Paper 254-31

Reading Between the Lines: Distinguishing Macro Code from Open Code in Macros

Mike Molter, Howard M Proskin & Associates, Rochester, NY

Abstract

Have you ever been confused by the sight of bad "SAS® grammar" such as statements that read *%IF &X=A %THEN IF X=A…*? Have you ever puzzled over how to iteratively generate DATA step DO loops or parts of them? Ever wonder why macros sometimes contain consecutive semicolons? If so, you are not alone. The macro facility enables us to generate open code that varies with circumstances, but of course the directions for code generation must be conveyed through more code. Inevitably, we find ourselves having to mix open code with macro code. Add to this the fact that macro code contains statements, keywords, and syntax that resemble their DATA step counterparts, and we begin to see the difficulties programmers experience with reading and writing macros. This paper attempts to clear up all the confusion by illustrating exactly what parts of a macro accept open code. Along with some supporting information such as macro syntax, quoting functions, and macro processing, we will discuss macros that only generate code (without any macro logic), as well as mixed macros or those that mix code generation with macro logic. Along the way comparisons are made to DATA step counterparts where applicable. Examples are provided to illustrate consequences of the rules such as bad SAS grammar and double semicolons. This paper is most useful to those beginning with macros and looking to solidify their current knowledge base of macros.

INTRODUCTION

Writing code is one thing, but writing code that writes code is quite another. For the most part, that is what we use macros for – providing instructions to the SAS system on how to generate code we want executed when called upon. Of course, code-generating instructions inevitably have to contain the code that is to be generated. Now that the obvious has been stated, let's state the dilemma: how does the SAS system know which parts of what you are typing are instructions, and which parts are code to be generated?

In order to begin talking about distinctions between open code and macro code, let's first make sure we know the meanings of the two terms. Code itself is a language in which we provide instructions to the SAS system. The difference between open code and macro code is "to whom" the instructions are being provided and what those instructions entail.

Open code is the code you write or the instructions you provide to the SAS system every day. When you write a PROC, you are telling the SAS system how to analyze a data set. A DATA step is a set of instructions for the SAS system (or the DATA step compiler) on how to build a data set. Certain DATA step instructions tell the SAS system how to set up the Program Data Vector while others instruct the SAS system how to populate variable values and ultimately generate observations for the new data set. Most open code (with the exception of a few isolated global statements such as TITLE, OPTIONS, and ODS) is organized into steps and the steps are run in the order in which they are found in the program.

While open code is a set of instructions for processing data, macros are sets of instructions for processing, manipulating, and generating text. When a macro is called or invoked, the text generated becomes open code at the location of the call. When called inside a PROC or a DATA step, it is only after this execution of the macro does the step execute. The macro facility does not require macros to contain any macro logic. The instructions for generating text may only provide the text to be generated, or they may contain logic that determines how the text will be generated. At points in this paper, I refer to macros of the former type as trivial, because they do not take advantage of the power of the macro facility, and they accomplish nothing more than can be achieved by other means without the overhead of macro compiling. In spite of this, they will come up in examples because lessons can be learned from them, but most of the focus will be on what I call mixed macros. A mixed macro is one that uses macro logic to generate open code. The idea is that a piece of code has the potential to change with circumstances from one use of the program to the next, and so such macros are often defined with parameters that reflect these changing circumstances. Directions to the macro processor, including what text to generate, are often based on these parameters. The necessity to distinguish between these two languages comes from the simple fact that for macro code to generate open code, both must co-mingle in a macro.

USE YOUR IMAGINATION

Because of its very nature, writing any kind of code requires imagination. Take a DATA step for example. You can picture in your mind what you want your data set to look like, but your set of instructions are for a computer, and so you cannot just tell the computer verbally what you need or draw it a picture. You have to use code that the software

understands, and you have to obey strict sets of rules or the computer will not understand. You have to make sure you spell key words correctly; semicolons have to be in the right places, and so on. Finally, after tests, repairs, reviews to make sure the data set looks like what you expected, your carefully, delicately constructed DATA step is complete. In some ways it can be like writing a paper for publication. Though your "audience" is human instead of a computer and can make sense out of flaws in the writing, these flaws are frowned upon by publishers. Once you have worked out the flaws, or in this case, the syntax errors and other things leading to undesirable results, you save it and don't want to touch it again. Then comes the need for a macro!

Once you have decided that your program can be made more flexible and used under multiple circumstances by introducing macro logic to it and allowing pieces of code to vary, the amount of imagination required has now more than doubled. Again, *any* kind of code requires imagination. Conveying to the macro processor the logic needed to generate a piece of code requires imagination. It requires testing, repairing, making sure the code generated is what you had in mind. This is in addition to making sure that the code you had in mind will actually produce the data set (or whatever you want from your open code) you need. In other words, at the same time you are considering how to write the DATA step, you are also considering how to write the macro code to produce that DATA step. Because macro code generates open code, the two exercises cannot be done independently. You cannot think of one without thinking of the other. The task now becomes to take that open code that you constructed so carefully and inject macro logic in the appropriate places to generate the pieces that can vary. This must be done in such a way that when the macro executes, the pieces get put back together again.

Beyond having to write two sets of code at once, the similarities between the two of them bring about other difficulties. Without macro code, you can look at the open code and tell exactly what is going on. You know what the statement key words mean, you know what the text that follows them means, you know why semicolons are where they are, and you know the order in which the statements are processed. You can look at it and have a pretty good idea what it is going to produce. By breaking all of that up with macro code, all of the open code gets much harder to find and read. We're not used to looking at our code that way. To make matters worse, macro logic often has a look and feel to it very similar to some of the logic we find in DATA steps. Despite the fact that certain aspects of macro logic have indicators that visually distinguish it from open code such as statement keywords being preceded by percent signs, this can lead to much confusion. One possible effect is the appearance of logic in places it does not belong. Another effect is confusion as to who any given instruction is intended for. When mixing open and macro code, it can be difficult to tell if a piece of code is part of the macro logic for generating code or if it is the code to be generated, ultimately serving as an instruction for a DATA step or a PROC step.

