Paper 237-30

HOLY MACRO! An Intuitive Approach to Understanding the Macro Facility

Michael J. Molter, Howard M Proskin and Associates, Rochester, NY

ABSTRACT

The macro facility allows us to write macro code that instructs SAS® how to generate the more familiar open code found in DATA steps and PROCs. With such a tool, we can consolidate families of related programs, execute the same piece of code repetitively, and develop applications for others to use. This replaces error-prone, high-maintenance jobs such as saving multiple versions of essentially the same program (with slight differences depending on the task) or excessive typing or copying and pasting. The best part is that most of the code is similar to what we find in open code. The key to understanding how the macro processor works is in understanding the differences – both in the code itself, and in the way the two types of code are processed. This theme will be emphasized throughout the paper.

Discussion begins with macro variables, beginning with initialization and referencing, followed by other subjects such as functions, quoting, and combinations. Macros are then presented, beginning with creation and referencing. Examples begin with "open-code" macros followed by a discussion of macro logic and examples of its use with open code in "mixed-code" macros. Finally, I will discuss different ways macros are written and used, depending on who they are written for. This paper is perfect for programmers who are comfortable writing in BASE SAS and are ready to take the next step toward automating their programs. It is applicable in any operating system.

INTRODUCTION

In your first algebra class one of the first exercises you learn is to evaluate an algebraic expression for a given value of the variable. For example, evaluate the expression *3x+2* when *x=5*. The idea is to take the given value of *x*, plug it into the expression and see what you get. At first, the way this exercise is expressed just seems like a fancy way of saying "triple 5 and then add 2". It is only when you start plugging in several different values of *x* and start understanding what a variable is, that you start to appreciate the value of this type of exercise. The macro facility is not much different. Imagine replacing the constants 3 and 2 in the algebraic expression with valid SAS code. The expression *3x+2* has now been replaced with a SAS program, but what does the *x* from the algebraic expression translate into? This is where the macro facility comes in. Now imagine that *x* can play a role in the program similar to that in the algebraic expression, except instead of a range of numbers constituting the valid values, imagine that different pieces of valid SAS code constitute valid values. By valid, I mean anything that when substituted into the "variable" generates syntactically correct SAS code. So numerical values are valid if the "variable" is in a part of the program where substitution of a number makes sense. Other times, substitution of a single word, a statement, or even groups of DATA steps and PROCs may be valid, depending on the reference. The analogy is now complete. The constants 3 and 2 correspond to "open" SAS code. The variable *x* in the algebraic expression, a stand-in for a range of numbers, now corresponds to a macro facility reference, a stand-in for SAS code. The exercise of replacing the variable with the given value and evaluating the expression corresponds to actual execution of the program. You see *x* and replace it with 5. The SAS system sees a macro facility reference, calls on the macro facility, generates the code, and executes the program.

Sounds simple, right? A SAS program laden with macro facility references is like an algebraic expression laden with variables. The process of replacing those macro facility references is the same as replacing the algebraic variables with a given value. On the surface, the two exercises are fundamentally the same. Below the surface, two parts of this exercise, referencing and definition, begin to illustrate why code-generation is a non-trivial task.

Referencing refers to the way you invoke or call the macro facility. In an algebraic expression, you use letters to represent variables because they are easily distinguishable from the numbers. However letters and numbers can be found in open SAS code. A macro facility reference represented only by a letter, a number, or even combinations of the two would not be distinguishable from open code. For that reason, though references to the macro facility contain letters or combinations of letters and numbers, when used in a program, they must be preceded by special characters known only to the macro facility. Not only that, but combinations of macro facility references and even combinations of these special characters can be used to provide specific instructions to the macro facility on how to substitute. Since a reference is replaced with SAS code, it can be used in any part of a program where the substitution results in valid SAS syntax.

Definition refers to the way you tell the SAS system which value is to be substituted. In the algebraic expression, the part of the exercise that specified "when *x=5*" told you which value in the range of *x* to substitute for *x.* But how do

you tell the SAS system which value to substitute for the reference, or in other words, what code to generate? The answer is not much different from the way you always communicate with the SAS system. We tell the SAS system what data sets to create, what output to produce, and what analyses to run through open code. Ironically, you tell the macro facility how to generate code through more code – macro code. Without macro code, almost all SAS code revolves around a set of data. If you are not creating a data set with a DATA step, you are using one in a PROC. The exception to this rule is the set of global statements such as TITLE, ODS, and LIBNAME, which help set up the environment. Just as the choice of 5 for *x* was made independent of the algebraic expression, defining the way code is generated is generally independent of the rest of the program and any data. For that reason, these definitions tend to exist outside of DATA steps and PROCs.

Up to this point, I have used the term "macro facility reference" as a generic term to mean a call to the macro facility for a substitution of code, but gave no indication as to how the macro facility knows what to substitute. I am now ready to be more specific. The macro facility gives us two ways to define through macro code how open code is generated. One way is by storing the code itself in memory. Macro variables are units of memory where you can store text. Part of initializing (or defining) a macro variable involves naming it. Once this is done, you can generate within open code the stored text by referring to its name preceded by an ampersand, the special character for macro variables, in the place where the substitution is to take place. The second way is by storing logic that produces the code. A collection of such logic is called a macro. Again, part of defining a macro involves naming it. After that, just as with macro variables, you can refer to it in code by name, but preceded by a percent sign rather than an ampersand. In both cases, before the DATA step or PROC in which a reference is found can be processed, the substitutions must be made.

The good news is that macro code is similar to DATA step code. Both follow a similar syntax that includes semicolons at the end of statements. Both have statements that allow you to direct messages to the log, create and initialize new variables, and define windows. Both contain IF-THEN logic to execute such statements conditionally, as well as DO-END blocks to execute several of them conditionally. The data set contains variables that reflect variability within a set of data, the reason you analyze data. The macro facility contains macro variables that reflect coding variability within a program, which reflects the changing circumstances that dictate what code to execute. In the DATA step, it is often the value of a data set variable that determines the result of a statement and ultimately the observation that is written to the new data set. Similarly, it is the value of a macro variable that determines the code that is generated. In addition, data set variable functions provide you with information about the value of a variable on any given observation while macro variable functions provide you with similar information about the value of a macro variable. Finally, the DATA step has iterative DO blocks that allow you to process a set of DATA step statements with each iteration of a data set variable, while similar DO blocks process macro statements and generate code with each iteration of a macro variable.

It is convenient that the means to the end for creating a data set is similar to that for creating code. Because of that, this paper will attempt to help you develop an understanding of the macro facility by drawing on your knowledge of the DATA step. In order to fully develop this understanding, you must learn where the similarities stop and the differences begin. Data step code is geared toward generating observations for a new data set, and macro code is geared toward generating open code. Because the code found in a DATA step applies to all observations being created, it must be processed iteratively or once per data record being read. Because macro code defines how to generate code, it does not have to be processed iteratively. Additionally, because macro code generates open code and open code is what is needed to execute a DATA step or a PROC, macro facility references must be defined and resolved prior to execution of the DATA step or PROC.

As the title suggests, this paper is intended to guide you through the workings of the macro facility using your intuition. It is assumed that you have a solid knowledge of what is available in the DATA step. We have discussed some of the similarities between macro code and DATA step code as well as differences. Throughout this paper we will continue to draw upon those similarities while continuing to emphasize the differences. In the end, your view of SAS code as a series of DATA steps, PROCs, and a few scattered global statements will be expanded. This is not intended to be a fully comprehensive instruction on the entire macro facility. Rather, the focus will be split between macro variables and macros. Other topics such as system options or macro statements will only be discussed within these contexts. The next section will discuss why you need code-generation capabilities and how they work. Following that, we will get into macro variable specifics. We will start with features and properties of macro variables, and then move into detailed discussions on definition and referencing. Comparisons with data set variables will be used throughout. Following that will be a section on macros, again detailing properties, features, definition and referencing. Comparisons to DATA step logic will be used throughout.

WHY DO YOU NEED CODE GENERATORS?

One advantage of macro facility references is that they allow you to broaden the scope of your programs by accepting and controlling changing circumstances in an organized way. Often times you find that while the general

purpose of a program is useful under many circumstances, each of those many circumstances has slightly different criteria, which requires slightly different code, but how do you change that code? If the program is small enough and the number of places in the program that require change is small enough and if the number of users with write-access to make these changes is under control, then it may be safe to scan through the program either manually or by searching and making the changes. This may mean replacing old code with new, adding new code but commenting old, or making changes and saving the program under a different name. Generally speaking, for many reasons this is not a good practice. Programs are often too long and many times the task is spread across multiple programs. Also, the primary user of the program who has the responsibility of implementing the new criteria is not the person who wrote the program, and may not be someone who should be changing code. The macro facility allows you to accommodate multiple circumstances without having to make changes to the core of the program. This is achieved by dividing a program or an application into three distinct pieces. One piece accepts user input, usually through initialization of macro variables. A second place, if necessary, is macro code that defines how user input generates code. The third piece is the core code. Rather than writing code in the core that is subject to change, the programmer replaces each of those pieces with a reference to the appropriate definition. Upon execution, when this reference is reached, code is generated according to the definition being referenced. With this setup, the input piece is designed for straightforward specification of the criteria by users and drives code generation throughout the program,

In our first example, an application is designed to provide statistical summaries of all employees whose salary is at a chosen threshold or higher. Because the database contains a data set for each department, several DATA steps must be written to extract data for all employees whose salary qualifies, each filtering on the chosen threshold. In addition, TITLE statements are issued that each states the threshold. Rather than asking users to find all places where this number is to be part of the code, you create a macro variable in the beginning of the program where they can enter a value. Each filter in each DATA step as well as each TITLE statement will now contain a reference to this macro variable, and upon execution, that code will be generated in these spots.

