

The How and Why of SAS® Macro Language

Carrie Mariner, Afton-Royal Training & Consulting, Richmond, VA

ABSTRACT

When starting in SAS® Macro Language, it's easy to feel overwhelmed. Understanding what happens behind the scenes makes it easier to code and debug your macro programs. This paper covers the basics of SAS® Language processing, SAS® Macro Language processing, and how the two differ.

INTRODUCTION

Anyone with a syntax guide or an example of a SAS® Macro program can jump right in and start programming. However, without an understanding of what happens when you submit that program, getting it to behave as you wish can be tough! Understanding what happens behind the scenes makes it easier to code and debug your programs. This paper covers the basics of SAS® Language processing, SAS® Macro Language processing, and how the two differ. We will walk through an extended example of the compilation of a short data step:

1. starting with a data step with no Macro Language,
2. then adding a macro variable definition and call, and finally
3. adding Macro Language to encapsulate our code and make the data step itself conditional.

Our stress will be on the details necessary to see the similarities and differences between processing of SAS® Language with and without SAS® Macro Language.

SAS LANGUAGE: NO MACRO

When you submit SAS code which does not contain Macro Language, the entire block of code goes first to the INPUT STACK, where it is stored in memory until it is ready to be used. The WORD SCANNER takes one statement at a time from the INPUT STACK and breaks it into tokens, then sends them to their proper destination (in this case the COMPILER). In the COMPILER, once the entire data step is received, the syntax is checked. Assuming no fatal errors have been found, the compiled code is ready to be executed.

This simple data step will read in a raw data file, check for records that are blank, and write log messages for any blank records and the final record.

```
data _null_;
  infile "C:\Data\SESUG06\myflat.txt" end=lastrec;
  input;
  if _infile_=" " then put "Blank rec#" _n_;
  if lastrec then put "Total records " _n_;
run;
```

We can now step through compile time in some detail, using this data step as our first example.

STEP 1: INPUT STACK

When we submit this entire data step, SAS sends all our submitted code to the INPUT STACK. The code waits there while SAS sends one statement at a time to the WORD SCANNER. Notice that this partly explains the great importance of the semicolon, since SAS relies on the semi-colon to parse the code into sections both at compile time and execution.

STEP 2: WORD SCANNER

The WORD SCANNER takes the first statement of our code and breaks it into three tokens, "data", "_null_" and ";". It then takes each of the three tokens in turn and sends them to the COMPILER, since none of them are macro triggers. The WORD SCANNER then repeats these two actions (tokenizing and sending to COMPILER) until it reaches the end of the step.

STEP 3: COMPILER

The COMPILER receives the tokenized statements from the WORD SCANNER, and holds them until the "run;" arrives. Then the COMPILER looks for syntax errors. Given clean code, the COMPILER then directs SAS to execute the step.

SAS TOKENS

Before we get into the added complexities of Macro Language, we should clarify our understanding of tokens. These are the basic units of our code, as defined by the following categories:

- Number tokens are digits or characters that resolve directly to digits, such as SAS date constants ("27jun06"d);
- Name tokens are regular "words", which can contain letters, digits, and underscores – plus periods only in formats or informat;

- Literal tokens are character strings enclosed in quotes;
- Special tokens are characters other than letters, digits, and underscores – these have special meaning to SAS, such as “+” or “/”; and finally,
- Special Macro tokens are a subset of special tokens. The symbols “&” and “%” are called macro triggers, and these are the only tokens that tell the WORD SCANNER to direct tokens to the MACRO PROCESSOR.

So the type of token the WORD SCANNER finds determines the destination of the token when it leaves the WORD SCANNER. In our example, the tokens were all numbers, names, literals, and special tokens, but none were macro triggers, so all the code was sent on to the COMPILER.