This paper is intended to clear up that confusion. For as long as macros are used to logically generate open code, they will always contain a mixture of open and macro code. There is nothing we can do about that. But if we can learn to read between the lines of a macro and identify where and what the open code is, no matter how widely it is scattered across the macro, then we can more effectively divide our attention between the code we wish to generate and the code used to generate it. This is not intended to be a lesson in macro logic and code generation per se, but because macros generate open code with logic very similar to that used by DATA steps to generate observations, we will take a closer look at these tools. Noting not only similarities but also differences between DATA step logic and macro logic will ultimately help us to identify exactly what parts of a macro can accept open code without confusing it with macro code.

LOGIC TOOLS

Though it is not their only capability, both the macro and the DATA step generate something. Though the generation of data may seem like a far cry from the generation of text to be used as open code making it a weak connection, it is the similarity in the logic used to do so that makes it a powerful one. Both need a mechanism to tell the SAS system when and what to generate. Both can use this mechanism in a trivial manner without the aid of any logic. For example, a DATA step may just read in data, create some data set variables and end, generating the observation implicitly. Similarly, a macro may just specify the code to be generated at any place the macro is called. Less trivially, the DATA step gives us ways to control the generation of observations. Based on conditions ranging from simple and involving one data set variable to complex and involving different functions of several data set variables, the number of observations to be generated for a new data set can be limited. The same can be said of the amount of code to be generated by a macro based on macro variables. In addition to limiting the number of observations, we can also expand the number of observations with looping logic (DO loops). By generating observations within a loop that iterates through a data set variable *n* times, we can create a new data set with up to *n* times as many observations. Sometimes it is only the iterating variable that changes value from one iteration to the next, but it is also common to change the value of another variable based on the value of the iterating variable. Similarly, code can be generated iteratively based on the values of iterating macro variables with a macro DO loop.

Differences between the DATA step and the macro are obvious, and the difference between logic based on data set variables and logic based on macro variables is a significant one. In most cases, the DATA step reads data, and

does so one record at a time. Each of the executable statements is processed for the current record, and when this is finished, the same process repeats for the next record. This iterative process continues until all records are read. The DATA step also consists of compile statements. These statements help the SAS system create something called a Program Data Vector (PDV), which serves as a memory buffer to hold values of variables while the current record is being processed. Executable statements look to the PDV to retrieve values of variables when necessary. In particular, conditional statements such as IF-THEN and ELSE look here to make a decision about which statement to execute.

More conceptually and less technically, the instructions provided in a DATA step are typed once but executed for each record read. Each data set variable can be thought of as an attribute of a set of data, and each value reflects an attribute of a record. The conditional generation of observations allows us to limit the data set we are creating based on what is found in the data set read. Because data set variables reflect attributes of data, we have no control over their values. This means that though we can define conditions for generating data, the decision as to whether to create the observation or not is driven by the data and out of our hands.

Macros, on the other hand, have nothing to do with data and have no need for a counterpart to the PDV. Macro logic is executed only once per call of the macro rather than iteratively. Macro variables can be thought of as units of memory that hold information too, but the information is nothing more than text, not necessarily a reflection of any data, and their values persist until explicitly changed. Macros are usually built to centralize common tasks that differ only by parameter values, and macro variables are used to store these parameters. So while it is **data** that determines how and if observations are generated, it is text as a representation of the current **circumstance** that determines the text to be generated. While conditional generation of an observation in a DATA step may take the form

IF an attribute of the data meets condition x, THEN generate the observation,

conditional generation of code may take the form

IF a user makes this choice THEN generate and execute this code.

or

IF today is Thursday, THEN generate and execute this code
ELSE execute that code

The difference between DATA step DO loops and macro DO loops is similar. The generation of observations within a DATA step loop results in an increase in the number of observations from the data read. Added to the PDV and hence included in the final data set (unless explicitly dropped) is the variable used to track the iteration. In some cases, this may be the only variable that distinguishes all the observations generated for the current iteration of the DATA step. Perhaps more often is the case that a pattern is developed for defining or using variables or variable values based on the current iteration value. One example of this involves transposing the values of several variables from one observation into values of one variable across several observations. By defining an array with the variables to be transposed and then using a DO loop to iterate through the dimension of the array, we define the value of the new variable by referencing the array and using the iterating variable as the array subscript. We then generate the observations within the loop.

Just as DATA step loops iteratively generate observations of a data set, macro loops iteratively generate code. Also, just as DATA step loops can generate the same observation with each iteration (with the exception of the iterating variable itself), macro loops can generate the same code every time. Again, perhaps the more common use makes use of the macro variable to generate text. However, since the macro variable is just a storage unit for text, their values can be used to generate patterns of text rather than just patterns observations. For example, if a DATA step requires a statement that reads

IF X1=1 AND X2=1 AND X3=1 AND.... AND X100=1 ;,

then a macro that generates *IF X1=1* on the first iteration of a %DO loop, and *AND X&j* on iterations 2 through 100, where *j* is a reference to the iterating macro variable, would save significant typing. Other uses of iterating macro variables might be found in references of the form *&&Y&j*, where *j* is still the iterating macro variable, but &*Yn,* the result of the initial resolution on the *n*th iteration, is a reference to a macro variable initialized by the user.

Even the generation of observations or open code without the aid of the iterating variable can be useful when one or both of the limits of the iterating variable can vary. In the case of the DATA step, this means replacing either or both

of the limits with data set variable names (e.g. DO I = X TO Y). Without any other references to the iterating variable in the loop, this will result in the generation of the current record *n* times (each observation differing only by the value of the iterating variable) where *n* is one more than the difference between the upper and lower bound (assuming the increment through the range is always 1). In these cases, it is again what is in the data, not us, that determines the generation of an observation. In macros, it is very common to iteratively generate a piece of code as many times as specified by a user, thereby using a macro variable as an upper bound to a macro loop.

