MWSUG 2017 - Paper SA101

# Before You Get Started: A Macro Language Preview in Three Parts

Arthur L. Carpenter, California Occidental Consultants

## OVERALL ABSTRACT

Using the macro language adds a layer of complexity to SAS® programming that many programmers are reluctant to tackle. The macro language is not intuitive, and some of its syntax and logic runs counter to similar operations in the DATA step. This makes the transfer of DATA step and PROC step knowledge difficult when first learning the macro language. So why should one make the effort to learn a complex counterintuitive language?

Before you start to learn the syntax; where to put the semicolon, and how to use the ampersand and percent sign; you need to have a basic understanding of why you want to learn the language in the first place. It will also help if you know a bit about how the language thinks. This overview provides the background that will enable you to understand the way that the macro language operates. This will allow you to avoid some of the common mistakes made by novice macro language programmers.

First things first – before you get started with the learning process, you should understand these basic concepts.

## KEYWORDS

Macro variables, scope, macro timing, %LET statement, symbol tables, %MACRO, %MEND, %LOCAL, %GLOBAL

## INTRODUCTION

There is a lot to learn concerning the SAS macro language. This is true even for those of us who have been using the macro language for years. This means that learning the language requires no small investment in time and energy. So is the reward worth the investment? For myself I would say absolutely! But before you start the learning process, there are some things that can be helpful as you gather knowledge. I believe that there are three basic areas of understanding that can be especially helpful to the learning process. The first provides a motivation – why you want to take the time and trouble to learn. Since a great deal of confusion revolves around timing issues, what happens when, the second area of discussion covers crucial timing relationships between the macro language and the rest of SAS. Finally the third topic revolves around the creation of macro variables and more importantly their assignment to symbol tables.

This is a high level introductory overview of the language, necessarily a number of topics will receive light treatment. I will try to use terminology that is understandable to the new programmer. Necessarily this means that the experienced macro programmer will find a number of areas in which I do not 'tell the whole story'.

## MACRO LANGUAGE PREVIEW: PART 1
## WHAT THE LANGUAGE IS, WHAT IT DOES, AND WHAT IT CAN DO

As complicated as the macro language is to learn, there are very strong reasons for doing so. At its heart the macro language is a code generator. In its simplest uses it can substitute simple bits of code like variable names and the names of data sets that are to be analyzed. In more complex situations it can be used to create entire statements and steps based on information which may even be unavailable to the person writing or even executing the macro. At the time of execution, it can be used to make queries of the SAS environment as well as the operating system, and utilize the gathered information to make informed decisions about how it is to further function and execute.

In the olden days, typewriters (you may remember seeing one in a museum) were used to write text to paper and the modern keyboard mimics the keys of the typewriter. When we write a SAS program, we generally type in the code from our keyboard. Our programs consist of DATA steps and PROC steps (along with the occasional global statement). We know the names of the data sets and variables that are to be included in the code, and we use them to in our DATA steps and PROC steps. The macro language can be used in this typing process. The macro language can serve as our intelligent typist which generates all or portions of our code for us.

Since the macro language is at its heart a code generator it can act as a typewriter[1] which we can control. We can use it to write all or any portions of our code for us. In a simple PROC PRINT step we may wish to print a selected number of lines of the specified SAS data set. The code must name the data set to be printed and any options that are to be applied.

```
proc print data=sashelp.class(obs=10);
   run;
```

In this PROC PRINT step we have named the data set of interest and the number of observations that are to be printed. Clearly if this is our entire program there is little motivation to automate or expand on its capabilities, but let's use it as a metaphor for a much longer more complex program. Perhaps the data set name appears multiple times throughout the program and when the data set name changes each occurrence must be edited separately. Instead the macro language can be used to make these changes for us. It can do so through the use of code substitution techniques, which use macro variables.

Macro variables are stored in memory and are used to hold text values. They can contain a single letter, a word, a list of words, SAS statements, or even complete steps. There are over a dozen ways to create macro variables and the simplest, and probably the most common, is through the use of the %LET statement.