SAS LANGUAGE WITH A MACRO VARIABLE

Now we will add a macro variable definition and a macro variable call to our simple data step. This will make the code more flexible by allowing us to easily change which raw data file we are reading.

```
%let path=C:\Data\SESUG06\myflat.txt;
data _null_;
  infile "&path" end=lastrec;
  input;
  if _infile_=" " then put "Blank rec#" _n_;
  if lastrec then put "Total records " _n_;
run;
```

The basic compile time processing will still be the same in the INPUT STACK and WORD SCANNER, but then we'll see the addition of the MACRO PROCESSOR and how it works to build SAS code.

STEP 1: INPUT STACK

The INPUT STACK works just the same as before, holding all our submitted code. The code waits here while SAS sends one statement at a time to the WORD SCANNER.

STEP 2: WORD SCANNER

The WORD SCANNER takes the beginning of the first statement

```
%let path=C:\Data\SESUG06\myflat.txt;
```

and breaks down the first two tokens, “%” and “let”. Because “%” is a special token called a macro trigger and it is followed by a non-blank token, the WORD SCANNER starts sending tokens to the MACRO PROCESSOR. It continues sending tokens there until the MACRO PROCESSOR completes its work, in this case at the “;”.

STEP 3: MACRO PROCESSOR

The MACRO PROCESSOR receives the tokens of the first statement from the WORD SCANNER. It processes the statement, putting the character string “C:\Data\SESUG06\myflat.txt” into the Macro Symbol Table as the value of “path”. No code is created by the MACRO PROCESSOR by the %let statement, so it simply returns processing to the WORD SCANNER.

BACK TO STEP 2: WORD SCANNER

The WORD SCANNER takes the second statement

```
data _null_;
```

and breaks it into three tokens, “data”, “_null_” and “;”. It takes each of the three tokens in turn and sends them to the COMPILER. It then starts taking the third statement

```
infile "&path" end=lastrec;
```

and breaking it into tokens (“infile”, the double quote, “&”, and “path”). The WORD SCANNER then sends “infile” and the double quote to the COMPILER, and “&” and “path” to the MACRO PROCESSOR, since it recognizes “&” followed by a non-blank token as a macro trigger. Then the WORD SCANNER waits for the MACRO PROCESSOR to finish its work before taking any more code from the INPUT STACK.

BACK TO STEP 3: MACRO PROCESSOR

The MACRO PROCESSOR receives the tokens “&” and “path” from the WORD SCANNER. It looks in the Macro Symbol Table for the macro variable “path” and finds the value “C:\Data\SESUG06\myflat.txt”. That text value is placed *back on the top* of the INPUT STACK, and processing control is passed back to the WORD SCANNER.

BACK TO STEP 2: WORD SCANNER

The WORD SCANNER takes the rest of the third statement from the top of the input stack. It is important that where

```
&path" end=lastrec;
```

used to be, there is now

```
C:\Data\SESUG06\myflat.txt" end=lastrec;
```

due to the replacement of the macro variable call (&path) with the value of the macro variable

(C:\Data\SESUG06\myflat.txt). The WORD SCANNER takes this new code and breaks it into tokens. It then takes the tokens in turn and sends them to the COMPILER, since all the macro triggers are gone. The WORD SCANNER continues taking the remaining four statements one at a time, tokenizing them and sending them to the COMPILER.

SKIP TO STEP 4: COMPILER

The COMPILER receives all the tokenized statements from the WORD SCANNER (they now have no Macro Language elements at all), and holds them until the “run;” arrives. Then it looks for syntax errors. Given clean code, the COMPILER directs SAS to execute the step.

THE KEY TO UNDERSTANDING

Notice that all macro processing takes places between cycles of tokenization and compilation. The Macro Language elements in the original code created portions of SAS® Language statements, which were then placed back on the input stack, so that no Macro Language elements remain when the code finally executes. This will always be true in “open code”, which means outside of macro programs.