GENERATING CODE VS GENERATING OBSERVATIONS

Up to this point, we have talked about similarities and differences between tools common to macros and DATA steps. The last tool and maybe the most important for this paper is the mechanism for generation. The DATA step has the OUTPUT statement. You may not find yourself typing it much, but when absent, one is always implied at the end of the DATA step. This is referred to as an implicit OUTPUT statement. In other words, you have to make an effort not to generate an observation (e.g. DELETE statement, subsetting IF). Also, the rigid structure of the PDV allows us to use the statement without telling the SAS system *what* to generate. Whatever is in the PDV when the OUTPUT statement is executed is what is generated in the observation. The macro facility, however, does not have a PDV. Because of that, you have to tell the macro processor what code to generate. On top of that, the macro facility has no counterpart to the OUTPUT statement. Combining these two facts,

the OUTPUT statement of the DATA step is replaced in the macro simply by the code to be generated.

Since, as we have discussed up to this point, code can be generated in macros with the same tools that observations can be generated in a DATA step, we can identify within a macro the text meant to be generated as open code and what text is intended for the macro processor (macro code) by simply thinking about where the OUTPUT statement is in the DATA step. Now armed with the knowledge of the macro's mechanism for generating code, we can now discuss these code-generation tools in some more details and illustrate with examples.

THE TRIVIAL CASE

We begin with the trivial case. By trivial case, I mean the generation of code without the aid of any looping or conditional logic. You do this all the time in the DATA step. As mentioned above, without an explicit OUTPUT, each DATA step contains an implicit OUTPUT statement at its end. For example, the DATA steps in the following two examples have the exact same effect.

```
data two ; /* Example 1 */
set one ;
*** implicit OUTPUT statement here *** ;
run;

data two ;   /*Example 2*/
set one ;
output ;
run ;
```

Now consider a trivial macro that generates code without the aid of looping or conditional logic. Such a macro may or may not contain macro statements, but rather than a statement to generate code like the OUTPUT statement that generates observations in a DATA step, the macro simply contains the code itself, as in the following example.

```
%macro two ; /* Example 3*/
%let a=San Francisco ;
open code to generate
%mend ;
```

Once again, the provision of the code to generate is not contained in a macro statement. Because of this, it is not preceded by a percent sign as all macro statements are, and does not end with a semicolon, as macro statements do. This brings us to an important point about a significant difference between the way a DATA step is constructed and the way macros can be constructed.

The DATA step (as well every PROC) is made up of nothing but statements. Each statement begins with what we call a statement keyword and ends with a semicolon. This is true from the very first statement of a DATA step – the DATA statement – and ends with the RUN statement. Of course every statement has its own syntax; some allow for options that may be expressed in parentheses, after slashes, or in other ways, but every statement ends with the semicolon, and immediately following every semicolon (except for the last one in the program) is another statement keyword to kick off another statement. We are now starting to see that macros are not necessarily the same. Every

macro statement begins with a statement keyword preceded by a percent sign and ends with a semicolon, but what follows ***does not*** have to be the beginning of a new statement. Unlike a DATA step, in a macro, there can be "stuff" in between the statements, and that stuff is open code. Below are the possibilities for what follows a statement-ending semicolon.

- A percent sign followed by a macro statement keyword – This is the beginning of a new macro statement.
- An ampersand followed immediately by text – This is a macro variable reference whose resolution will be part of the open code to be generated.
- A percent sign followed immediately by a macro function keyword – This is a macro function call whose resolution will be part of the open code to be generated.
- A percent sign followed immediately by the name of a macro – This is a macro call whose generated code will be generated here.
- Anything else is part of the open code to be generated.

In other words, everything that follows a macro statement-ending semicolon up until the next percent sign that precedes a macro statement keyword will be generated open code.

Consider the following examples:

```
%macro example4 ;  /* Example 4 */
data two ;
  set one ;
  array ex1(*) var1 var2 var3 ;
  do onevar=1 to 3 ;
    value=ex1(onevar);
    output;
  end;
  drop var: ;
  run;

proc freq data=two ;
  tables onevar*value ;
  run;
%mend example4 ;

%macro example5 ;  /* Example 5 */
data three ;
  set two ;
  if value gt 100 then saleprice=0.8*value ;
  else if value gt 90 then saleprice=0.85*value;
  else saleprice=0.9*value ;
  run;
%mend example5 ;

%macro example6 ;  /* Example 6 */
  do onevar=1 to 3 ;
    value=ex1(onevar);
    output;
  end;
%mend example6 ;

%macro example7 ;  /* Example 7 */
  else if value gt 90 then saleprice=0.85*value ;
%mend example7 ;

%macro example8 ;  /* Example 8 */
  or x2=1
%mend example8 ;

%macro example9;  /* Example 9 */
  or
%mend example9;
```

```
%macro example10;  /* Example 10 */
  "09"x
%mend example10 ;
```

The first example generates multiple steps and each subsequent example generates progressively less code. Some of these are not very practical examples because their effects can be achieved in more efficient ways. Code generated in any of the first few examples could be generated by saving it in its own program and then calling it with %INCLUDE rather than compiling a macro. Portions of statements might just as well be typed with the rest of the code rather than generate them with a macro. A few lessons, however, can still be learned from each.

The common thread among all of these examples is that they all generate code without the aid of any logic. As we know, each begins with a %MACRO statement. More specifically, they begin with a percent sign followed by a macro statement keyword (MACRO). Everything up until the semicolon is part of the statement. Following the semicolon is code to be generated, which continues until the next percent sign that precedes a macro statement keyword – in these cases, %MEND.