## Code Substitution

Analogous to the assignment statement in the DATA step, the %LET statement is a macro language statement, which is used to assign a text value to a macro variable. The %LET statement, like all macro statements, begins with a percent sign (%) which precedes a keyword, and ends with a semicolon. Macro variables will be discussed in more detail in the third section of the paper, however for now it is only important to know that macro variables store text values in memory during the execution of a SAS program.

```
%let dsn = sashelp.class;
%let cnt = 5;
```

In order to take advantage of macro variables once they are created, we need to be able to call them by name when they are needed. In our SAS programs macro variables are named using an ampersand. Essentially these names are used in our program as symbolic references, dummy values, for the true values of the items which will be substituted later during program execution.

```
proc print data=&dsn(obs=&cnt);
   run;
```

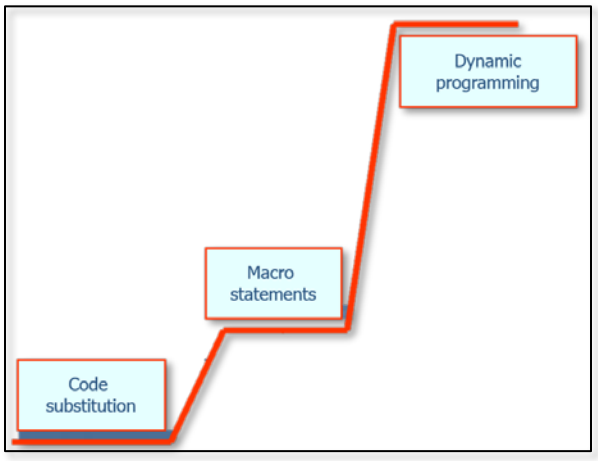
When we write our program using macro variables, we give ourselves us a deeper level of programming flexibility. Imagine for a moment that the simple PROC PRINT shown here is actually a program consisting of a number of steps, perhaps of several hundred lines in length. If the items specified in the macro variables appear throughout the program, we need only change the %LET statements to radically alter what the program will do. Essentially before the PROC PRINT step can execute the macro variables must be resolved. The macro facility, the typewriter, replaces the macro variables with their stored values. Since this code substitution is done before the PROC step is scanned, the macro language is essentially typing code for us.

```
%let dsn = sashelp.class;
%let cnt = 5;

proc print data=&dsn(obs=&cnt);
   run;
```

## Stages of Macro Language Learning

Most programmers who learn the macro language go through three stages or levels of learning. Code substitution, which is mentioned above, is the first, and generally the most straightforward, of the three. Generally even those new to SAS can easily make use of code substitution techniques without encountering any of the macro language issues that can impede the learning process.




The second level makes use of macro statements, macro functions, and macro logic. The learning curve between the first and second learning step is fairly short and not too steep. With practice, and a reasonably good understanding of the basic SAS programming language without the use of macros, most programmers are able to start to use the basic aspects of these elements of the macro language fairly quickly.

The learning curve for the third level, dynamic programming techniques, is much steeper and longer. Often programmers who have gained proficiency in the second level will take months of practice before even starting to become comfortable using the techniques of this third learning stage. Indeed I have known SAS programmers who are quite adept at the macro language that have little or no understanding of the elements of the third level. Actually (true confession time), I programmed in the macro language for years, blissfully unaware that this third level even existed. Then I was confronted by a problem that could only be solved using these techniques, and I was dragged, with much resistance (read kicking, screaming, gnashing of teeth), into this new (to me) world of dynamic programming. However the difference in power and capability between the techniques in the second and third stage make the effort to acquire the necessary knowledge well worth the effort. I cannot imagine not being able to take advantage of dynamic programming techniques, and I am still hoping that I will one day find that there is a fourth level.

## Using Macro Language Elements – The Second Stage of Macro Language Learning

In the second level the programmer goes beyond code substitution by taking advantage of more of the elements of the macro language. Some of these elements bear a strong resemblance to similar elements in the DATA step. This reinforces the notion that you cannot be a better macro language programmer than you are a DATA step programmer.

Similar elements include:

- functions — many macro functions have analogous counterparts to the DATA step functions
- statements — some executable DATA step statements, such as the DO and IF, can also be found with similar syntax in the macro language
- options — system options can be used to affect the behavior of the macro language

Mastering this level of the macro language allows you to control the flow of your program at the step level. The DATA step IF statement gives you control within a DATA step, but not across steps. The macro %IF statement operates on the program-code level (remember macro language is a code generator) and writes the steps that are to be executed. We can use macro functions to determine the status of our program, and then using logic, apply that information to guide the execution of the program itself. Imagine a program that creates a data set and then depending on the number of observations either performs a PROC PRINT or a PROC MEANS on that data set.