DEFINING AND CALLING A MACRO PROGRAM

Finally, we need to know what happens when we submit an actual macro program, which contains both SAS® Language elements and Macro Language elements, including macro logic and macro variables. We modify our previous example data step by wrapping it in a macro definition. This allows us to make the entire data step conditional, so that when we call the macro, the data step won't even compile if the raw data file doesn't exist. Instead, SAS will write a log message for us.

```
%macro checkdata;
%if %sysfunc(fileexist("&path")) %then %do;
data _null_;
  infile "&path" end=lastrec;
  input;
  if _infile_=" " then put "Blank rec#" _n_;
  if lastrec then put "Total records " _n_;
run;
%end;
%else %put "File &path not there";
%mend;

%let path=C:\Data\SESUG06\myflat.txt;
%checkdata
```

When we submit our macro program definition, the code “pre-compiles”, but final compiling won't happen until the macro program is called. During this “pre-compile”, the INPUT STACK and WORD SCANNER behave just as we saw above. When the WORD SCANNER sees the macro trigger “%macro” it turns processing over to the MACRO PROCESSOR, which returns processing to the WORD SCANNER after the macro trigger “%mend”.

While it is in control, the MACRO PROCESSOR “pre-compiles” some Macro Language elements which create macro logic, like “%if”, “%do”, and “%end”. It does not “pre-compile” some Macro Language elements, such as “%sysfunc” and “&path”, which might depend on information to be submitted along with the macro call later. These are simply saved as text in the “pre-compiled” macro definition. Our “pre-compiled” macro definition would look something like this (bold portions would actually be compiled instructions to the processor, while the rest would be just the text as written):

```
%if %sysfunc(fileexist("&path")) %then %do;
data _null_;
  infile "&path" end=lastrec;
  input;
  if _infile_=" " then put "Blank rec#" _n_;
  if lastrec then put "Total records " _n_;
run;
%end;
%else %put "File &path not there";
```

Then, when we submit the macro variable definition and macro call

```
%let path=C:\Data\SESUG06\myflat.txt;
%checkdata
```

the code goes back into the compiling process, and these two statements are compiled. The first statement

```
%let path=C:\Data\SESUG06\myflat.txt;
```

is sent to the MACRO PROCESSOR, and SAS puts the value of “C:\Data\SESUG06\myflat.txt” into the variable “path” in the Macro Symbol Table. The second “statement” isn't really a statement at all (notice it has no semicolon), but is a macro call. This is also sent to the MACRO PROCESSOR, which retrieves the “pre-compiled” macro definition and proceeds to compile the code as before. Thus, the reference to “&path” which was part of the “pre-compiled” macro definition will resolve, since SAS doesn't attempt this until we have defined the macro variable “path” in the %let statement. If SAS tried to fully compile the macro when we submitted the macro definition, it would fail due to this type of unresolvable reference, and we would not be able to utilize macro variables or parameters which changed at the time of the macro call.

CONCLUSION

By examining and understanding the mechanics and timing of macro processing, we are able to make more intelligent decisions when creating and editing SAS code using SAS® Macro Language elements. Knowing when each type of code (SAS® Language, macro variable definitions and calls, macro program definitions, SAS® Macro Language elements) compiles, and how the pieces fit together to create the final compiled code to be executed, allows us to create macro code more quickly with fewer errors. Then, by harnessing the power of SAS® Macro Language, we can make our programs more flexible and more powerful.

REFERENCES

Burlew, Michele M. 2002. *SAS® Macro Programming Made Easy*. Cary, NC: SAS Institute, Inc.
SAS Institute, Inc. 2002. *SAS® 9 Macro Language Reference*. Cary, NC: SAS Institute, Inc.

CONTACT INFORMATION

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

Carrie Mariner
Afton-Royal Training & Consulting
11901 Heathmere Crescent
Midlothian, Virginia 23113
Email: AftonRoyal@comcast.net
Web: www.Afton-Royal.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.