Though maybe not practical, each of the examples above is a legitimate macro. In other words, each will compile and each will execute (generate code) when called upon. At first this might seem counterintuitive, particularly with the later examples. We are used to writing code that falls between the beginning and end of a step in statements, not fragments. At this point, two important facts about macros are worth remembering. First, as we stated earlier, not everything in a macro between the %MACRO statement and the %MEND statement is contained in a statement. Second, there is no restriction on the amount of code that can be generated. Often times it is the case that a part of a statement is expected to vary from one circumstance to the next, and another part is not. In such cases, a macro might be called within an open code statement. The lesson here is that every macro we write, we do so with an idea of exactly where in our open code it will be called. Because the first two examples generate steps, each would be called between steps in a program. Illustrated below are examples of where in a program each of the rest of the above examples could be called.

```
data two ;  /* Example 11 */
set one ;
array ex1(*) var1 var2 var3 ;
%example6
run;

data three ;  /* Example 12 */
set two ;
if value gt 100 then saleprice=0.8*value ;
%example7
else saleprice=0.9*value ;
run;

data four ;  /*Example 13*/
set three ;
if x1=1 %example8 ;
run;

data four ; /* Example 14*/
set three ;
if x1=1 %example9 x2=1;
run;

filename spread dde 'excel | sheet1!r1c1:r100c2' notab ; /* Example 15*/
data _null_ ;
file spread ;
set four ;
put x1 %example10 x2 ;
run;
```

It cannot be overemphasized that it is everything between macro statements in a macro that will be generated as open code. This includes semicolons. Semicolons are worth special attention because they have the same special meaning in macros as they do in open code. For that reason, we will continue to revisit them throughout. Once again, everything between the semicolon that ends the %MACRO statement up until the next macro statement is generated code and that includes semicolons. Because it is between statements, we know that these semicolons are

not instructions to the macro processor, but rather code to be generated. Note how this affects the way such macros are called. Macro calls within open code are not considered statements and do not require semicolons per se. They would only require a semicolon when the code they generate does not include a semicolon to end a statement. The macro EXAMPLE7 generates a semicolon and so the call of this macro in example 12 is not followed by a semicolon. The opposite is true of the macro EXAMPLE8 and its call in example 13.

LOOPING LOGIC

We are now ready to move on to non-trivial code generation. We know that by inserting an OUTPUT statement into a DATA step DO loop, the observation will be generated as many times as the loop iterates, and each observation will differ from each other by at least the value of the iterating variable if it is kept. Similarly, code such as that in the examples above can be generated iteratively with macro loops. Most of the time, as with the generation of observations in a DATA step loop, such loops are useful when the code to be generated is a function of the iteration.

Our first example is a rare instance of iterative text generation without the use of the iterating variable. In it we re-visit example 10 from above. "09"x is the hexadecimal tab character, and is useful when writing variable values into multiple columns of an Excel spreadsheet using DDE as was done in example 15. We stated earlier that the generation of this character by a macro is no more effective than just typing the character with the rest of the open code. Suppose however that you do not necessarily want to write into adjacent columns. To leave a blank column in between the two you are writing to, your PUT statement would include the name of the variable to write in the first column, followed by two tab characters to move the pointer over two columns, followed then by the name of the second variable. Similarly, to leave ten blank columns, you need to type eleven tab characters. As a convenience to yourself, you can insert a macro DO loop into %EXAMPLE10 where the number of iterations is provided in a macro parameter.

```
%macro example16(tabnbr) ;  /* Example 16 */
%do i=1 %to &tabnbr ;
    "09"x
%end;
%mend example16 ;
```

The following illustrates how the macro might be called inside a DATA step.

```
put var1 %example16(4) var2 %example16(2) var3 %example16(5) ;
```

It is often the case that the user needs more control over the generation of code than just the number of iterations. We now return to example 5 in which a DATA step that creates a variable based on another variable is created. Suppose you wish to insert a loop that iterates according to a macro parameter. If you simply generate the open code in this macro multiple times, then you will be creating the same data set over and over again. Instead of supplying the macro with a number, the user supplies the macro with a list of data sets to which this logic (creation of a DATA step variable) must be applied. Given a list of data sets but not the number of data sets, it is up to the macro to determine when iterations of the loop stop. Two methods come to mind and both include %DO %UNTIL. One involves using this tool to count the number of data sets provided and store it in a macro variable. Once this is determined, use it as an upper bound of a regular %DO loop. Another, which is illustrated here, is to simply generate the code within the %DO %UNTIL loop itself. In %EXAMPLE17 below, for each data set named in the macro parameter (the parameter is delimited by tildes (~)), a data set of the same name but with a 2 at the end is created with the variable-creation logic implemented. This naming convention guarantees unique names of the new data sets while tying each back to its predecessor.

```
%macro example17(datasets=);   /* Example 17 */
%let i=1;
%let dset=%scan(&datasets,&i,~) ;
%do %until(&dset= ) ;
    data &dset.2 ;
    set &dset ;
    if value gt 100 then saleprice=0.8*value ;
    else if value gt 90 then saleprice=0.85*value;
    else saleprice=0.9*value ;
    run;

    %let i=%eval(&i+1) ;
    %let dset=%scan(&datasets,&i,~) ;
%end;
%mend example17 ;
```

Just as SCAN is a useful function for parsing data set variable values, %SCAN helps to parse macro variable values, and is especially common in situations like these when the user is free to supply a list of varying length. We now walk through this one step by step. Following the semicolon that ends the %MACRO statement immediately is a percent sign followed by a macro statement keyword (LET). In fact, after the %MACRO statement are three more macro statements without any open code to be generated. The first %LET statement initiates a macro variable which will ultimately track iterations through the loop. Similarly, the next %LET statement initiates another macro variable which will ultimately hold the name of a data set at each iteration. Following the semicolon that ends %DO %UNTIL though is the character "d". Since it is not a percent sign, it must be the beginning of the open code to generate, which does not end until the %LET statement that follows the text *run;*. At that point, the macro variable *i* is incremented, *dset* is re-assigned, and the loop iterates again. This continues until the value of *dset* is null (i.e. no more data sets to process).