You can also use macro logic and macro language elements to control the statements and flow of your program within a step. These could include things like which data set to read, which observations to remove, and the naming of variables based macro variables.

As is so often the case when learning something new, it is not immediately obvious why it is worth the trouble to learn the macro language. But once you start to accumulate an understanding of the macro language elements, you will find more and more uses for them. Soon you will be wondering how you ever managed to program without them.

## Dynamic Programming

The true power and flexibility of the macro language becomes available to you as you master the techniques at this level. In dynamic programming it is less about the macro language itself, and more about the nuances of how it is applied. Although there are some macro language elements that you are less likely to use in the first two levels, for the most part you will be using these elements in different ways.

The construction and use of lists is a very important aspect of these kinds of programs. List processing allows us to execute a program many times, with each iteration based on different items, such as data sets or variables. There are four distinct types of list processing (Fehd and Carpenter, 2007), and the two most common and flexible approaches strongly utilize many of the macro language elements learned in stage two (Rosenbloom and Carpenter, 2014).

Dynamic programs adjust to their environment. Rather than telling the program which data sets or variables to process against, these programs tend to make these determinations at execution time. This means that you can write programs that will execute against data sets that have not yet been created or perhaps even named when you do your coding.

In our PROC PRINT example we have selected one data set to print. What if we want to print every data set in a given directory, and we do not know the data set names or even how many data sets there are in that directory? A dynamic program can determine the names, make a list, count the items in the list, and then "type" the code to generate the PROC PRINT steps for each of the data sets. If the list of names changes, our program will not care; it will dynamically adjust to different data set names and different numbers of data sets. Dynamic programs tend not to have hard coded information in the program when that information can be determined by the program at execution time.

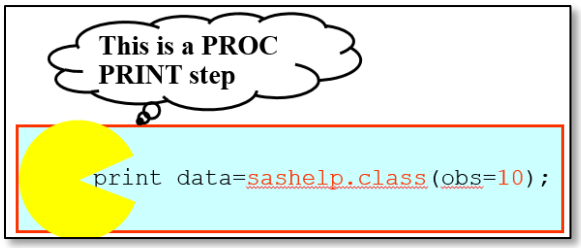
There are a great many tools in SAS and in the macro language that can be used by the programmer to help the macro language to dynamically determine the information that it needs to execute. SAS has special data sets and tables that contain all kinds of runtime information about the operating system, about the SAS environment, and about the data sets themselves. There are functions and statements that can retrieve this information and position it in such a way so that the macro language can take advantage of it.

## MACRO LANGUAGE PREVIEW: PART 2

## IT IS ALL ABOUT THE TIMING; WHY THE MACRO LANGUAGE COMES FIRST

Because the macro language is primarily a code generator, it makes sense that the code that it creates must be generated before it can be executed. This implies that execution of the macro language comes first. Simple as this is in concept, timing issues and timing conflicts are often not so simple to recognize in application. But as we use the macro language to take on more complex tasks, it becomes even more critical that we have an understanding of these timing relationships.

First remember that SAS is a parsed language. This means that the code is processed one portion at a time. If a program contains three DATA steps, a syntax error in the second will not be even noticed until the first step has been fully completed. The same is true at the statement level. The first statement is parsed before the second is looked at. Think of the parser as a Pac Man game that gobbles up code instead of energy dots. In reality there is more than one parser including a separate one for macro language elements.


Let's say that we submit a PROC step for execution without using any macro language elements in our program. The submitted code is parsed, then compiled, and then executed. These steps always happen and they have to take place in this order. When macro language elements are present (remember that macro language elements are always marked using either an ampersand or percent sign), the process is interrupted. When the parser encounters a macro language element, it cannot continue

