining code that will be executed by SAS looks like this:

```
data _null_;
  set sashelp.class;
  where name='Alfred';
run;
```

If the goal is to store the value of the variable AGE associated with the record for Alfred from the SASHELP.CLASS data set into a macro variable, this code will not accomplish that task. The %LET statement is processed before the DATA step begins executing, at which point the data set has not even been read.

WORKING WITH MACRO VARIABLES AT EXECUTION TIME

Since our goal in this paper is to use macro variable lists to create dynamic programs driven by our data, we need a way to transfer values from data sets into macro variables. We need to create macro variables at execution time, long after any %LET statements will have already been handled by the macro processor. Two methods for doing this are by calling the SYMPUTX routine in the DATA step and by using the INTO clause in the SQL procedure.

The SYMPUTX routine provides a way for us to assign values to macro variables during DATA step execution. Since this assignment takes place at execution time, we have access to values which have been read from data sets and placed in the program data vector. We simply provide a macro variable name and the value to be assigned as arguments to the SYMPUTX routine:

```
data _null_;
  set sashelp.class;
  where name='Alfred';
  call symputx("alfred_age",age);
run;
```

After the DATA step above has finished executing, there will be a macro variable called ALFRED_AGE containing a value of 14, which comes from the SASHELP.CLASS data set variable called AGE. Naturally, if this macro variable already exists, its value will be overwritten.

The SQL procedure provides an alternative means of achieving the same result using its INTO clause. The INTO clause is a SAS-specific feature of PROC SQL that is not part of the industry-wide ANSI SQL standard used broadly across various data processing systems.

```
proc sql noprint;
  select age into :alfred_age
    from sashelp.class
    where name='Alfred';
quit;
```

Just like the DATA step above using CALL SYMPUTX, this procedure call will create a macro variable called ALFRED_AGE which will contain a value of 14. It will also send the query results to any open output destinations unless the NOPRINT option is specified on the PROC SQL statement. Note also the syntax requires a colon preceding the macro variable name into which the query result will be placed.

CREATING MACRO VARIABLE LISTS

Now that we've reviewed the basics of creating macro variables, we turn our attention to the creation of macro variable lists. Macro variable lists are a powerful tool for creating programs with logic that dynamically adapts based on some values in our data. We will see later how macro variable lists enable us to use the macro language to generate SAS code that depends on our data.

HORIZONTAL VS VERTICAL MACRO VARIABLE LISTS

There are two different types of macro variable lists, which we will refer to as "horizontal" and "vertical". Although the syntax and some of the details are different, both types of

macro variable lists provide similar functionality. Selecting which type to use may depend on the particular situation. In many cases, either type can be readily used and the decision comes down to the convenience and preference of the programmer.

A horizontal macro variable list simply refers to a list of values concatenated together, separated by delimiters, and stored in a single macro variable. For example, suppose we wish to store the unique values of the variable ORIGIN from the data set SASHELP.CARS (one of the built-in sample data sets included with SAS). This variable happens to have three unique values: Asia, Europe, and USA. We might create a horizontal macro variable list containing these values like this:

```
%let origin_list = Asia Europe USA;
```

Note that all three values are stored in a single macro variable. Here, we have used the space character as our delimiter, but it is important to select a delimiter that does not appear within any of the data values themselves. If we thought we could potentially have spaces within our data values, we would choose another character, such as the tilde:

```
%let origin_list = Asia~Europe~USA;
```

The other type of list is the vertical macro variable list. It consists of a separate macro variable for each value in the list, but the macro variable names are structured in a way that facilitates looping. Specifically, we use a common prefix followed by a sequential number. Returning to our example from above, we could create a vertical macro variable list in the following manner:

```
%let origin1 = Asia;
%let origin2 = Europe;
%let origin3 = USA;
```

As we discussed earlier, creating a macro variable list (either horizontal or vertical) using the %LET macro statement requires that we hard-code the specific values in our program. We want a macro variable list that is generated dynamically based on the values found in our input data set. In order to accomplish that, the macro variable list must be generated at execution time. For that, we revisit the two methods discussed previously for creating macro variables at execution time − the SYMPUTX routine in the DATA step and the INTO clause in the SQL procedure.