This example is characterized by open code containing references to a macro variable whose value is directly set by the user. Though any user could scan through the program and insert their choice of threshold, this use of macro facility references offers the advantage of making changes in one place instead of many, reducing the likelihood of overlooking substitutions. In this case the generated code represents a data set variable value, but this does not have to be the case. This approach can be useful whenever a programmer feels that the code to be generated can be specified by the user. For example, if a group of users is familiar with variables in a data set, the programmer can create a user-defined macro variable to be referenced in a KEEP statement. If users are familiar with the statistics keywords that the MEANS procedure recognizes, then a user-defined macro variable populated with these can be referenced in PROC MEANS. If users are familiar with the directory structure, they can specify the directory path where they want data sets to be saved in a macro variable to be referenced in a LIBNAME statement.

For our second example, we return to the situation in example 1. Because the macro variable was to be used as a filtering value of a numerical data set variable, users were instructed to supply only the number, without other characters such as dollar signs and commas. You now realize that you do want these formatted values in TITLE statements. One solution is to ask users to initialize two macro variables. The first is the same as in the first example, the unformatted value, and the second is the formatted value. For example, the macro variable NBR1 is initialized at 30000, and the macro variable NBR2 is initialized at $30,000. A second option is to ask users for one of these and through macro functions and/or macro logic, derive the other. Something to consider when choosing one of these options is the fact is that the more you rely on user input, the higher the likelihood for error. If you allow users to initialize both, they may forget to specify one in which case the titles do not reflect the actual summaries generated. Inconsistency is also a possibility if, for example, commas or dollar signs are sometimes neglected. In instances where the input is not numerical, inconsistencies in spelling or case can also occur. Generally speaking, deriving values from minimal user input generates more consistent code than relying on that from users.

While code such as data set variable names or values or data set names may be reasonable for users to supply as part of the criteria specification, sometimes the code to be generated is too long or too technical to expect users to specify. In such cases, you define for users meaningful keys to use for input. You then use these keys to create links between the input and the code to be generated. This can be accomplished using a combination of macros, macro functions, and other macro variables. For our third example, consider an application that creates daily, weekly, monthly, or annual reports based on user input, but the layout, and therefore the TABLE statement in the TABULATE procedure for each type is different. Of course, not only is the text used to construct a TABLE statement too complicated to ask a user to specify, but there is also no meaningful relationship between such text and the type of report it is supposed to represent. For that reason, you create a macro variable in the beginning of the program where users enter a value such as D, W, M, and A (for Daily, Weekly, Monthly, and Annually respectively), only this time, instead of generating code in the core of the program from that value, you generate code based on macro logic that links these keys with their respective text. The advantage is obvious for users who are not programmers and do

not know how to construct a TABLE statement, but even the world's most knowledgeable experts on PROC TABULATE draw benefits. Every possible TABLE statement only has to be thought about and written once. Once these are part of the macro definition, every subsequent call of the macro consistently generates the defined code.

Why use D, W, M, and A instead of asking users to spell out the words? There is no technical reason why you cannot use the words for input, but keep in mind that this is more vulnerable to differences in spelling and casing. For example, if your macro code constructs the TABLE statement based on user input values of "Annual", "Weekly", "Monthly", and "Daily", then nothing will be generated for users that specify "Annually", "Weeky", "DAILY", or "monthly". In such a case, extra caution should be taken when specifying in your macro logic the conditions to be met to generate the TABLE statement. For example, you might consider executing based on the UPCASE of the first character of the input. This would catch all misspellings that at least started with the correct character, and would also erase any concerns regarding casing.

Sometimes you may decide to create input keys for the user, not because the generated code is too technical or too long, but simply to create consistency in output. Consider example 4 with the same situation as example 3, but now you do allow users to spell out the words rather than supply the first letter. As discussed, proper care has been taken to ensure that the TABLE statement will be generated, even from different spellings and casings of the input. Unfortunately, it is discovered that the inconsistencies in the input has led to inconsistencies in the titles on these reports, generated from TITLE statements containing references to the macro variables used for input. For example, some daily reports contain the text "Daily reports" in their title, some contain "DAILY reports", and some contain "daily reports". To create consistency across reports as well as within the title, you derive a macro variable from the input with the spelling and the casing you require for all reports.

The examples in this section thus far have involved code generation in relatively small sizes, but there is no limit on the amount of code the macro facility can generate. As the scope of an application grows to handle a wider variety of different tasks, it may be entire sections of code consisting of multiple DATA steps and PROCs that change. For our fifth example, consider an application that creates a series of reports for several divisions of a company. The reports are to look the same regardless of which division they represent, but each division stores their data in much different ways. One division stores all of their data in a SAS database in which each data differs from the others according to the year it represents. A second division stores all of their data in an Oracle database and tables differ from each other not by time, but by the class of parameters they represent. For example, one table represents demographic information while another represents sales information. A third division uses a DB2 database and also separates their tables by parameter groups, although not the same as the second division. Additionally, variables in different databases that appear to represent the same parameter may use different values from one database to the next (e.g. "F" and "M" represent GENDER in one database whereas "1" and "2" are used in another). Not only should reports from different divisions look the same but they should also represent the same data. For example a report by age should calculate age consistently across the divisions. The general purpose of the application is to extract data from the appropriate database based on the division supplied by the user. Following extraction is whatever code is needed to transform data from that database into a common set of variables with common definitions. After this, a common set of code will create the reports regardless of the data source.

Conceptually, this example is not much different from the last two examples. The programmer provides keys for users to indicate a division, and macro code is used to link them with generated code. The difference between this example and example 3 is the size of the task that is subject to change, which reflects the amount of code necessary to generate. In example 3, all that differed between the types of reports was the layout of the table. From a programming perspective, all that differed was one statement. In example 5, it was the entire preparation of the data that differs between divisions. With division 1 you can use a DATA step to extract, but with divisions 2 and 3 you need to run SQL native to the environment. After that, each division will require its own code to create the common variables based on variable names, values, and definitions used in the database from which it came.

Through many examples, we have now seen that code that is likely to change from one execution of a program to the next based on different criteria and circumstances is a good candidate for being generated according to a previously-defined specification or algorithm. Such code can represent values of a data set variable, names of data set variables or data sets, entire statements or even sections of code. Sometimes, it is more than just the content of the code that needs to be generated, but also the quantity. Sections of code that contain repeated patterns can be generated by defining the pattern in a macro. For example 6, we return to example 1 where we extracted from multiple data sets those whose salaries exceeded a threshold. Suppose you have fifty data sets from which you need to extract. One way to write this part of the program would be to write the DATA step, copy it, and paste it forty-nine times. After that, you have to change the data set name in each SET statement as well as each DATA statement. On the other hand, if you can use a predictable text pattern to name each of the fifty data sets being read and another pattern to name each of the fifty being created, then you can use the iterations of a macro DO loop to define a general pattern for this DATA step. Calling the macro will generate the DATA step as many times as the

loop iterates. The macro variable references present in the DATA step that represent the number of the current iteration will be replaced by that number as the macro executes. This way, any spec changes can be made in the pattern definition rather than making the change in all fifty DATA steps.

Of course you can generate repeating patterns of any amount of code. In the current example, we have generated a DATA step fifty times. In order to generate summaries based on all employees, we need to concatenate all fifty data sets that were just created. For this there is no need to generate fifty DATA steps but we do need to generate the names of all fifty data sets in the SET statement. Again, we use the iterations of a macro DO loop to define the pattern of data set names to be listed in the SET statement. More details on macro DO loops will be provided in the Macros section.

We generate code for two main reasons. One is to accommodate regularly changing criteria in a way that preserves the general purpose of the application while leaving the core code untouched. A second is that the necessary code contains repeated patterns. By supplying the macro facility with the block of code that needs repeating and a definition of the pattern, you can save yourself the time of either typing or copying and pasting all the necessary occurrences of the block and manually changing whatever differs between them. Sometimes both of the reasons are applicable. In such cases, a predictable pattern may exist but the number of iterations may vary from one execution to the next based on user input. For our seventh and final example of this section, suppose a summary of a data set variable called VAR1 is to be created in which users choose the statistics to be computed by listing them in a macro variable assignment, each separated by spaces. For each statistic requested, the resulting variable in the output data set will be named by the name of the keyword that represents the statistic preceded by the text VAR1 (e.g. max=VAR1max), specified in the OUTPUT statement. Because you have given users the flexibility to specify as many statistics as they want, you have no upper bound for an iterative DO loop and so cannot include any set number of variable naming specifications of the form *stat=VAR1stat* in the OUTPUT statement. However just as the DATA step has DO-WHILE and DO-UNTIL loops to handle these kinds of situations, the macro facility has corresponding functionality. With the help of %SCAN (see Macro Functions section later) to parse the input macro variable into the individual statistics, the code to request all of the statistics can be generated. Alternatively, macro variables can be created to represent each of the statistics requested, as well as one that counts them. This last macro variable can then be used as an upper bound for an iterative DO loop to generate the code.

HOW DOES IT WORK?

At some point prior to referencing a macro or a macro variable, it has to be defined. Standard practice is to define macros outside of steps since they are usually independent of any data. References to macros defined within steps will be resolved as long as they follow the definition. However in the DATA step, if a definition follows a reference, the reference will not be resolved for ANY of the iterations of the DATA step. In this paper, all discussions on macros will assume it has been defined outside any step. All macros are defined by the percent sign followed by the string MACRO. Macro variables are also generally defined outside of steps (we will see exceptions to this rule in the Macro Variables section), either within macros (in which case their use may be limited by the environment in which they were defined) or in open code (in which case their use is unlimited). Details on defining macros and macro variables will be discussed later in their respective sections.