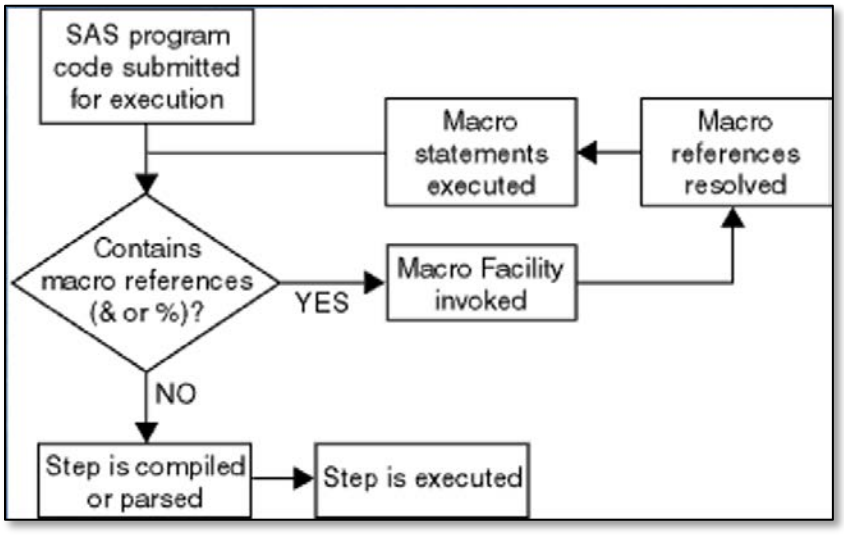
until the element is resolved or executed. This resolution or execution takes place in the macro facility, while the statement parser waits patiently for a response.

In the PROC PRINT step shown earlier there are two macro language elements (macro variables). Before the PROC statement can be fully parsed these macro variables must be resolved. The parser encounters the &DSN, the &DSN (which is known as a token) is then passed to the macro facility. Within the macro facility the &DSN is resolved, in our example, to SASHELP.CLASS. The SASHELP.CLASS is 'typed' (replaces the &DSN in our PROC statement), and it is this 'typed' value that the parser sees. This same process then repeats a few characters later when the &CNT is encountered.

```
proc print data=&dsn(obs=&cnt);
   run;
```

Think of the program that we have written as stream of water with us (the parser) standing on a bridge watching it flow by. Slightly upstream of us is a partner on another bridge scanning for macro elements which might be floating downstream. These macro language elements are recognizable by the macro triggers, the ampersand (&) and percent sign (%). If the scanner detects a macro language trigger, the water upstream of us immediately freezes (ceases to flow – hey this is an analogy it can only go so far). The macro language element (token) is lifted from the water, processed, and then the resulting value placed ('typed') back into the stream at the same spot. The stream now thaws (in Alaska we call this breakup), starts to flow, and the process continues. This means that necessarily by the time the water gets to us there are no more macro language elements in the water.

Reality is of course more complex, but if we can live with this simplification we can diagram the process. The diagram shown here can be applied at the step level, statement level, expression level, and even the word level. Regardless of the level of application, the code is scanned for macro language elements ***BEFORE*** it can be parsed. This timing is very important. There are some things in the macro language that just will not make sense to you if you do not understand this process. The trouble is that the discernment of the timing of events is not always obvious




In fact it is not uncommon for very successful macro programmers to have no understanding of the order of these events. However if you do not understand these timing relationships there are some things that you will not be able to do in the macro language.

A simple, and unfortunately not uncommon, application of this timing issue and its related confusion can be seen in the DATA step to the right. We would like to retrieve the value in the variable AGE and write it into the macro variable &J_AGE using the %LET statement, and this simply ***cannot*** work. According to the timing diagram the %LET will execute before the DATA step has compiled, and this will be before the variable AGE exists on the PDV, and this is even before the PDV exists, and certainly before any data has been read. Fortunately there are *DATA step tools*, such as the SYMPUTX routine shown here, that can be used instead of the %LET statement that will solve this problem. Emphasis on DATA step tools, and NOT macro language elements.

```
data _null_;
   set sashelp.class(keep=name age
                     where=(name='Jane'));
   %let j_age = age;
   run;
```

```
call symputx('j_age',age,'l');
```

The application of this timing diagram is not always this obvious and it takes practice to look for instances where it applies, but remember – Macro language elements will execute BEFORE the DATA step even exists. Macro language elements are therefore not used to access values on the Program Data Vector.