USING THE DATA STEP

Because the DATA step is an implied loop, it lends itself to creating vertical macro variable lists where each list item corresponds to an observation from a data set. In our example above, we only want to add each unique value of ORIGIN to the macro variable list once, so we start by using PROC SORT with the NODUPKEY option to create a data set containing only one observation per unique value of ORIGIN:

```
proc sort data=sashelp.cars out=unique_origins(keep=origin) nodupkey;
  by origin;
run;
```

In a subsequent DATA step, we call the SYMPUTX routine once for each observation. We take advantage of the automatic variable _N_, which provides a convenient way to get a unique sequence number for each macro variable name:

```
data _null_;
  set unique_origins;
```

```
  call symputx(cats('origin',_n_),origin);
run;
```

Upon completion of this DATA step, three macro variables (origin1, origin2, and origin3) will have been defined, each containing one of the unique values of the ORIGIN variable from SASHELP.CARS. Note that this was accomplished without ever referring directly to any of the values within our code. If we later receive a new version of the data set having a different set of unique values, this code would function perfectly with no changes needed.

Since this macro variable list is data-driven, we do not know in advance how many elements it contains. For subsequent use of the list, a count of the number of items in the list is very helpful. This is easily obtained with a slight modification to our DATA step:

```
data _null_;
  set unique_origins end=eof;
  call symputx(cats('origin',_n_),origin);
  if eof then call symputx('numorigins',_n_);
run;
```

The variable EOF will only evaluate to true after the last record has been read from the UNIQUE_ORIGINS data set, which is the last time through the DATA step. At that point, the value of _N_ will correspond with the number of observations in the data set, which is 3. That value will be stored in a standalone macro variable called NUMORIGINS.

We can also create a horizontal macro variable list using the SYMPUTX routine in a DATA step. However, the code is a bit more complex due to the need to concatenate the values of ORIGIN across multiple observations. Only after the last observation has been read are the macro variables ORIGIN_LIST (a horizontal macro variable list) and NUM_ORIGINS created:

```
data _null_;
  set unique_origins end=eof;
  length origin_list $200;
  retain origin_list;
  origin_list = catx('~',origin_list,origin);
  if eof then do;
    call symputx('origin_list',origin_list);
    call symputx('numorigins',_n_);
  end;
run;
```

USING PROC SQL

Both horizontal and vertical macro lists can also be created using PROC SQL. In many ways, it is easier than calling the SYMPUTX routine from within a DATA step. Moreover, using the DISTINCT argument on the SELECT statement makes it simple to add only unique values of a variable to a macro variable list without the need to do any pre-processing such as invoking the SORT procedure as we did above. The following code creates a vertical macro variable list:

```
proc sql noprint;
  select distinct origin into :origin1-
    from sashelp.cars
      order by origin;
quit;
```

Notice we use the ORDER BY clause to order the list items as desired. In addition, we need only specify the name of the first macro variable followed by a dash to indicate that additional query results should be stored in sequentially numbered macro variables (origin2, origin3, etc.). Prior to SAS 9.3, an upper bound was required, although one could specify an arbitrarily large number (e.g. `:origin1-:origin999`) and PROC SQL would create only the macro variables needed.

The SQL procedure also provides a convenient way to obtain a count of the items in the list. Each time a SQL query is performed, the number of resulting rows is stored in an automatic macro variable called SQLOBS. It is a good practice to copy this value to another macro variable for later use as it will be overwritten if another SQL query is performed.

```
proc sql noprint;
  select distinct origin into :origin1-
    from sashelp.cars
      order by origin;
  %let numorigins = &sqlobs;
quit;
```

It may seem contradictory to use the %LET macro statement here after the earlier caveats about the timing of its execution. However, because PROC SQL is a fully interactive procedure, each query is executed as soon as the parser reaches the end of the statement (rather than being held until a step boundary as is the case for DATA step statements). Thus, the SQLOBS macro variable is populated before the %LET macro statement is parsed from the input stack and sent to the macro processor to create the NUMORIGINS macro variable.

Creating horizontal macro variable lists with PROC SQL requires only a slight modification to the code. We specify only a single macro variable name and add the SEPARATED BY clause to specify a delimiter:

```
proc sql noprint;
  select distinct origin into :origin_list separated by '~'
    from sashelp.cars
      order by origin;
  %let numorigins = &sqlobs;
quit;
```

USING MACRO VARIABLE LISTS

We have seen how to create both horizontal and vertical macro variable lists using the SYMPUTX routine in a DATA step or using the INTO clause in PROC SQL. To make effective use of these lists, we must have access to the individual list items. Since macro variable lists are frequently used within the context of a macro loop, we need to know how to access a list element on the basis of an index number.