Almost every key on your keyboard, when found in a SAS program, is accepted as literal text by the SAS system. Alone they may not mean much, but certain combinations of them have special meaning within the SAS syntax. For example, the combination of the letters D-A-T-A in the beginning of a program or following a semicolon tell the SAS system that the following text will name a new data set. The combination of the letters K-E-E-P following a semicolon tells the SAS system before execution the list of variables to keep in the data set it is about to create and to make room for them in the program data vector. The two keys NOT always accepted as literals are the ampersand and the percent sign. When the SAS system sees either of these two characters followed immediately by text allowable in naming macro variables and macros (mostly letters, underscores and numerals, though numerals cannot begin the name of a macro variable or macro), then the combination is not part of the code to be compiled or executed. Rather, it is a call to the macro facility to generate code in its place. The exceptions to this are %INCLUDE, %LIST, and %RUN which are not executed with the macro facility.

You know that the steps (DATA steps and PROCs) that make up a program are executed sequentially, but once inside any given step, compile-time statements are processed before execution-type statements. For example, before creating observations, the SAS system has to survey the situation or look over the code in order to set up a program data vector. If no macro facility references are present, then the SAS system can proceed through compiling, executing and moving to the next step. On the other hand, once the SAS system reaches a step that does contain macro facility references, before executing or even compiling any code, the SAS system has to know what the code is! Therefore when an ampersand or a percent sign is seen, the macro facility is called in to generate code. When the character is an ampersand, the macro facility looks through its list of macro variables for one named by the string of characters that follow the ampersand. When the character is a percent sign, a few possibilities exist. One

possibility is that the string of characters that follows the percent sign matches that of a macro function. If an argument of a function contains a reference to a macro variable, the list of macro variables is consulted to resolve it. Either way, the macro facility evaluates the function after macro variables have been resolved, and generates the result at the reference. A second possibility is that if the string of characters does not match that of any keywords for definitions or functions, but is allowable for naming macros, then it must represent a macro. Again, the macro facility is consulted to generate the code. Of course these calls to the macro facility do not have to be found within steps. When a macro contains entire steps, it can be found on its own, independent of any step. Also, global statements such as TITLE and LIBNAME can contain macro variables and functions of them.

MACRO VARIABLES

With a conceptual idea of what a macro variable is, we are now ready to dive into the details. We will start by observing some of the high-level similarities and differences between macro variables and data set variables. For starters, they are both variables. That means that both of them are representations of variability whose values can change with circumstances. Second, you can work with each type of variability with SAS code. With this code, you can create new variables whose values are either literals or based on other variable values. In the case of the latter, you have text functions as well as mathematical functions. You can also create both kinds of variables with iterative DO loops and use their values as conditions on which to base further action. Where they start to differ is in what they represent and what their ultimate goal is.

While a macro variable represents variability in potential code, the data set variable represents variability among observations of a data set. Because of its tie to the data set, a data set variable must be created in the DATA step, making use of the iterative process to define values for every record of data being read and observation being created. As a result of this, the value of a data set variable value is in memory only as long as the corresponding iteration is processed. With the help of RETAIN it stays in memory longer, but it is gone after the DATA step is completed. Macro variables are generally independent of data, and so do not need to be created within a DATA step. This means that its value is remembered not only across iterations of a DATA step, but also outside of DATA steps and across steps of a program until explicitly changed. References to data set variables in a DATA step are references to the value of the variable for the observation under consideration. In a PROC, these references pertain to the variability of the data set being analyzed. Macro variables references are references to strings of text, possibly to be used as code. References to data set variables and their variability can only be made in areas of steps where the syntax allows. For example, in several procedures, you obtain descriptive statistics on a data set variable by referencing it in the VAR statement. You separate analyses according to the values of a data set variable by referencing it in a BY statement. You create a data set variable by referencing it immediately after a semicolon and following it with an equal sign. If a string of text is the same as that of a data set variable name and it is found anywhere else such as within quotation marks of a FOOTNOTE statement or following DATA in a DATA statement, it is not a reference to the variable. **However, since a macro variable is nothing more than a string of text, its definition presumes nothing about how it will be used**. For that reason, a macro variable reference can be found anywhere in code, as long as resolution of it leads to syntactically correct code. For example, a reference to a macro variable might be found in a DATA statement to name a data set and in a CLASS statement in a subsequent PROC TABULATE if its value happens to coincide with the name of a variable found in the data set being analyzed, as is the case in the following piece of code.

```
data &sample ;
data step code ;
keep &sample other variables ;
run;

title “Average &sample by department for 2004” ;
proc tabulate data=&sample ;
class &sample ;
other statements ;
run;
```

Here the macro variable SAMPLE serves two purposes. First, its value matches the string of text used for a data set variable name. For that reason, it is reasonable to find it wherever you might find data set variable references, such as the KEEP statement in the DATA step and the CLASS statement of the PROC. Second, the same text string, or the same value of the macro variable is being used to name a data set. With this example we see the value of a macro variable being used across multiple steps of a program, and used in different ways. In all cases, the effect is the same – code generation. On the other hand, the use of a text string inside a TITLE statement or to name a data set that happens to match the name of a data set variable is in no way a reference to the variability represented by that variable.

Data set variables are the object of our analysis. We also use them for other reasons while creating data sets, such as filtering, creating other data set variables, iterative DO loops, etc., but in the end, it is the variability within a group

of data that drives our analyses and the decisions we make based on them. Though by definition, macro variables vary in their values too, we usually are not interested in analyzing them. Their greatest contribution is in helping us create code. To the extent that they share characteristics, the SAS system provides us with functionality for macro variables similar to that of their counterparts. In these cases, we can use what we know about data set variables to help us learn more about using macro variables.

PROPERTIES

Because macro variables are text strings by definition and primarily used for code substitution, they do not have the attributes that data set variables have such as length, formats, informats, and most notably, type. Unlike data set variables that can be numeric or character, macro variables are always character. Just as data set variables can be defined as literals or in terms of other data set variables, macro variables can be defined as literals or in terms of other macro variables. The difference is in how literals are distinguished from variable references. With data set variables, literals are marked by quotation marks. You use quotation marks when you define them (e.g. NAME="Mike") and everywhere you refer to such values (e.g. if NAME="Mike"). Anything is fair game within the quotation marks too. In other words, special characters (e.g. semicolons and spaces) and mnemonics (OR and AND) or anything else that may otherwise have special meaning outside of quotes are masked or considered part of text within quotation marks. The absence of quotation marks represents a reference to other data set variables or a function of them. Quotation marks can also mask macro variable values but unlike data set variables, when used to define macro variables, they are included as part of the text value. Also unlike data set variables, they are not required to define a macro variable as a literal. While the absence of a marker indicates a reference to another data set variable or a function in the DATA step, reference to another macro variable is marked by an ampersand. Additionally, macro functions of macro variables are marked with percent signs. The absence of such a marker is interpreted as a literal by the macro facility. Consider the difference in the way quotation marks are used in the following table. Specifics of the macro statements and functions will be discussed in later sections.

DATA Step Statement	Macro Facility Statement
`DEPARTMENT="SALES" ;`	`%LET DEPARTMENT=SALES ;`
`IF DEPARTMENT="SALES" THEN …`	`%IF &DEPARTMENT=SALES %THEN …`
`PUT "THE DEPARTMENT OF THE MONTH IS " DEPARTMENT ;`	`%PUT THE DEPARTMENT OF THE MONTH IS &DEPARTMENT ;`
`SCAN(DEPARTMENT,1,"L")`	`%SCAN(&DEPARTMENT,1,L)`

Of course in addition to marking literals, quotation marks also mask special characters and mnemonics. Consider the following DATA step statement.

```
VAR3="PENCILS ; ERASERS" ;
```

Because of the quotation marks, the DATA step is not tempted to end the statement at the semicolon that follows PENCILS. Without quotation marks, how does the macro facility know which semicolons to interpret as part of the text, and which indicates the end of the statement? For another example, consider a macro statement that conditionally executes a statement based on the value of a macro variable. Suppose the value of the macro variable contains a space followed by OR followed by another space. OR is a logical operator in the macro facility. Does the macro facility interpret this as text or as a logical operator? Finally, consider a macro function that expects three arguments, but one of the arguments is populated with a macro variable whose resolution contains a comma. How do we tell the macro facility to interpret this comma as text rather than an argument delimiter? Just as quotation marks mask the meanings of special characters in the DATA step, macro quoting functions are available to clear up ambiguity that can arise from situations similar to those mentioned here.

Another consequence is that you can assign numbers to macro variables, but again, it is only the numeral text that is assigned and not any quantity as with numeric data set variables. To extend that idea, you know that the DATA step statement VAR2=8+4 will store the value 12 in VAR2, but storing the string "8+4" in the macro variable VAR2 will only store the sequence of characters, not the arithmetic result in the macro variable. We will see later that the macro facility does have special functions that allow for such calculations.

With the help of macro quoting functions, macro variables can be as flexible as you need them to be in terms of possible values, but you need to determine what the best way is to use that flexibility for a given application. Macro variable values can be one "word" (e.g. a data set variable name, a variable value, library references, a data set name, any alphanumeric string without spaces) or many words. They can be partial words (e.g. data set variable name prefixes) or they can be null (e.g. when what varies is the decision to include a piece of code or not). At the other extreme, macro variables can be as long as 65,534 characters. Also, you are unlimited in the number of macro variables you have. Technically, you can write a program with nothing but several macro variable references, or a program containing nothing but one macro variable reference, where that variable includes several DATA steps and PROCs. The price you pay for macro variables, however, is memory, maintenance, and readability. As a programmer, just as a consumer in a department store, you have to ask yourself what you have bought with macro

variables and if it is worth it. Generally macro variables are most useful when they represent code that has the potential to change relatively frequently. A program laden with macro variable references may help address all the variability in a task, but also may be hard to read. Just as a book is harder to read when it is written in French, forcing you to continually consult a French-English dictionary, than if it were written in English, programs with macro variable references are harder to read than those without. Also, more macro variables mean more macro variables to maintain. When these get lost in the shuffle, they may accidentally be overwritten or not overwritten when they should have been, and can lead to unexpected results.