## MACRO LANGUAGE PREVIEW: PART 3
## CREATING MACRO VARIABLES AND DEMYSTIFYING THEIR SCOPE

Macro variables and the text values they hold are stored in symbol tables, which in turn are held in memory. Not only are there a number of ways to create macro variables, but they can be created in a wide variety of situations. How they are created and under what circumstances effects the variable's scope - how and where the macro variable is stored and retrieved. There are a number of misconceptions about macro variable scope, and about how the macro variables are assigned to symbol tables. These misconceptions can cause problems that the new, and sometimes even the experienced macro programmer, does not anticipate. Understanding the basic rules for macro variable assignment can help the macro programmer solve some of these problems that are otherwise quite mystifying.

### Using Symbol Tables

Although macro variables can be created in a number of different ways (Carpenter, 2004), they are always stored in symbol tables and there are only two basic types of symbol tables – global and local. The assignment of a macro variable to a symbol table is *not* however this straight forward. A given macro variable is written to a specific table, and the rules that determine which table is to receive a macro variable can be very arcane. Often when you are writing macro code it is not always possible to anticipate which symbol table will receive the macro variable. To make matters more complicated, there can be any number of local symbol tables (thankfully there is always only one global symbol table).

One of the features of the macro language is the ability to create reusable code known as macros. Macros, which are defined using the %MACRO and %MEND statements, are important because only some of the power of the macro language can be utilized outside of a macro definition. When you write code that is outside of a macro definition (your code is not between a %MACRO and %MEND statement), you are said to be in open code. Macro variables that are created in open code are always stored in *the* global symbol table. When a macro executes it will generally, but not always, have a local symbol table associated with it. Macro variables created within a macro will *usually*, but not always, be written to the local symbol table associated with that macro. When the macro finishes executing its local symbol table is deleted.

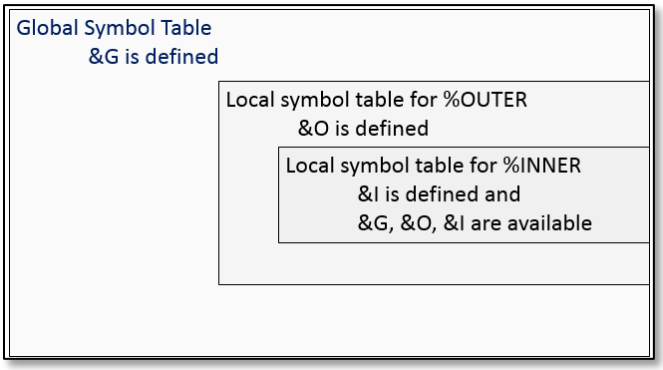
We know that data sets stored in the WORK directory will be deleted at the end of the SAS session, while data sets stored in user defined libraries can be stored permanently. How long the data set is stored is known as its scope or persistence. Once a TITLE is defined in a SAS program, its definition remains until the end of the SAS session or until it is redefined. Its scope is the SAS session. PROC PRINT options defined within a PROC PRINT step are not carried forward to the next step, so their persistence or scope is at the step level. Macro variables also have scope or persistence levels. The global symbol table and its macro variables have a scope of the SAS session. A local symbol table, however, only exists during the execution of its associated macro. The scope of a local symbol table is limited to its macro's execution, therefore macro variables stored in a local table are no longer available after the macro terminates.

### Nested Symbol Tables

Imagine that you are executing a macro which has a local symbol table. There will be two different symbol tables available to store macro variables – the global symbol table and the local one associated with the macro. If you create a macro variable and it is written to the global symbol table it will be available after your macro terminates, however if it is written to the local table, the macro variable and its value will be available only during macro execution. These two symbol tables are said to be nested, with the local table nested within the global table (the global table is said to be 'higher'). From within the macro you can use macro variables in either table - that is you can always look outward. You cannot look inward because the inner table only exists during the execution of its macro.