Macros are an efficient and manageable way for us to build flexibility into our programs. In moving from example 5 to example 17, observe what you have accomplished. In order to use %EXAMPLE5, a user had to have a data set called TWO and be content with creating a data set called THREE from TWO. More importantly, TWO was the only data set to which the logic could be implemented, and THREE was the only data set that would be created. In other words, without significant renaming of datasets between macro calls, a user could only implement this logic once. With %EXAMPLE17, the user is no longer tied to data sets named TWO, and no longer restricted to one data set. Other restrictions, however do still exist. For example, the user is still limited to exactly three conditions – VALUE>100, VALUE>90, and VALUE<=90. The user is also stuck with the thresholds – 100 and 90 as well as the multiplying factors to create SALEPRICE – 0.8, 0.85, and 0.9. With some imagination, you can come up with ways to use loops to allow for flexibility in each of these restrictions.

QUOTING FUNCTIONS

Up to this point we have discussed how the OUPUT statement of the DATA step is replaced with the open code to be generated in a macro. We find the OUTPUT statement being executed iteratively, conditionally, and without any logic. We have seen examples of code generated by a macro in the absence of logic as well as iteratively, but generating code conditionally introduces new challenges. For the first time in this paper, the generation of code is part of another macro statement. To the extent that the code to be generated contains special characters or mnemonics that have meaning to the macro processor, this can cause undesirable results. Because this issue can come up in conditional code generation, we first see what the macro facility has to alleviate this problem.

Any characters or mnemonics that have meaning to the macro processor are always assumed to carry that meaning where it would make sense, and are not considered to be part of the generated code, unless instructed otherwise by macro quoting functions. We now return to example 6. Suppose that you want to generate the code by assigning the text of the first two statements to a macro variable, and then referencing the macro variable. Without knowledge of quoting functions, you write a macro as follows.

```
%macro example18 ;  /* Example 18 */
%let x= do onevar=1 to 3 ;  value=ex1(onevar) ;

&x ;
output;
end;
%mend;
```

Your intention was for the first semicolon to be generated as text and the second to end the %LET statement, but the macro processor doesn't know that. The effect is that the first semicolon ends the macro statement, and everything that follows is generated as text. The resulting generated code is the following.

```
value=ex1(onevar) ;
do onevar=1 to 3 ;
output;
end;
```

%STR and %NRSTR are two examples of macro quoting functions, and their arguments are all the text whose special meaning is to be masked so that the macro processor does not use it for its special purpose. In addition to all the characters that %STR masks, %NRSTR also masks ampersands and percent signs so that you can use them in text without tempting the macro processor into thinking they are macro variable references or macro calls and resolving them. The desired result in the above example could have been achieved with the following.

```
%macro example18 ;  /* Still example 18 */
   %let x= do onevar=1 to 3 %str(;) value=ex1(onevar) ;

   &x ;
   output;
   end;
   %mend;
```

The semicolon within the %STR function is part of the text assigned to the macro variable. The semicolon at the end serves the purpose of ending the statement and is not part of the text. For that reason, a semicolon is needed after the reference to the macro variable.

Other characters that may require masking are unmatched parentheses and quotation marks, commas, blanks, and others. Other arguments for these two functions can be text strings that contain mnemonic operators such as OR, AND, EQ, NE, and others. Suppose that the data sets listed in example 17 by the user in the macro parameter are to be delimited by spaces rather than tildes. That means that the third argument of the %SCAN function needs to be a space, but in order to tell the macro processor that the space means something, you must use %STR as follows.

```
   %let dset=%scan(&datasets,&i,%str( )) ;
```

%STR and %NRSTR mask the meaning of special characters and mnemonics at the time that a macro compiles. It is, however, conceivable that the resolution of a macro variable or a function of a macro variable may lead to the processing of unintentional logic, and so may require masking. Since macro variables resolve when a macro executes, not when it compiles, you need functions that mask at execution time. Most common for the masking of macro variable references are %BQUOTE and %NRBQUOTE. Finally, many macro functions such as %SUBSTR and %SCAN have counterparts of the same name preceded by a Q (i.e. %QSUBSTR and %QSCAN) that mask the result of the resolution of the function.

CONDITIONAL LOGIC

We are now ready to turn our attention to conditional code generation. We know that OUTPUT in a DATA step can appear on its own either implicitly (which does not have a macro counterpart) or explicitly, or in a DO loop. We have discussed and illustrated the macro counterpart of generating code without the aid of any logic or within a %DO loop. The only other way to generate an observation in a DATA step is by executing OUTPUT conditionally. Conditional output is the only time that the OUTPUT statement appears as part of another statement. The form is as follows.

IF (*data set variable*) (*operator) (operand)* THEN OUTPUT ;

Conditional generation of open code has the same form.

%IF (*macro variable*) (*operator*) (*operand*) %THEN *code to be generated* ;

Of course the form (ELSE OUTPUT) in the DATA step corresponds to the form (%ELSE *code to be generated*) in the macro.

Up to this point we have said that everything between the macro statements is open code to be generated. The rule of thumb was that everything between the semicolon and the next percent sign that preceded a macro statement keyword was open code to be generated. As noted in the bullet points above, exceptions include macro variable references, macro calls and macro functions whose resolution would be generated code. Again, using the DATA step and particularly, the OUTPUT statement as a model, we now see that the same rules apply to whatever follows %THEN and %ELSE. If what follows %THEN and %ELSE is anything but a macro statement keyword with a preceding percent sign (or macro variable references, macro calls, or macro function calls), then it must be code to be generated. This is true for everything up until the semicolon that ends the statement.