OTHER PROPERTIES

Macro variables cannot resolve inside of single quotation marks. Keep this in mind when using a macro variable inside a TITLE or a FOOTNOTE statement, or a PUT statement. Suppose also that you assign a string of text to a macro variable that you plan to use as a literal in a DATA step. Since literals must be surrounded by quotation marks in a DATA step, any reference to this macro variable must be surrounded by double quotes. For example, you are creating a data set with a variable called OFFICER whose value at any given execution will be the title of an officer of choice (e.g. PRESIDENT, SECRETARY, etc.). Following a macro variable definition statement that initializes a macro variable that is also called OFFICER, you have a DATA step with a statement that reads OFFICER="&OFFICER";.

Macro variables are not always available for referencing. In general, macro variables defined outside of macros are always available and are described as "global" macro variables, whereas macro variables defined inside macros are available only during the execution of that macro and any sub-macro, and are described as "local" to that macro or environment. The exception to this is when a macro variable of the same name already exists "more globally". This exception can be overridden with a %LOCAL statement inside a macro, which keeps separate two macro variables of the same name defined in two environments. References to the macro variable within the macro affect the value of the one defined in the macro while references outside the macro affect the one defined outside. The use of this statement is recommended in applications that use many macro variables defined in many different environments. On the other hand, a macro that would otherwise be local can be made global by naming it (without an ampersand) in a %GLOBAL statement before it is defined.

DEFINING MACRO VARIABLES

The method by which you define a macro variable somewhat depends on how you plan to use it and who will be initializing it. You know that the ultimate goal is to generate code with it, but you now know that that can be accomplished in a couple of ways. Sometimes a macro variable is directly referenced in open code, and sometimes it is used in macros only to create other macro variables or generate code directly without creating other macro variables. You know that a macro variable is the means by which a user specifies a set of circumstances, and you know that the user may only be ourselves, a group of other programmers, or a group of non-programmers who need a user-friendly interface on a multi-purpose application. This section will describe three general ways of initializing macro variables, none of which require a macro to do so. Two other methods that require a macro will be discussed in the Macro section.

The most versatile and straightforward method for defining a macro variable is with the macro statement %LET, a statement that is useful inside and outside of macros. Found outside of macros, this is one common way of accepting input from a user. Rather than asking a user to search through a program to make parameter changes, you create a block of %LET statements in the beginning of the program and provide instructions (by way of comments or a separate document) on how to use them. Returning to an earlier example, users specifiy a salary to use as a threshold for filtering data sets, both in an unformatted and formatted way.

```
%LET NBR1=30000 ;
%LET NBR2=$30,000 ;
```

This code is in the beginning of the program, and references to NBR1 and NBR2 throughout the program are replaced by their respective values. Inside of macros, %LET is a quick and convenient way to create links between user input and code to be generated. Reasons for using it mirror those for creating data set variables in the DATA step. Just as created data set variables may be the target of analysis or just a means to an end in a DATA step, we can create macro variables to be referenced in code or to lead to other code to be generated. Also like its DATA step counterpart, we can use %LET to create macro variables based on the values of other macro variables or as literal text. These statements can also be executed conditionally using macro conditional statements.

Before moving on to other methods, consider the following illustration of how some of the differences between macro variables and data set variables manifest themselves using %LET.

```
%LET GAME=CHESS ;
DATA HOBBIES1 ;
LENGTH GAME $10 ;
SET HOBBIES0 ;
```

```
GAME='CHESS' ;
RUN;
```

Of course, to create a data set variable, you need to do so in a DATA step, whereas the macro variable does not. Whether in a macro or in open code, %LET is processed once, while the data set variable is processed for each observation read from the data set HOBBIES0. The value of the macro variable persists through the data set and into other steps until it is explicitly changed. The data set variable is restricted to contexts involving HOBBIES1. As emphasized by the LENGTH statement, data set variables have attributes such as length while macro variables do not. While both types of assignment follow the form *new variable* = *value*, the macro variable assignment needs the percent sign to signal the macro facility, whereas the DATA step assignment begins with the new variable name. Finally, the macro variable assignment needs no quotation marks.

Depending on the user, you may feel that allowing users to specify input with a %LET statement is letting them too close to the code. For that reason there is %WINDOW. %WINDOW is a method of creating macro variables exclusively from user input by way of attractive, user-friendly input screens created by the programmer containing messages from the programmer (e.g. instructions) and fields for input. While its scope is not as broad as that of %LET, it does keep users further away from the code. Just as with its DATA step counterpart, the WINDOW statement, %WINDOW allows the programmer to specify the background color of the screen, the font color, number of columns and rows, and where to direct messages and input fields based on the current position of the pointer. Fields can be assigned attributes such as length and appearance (e.g. underline, reverse video, blinking). While the name of the field in the WINDOW statement of a DATA step corresponds to a data set variable name, the field name corresponds to a macro variable name in %WINDOW. When %WINDOW executes, it creates macro variables for each field if they are not defined already. With %DISPLAY, users have the opportunity to initialize these macro variables by typing their input into the field.

Until now, we have talked about the macro facility as a separate entity from the DATA step with processing taking place outside of any data context. There are exceptions. Sometimes code needs to be generated based on data. In other words, information from a data set is needed outside of the DATA step to determine what happens next. CALL SYMPUT is a tool in the DATA step that allows you to create macro variables whose values are found in the data set. Because it is found in the DATA step, it does get processed iteratively with the rest of the executable statements, and its syntax is more typical of the DATA step. The macro variables created by CALL SYMPUT are local to the environment that is current when the end of the step is reached, but are not available until after the step is complete. If that current environment contains no macro variables, it will be created in the nearest non-empty environment in which the current environment is nested (though exceptions do occur).

CALL SYMPUT has two arguments. The first is the name of the macro variable you are creating, and the second is the value being assigned. The first argument can be literal text, a combination of variables from the data set, or a combination of literal text and data set variables. As is always the case in the DATA step, literal text must be enclosed in quotation marks. Anything else is assumed to be a variable name. Of course, using only literal text means that only one macro variable is being created. This is useful when you only need one piece of information from the data set, such as the number of observations. For that reason, in such cases, CALL SYMPUT is usually executed conditionally, where the condition is true for only one iteration of the DATA step, such as the first or last observation. To the extent that it gets executed more than once, the macro variable continues to be overwritten. On the other hand, the first argument could be the name of one of the data set variables, or a function of multiple data set variables. In this case, the name of the macro variable created in a given iteration is the result of the expression for that iteration of the DATA step. Suppose a data set has an ID variable in which every value is found in only one observation. By naming this variable in the first argument, we create as many macro variables as there are observations in the data set, and carry outside of the data set, information from each observation. Of course even without such a variable, every data set has available the automatic variable _N_. Combining this with literal text is another way to create a macro variable for each observation.

The second argument is the same as the first in terms of the syntax. Literal text is indicated with quotation marks. Any valid SAS expression involving combinations of literal text and functions of DATA step variables is allowed. When the second argument is the name of a character value, the SAS system will write it using a format that reflects the length of the variable. Values shorter than that length will be padded with spaces on the right. For this reason, it can be a good idea to use LEFT and TRIM or COMPRESS to remove unwanted spaces. Similarly, when the variable is numeric, the SAS system uses BEST12 to convert the number to character. Without suppressing spaces, written instances of this macro variable will be right-aligned. Version 9 offers CALL SYMPUTX, which relieves you of the hassle of using these compression-type functions to remove unwanted spaces.

Consider the following three examples. Example 1 contains a data set called GEOGRAPHY with a variable called STATE, in which each value of STATE can be found in any number of observations. You need to use the number of unique values of this variable as the upper bound of an iterative DO loop later in the program, so you obtain this

information with a DATA step that reads GEOGRAPHY and passes the information out of the DATA step by way of CALL SYMPUT. Assuming that GEOGRAPHY has been sorted by STATE, observe the following code.

```
data _null_ ;
set GEOGRAPHY end=thatsit ;
by STATE ;
if first.state then counter+1;
if thatsit then call symput('stateno',put(counter,2.));
run;
```

A few observations are noteworthy. First, you only needed this DATA step to pass along this information, but you did not need a data set, so DATA _NULL_ makes sense. Second, the second argument of the CALL SYMPUT turns the numeric variable into a character variable before assigning it to the macro variable. We know that macro variables are always character strings and this is no exception. Without this step, you leave the conversion to the SAS system. This example is characterized by the fact that you only needed one piece of information about the DATA step, and so CALL SYMPUT is executed only once, and a literal text string is used to name the macro variable.

For example 2, in addition to the number of unique values, you need to pass along the values themselves. Consider the following change to the code.

```
data _null_ ;
set GEOGRAPHY end=thatsit ;
by STATE ;
if first.state then do;
     counter+1;
     call symput(compress('VAR'||put(counter,2.)),state);
end;
if thatsit then call symput('stateno',put(counter,2.));
run;
```

In addition to incrementing the counter by 1 at each new value of STATE, we also assign the value of STATE to a macro variable. The name of the macro variable begins with the text 'VAR', and is suffixed by a unique number, thereby creating unique macro variables. In the Macro section, we will see how you can use and reference these macro variables.