When accessing macro variable lists, it doesn't matter how they were created. However, the methods for accessing horizontal macro variable lists are different from those for vertical macro variable lists, so we will cover each in turn.

USING HORIZONTAL MACRO VARIABLE LISTS

We have seen that a horizontal macro variable list consists of values concatenated together, separated by delimiters, and stored in a single variable. Fortunately, it is relatively simply to deconstruct such a list using the %SCAN macro function.

The %SCAN macro function allows us to specify a string, an index, and one or more delimiters. It parses the string into separate words using the delimiter(s) specified and

returns the word corresponding with the value of the index. Its functionality is identical to that of the SCAN function in the DATA step except it operates in the macro language. There is no need to place quotation marks around the parameters as one would with the the DATA step SCAN function.

Recall from our example that the macro variable ORIGIN_LIST is a horizontal macro variable list and has the value "Asia~Europe~USA" (without quotation marks). We can reference the individual list items using the %SCAN macro function as follows:

```
%scan(&origin_list,1,~)  → Resolves to: Asia
%scan(&origin_list,2,~)  → Resolves to: Europe
%scan(&origin_list,3,~)  → Resolves to: USA
```

More generally, if we have an index stored in a macro variable I, we could reference the item from the horizontal macro variable list corresponding with that index as follows:

```
%scan(&origin_list,&i,~)
```

This syntax can this be used in conjunction with the NUMORIGINS macro variable we created earlier which contains a count of the number of items in the ORIGIN_LIST macro variable:

```
%do i = 1 %to &numorigins;
      %put Item &i: %scan(&origin_list,&i,~);
%end;
```

Note that the %DO macro loop is only valid inside a macro definition and cannot be used in open code at the time of this writing. Executing the above code will product the output below:

```
Item 1: Asia
Item 2: Europe
Item 3: USA
```

USING VERTICAL MACRO VARIABLE LISTS

Compared with using horizontal macro variable lists, the syntax for using vertical macro variable lists is simpler, although some may find it intimidating because it involves the dreaded double ampersand. Recall from our example that our vertical macro variable list consists of three macro variables: ORIGIN1, ORIGIN2, and ORIGIN3. These macro variables contain the values "Asia", "Europe", and "USA", respectively (again, without quotation marks). We can reference the individual list items directly using their respective macro variable names, each preceded by a single ampersand:

```
&origin1  → Resolves to: Asia
&origin2  → Resolves to: Europe
&origin3  → Resolves to: USA
```

Suppose again we have a macro variable I that contains an index, and we'd like to refer to individual list items using that index. We cannot simply write `&origin&i`. The macro processor will interpret this as two distinct macro variable references, the first being to a nonexistent macro variable called ORIGIN.

Instead, we delay the attempted resolution of this macro variable by doubling the first ampersand: `&&origin&i`. On the first pass of the macro processor, the double ampersand will resolve to a single ampersand and `&i` will resolve to the actual value of the index. For

example if the macro variable I contains a value of 1, `&&origin&i` will resolve to `&origin1`. Since this result is itself another macro variable reference, it is passed back to the macro processor for further resolution. This second pass will resolve to the first value from our vertical macro variable list, which is "Asia" (without quotes). This is illustrated below:

Original expression:	`&&origin&i`	
After first pass:	`&origin1`	(`&&` resolves to `&`, `origin` is just text, `&i` resolves to `1`)
After second pass:	`Asia`	(resolved value of macro variable `origin1`)

As we did above with the horizontal macro variable list, we can combine this syntax with the NUMORIGINS macro variable to loop through the items in our vertical macro variable list:

```
%do i = 1 %to &numorigins;
      %put Item &i: &&origin&i;
%end;
```

The result is the same:

```
Item 1: Asia
Item 2: Europe
Item 3: USA
```

DATA-DRIVEN PROGRAMMING EXAMPLES

We now have the pieces in place to write dynamic programs that are driven by values in our data. We present several examples below to demonstrate the power and versatility of this approach.