For our first example, suppose you wish to generate the code of example 8 conditionally. In the following example, we expand %EXAMPLE8 to contain more open code, and the text from that example (*OR X2=1*) will be generated conditionally. Of course this text cannot stand alone as a statement so the conditional macro logic will interrupt the open code statement.

```
%macro example19(x=) ;  /* Example 19 */
data this ;
set that ;
if x1 = 1 %if &x=Y %then or x2 = 1 ; ;
run;
%mend ;
```

Several aspects of this example are noteworthy, so let us step through it carefully. Following the semicolon that ends the %MACRO statement is the character “d”, so everything from that point up to the percent sign in the subsetting IF statement is open code to be generated unconditionally. Note that at the point that we reach the percent sign, we have no semicolon, so the subsetting IF statement has not yet ended. Following %THEN is the character “o”, so if the condition based on the macro parameter is met, everything from this point to the next unmasked semicolon will be generated. Now note the consecutive semicolons. We know that the first ends the macro statement. We also know that everything immediately after that first semicolon up to the next macro statement is open code to be generated. Therefore, the second semicolon is to be generated – unconditionally. Why do we need the second semicolon? Because we know that whether the macro condition (&x=Y) is true or not, we are going to need a semicolon generated to end the subsetting IF statement. In the next example we note how this changes when we generate an entire statement conditionally, such as that in example 7.

```
%macro example20(x=) ;  /* Example 20 */
data three ;
  set two ;
  if value gt 100 then saleprice=0.8*value ;
  %if &x=Y %then else if value gt 90 then saleprice=0.85*value %str(;) ;
  else saleprice=0.9*value ;
  run;
%mend ;
```

This one is not quite as hard to look at because the macro logic is not interrupting an open code statement, though it is still disrupting the flow of the DATA step. Once again, following %THEN is the character “e”, so despite the fact that the macro processor has “ELSE functionality,” this ELSE and everything up to the next unmasked semicolon is nothing more than generated text. Notice that on the way there, we run into a percent sign followed by a macro function whose resolution is generated text. In this case, the resolution is a masked semicolon. Because it is masked, it will be generated as code and will not end the macro statement. Because it comes before the unmasked semicolon, it is part of what is conditionally generated. In this case, because an entire statement may or may not be generated, we only want a semicolon generated when the rest of the statement is. Imagine if instead of this, we would have used consecutive semicolons as in the last example. Furthermore, imagine that the macro condition is false. Then a portion of the generated code would be as follows.

```
if value gt 100 then saleprice=0.8*value ;
;
else saleprice=0.9*value ;
```

Because you have an ELSE following an empty statement (the lone semicolon) and not an IF statement, this DATA step would not execute. On the other hand, if you exhaust all possibilities with %ELSE, you can go back to consecutive semicolons.

```
%example21(x=) ;  /* Example 21 */
data three ;
  set two ;
  if value gt 100 then saleprice=0.8*value ;
  %if &x=Y %then else if value gt 90 then saleprice=0.85*value ;
  %else else if value gt 90 then saleprice=0.85*value ; ;
  else saleprice=0.9*value ;
  run;
%mend ;
```

Once again, since a statement is going to be generated one way or another, a semicolon (the second of the consecutive semicolons) can be generated unconditionally.

Often times in the DATA step we find it useful to execute multiple statements under the same condition. For this we have the DO block, and it has the following form.

```
if (data set variable) (operator)(operand) then do ;
DATA step statement 1 ;
DATA step statement 2 ;
more DATA step statements
end;
```

Of course nothing says that DO blocks must contain multiple statements, but it can be a little excessive when they do not. The macro also has %DO blocks of a similar form, but like the rest of the macro and unlike the DATA step DO block, it does not have to be made up exclusively of statements.

```
%if (macro variable) (operator) (operand) %then %do ;
open code
macro statement 1 ;
open code
macro statement 2 ;
more macro statements and open code
%end ;
```

Just like the DATA step DO block, a programmer is not obligated to accomplish several tasks in a %DO block. In fact some prefer using this method of conditionally generating code as opposed to specifying the code immediately after %THEN or %ELSE. One advantage is that you do not have to worry about masking anything because the code to be generated is between macro statements rather than part of the statement. It is also a little easier to identify it as open code. The example below accomplishes the same task as example 20 above.

```
%macro example22(x=) ;  /* Example 22 */
data three ;
  set two ;
  if value gt 100 then saleprice=0.8*value ;
  %if &x=Y %then %do ;
    else if value gt 90 then saleprice=0.85*value  ;
  %end ;
  else saleprice=0.9*value ;
  run;
%mend ;
```

Unlike example 20, the semicolon to be generated no longer needs to be masked. Everything after the semicolon that follows %DO is known to be generated code, including the semicolon. Generally, the more open code there is to generate, the more convenient the %DO block becomes. Generating code such as that contained in examples 1 through 3 with %DO blocks saves you from having to wrap each semicolon in a %STR function.

You now know all the rules. You have always been able to tell the difference between a macro statement and an open code statement by the presence or absence of a percent sign. You now know exactly where in a macro open code is allowed to show up. You know that macros can generate *any* text including text that means something to the macro processor, but you can force such text to be interpreted as text to be generated with the help of quoting functions. You know all the rules, yet the mixture of open and macro code still presents difficulties. You now know where open code *can* go, but it is not always obvious where it *should* go in order to generate it the way you want it. With so much concentration on where to place the open code along with the combination of macro conditional logic and DATA step conditional logic, or macro iterations and DATA step iterations, the line between macro logic and open code logic, particularly DATA step logic tends to get blurry. What is the difference between IF and %IF, DO and %DO? We saw these issues come up in examples 19 through 22. Consider the following situation. You are writing your SUGI paper and need a sample data set to illustrate something. Having no good examples, you decide to make one up in the following manner.

```
data sample ;  /* Example 23 */
  do a1 = 1 to 5 ;
     do a2 = 1 to 5 ;
    random = ranuni(0) ;
    output ;
     end ;
  end;
run;
```

You now have two numeric variables, *a1* and *a2*, that act as classification variables and each combination of the two corresponds to a randomly assigned analysis variable value. Suppose that you decide you want to make this program more flexible by allowing the number of classification variables and hence, the number of DO loops, to vary according to a macro parameter. If the value of the parameter is 4, then four levels of nested loops will be needed. By imposing *a3* and *a4* as the iterating variable names for these additional loops, and generally, *a1* through *an* for *n* loops, you build a pattern of text generation that can be specified for a %DO loop. What is not necessarily obvious is how you can make it interact with the DATA step loops. Does it come before, after, in between, or a combination of all the above?