This example is also characterized by the fact that you did not know which states would appear in the data set, so you used macro variables to store them. For example 3, suppose in the same database but in a different data set you have the variables STATE and CAPITAL, where each state in the United States is represented by STATE, and the corresponding value of CAPITAL is the capital of that state. Consider the following code.

```
data _null_ ;
set capitals ;
call symput(state,capital) ;
run;
```

Here CALL SYMPUT is executed for each observation of the data set. In this case, we know that all states are represented, and so each state name is the name of a macro variable.

The SQL procedure also offers macro variables through data, but with less options and flexibility. Using an INTO clause between SELECT and FROM, you can specify a comma-delimited list of macro variable names, each preceded by a colon. The value of the *n*th variable listed in the SELECT clause will be assigned to the *n*th macro variable listed. When specifying only macro variable names and not ranges of variable names, only the values of the first observation of the data set will populate the macro variables. On the other hand, when specifying a range of *n* macro variable names, either with a dash or with the word THROUGH or THRU, (e.g. :MACVAR1 thru :MACVAR*n*), values of the corresponding data set variable from the first *n* observations will populate the *n* macro variables. Finally, by specifying one macro variable name followed by the keyword SEPARATED BY and a delimiter of your choice enclosed in quotation marks, you can create one macro variable whose value is the string of all values of the corresponding data set variable, separated by the chosen delimiter. Note that the ability to create macro variables named by values of a variable does not exist with PROC SQL. Observe the following code that accomplishes the same task as that of example 2 above in which each value of STATE was assigned to a macro variable along with the total number of unique values of STATE.

```
proc sql noprint ;
select count(distinct state) into :stateno
from GEOGRAPHY ;

select distinct state into :var1 thru :var%left(&stateno)
from GEOGRAPHY ;
quit;
```

The first statement counts the number of unique values of STATE without having to create the COUNTER variable as was done in the DATA step. The second statement uses the value of the macro variable created in the first statement as an upper limit on the number of macro variables to be created. As we will see later in the Macro Functions section, the effect of %LEFT is to left-align its argument. As with CALL SYMPUT, when creating macro variables for only the first observation of the data set being read, numeric values will be right-aligned. Without %LEFT, VAR&STATENO would resolve to the text string VAR followed by padded blanks followed by the value of the macro variable STATENO, thus causing an error. When specifying ranges of macro variables or when using the keyword SEPARATED BY, leading and trailing blanks are removed.

Finally, a group of macro variables that need no defining are the automatic variables supplied by the SAS system. These macro variables mostly contain information about the environment, such as the name of the current PROC or macro that is executing, recent return codes, operating system information, and information known to your system such as the current time, day, and date. Because the names of most of these macro variables begin with the string SYS, it is a good idea to avoid creating macro variables named in this way. PROC SQL also creates macro variables to provide information about how many rows were processed by PROC SQL and SQL return codes. These macro variables begin with the string SQL. Other macro variables that begin with the string SQLX provide information about the results of an SQL query submitted through the Pass-Through facility.

MACRO VARIABLE REFERENCING

At this point, what else can we say about macro variable referencing? At the sight of an ampersand followed immediately by an allowable macro variable name, the macro facility is called upon to generate code in its place based on what has been stored in memory. We know that what is stored in memory is nothing more than text without any presumption about how the text will be used, once generated. For that reason, macro variables can be referenced anywhere in open code where the resolution generates syntactically correct code. We have talked about how big the value of a macro variable can get. You can assign to them such “words” as data set variable names or variable values or data set names, etc, or you can assign longer text such as a list of variable names or data set names or even entire statements or blocks of code. What we have not talked about is the opposite situation – shorter values of macro variables, to be used as portions of “words”. How do you combine macro variable references with text to produce a “word”? For example, how do you combine the text string “20” with a reference to a macro variable that represents two-digit years to generate four-digit years? In this section, we will discuss combining macro variable references with text as well as with other macro variable references. Along with generating text in open code, you can also generate with macro variables the names of other macro variables. When this happens, how is the macro processor to know whether these are to be used as text, or whether the macro facility should attempt to resolve them? We will discuss the notation used to instruct the macro facility to treat the result of macro variable resolution as another macro variable reference. Finally, we will discuss macro functions and their relationships, not only to DATA step functions, but also to macro variable references.

For our first example, we will use the example alluded to above. Knowing that a macro variable reference is replaced by its value upon execution, it should be of no surprise that 20&YR2 would generate a four-digit year if YR2 is the name of the macro variable representing the two-digit year. When using a reference like this, make sure to consider how YR2 was initialized. If it was assigned through CALL SYMPUT or PROC SQL, make sure that the proper care was taken to eliminate leading blanks or stripping of leading zeros.

Combining macro variables works the same way. For our second example, consider a medical claims database in which the data sets are named both for the type of claims they contain (IP for inpatient and OP for outpatient) and the year in which they were incurred (four digit years). With the macro variables TYPE and YEAR, users specify which claims to extract from the database. Combining the two macro variables references in a SET statement generates the name of the appropriate data set to process.

```
data extract ;
set db.&type&year ;
more code
```

Ambiguity arises though when the combination of text with a macro variable reference requires the reference to precede the text. For our third example, suppose that a smaller application than that of example 2 extracts only 2004 claims, but users still have a choice of claim type. If you take out the reference &YEAR and replace it with 2004, the SET statement now looks like this.

```
set db.&type2004 ;
```

The macro facility looks for a macro variable named TYPE2004, a valid name for a macro variable, does not find it, and issues an error. This was not an issue in the first two examples. In the first, the macro variable reference came at the end of a “word”, and in the second, it was followed by an ampersand. The question here is how does the macro facility know when it has reached the end of a reference? Remember that certain characters are not allowed as part of macro variable names. The space is one such character. For that reason, in example 1, once a space is reached after the 2, the macro processor knows that this must be the end of the reference. Ampersands are also

prohibited; thereby removing any ambiguity that otherwise may arise in example 2. Numerals such as 2, however, are allowed. In the reference &TYPE2004*,* the macro processor has no way of knowing that the reference ends after the *e*. For that reason, the period is provided as a delimiter for the macro processor. The period allows you to concatenate macro variable references with text without having to separate them with unwanted characters such as spaces. You insert the period to signify the end of the macro variable reference, but because of its special status as a delimiter, it does not become part of the code.

```
set db.&type.2004 ;
```

If the value of TYPE is initialized to IP*,* then the generated code becomes

```
set db.ip2004 ;.
```

So a period that immediately follows a macro variable reference is always “swallowed” by the reference, or never becomes part of the code. Of course, this may not always be the desired effect. For our fourth example, suppose that the first level of the data set name is represented by a macro variable LIBREF, instead of “db”. By removing “db” and inserting the macro variable reference, you get the following:

```
set &libref.&type.2004 ;.
```

Just as the reference to TYPE swallows the period that follows, the reference to LIBREF does the same. If TYPE is initialized as IP and LIBREF is initialized as DB, upon resolution, the following is generated.

```
set dbip2004 ;
```

Rather than trying to process a permanent data set, the SAS system will try to process a WORK data set called DBIP2004. To fix this, keep in mind that the period is the end of a reference. Anything that follows it up until the next ampersand is just part of the open code. Therefore, following the delimiter period with another period will leave one period in the open code.

```
set &libref..&type.2004;
```

We know that we use macro variables primarily to represent different possibilities for a given piece of code. What if it is the case that not only does a piece of code vary, but also the macro variable to be used could vary from one execution to the next? We now return to the example where we had a data set with two variables, STATE and CAPITAL, where each value of CAPITAL represents the capital city of the corresponding value of STATE. In that example, we used CALL SYMPUT to create macro variables named for each value of STATE, whose values were the corresponding values of CAPITAL. With the user’s choice of a state, specified through a macro variable called PICKASTATE, you write an application that writes a message to the log indicating that state’s capital. It sounds simple enough, but there is one problem – because it coincides with the choice of the user, the macro variable that generates the capital city changes with every execution. Certainly if the message was always to generate a statement about the capital city of Oklahoma, the %PUT statement could read as follows:

```
%put The capital city of Oklahoma is &Oklahoma; .
```

However, since users are free to choose other states through the macro variable, we know that the statement would begin as follows:

```
%put The capital city of &pickastate etc...
```

How do you finish this statement?

We know that the result of macro variable resolution is text. Sometimes, that text may be the name of a macro variable, but of course, once the resolution is over, the ampersand is gone, and the name of a macro variable is just another piece of text. In this example, if the user chooses Oklahoma, then the resolution of &PICKASTATE is Oklahoma. Though Oklahoma is the name of a macro variable, without any ampersands, it is just another string of text. However, when resolution of a macro variable reference yields an ampersand, the macro processor will try to resolve this result. In this case, this second read will generate the name of the capital city in the code.