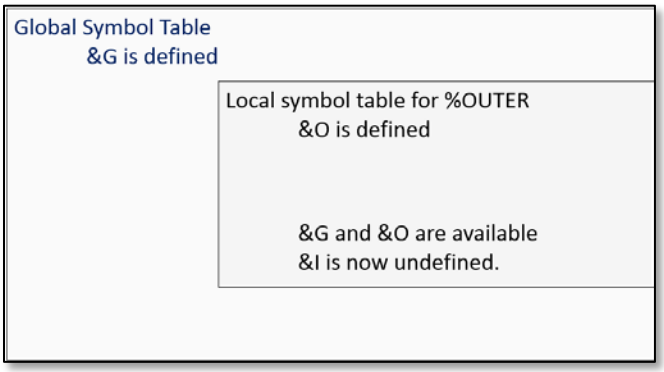
Have you ever seen the Russian dolls known as Matrioskas? These are a series of dolls, which are usually carved from wood so that they nest within each other. The number of dolls in the set depends on the skill of the artisan, and each doll traditionally has a painted lady with a colored scarf and apron. As you open the doll set for the first time, you can see the colors of the dolls that have been opened including the one unopened doll, but you have no idea how many more dolls are in the set or the colors of their scarves. In the picture we can see that the fourth doll is unopened, is there a fifth? We have no way of knowing, but at any point in the process we do know how many have already been opened and the colors associated with each doll that has been revealed.

Although you might not recognize it as such, one of the powers of the macro language is the ability to have macros that call other macros that call macros, and so on. These nested macro calls can result in nested symbol tables. Like with the Matrioskas, where you can see the scarves of the opened dolls, you can use the macro variables in the current local table (the revealed, but unopened doll) or any higher symbol table (opened doll), but you cannot access any aspect of the dolls that have yet to be revealed.




In the illustration shown here, the macro variable &G is defined in the global symbol table. The %OUTER macro is called (macros are named and the macro is called, executed, by preceding its name with a percent sign), and it defines &O in its local table. The %INNER macro is then called from within %OUTER and &I is defined. Although each of these macro variables resides in a different symbol table, during the execution of the %INNER macro, all three macro variables are available to be used.

Once the %INNER macro has completed executing, its symbol table is removed, and any macro variables that it contained are no longer available. Essentially we have closed up one of the nesting dolls.




So when no macros are executing, only the global symbol table is available. When macros are executing, nested symbol tables are available, and your program can use macro variables from any of the symbol tables from the most local to the highest (global).

### The %LOCAL and %GLOBAL Statements

When the %LET statement is used to create a macro variable that macro variable and its value is written to a symbol table, however there is nothing on the %LET statement that can force that macro variable to any particular symbol table. Instead there are a series of arcane rules that govern into which symbol table the macro variable will be written. You do NOT want to learn these rules, besides knowing them is not enough, as very often their application is unknowable until the macro executes. There are a couple of things that you can do to help control where a macro variable is to be written. The %GLOBAL statement is used to force macro variables into the global symbol table and the %LOCAL statement forces them onto the most local symbol table.

The English idiom 'Rule of thumb' is used to indicate the way things should work. One might say, "As a rule of thumb, moss grows on the North side of trees", or "As a rule of thumb there should be at least five dolls in a set of Matrioskas". Usually these 'rules' are based on one's experience or perceived experience. One of the more dangerous 'rules of thumb'

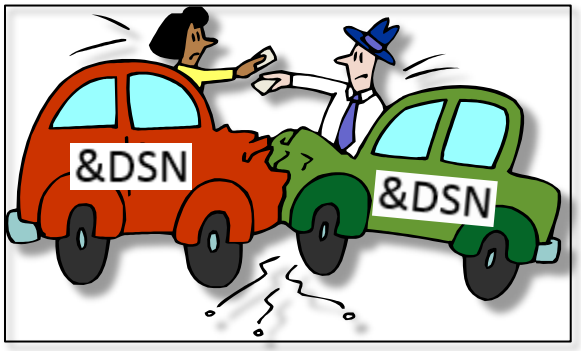
in the macro language has to do with the writing of macro variables to symbol tables. I say dangerous because it can be very wrong (moss doesn't necessarily grow on the north side of the tree either). This rule of thumb that I often hear is: "Macro variables created during a macro's execution will be written to that macro's local symbol table". This rule is dangerous because it is almost always right, and in the world of computing the word 'almost' can be very scary. Those that hold to this 'rule' (think that it is true) do not know those arcane assignment rules or how they are applied. Or even potentially the trouble that can be caused by their lack of understanding.

We can greatly simplify the assignment rules through the use of the %LOCAL and %GLOBAL statements. Use them when you need them. Use them when you think you don't need them. Although I rarely used the %GLOBAL statement, virtually every macro that I write uses a %LOCAL statement. Placing macro variables on the local table means that I know where they are and I know what other variables are on that particular symbol table. Faithfully using the %LOCAL statement prevents macro variable collisions.