EXAMPLE 1: SPLITTING A DATA SET

This example uses a vertical macro variable list to split the SASHELP.CARS data set into a separate data set for each value of the variable ORIGIN. The result will be three separate data sets, CARS_ASIA, CARS_EUROPE, and CARS_USA, but these are generated dynamically without any direct reference to those values.

```
%macro split_data;

  * Create the vertical macro variable list.;

  proc sql noprint;
    select distinct origin into :origin1-
      from sashelp.cars;
    %let numorigins = &sqlobs;
  quit;


  * Loop through each value and generate a data step ;
  * to create the corresponding subset.              ;

  %do i = 1 %to &numorigins;

    data cars_&&origin&i;
      set sashelp.cars;
      where origin = "&&origin&i";
    run;

  %end;
```

```
%mend split_data;

%split_data;
```

EXAMPLE 2: DYNAMIC REPORT CREATION

This example uses a horizontal macro variable list to create a separate plot for each value of the variable STOCK in the SASHELP.STOCKS data set. Each plot is written to a separate output file.

```
%macro graph_stocks;

  * Create the horizontal macro variable list.;

  proc sql noprint;
    select distinct stock into :stock_list separated by '~'
      from sashelp.stocks;
    %let numstocks = &sqlobs;
  quit;

  * Loop through each value and generate a call to SGPLOT ;
  * with ODS statements to output the graph to a PDF.     ;

  %do i = 1 %to &numstocks;

    ods pdf file="%scan(&stock_list,&i,~).pdf";

    proc sgplot data=sashelp.stocks;
      where stock = "%scan(&stock_list,&i,~)";
      highlow x=date high=high low=low;
    run;

    ods pdf close;

  %end;

%mend graph_stocks;

%graph_stocks;
```

EXAMPLE 3: COMPARING MULTIPLE DATA SETS

This example uses a vertical macro variable list to compare all data sets which are common to two specified libraries. The names of the two libraries are specified as macro parameters. Any data sets only existing in one of the two libraries are ignored. By using dictionary tables to get information about the data sets which exist in each library, no data set names need be hard-coded. Refer to Lafler (2010) for more information about dictionary tables.

```
%macro compare_all(lib1,lib2);

  * Create a vertical macro variable list using dictionary ;
  * tables to find data sets common to both libraries.     ;

  proc sql noprint;
    select distinct a.memname into :ds1-
      from dictionary.tables a, dictionary.tables b
```

```
      where a.libname = upcase("&lib1")
        and b.libname = upcase("&lib2")
        and a.memname = b.memname
        and a.memtype = "DATA";
    %let numds = &sqlobs;
  quit;

  * Loop through each data set and generate the PROC COMPARE. ;

  %do i = 1 %to &numds;

    proc compare
      base    = &lib1..&&ds&i
      compare = &lib2..&&ds&i;
    run;

  %end;

%mend compare_all;

%compare_all(mylib1,mylib2);
```

CONCLUSION

A robust program is one which is designed to properly handle various combinations of input. Macro variable lists are one tool that allows the SAS programmer to create more robust programs. These programs avoid hard-coding and instead include dynamic logic that adapts to changes in the input data or the computing environment. Such programs are easier to maintain, have far greater potential for reuse, and thus make the programmer more productive.

REFERENCES

Carpenter, Art. 2016. *Carpenter's Complete Guide to the SAS® Macro Language, Third Edition.* Cary, NC: SAS Institute Inc.

Carpenter, Arthur L. 2017. "Five Ways to Create Macro Variables: A Short Introduction to the Macro Language." Proceedings of the SAS Global Forum 2017 Conference. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings17/1516-2017.pdf.

Lafler, Kirk Paul. 2010. "Exploring DICTIONARY Tables and SASHELP Views." Proceedings of the SAS Global Forum 2010 Conference. Cary, NC: SAS Institute Inc. http://support.sas.com/resources/papers/proceedings10/155-2010.pdf.

Li, Arthur X. 2010. "When Best to Use the %LET Statement, the SYMPUT Routine, or the INTO Clause to Create Macro Variables." Proceedings of the SAS Global Forum 2010 Conference. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings10/028-2010.pdf.

Lyons, Lisa. 2004. "Going Under the Hood: How Does the Macro Processor Really Work?" Proceedings of the NorthEast SAS Users Group (NESUG) 2004 Conference. https://www.lexjansen.com/nesug/nesug04/pm/pm07.pdf.

Slaughter, Susan J. and Lora D. Delwiche. 2004. "SAS Macro Programming for Beginners." Proceedings of the Twenty-Ninth Annual SAS® Users Group International Conference. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/243-29.pdf.

CONTACT INFORMATION

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

Joshua M. Horstman
Nested Loop Consulting
317-721-1009
josh@nestedloopconsulting.com
www.nestedloopconsulting.com