The macro processor will generate ampersands only from references that contain consecutive ampersands. Knowing how many to use and where to put them in a reference requires knowledge of three rules the macro processor follows for resolving such references. The first rule states that pairs of ampersands always resolve to one ampersand. Second, the macro processor makes a complete read of a macro variable reference from left to right before making a second read. Third, the presence of at least two consecutive ampersands guarantees a subsequent read and in fact, is the only way to get a subsequent read. Several consequences evolve from this set of rules. First, because of rules one and two, the number of ampersands present at any one time will be cut in half after a single read by the macro processor. In the case of an even number of ampersands, this means that no resolution of the macro variable takes place. When the number is odd, the last ampersand resolves the macro variable while the remaining even number preceding it reduces to half. A second consequence not unrelated to the first is the fact that the placement and the number of ampersands used is anything but arbitrary. Suppose that in the example above, we finish the %PUT statement in the following way.

```
%put The capital city of &pickastate is &&pickastate... ;
```

How does the second reference resolve, according to the rules outlined above? We know that the pair reduces to one ampersand. Continuing to the right, you have no more ampersands, but only the text PICKASTATE. After the first read, you are left with &PICKASTATE. Because of rule 3, we know that the macro processor will turn around

and make another read, but now it is reading the same reference as the first reference in the statement. The result is a message in the log that reads THE CAPITAL CITY OF OKLAHOMA IS OKLAHOMA., not the message you wanted. Generally, a reference containing only one macro variable with a pair of ampersands gives you nothing more than what you get with one ampersand. Suppose you add a third ampersand. This way, during the first read, the first two reduce to one and the third resolves the macro variable. What remains is &OKLAHOMA. Being guaranteed a second read, final resolution generates the capital city. To generate the message in the log, you use the following statement.

```
%put The capital city of &pickastate is &&&pickastate... ;
```

Sometimes an even number of ampersands is used to delay resolution of one part of a reference until another part is resolved. Suppose an application contains a FOOTNOTE statement that sometimes is to indicate the current day and sometimes the current date, depending on the value of the macro variable FOOTNOTE, initialized by the user. You have available the automatic macro variables SYSDAY and SYSDATE, but how do you make a reference that generates the user's choice? One option would be similar to above. If the two choices for the user are the text strings SYSDAY or SYSDATE, then by referencing the macro variable FOOTNOTE with three ampersands, you would generate the desired output. However the prefix SYS does not mean anything to users. All they care about is the choice between the day and the date. For that reason, you make these the two choices. Of course these two choices are the suffixes of the two automatic macro variables of choice, and so you are faced with the situation of combining macro variables with text as described above. If you simply combine the text string SYS with a reference to the macro variable FOOTNOTE, then the result of SYS&FOOTNOTE is the text string SYSDAY or SYSDATE, depending on the user's choice. Again, these are macro variable names, but without any ampersands left, they are just text. Suppose you place an ampersand in the front of the reference. Then &SYS&FOOTNOTE asks the macro processor to resolve a macro variable called SYS which does not exist. What you need is to use enough ampersands in the beginning of the reference to force the macro processor to wait until the reference in the suffix is resolved to resolve anything in the beginning. Notice what happens with a second ampersand in the beginning. With &&SYS&FOOTNOTE, the first pair resolves to one, but the text string "SYS" is left alone. Continuing from left to right, &FOOTNOTE resolves to DATE (if that is the user's choice), and after the first read, you have the reference &SYSDATE. With a second read, the current date is generated in code.

```
Footnote "Today is &&sys&footnote..." ;
```

Note the use of the three periods following the macro variable reference. Since this reference will be read twice, the first two periods will be swallowed, leaving one period to end the sentence after complete resolution.

For a more detailed discussion on the use of consecutive ampersands, please see *Molter, 2004*.

FUNCTIONS

Just as the DATA step has functions to provide you with information about data set variables, the macro facility has functions to provide you with information about macro variables. Just like macro variables, macro functions can be referenced in macros or in open code. Since the macro facility treats anything without a percent sign or an ampersand as literal text, macro functions must always be preceded by a percent sign. Also because of this property, the macro facility provides us with other functions the DATA step does not have or need. Finally, the macro facility provides us with one function that allows us to use DATA step functions that have no macro counterparts, such as numeric functions (e.g. MIN and MAX).

One class of macro functions that requires little discussion is the text functions. These include %LENGTH, %INDEX, %SCAN, %SUBSTR, and %UPCASE. These work exactly the same way as their DATA step counterparts with the same arguments. As is the case with the rest of the macro facility, literal arguments need no quotation marks.

Of course these correspond to only a small subset of DATA step functions, but more are available through autocall macros. Though technically these are macros and not macro functions, I include them in this discussion because many of them remind us of DATA step functions. Autocall macros are macros that come with your SAS system. To use them, two system options must be in effect. The SASAUTOS option points to where these macros are. This should be part of your configuration file so that you do not have to worry about remembering to activate it. Also, MAUTOSOURCE should be turned on to direct SAS to search these libraries for macros. This option is turned on by default. Included in these macros are %LEFT, %TRIM, %LOWCASE, and %CMPRES (a limited macro counterpart to the COMPRESS function).

In another class of functions are the quoting functions. Because literals are not enclosed in quotation marks in the macro facility, you need another way to distinguish between literals and characters or mnemonics that have special meaning such as commas, semicolons, logical operators, etc. Suppose for example that you need %SCAN to parse a text string delimited with commas. While the third argument of the DATA step counterpart would surround the comma with quotation marks, without quotation marks, the third argument of the macro function is indistinguishable

from the commas used to separate arguments of the function. For a second example, imagine trying to assign with %LET a string of text that contains a semicolon to a macro variable. The quotation marks that surround the literal in a DATA step assignment statement mask the special meaning of the semicolon, but without the quotation marks, the semicolon intended to be part of the macro variable value ends the macro statement. For instances like these where characters intended to be literals could be interpreted as special characters during the definition or compilation of a macro or macro variable, the macro facility offers %STR to mask special meanings and treat them as literals. Also available are quoting functions such as %BQUOTE that mask special characters during execution. While %STR allows you to define a macro variable or a macro without ambiguity, what guarantee do you have that the resolution of a macro variable will not lead to more ambiguity? For example, if a conditional macro statement is based on a string of text stored in a macro variable that happens to contain a space, followed by the text string OR, followed by another space, then without a quoting function, the macro facility will interpret this as a logical operator within the condition and likely lead to errors. In order to prevent against this, you use %BQUOTE to mask any special characters that may be contained in the resolution of macro variables, especially when values of macro variables can be long and unpredictable (e.g. when the user specifies syntactically correct filtering criteria for a step). Both %STR and %BQUOTE also allow you to use unmatched quotation marks or parentheses without the SAS system expecting them to be matched, though %STR requires these to be marked with special characters. By specifying %NRSTR and %NRBQUOTE, percent signs and ampersands are also masked. Finally, each of the text functions mentioned earlier has a corresponding function with the same name preceded by a Q (e.g. %QSUBSTR) in order to mask the result of the function at execution time.

We know that to assign the result of a mathematical operation to a data set variable, you can simply provide the numbers or the names of numeric variable along with the operator symbols (e.g.“+”). For example, the assignment statement X=10+5 assigns the value of 15 to the numeric variable X, but without a percent sign, the macro statement %LET X=10+5 simply assigns the text “10+5” to the macro variable. For that reason, the macro facility gives you %EVAL to allow you to perform mathematical operations on integers. To assign 15 to the macro variable X, you use the statement %LET X=%EVAL(10+5). %SYSEVALF is provided to perform mathematical operations on numbers with decimal points.

Finally, for most of those DATA step functions without macro function counterparts, the macro facility gives us %SYSFUNC and %QSYSFUNC. For example, to compress *b* out of the string *abc* and assign the result to the macro variable *x,* you use the macro statement %LET X=%SYSFUNC(COMPRESS(abc,b)). Be careful though because you cannot nest DATA step functions within one %SYSFUNC function. You can however nest multiple %SYSFUNC functions. For example, suppose you create a macro variable with the statement %LET X=%STR( )ABC%STR( ). The macro variable X contains a leading and a trailing blank, which can be trimmed out with %SYSFUNC(LEFT(%SYSFUNC(TRIM(&X)))).

MACROS

We have discussed two main reasons for generating code. We also know that the macro facility provides us with two ways of doing so. One way is by storing potential code in memory, but we have seen that that is not always enough. One reason to generate code is to repeatedly execute a piece of code, with each execution sometimes differing from the others according to a defined pattern. While a macro variable can help us track iterations of this kind of execution, it takes more than the storage of a text string in memory to define the pattern. We also generate code to allow a program to run under multiple circumstances. When writing such a program to accept user input, we do our users a favor by offering them meaningful, intuitive choices for input specification. If we allow a user to specify gender, we ask them to use M or F, or Male or Female. We do not ask them to specify 1 and 2 just because those are the coded values in our database. When the code to be generated is more than what we want to ask of our users, we need logic to make the translation.

Consider a task in which you are to analyze several variables. In front of you is a database with several data sets, each containing several variables, some of which will be analyzed, some will not. Some of those variables not being analyzed are still used to create additional variables not present in the database that do need to be analyzed. These are the two roles a DATA step variable plays – either it is analyzed, or through the DATA step, using DATA step tools, it is used to create other variables to be analyzed. An analogous statement can be said of macro variables. While the DATA step facilitates analysis, the macro facility facilitates code generation. Just as some database variables are ready from the beginning for analysis without manipulation, some macro variables can be referenced in open code. Just as sometimes we need logic to get to the other analyses, we also need logic to generate other code. To the extent that a tool is applicable to both environments, macros allow you to do with macro variables what the DATA step allows us to do with DATA step variables. Of course DATA step statements that help define a data set such as KEEP, RETAIN, and FORMAT are not necessary in the macro facility. However, a DATA step statement of the form IF *(DATA step variable) (operator) (DATA step variable value)* THEN (*DATA step statement)* clearly has use in the form of %IF *(macro variable) (operator) (macro variable value)* %THEN (*macro statement)* in the macro facility. Notice the percent signs to signal the macro processor. Among the possibilities for *macro statement* is %DO, to be

followed later by %END. Similarly, the DATA step statement of the form DO *(data set variable) = a* TO *b,* where *a* and *b* can be numbers or data set variables with numeric values, has a useful macro facility counterpart of the form %DO *(macro variable) = x* %TO *y,* where *x* and *y* can be numbers or macro variables with numeric values. DO-WHILE and DO-UNTIL also have macro facility counterparts. Finally, maybe the most important analogy to understand is that between the DATA step OUTPUT statement and its macro facility counterpart. Whether executed conditionally or not, implicitly or explicitly, this is what creates the observation in the new data set. In this section, we will see what macros use to create code.