### Macro Variable Collisions

Just like with data set variables, where a given variable name can be used in more than one data set, macro variable names must only be unique within a symbol table, but not necessarily across symbol tables. This means that I can have a macro variable &DSN in the global symbol table, and a different &DSN in a local symbol table. Although these two macro variables with the same name can coexist, the programmer must be careful. Since the two variables have the same name, &DSN, there is no easy way for the programmer to distinguish between them in the program itself. Also if one of the values of &DSN is to be updated, say with a %LET statement, which one will be updated?

A macro variable collision (Carpenter, 2005) occurs when two or more macro variables share a name, and when being updated the value of the wrong one is changed. In earlier examples, we have already established the global macro variable &DSN to contain the name of the data set of interest. If within a macro, we attempt to establish another &DSN to contain say a number, will we create a new local &DSN or will we inadvertently write over the existing global variable? The answer is 'Well it depends . . . .'. Some of the factors that it depends on are out of our control, but the best way to protect ourselves and our macro variable symbol tables is to use the %LOCAL statement whenever possible within our macro definitions.




If all your macros contain a %LOCAL statement forcing all of the macro variables to be local (even if they would be local anyway according to the arcane rules), you will totally avoid macro variable collisions. The %LOCAL statement costs nothing, except a little typing but can save hours of heartache.

## SUMMARY

The high level discussion of the macro language in this paper is designed to motivate you to get motivated to start learning the macro language. It is intended to show you the value of the macro language while pointing out a few of the catch points that sometimes make the learning process more difficult. There is a lot to learn, however you can do a great deal with the macro language, even if you only master the basics of code substitution.

As you become a stronger macro language programmer, you will also become a better DATA step and PROC step programmer, and as you master the macro language you will find that its power and flexibility will make other programming tasks easier and faster.

## ABOUT THE AUTHOR

Art Carpenter's publications list includes; five books, two chapters in *Reporting from the Field*, and numerous papers and posters presented at SAS Global Forum, SUGI, PharmaSUG, WUSS, and other regional conferences. Art has been using SAS since 1977 and has served in various leadership positions in local, regional, and national user groups.

Art is a SAS Certified Advanced Professional Programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

Recent publications are listed on my sasCommunity.org Presentation Index page.

## AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

## FOOTNOTES

1 Cynthia Zender, renowned trainer for SAS Institute, uses the concept of the macro language as a typewriter extensively when she teaches the macro language. If you found this analogy helpful, I probably stole the idea from her.

## REFERENCES

Carpenter, Arthur L., 2004, "Five Ways to Create Macro Variables: A Short Introduction to the Macro Language", Proceedings of the 12th Annual Western Users of SAS Software, Inc. Users Group Conference (WUSS), Cary, NC: SAS Institute Inc. Also in the proceedings of the Pharmaceutical SAS User Group Conference (PharmaSUG), 2004, Cary, NC: SAS Institute Inc., paper HW02, and in and in the proceedings of the South East SAS User Group Conference (SESUG), 2005, Cary, NC: SAS Institute Inc., and in the and in the proceedings of the Mid West SAS User Group Conference (MWSUG), 2005, Cary, NC: SAS Institute Inc. Also presented at WUSS (2009, and PNWSUG (2009).

Carpenter, Arthur L., 2005, "Make 'em %LOCAL: Avoiding Macro Variable Collisions", proceedings of the Pharmaceutical SAS User Group Conference (PharmaSUG), 2005, Cary, NC: SAS Institute Inc., paper TT04. Also published in the proceedings of the 13th Annual Western Users of SAS Software, Inc. Users Group Conference (WUSS), Cary, NC: SAS Institute Inc., paper SOL_Make_em_local_avoiding.

Fehd, Ronald J. and Arthur L. Carpenter, 2007, "List Processing Basics: Creating and Using Lists of Macro Variables", proceedings of the SAS Global Forum Conference , 2007, NC: SAS Institute Inc., paper 113-2007.

Roosenbloom, Mary F. O. and Arthur L. Carpenter, 2014, "Are You a Control Freak? Control Your Programs – Don't Let Them Control You!", proceedings of the Western Users of SAS Software Conference, 2014, NC: SAS Institute Inc.

## TRADEMARK INFORMATION