On the surface, the DATA step DO loop and the macro %DO loop do not look much different. Superficially, one has a percent sign and the other does not; one iterates through a data set variable and the other through a macro variable, but nonetheless, each is the beginning of a block of logic that will execute iteratively. The big difference is how the contents of the block are used. Inside a DATA step DO loop is DATA step statements. These statements are executed sequentially each time the loop iterates. Of course the DATA step is always processing data one record at a time. It is important to note that each iteration processes the same record. Macro loops can contain macro statements as well as text to be generated. Sometimes this text might look like what you would see in a DATA step loop. For example, both of the following are legitimate loops.

```
do i=1 to 5 ;
  output ;
end ;

%do i = 1 %to 5 ;
  output ;
%end ;
```

The difference is this: in the first example, the record currently being processed by the DATA step is being generated as an observation in the new data set five times. In the second, the text *output ;* is being generated five times. Of course, the text happens to be a DATA step statement, but that is irrelevant to the macro processor.

In example 23 above, the number of loop iterations directly translates into the number of DO statements and END statements to be generated in the DATA step (but not the number of assignment statements or OUTPUT statements, each of which should appear only once). Since we are iteratively generating code, we will place that code within the macro loop as follows.

```
%macro example24(varnbr=) ;  /* example 24 */
%do j = 1 %to &varnbr ;
    do a&j=1 to 5 ;
%end ;

    random=ranuni(0)
    output ;

%do j = 1 %to &varnbr ;
    end ;
%end ;
```

We now make a few observations. First, note the use of the iterating macro variable *j*. In the end, we do not want each level of nesting in the DATA step to loop through the same data set variable. We therefore make sure that each iteration of the macro loop generates slightly different text, particularly, the text that will name the iterating variable for a given level of nesting. Second, notice that we went from two DO statements in example 23, to the presence of only one string of text containing *DO* in the macro. In example 24 this difference would have been even more pronounced if without the macro we had more levels of nesting. Of course we use macro loops for this very reason – so that we can provide a pattern of code generation, often times with the help of the iterating macro variable. Of course the convenience of this is somewhat offset by the difficulty in assessing if the code that will be generated is what is needed, as well as difficulties in troubleshooting if the results after execution are not what was expected. Finally, notice the presence of *end ;* followed by *%end ;*. This can be confusing as well as misleading. It is common to find consecutive END statements in every day code, but it is important to know that this is not serving the same purpose. Once again, to the macro processor, the presence of the text string *end;* within the macro loop is nothing more than just that – text to be generated, and has no meaning beyond that.

The same kind of confusion can arise with conditional logic. Macro conditional logic differs from DATA step conditional logic partly by what is being compared. Just as the contents of a macro loop might look like what a DATA step loop can contain, the same can be said of what is compared with conditional logic. For example, the following are both legitimate.

```
if &x=y then etc ;
%if &x=y %then etc ;
```

Though the first contains a macro variable, the absence of percent signs preceding IF and THEN make this a DATA step conditional statement. The macro processor resolves the macro variable *x*. As long as the text stored in this macro variable is the name of a variable in the data set being processed, this statement will execute in a DATA step. Specifically, the DATA step will check the current value of this variable, and if it is the same as the current value of the variable Y, the statement following THEN will execute. The second is a macro statement, which means that it is not necessarily part of a DATA step. The macro processor is simply checking to see if the text stored in the macro variable is *y*.

Similarly, the following are both legitimate.

```
if x=y then etc
%if x=y %then etc
```

The second is not very common because it is never true. It is testing to see if the text string *x* is the same as the text string *y*. Of course the DATA step conditional is checking to see if the values stored in each of the variables for the current record being processed are the same.

We could go through this exercise, comparing what follows THEN with what follows %THEN, but at this point maybe you can see what is coming. Following THEN in the DATA step is a DATA step statement. Following %THEN can either be a macro statement, which we know is preceded by a percent sign, or text to be generated. That text could look like a DATA step statement, but since it is not a macro statement, to the macro processor, it is nothing more than text to be generated.

A FINAL EXAMPLE

Before concluding, we now derive a macro from start to finish that implements a little of everything we have covered up to this point, including the thinking process behind building variability into an otherwise static piece of a program. In this final example we use SQL to count the number of unique values of the variable SUBJECT by some grouping variables. If you are not very comfortable with SQL, it's ok – that is not an emphasis in the example. Also, we will build this macro step by step, so that even if the later steps get difficult to follow, you will still have gained something in the initial steps.

The DATA step is capable of just about everything that PROC SQL is, but certain tasks can be accomplished a little more compactly with PROC SQL, and counting unique values may be one of those tasks. Because not everyone in your department is familiar with SQL, you decide to write a macro. In Version 1 of this macro, the user is allowed to specify three grouping variables through macro variables. The code to be generated should have the following form.

```
proc sql ;
create table sample2 as
select var1, var2, var3, count(distinct subject) as subject
from sample1
group by var1, var2, var3 ;
quit ;
```

Of course several approaches are possible, and if this was all the text generation that was necessary, you might select something else, but because we will be building on to this macro, for the sake of illustration, we will make use of a %DO loop. If all that varies are the three grouping variables, then the generation of this text might be all that you include in the macro.

```
%macro version1(gvar1= , gvar2= , gvar3= );  /* Example 25 */
%do x=1 %to 3 ;
  &&gvar&x
  %if &x<3 %then , ;
%end;
%mend;
```

Notice that a comma is generated after the generation of the macro variable value only on the first two iterations of the loop. This is because it is not wanted in the GROUP BY clause. With this macro the rest of the SQL could have been written outside of any macro, and this macro is called only when this text needs to be generated.

```
select %version1, count(distinct subject) as subject
.
.
group by %version1 ;
```

Notice the presence of a comma after the invocation of the macro in the SELECT clause, but the absence of a comma in the invocation in the GROUP BY clause.