Before getting into these details, we will briefly discuss the syntax of macro definition and referencing. A macro definition always begins with a %MACRO statement and ends with a %MEND statement. Following %MACRO is the name you give the macro along with parameter definitions if you choose to use them. As with other macro statements, a semicolon ends the %MACRO statement. The %MEND statement ends the definition of a macro. Optionally, the name of the macro that is being ended can follow %MEND. This is recommended particularly when macros are defined within other macros in order to easily detect which macro definition is ending. When referencing or calling a macro, you type the name of the macro immediately after a percent sign.

Macro parameters are tools used to accept user input. The value supplied by the user becomes the value of a macro variable named for the parameter. This macro variable is always local to the macro that defines it. To define macro parameters, you follow the name of the macro in the %MACRO statement with a set of parentheses that enclose their definitions. If you are defining more than one parameter, each definition is separated by a comma. The way you define any given parameter depends on which of two types you choose to use. One is the keyword parameter. When defining a keyword parameter, you indicate the name of the parameter followed by an equal sign. Optionally, you can assign a default value on the right side of the equal sign. The following example illustrates the %MACRO statement for a macro that uses two keyword parameters, the second of which assigns a default value.

```
%macro example1(parm1=,parm2=75);
```

The second kind of macro parameter is the positional parameter. When defining a macro with positional parameters, you must define them before any keyword parameters. With these parameters, an equal sign is not used and default values cannot be assigned. The following example defines a macro with both types of parameters.

```
%macro example2(posparm1,posparm2,kparm1=,kparm2=75);
```

To reference a macro with parameters, you specify values according to the same rules by which they are defined. Keyword parameter values are specified by indicating the name of the parameter, followed by an equal sign, followed by the desired value. The keyword parameters can be specified in any order, but no keyword parameter value can be specified before any positional parameter value. Finally, these parameter values are optional. When not specified, the value assigned will be the default specified in the definition, or the null value if no such default was assigned. Positional parameter values are specified by indicating only the desired value without the parameter name. They must also be specified in the same order they were defined, and their specification is mandatory. The following illustrates a call to the macro EXAMPLE2 defined above.

```
%example2(5,10,kparm2=50,kparm1=0);
```

In this case, &POSPARM1=5, &POSPARM2=10, &KPARM1=0, and & KPARM2=50. Note that the keyword values do not need to be specified in the order in which the parameters were defined.

Once inside the macro, a number of macro statements, some of which we have discussed in varying amounts of detail, are available to you. You know about %LET, %WINDOW, and %DISPLAY. %INPUT provides functionality similar to that of %WINDOW and %DISPLAY. You also know that %GLOBAL and %LOCAL allow you to change referencing environments for macro variables. Two other classes of macro statements that we have alluded to are conditional statements and DO loops. As discussed earlier, the macro facility has the %IF-%THEN statement whose form mirrors that of its DATA step counterpart. While the DATA step statement checks the value of a data set variable, the macro statement checks the value of a macro variable. Following THEN is a DATA step executable statement. Among these is the DO statement, in which case, everything that follows until the END statement is to be executed when the condition is satisfied. OUTPUT is also available, in which case observations are created conditionally. Following %THEN can be macro statements such as some that we have discussed such as %LET, %WINDOW, %DISPLAY, and %PUT. %DO can also follow, in which case, everything until %END, macro statements or code generation, will be executed if the condition is satisfied. Just as the generation of an observation can follow THEN with the OUTPUT statement, generation of code can follow %THEN. Also, as is the case with ELSE in the DATA step, %ELSE offers alternatives when the macro condition is not satisfied. Macro DO loops also mirror in form their DATA step counterparts. While iterative DO loops in the DATA step iterate through values of a data set variable, iterative DO loops in the macro facility iterate through values of a macro variable. Of course the macro facility requires the percent sign in front of DO and TO. Similarly, a percent sign is required to precede WHILE and UNTIL. Inside parentheses that follow either of these macro keywords is a valid macro expression. As you would expect, all %DO loops end with %END. Until that %END, macro statements as well as code generation can appear.

There is nothing in the rules that says that a macro has to generate code, or has to have statements that may generate code. You might need a macro that does nothing more than creates other macro variables or writes a message to the log. We now return to the example in which users indicate with a macro parameter the type of report they want. By asking them to spell the word that describes the report, we introduce the possibility of inconsistent spelling or casing in the title. The following macro keys on the first letter of the input to create a macro variable to be referenced in the TITLE statement.

```
%macro title(report=);
%global rpt;
%if %upcase(%substr(&report,1,1))=A %then %let rpt=Annual;
%if %upcase(%substr(&report,1,1))=M %then %let rpt=Monthly;
%if %upcase(%substr(&report,1,1))=W %then %let rpt=Weekly;
%if %upcase(%substr(&report,1,1))=D %then %let rpt=Daily;
%mend;
```

Referencing &RPT in the TITLE statement instead of &REPORT yields a consistent report title. Of course %LET is allowed in open code, but executing them conditionally is not, which is why we used the macro.

The next example uses the macro variables created earlier where the name of each state is a macro variable whose value is its capital city. From a data set, you have selected certain states whose capital cities will be identified in a statement in the log. The number of such states is stored in the macro variable STATENO, and each state name is stored in a macro variable named with the text string VAR followed by a unique number. Consider the following macro.

```
%macro logmessages ;
%do %i=1 %to &stateno ;
    %let pickastate=&&var&i ;
    %put The capital city of &pickastate is &&&pickastate ;
%end;
%mend;
```

We know that %PUT can be used in open code without the help of a macro. Of course you can also use it multiple times in open code, but with a macro variable representing the upper limit of the loop, this macro is suggesting that the number of states, and therefore, the number of messages written to the log, can vary, making the loop, and therefore the macro necessary.

We now turn our attention to macros that generate code. Unlike the DATA step which has the OUTPUT statement to generate observations of a data set, the macro facility has no statement to generate code. In place of an OUTPUT statement, you simply provide the code to be generated. Certain ways of using OUTPUT in the DATA step have no counterpart in the macro facility, such as naming a data set after OUTPUT to direct the observation to a particular data set, but other ways do have counterparts. One way is based on conditional logic. Just as the OUTPUT statement can be executed conditionally based on the values of data set variables, you can also follow %THEN with code to be generated when conditions based on macro variables are met. You can also execute OUTPUT unconditionally, either inside or outside DO loops. Similarly, we can provide code between macro statements, either inside %DO loops or on its own.

We are now ready to take a look at examples of macros that generate code. For example 1, consider a macro that contains no macro logic.

```
%macro nologic ;
and
%mend;

data test1 ;
set test0 ;
if var1 %nologic var2 ;
run;
```

The macro NOLOGIC does nothing more than generates the text AND. It is not a very practical macro since a user can type the word AND in the program as easily as calling a macro, but it is worthy of some observations. Of course, the first semicolon of the macro ends the %MACRO statement. Since what follows is not preceded by a percent sign, it is code that will be generated at the location the macro is called. Notice that there are no semicolons other than those that end %MACRO and %MEND. This is ok. Again, in the absence of macro statements, everything is just code to be generated. Here a semicolon is not desired in the generated code. For example 2, consider the following change to the macro.

```
%macro nologic ;
var2 ;
```

```
%mend;

data test1 ;
set test0 ;
if var1 and %nologic
run;
```

In this case, the macro generates the text VAR2 plus a semicolon. For that reason, the IF statement in the DATA step does not need a semicolon because the semicolon is generated unconditionally by the macro. Note also that the semicolon is unmasked. Because it is not part of a macro statement, there is no ambiguity as to how it will be used, and so masking is unnecessary.

The code generated by a macro that contains no macro logic or macro statements can also be generated by a %INCLUDE statement, without the overhead of compiling. For that reason, most macros contain a mixture of macro logic and code to be generated. Such macros present the biggest challenge because it can be difficult to distinguish macro code from open code. The key to this understanding is that macro statements begin with percent signs and end with the first unmasked semicolon. Anything in between is part of the statement and anything after the semicolon and before the next percent sign is code to be generated unconditionally. Anything following %THEN that does not follow a percent sign is code to be generated conditionally. The remainder of this section will focus on these types of macros.

For example 3 consider the following macro in which the operator that is generated is done so based on the value of the macro variable OPERATOR.

```
%macro mixed1 ;
%if &operator=ANY %then or ;
%else %if &operator=ALL %then and ;
%mend;

data test2 ;
set test1 ;
if var1 &operator var2 ;
run;
```

For beginning macro writers, this code is hard to look at. We are not used to seeing THEN followed by an operator like OR or AND. The best way to get around this confusion is by separating what is to be generated from the macro code. Since the text following %THEN is not preceded by a percent sign, it is code to be generated when the condition is met. Because the semicolon ends the macro statement, it is not considered part of the generated text.