For Version 2, suppose you want to create subtotals for each value of the first variable, plus each combination of values for the first and second variables. Again, PROC SQL allows us to achieve this quickly and efficiently with the help of the UNION operator to concatenate results of multiple queries. The desired code to be generated is below.

```
proc sql ;
create table sample2 as

select var1, var2, var3, count(distinct subject) as subject
from sample1
group by var1, var2, var3
union

select var1, var2, 'TOTAL' as var3, count(distinct subject) as subject
from sample1
group by var1,var2            /* Subtotals for var1-var2 combination */
union

select var1, 'TOTAL' as var2, 'TOTAL' as var3, count(distinct subject) as subject
from sample1
group by var1 ;         /* Subtotals for var1 */
quit ;
```

In example 25 we used a loop to generate the SELECT variables, making use of the pattern in the names of the macro variables. In this case other patterns can be exploited. First, note that each query beginning with SELECT and ending with GROUP BY contains similar text. This suggests a loop that contains an entire query.

```
%macro version2(gvar1= , gvar2= , gvar3= );
proc sql ;
create table sample2 as
%do x = 1 %to 3 ;
  query code
%end ;
quit ;
%mend ;
```

For a second pattern, notice from one query to the next that the number of text strings in the SELECT list of the form *varx,* decreases from 3 in the first iteration to 1 in the third iteration, and is always equal to 4 - &x. For a third pattern, notice that the number of text strings of the form *'TOTAL' as varx,* increases from 0 to 2. Each pattern suggests a sub-loop within the query code. One will begin at 1 and stop at 4 - &x. The second will pick up where the first left off, starting at one more than upper limit of the first and stopping at 3. For a fourth pattern, the number of variables listed in GROUP BY deceases by 1 as &x increases by one. The text string UNION is generated while &x is under its limit.

```
%macro version2(gvar1= , gvar2= , gvar3= );
proc sql ;
create table sample2 as
%do x = 1 %to 3 ;
    select
    %do y = 1 %to %eval(4 - &x) ;
       &&gvar&y,
    %end;

    %if %eval(4 - &x)<3 %then %do z = %eval(4 - &x + 1) %to 3 ;
       'TOTAL' as &&gvar&z,
    %end ;

    count(distinct subject) as subject
    from sample1
    group by

    %do y = 1 %to %eval(4 - &x) ;
       &&gvar&y
       %if &y<%eval(4 - &x) %then , ;
    %end;

    %if &x<3 %then union ;
%end ;
; quit ;
%mend ;
```

The fact that using macros to generate open code has a tendency to break up open code has been a central theme within this paper and nowhere is it better illustrated than in this example. We have also pointed out that because %DO loops generate open code according to a pattern defined by the macro programmer, it can be much more difficult to predict what open code will be generated. Again, with four loops, three being nested inside one, this fact becomes evident in this example. For that reason, when I want to predict what code will be generated from %DO loops, I pick a value for the iterating variables and see what they generate. To walk through this example, let's see what comes out of the first iteration of the outer loop (&x=1). This means that the first inner loop has an upper bound of 3 (4 - 1). On the first iteration of this inner loop, &&var&y resolves first to &var1 (because &y=1), and then to the first variable supplied by the user in the macro parameter. A comma is also generated. The same is true on the second and third iterations of this loop.

The second inner loop is executed only under the condition that the expression (4 - &x), which in this case is 3, is less than 3. Since this condition is false, the text string of the form *'TOTAL' as &&var&z* will not be generated on this iteration. Following some trivial code generation, we come back to the same inner loop as we had earlier. The difference is that inside this loop, the text that results from the resolution of &&var&y is generated unconditionally, while a comma is generated only for all but the final iteration of this loop. This conditional generation of the comma wasn't necessary in the first inner loop because we knew that if nothing else, the text string *COUNT(DISTINCT SUBJECT) AS SUBJECT* would follow in the SELECT list. Finally, because we are not on the last iteration of the outer loop, the text string UNION is generated. By putting this all together and comparing it to the open code above, we see that this macro generates exactly what is needed at least for the first query. Since the second inner loop did not execute for this iteration, it might be wise to run through this exercise for another iteration. Finally, once we escape the outer loop, the semicolon that ends the CREATE TABLE statement is generated, followed by the QUIT statement.

In moving from Version 1 to Version 2, we have added more information to the output but we have not added any more flexibility for the user. This I leave for you in Version 3. At this point, we are still stuck with forcing upon the user exactly three grouping variables. Try leaving this to the user. Rather than having three macro parameters, you may have one in which the user specifies as many grouping variables as desired. It is then up to the macro to determine how many grouping variables there are. Whatever it is, this will have to be an upper bound on the first loop. The number 4 in the current argument of the %EVAL function would have to be a function of this. With more practice, you will gain more comfort in allowing more and more flexibility to the user.

CONCLUSION

We build macros to make a program more flexible, usable under more circumstances and conditions, more convenient. When it comes down to it, the convenience we are providing for the user, or the burden from which we

are alleviating them is now becoming our own. As we shift responsibility away from them over to ourselves, we are imposing on ourselves the responsibility not only of knowing what code will get the job done, but knowing which sets of code will get all the jobs done, and how to tell the SAS system how to build the code for each specified circumstance. Like trying to carry on two different conversations with two different people at the same time, it is difficult to give attention to two different thought processes at the same time, and as we concentrate on one, we lose focus on the other. Concepts we took for granted before we ever started writing macros like conditional logic now become jumbled in our minds because the "person on the other line" is talking about the same thing in a completely different context. This is why I wrote this paper. Macros can still be tough to write – even for the world's greatest – but knowing the rules and always concentrating on who you are talking to is a great start.

CONTACT INFORMATION

Please feel free to contact me with any comments and questions.

Mike Molter
Howard M. Proskin and Associates
Rochester, NY 14612
(585) 359-2420
mmolter@hmproskin.com