When generating code conditionally, because the code to be generated is actually part of the macro statement, a challenge is presented when a semicolon is part of that code. The first unmasked semicolon after %THEN will end the macro statement. Therefore, any semicolon to be generated conditionally should be masked with %STR. Observe the following example in which the code intended to be generated is a DATA step DO loop, but the semicolon is unmasked.

```
%if &a=1 %then do i=1 to 2 ; output ; end;
```

Because the first semicolon is unmasked, it ends the macro statement. That means that the code DO I=1 TO 2 is generated only when the value of the macro variable *a* is 1. On the other hand, the OUTPUT and the END statements are always generated. IF *a* resolves to 1, then the DO loop is generated without a semicolon between the DO statement and the OUTPUT statement, resulting in an error. When *a* does not resolve to 1, the DO statement is not generated, and an error results from having an END statement without a DO or SELECT statement. The following fixes the problem.

```
%if &a=1 %then do i=1 to 2 %str(;) output %str(;) end %str(;) ;
```

Alternatively, when the code to be generated from a %IF statement includes multiple semicolons, a %DO-%END block saves us from having to worry about masking.

```
%if &a=1 %then %do;
    do i=1 to 2 ;
    output ;
    end;
%end;
```

Here we have replaced the code to be generated in the %IF statement with %DO. Following the semicolon, everything until the next percent sign will be generated when *a* resolves to 1. Now the semicolons that were masked when they followed %THEN can be unmasked because they are no longer part of a macro statement.

For example 4, we return to the DATA step in examples 1 and 2. Suppose that the data set TEST0 contains 100 variables, all named with the string VAR followed by a unique number between 1 and 100. The positional parameter

CONDITIONS represents how many of the first 100 variables must be listed in the subsetting IF. If no such statement is needed, the macro variable will be initialized to 0.

```
%macro mixed2(conditions) ;
%if &conditions>0 %then %do;
    if var1
    %if &conditions>1 %then %do i=2 %to &conditions ;
           and var&i
    %end; ;
%end;
%mend;
```

The following generates a subsetting IF that checks the values of five variables.

```
data test1;
set test0;
%mixed2(5)
run;
```

This time the macro contains a mixture of macro code and code to be generated. We will step through this carefully to distinguish between them. First, nothing is necessary when CONDITIONS resolves to 0. Next, everything after the %IF statement until the next percent sign is code to be generated. In this case, it will be used as *the start* of a subsetting IF, but without a semicolon to be generated, it is not the whole statement. After this is another percent sign signifying the beginning of another macro statement. The text (with the macro variable resolved) that follows the semicolon will be generated for each iteration of the %DO loop. For example, if the value of CONDITIONS resolves to 3, the text AND VAR2 AND VAR3 will be generated. If CONDITIONS resolves to 1, nothing further will be generated. Once the loop completes, the subsetting IF is completed except for a semicolon to end it. Again, everything between the semicolon that ends %END and the next percent sign is code to be generated. In this case, a semicolon is generated.

These macros have generated relatively short pieces of code. Some examples have generated a part of a statement, while others have generated part of a DATA step. For example 5, we revisit several earlier examples with a macro that accomplishes several tasks. Consider an application that retrieves data from a database that depends on the division of the company, as specified by the user through a keyword parameter called DIVISION. As before, division 1 data is held in a SAS database against which SAS DATA steps can extract. Division 2 data is stored in an Oracle database, and Division 3 data in a DB2 database. The keyword parameter COST allows users to request certain types of cost reports. Specifying A produces an annual report, M a monthly report, W a weekly report, and D for a daily report. Finally, through the keyword parameter SALESSUMMARY, a SAS data set is created containing any number of summaries, as chosen by the user.

We begin with a look at the macro code that decides from which database to extract.

```
%if &division=1 %then %do;
    data extract1 ;
    other DATA step code
%end;

%else %do;
    proc sql;
    connect to
    %if &division=2 %then oracle ;
    %else %if &division=3 %then db2 ;
    as mydbms (user=username password=password) ;
    create extract1 as select * from connection to mydbms (
%if &division=2 %then oracle specific SQL query ;
%else %if &division=3 %then db2 specific SQL query ;
    ) ; disconnect from mydbms ; quit;
%end;
```

Once again, %DO-%END blocks are used for division 1 and for the combination of divisions 2 and 3. Within the latter though are interruptions of code to be generated with macro code. The first place is in naming the DBMS. Once again, following the %IF and %ELSE statements are semicolons, but they end the macro statements and are not to be generated. Without a percent sign, the line that begins AS MYDBMS is unconditionally generated until the next percent sign. This example assumes the same username and password for both Oracle and DB2. If this is not the case, macro logic similar to above may be used within the parentheses to generate the appropriate username and password. Finally, after the semicolon following the last %ELSE statement is code to be generated – a closing parenthesis, a semicolon, the DISCONNECT statement, and the QUIT statement.

This portion of the application could have been coded in several different ways. First, it might make sense to require the DBMS specific username and password in order to even use the application. If so, you can make these either keyword parameters or ask for them in a prompt screen. Second, this portion was written with the approach that only minimal code was to be generated conditionally. Anything that would be common to both choices of DIVISION would not be part of the %IF-%THEN statements. This meant using multiple %IF-%THEN statements. Another approach would have been to create separate blocks for Division 2 and Division 3. Some of the code would be redundant but as the number of places that require conditional logic increase, it may be less cumbersome.

After data is extracted and put into a common format, PROC TABULATE generates the report of choice for cost.

```
proc tabulate data=extract2 ;
class year month week date ;
var cost ;
table
    %if &cost=A %then TABLE statement 1 ;
    %else %if &cost=M %then TABLE statement 2 ;
    %else %if &cost=W %then TABLE statement 3 ;
    %else %if &cost=D %then TABLE statement 4 ;
; run ;
```

In this case, the TABLE statement is interrupted by macro logic. Once it is complete, a semicolon and a RUN statement are generated.

Finally, the SUMMARY procedure creates data sets that summarize sales. The types of summaries as well as the number of them are up to the user.

```
proc summary data=extract2 ;
class salesperson ;
var sales ;
output out=summaries
    %let i=1;
    %let summary=%scan(&salessummary,1,%str( ));
    %do %until(&summary=%str()) ;
       &summary=&summary.sales
       %let i=%eval(&i+1);
       %let summary=%scan(&salessummary,&i,%str( ));
    %end; ;
run;
```

Here the OUTPUT statement in PROC SUMMARY is interrupted by a series of macro statements including an entire %DO-%UNTIL loop. At the i^{th} iteration of the loop, the macro variable SUMMARY represents the i^{th} summary requested in the parameter. The only place within this macro logic where code is generated is immediately after the %DO statement. For example, If the 2nd summary requested is MIN, then MIN=MINSUMMARY is generated after the generated code for the first statistic requested (note the role of the period as a macro variable delimiter here). Once out of the loop, a semicolon is generated to end the OUTPUT statement, followed by a RUN statement.

CONCLUSION

The macro facility allows you to build flexibility into your applications. An application becomes more flexible as the number of circumstances under which it can be run with minimal user intervention increases. Of course, different circumstances require different pieces of code. Through the macro facility you can efficiently generate circumstance-dependent code without having to touch the core of the program. This is accomplished by designating an area of the program for specification of circumstances, possibly followed by logic that defines how to turn any given circumstance into code. Places in the core of the program that would otherwise depend on the circumstance are replaced by references either to the specification itself or the logic that generates the code that is based on the circumstance. Rather than searching throughout a program for all pieces of code that depend on the current circumstance, a user can specify the circumstance in the designated area and the references in the core of the program will generate the appropriate code. In addition to flexibility, the macro facility can also make applications easy to use. Since macro logic can be used to translate specifications into code, you can write the macro to accept meaningful, easy-to-remember input keys from the user. In the end, users are kept away from the core of the program.

In this paper we have seen that you can develop an understanding of the macro facility by starting with your knowledge of the DATA step. Concepts such as variables and DO loops and conditional logic are already familiar to you, and so the understanding of slight differences such as the use of quotation marks, quoting functions and %EVAL is not too much to ask, especially with a solid understanding of the difference in objectives between the DATA step and the macro facility. With all that said, the role of the macro facility to generate open code makes inevitable the mixing of macro code and open code, and this is where confusion often arises. Statements such as %LET

VAR1=VAR1=5 and %IF &VAR2=5 %THEN IF VAR2=5 do not have a place in your understanding of the SAS language. In this paper I have attempted to clear up some of this confusion. Just like data set variable assignments, the text on the right side of the equal sign is the value assigned to the variable. The only difference is the lack of quotation marks and %LET. Just as conditional creation of an observation is accomplished with the OUTPUT statement following THEN, the code to be generated conditionally follows %THEN. In summary, among all the mixing of open code and macro code, open code does have certain "places" it is allowed to be. In macros, these places correspond to those where OUTPUT can be found in DATA step code. By remembering these facts, distinguishing macro code from open code becomes easier, and ultimately, so does reading and writing it.

The SAS system is famous for having many ways to accomplish almost any given task. Many programmers get along fine without the macro facility while others use it every chance they get. Among those who do, some create macros to generate only the code with potential to change while others may include static code in their macros. If given a choice, some may use macro variables for situations that others would use macros for. Issues to consider include memory and performance, but maybe most important is who will be using it. Liberties can be taken when you write a macro only for your own convenience. When other programmers use it, you may get away with asking them to supply syntactically correct SAS code into a macro variable to be directly referenced in code. However as the number of users increase, especially users who are not programmers, extra efforts should be taken to keep them away from the code while making input as intuitive and easy as possible. This may involve allowing them to populate the right side of the equal sign in a %LET statement with text that will become part of the code, allowing them to call a macro with keyword parameters whose values, for example, are the first letters of the possible types of reports, or creating an icon for their desktop which, upon double-clicking, invokes a prompt screen with a list of choices they can mark with an "x". As is the case with everything else, you have many choices.

REFERENCES

Molter, Michael (2004), "The Role of Consecutive Ampersands in Macro Variable Resolution and the Mathematical Patterns That Follow, " *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference.*

CONTACT INFORMATION

I am happy to answer any questions you may have regarding this subject. Please feel free to contact me in any of the following ways.

Mike Molter
Howard Proskin and Associates
300 Red Creek Drive
Rochester, New York
Phone: (585) 359-2420
Fax:
E-Mail: