# Moving from Macro Variables to Macros

Lisa Sanbonmatsu, Consultant, Somerville, MA

**ABSTRACT**

The macro facility is one of the most powerful features of base SAS. Macros and macro variables allow users to write more flexible code, perform repetitive tasks, pass information between data or proc steps, conditionally execute SAS code, and generate SAS statements based on what is in the data. Although beginning SAS users sometimes shy away from macros, getting started with macros is actually easier than it looks.

This paper is designed to help the beginning user step into macros. The paper starts with built-in and user-defined macro variables, illustrating how users can add flexibility to their code with the simple addition of a few %LET statements and ampersands. Passing information between data steps using CALL SYMPUT is also discussed. Next the paper demonstrates how to actually create a macro. The paper breaks the process down into steps and incorporates debugging into the coding process to help beginners avoid the frustrations of trying to figure out why a macro does not work. For users who want to write more complex macros, the paper concludes with a few more advanced macro examples.

**INTRODUCTION**

The macro facility is like a substitution machine: it replaces one sequence of text with another (often larger) sequence of text. There are two forms of substitution: 1) macro variables and 2) macros. For example, with a macro variable you could use a 10-letter abbreviation as a short-hand for a text phrase of 250 characters. With macros you can create short-hands for entire sections of code rather than just phrases.

Sound simple? The basic idea behind macros is simple. Creating and using macro variables is also simple. For example, if we frequently use a trademark notice in our footnotes and titles, we might want to create an abbreviation or short-hand for this notice. We can create a macro variable called TRADEMRK and assign our notice to it with a %Let statement:

```
%Let trademrk = ACME is a registered
trademark of ACME, Inc. Other brand and
product names are trademarks of their
respective companies.;
```

Now we can insert &trademrk (the ampersand identifies the word as a macro variable) into a footnote or wherever else we want it:

```
footnote1 "Notice:  &trademrk";
```

When the macro facility processes our code, it will substitute the full phrase for "&trademrk" to produce the following footnote:

Notice: ACME is a registered trademark of ACME, Inc. Other brand and product names are trademarks of their respective companies.

Through text substitution the macro facility allows you to accomplish many functions. Just by using macro variables you can:

1) Automatically insert the date and other session information into your code,
2) Write more flexible code, and
3) Pass data between steps.

With macros we can add even more functions to the list:

4) Perform simple repetitions,
5) Conditionally execute program steps,
6) Perform highly repetitive tasks using loops and nesting,
7) Generate data dependent code.

This paper will show you how to get started with macros and will demonstrate how each of the above functions (except for data dependent code) can be achieved. Although the basic concept behind the macro facility is simple text substitution, macro solutions can be quite complex. Getting started with macros is quick, mastering them takes more time.

**1. AUTOMATICALLY INSERTING THE DATE**

SAS provides "automatic" or pre-defined macro variables. These variables contain information about the current SAS session.

To reference a macro variable you just type an ampersand immediately followed by the name of the macro variable. For example, SYSDATE is one of the most commonly used automatic macro variables. Wherever &SYSDATE appears in your code, the macro facility will substitute in the date on which the current session began. (Note that the date SAS was invoked is not necessarily the current date.) Below is an example of how you can easily add the date into your footnotes:

```
footnote "Processed on &SYSDATE";
```

On the printout, this generates the following footnote:

Processed on 30SEP00

When you use a macro variable within a quote (as in the above example), you must use double quotes: SAS treats macro variables enclosed in single quotes as regular text. Thus the following statement:

```
/* mistakenly using single quotes */
footnote 'Processed on &SYSDATE';
```

would produce the following footnote:

Processed on &SYSDATE

In addition to SYSDATE, there are a number of other commonly used automatic macro variables:

| VARIABLE | DESCRIPTION | EXAMPLE |
|---|---|---|
| SYSDATE | date SAS session started | 30SEP00 |
| SYSDATE9 | date in DATE9. format | 30SEP2000 |
| SYSDAY | day session started | Saturday |
| SYSLAST | last data set created | WORK.TEMP |
| SYSSCP | host system | WIN |
| SYSTIME | time session started | 10:09 |
| SYSVER | SAS Version | 7.00 |
| SYSUSERID | userid | USER |

Automatic variables are easy to spot in code because they begin with "&AF" and "&SYS". To see the values currently assigned to all automatic macro variables, type the following line of code:

```
%put _automatic_;
```

This will send a list of the current values (including empty values) to your log:

```
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
...
```

Note that "%Put" starts with a percent sign. The percent symbol indicates to the macro facility that what follows is a macro keyword (%IF, %DO, etc.) just as an ampersand indicates the start of a macro variable. The many percent signs and ampersands that are used in macro code can look confusing at first. Just keep in mind that these symbols are necessary to let the macro facility know that this is macro-related code.

Macro variables can be used throughout your program, not just in quoted text. SYSLAST is a macro variable that contains the two-level name of the last data set that you created. At the start of your SAS session it is empty. Each time you create a data set, SYSLAST is updated to reflect the most recent data set. We can insert &SYSLAST into our code to print out the first 10 observations from our most recently created data set:

```
proc print data = &SYSLAST
            (obs = 10);
   footnote "Obs from &SYSLAST";
   footnote2 "Printed on
&SYSDATE..";
run;
```

If the last data set we created is WORK.TEMP, the SAS macro facility would substitute "WORK.TEMP" wherever "&SYSLAST" appears. At run time, the facility will generate the equivalent of:

```
proc print data=WORK.TEMP (obs =
10);
    footnote "Obs from WORK.TEMP";
    footnote2 "Printed on 30SEP00.";
run;
```

And produces the following footnotes:

Observations from WORK.TEMP
Printed on 30SEP00.

In footnote2 we needed to use a double-period after &SYSDATE: the first period is interpreted by the macro facility as the end of the macro name and the second period is treated as text. No period is necessary after the other automatic variables in the above code because they are followed by a space, double quote, or other punctuation that makes the end of the variable name clear.

Automatic Macro Variables At-A-Glance

Assigning:

- No need to assign. These are assigned automatically by SAS.
- All automatic variables are global (except for SYSPBUFF) so a value is in effect until the SAS session ends or the value changes.

Using:

- Insert into your code using:

"&" + <macro variable name>

- Use double quotes when quoting ("Today is &SYSDATE")
- Use a single period to separate the macro variable name from any text immediately following it ("&SYSLAST.TEXT")
- Use a double-period to produce a single period immediately after the macro variable ("&SYSDATE..")

Displaying values assigned (in your log):

- %put _automatic_;

**2. WRITING MORE FLEXIBLE CODE**

In addition to using the macro variables assigned automatically by SAS, you can create your own custom macro variables. The easiest way to create a user-defined macro variable is with a %Let statement:

```
%Let <macro var name> = <text string>;
```

Macro variables are an easy way to add flexibility to your code by reducing "hard coding." Typing specific information into your program is "hard coding" it. In the example below, all the specifics such as the month, the type of books, and the number of top books are hard coded:

```
/* TOP BOOKS */
/* This program prints a list of top
   5 Sci-Fi books for June */

/* Each observation is one book */

/* book = name of the book */
/* author = author of book */
/* type = type of book */
/* volume = number of books sold*/
/* profits = profits from book */

proc sort data = sales06
      (where = (type = "Sci-Fi"))
      out = sort06;
   by descending volume;
   /* sorts from most to least books */
run;

proc print data = sort06 (obs=5)
                  noobs;
   /* noobs suppresses obs #s */;
   var book author volume;
   title1 "Top 5 Sci-Fi Books for 0600";
   title2 "Based on Volume";
run;
```

The output from our code is:

```
Top 5 Sci-Fi Books for 0600
Based on Volume

book              author
volume

Seven Suns        E. Ernest
887
World of the      C. Chaudry
550
Planet XYZ        J. Johnson
333
More SciFi        W. Wu
200
Jumping Jupiter   D. Diullo
179
```

The above code is not very flexible. When we want to print out the same information for July, we would need to change "06" to "07" in 4 separate places. Instead of hard coding the month, we could instead assign the month to a macro variable and then refer to this macro variable throughout our code. This would allow us to change the month by changing only one %Let statement.

Values that are likely to change and appear multiple times in your code should generally be assigned with a %Let statement. Interest rates, selection criteria, file names, dates and other specifics that are likely to change are good candidates for macro variables.

To make a program more flexible:

1) Identify specific information that occurs multiple times in your code and is likely to change.
2) Replace each occurrence of this information with a reference to a macro variable (i.e., &MONTH).
3) At the top of your code use a %Let statement to assign the specific information to the macro variable name:

```
%Let month =06;
```

Using the above steps we can make our "Top Books" program more flexible. After adding macro variables and changing the %Let statements to print the top 10 Fiction books rather than top 5 Science Fiction books, here is how it looks:

```
%let month = 06;
%let type =  Fiction;
%let criteria=Volume;
%let topnum= 10;

proc sort data = sales&month
      (where = (type = "&type") )
      out = sort&month;
   by descending &criteria;
run;

proc print data = sort&month
```

```
      (obs = &topnum) noobs;
   var book author &criteria;
   title1 "Top &topnum &type
Books for &month.00";
   title2 "Based on &criteria";
run;
```

Since the macro facility will ignore leading blanks after the equals sign in a %Let statement, it does not matter whether you start the text directly after the equal sign or put a space in between.

We can list all of our user-defined macro variables in our Log by typing:

```
%put _user_;
```

For the above example, this will produce the following lines:

```
GLOBAL MONTH 06
GLOBAL TOPNUM 10
GLOBAL TYPE Fiction
GLOBAL CRITERIA Volume
```

**3. PASSING DATA BETWEEN DATA STEPS**

Call Symput provides another of creating a user-defined macro variable. The format for Call Symput is:

```
Call Symput(<macro varname>, <expression>);
```

Unlike %Let, Call Symput allows you to assign the value of a variable or other expression to a macro variable. This makes it possible to "share" information between different steps in a program.

Returning to our book store example, suppose we wanted to add the name of the top selling book into our proc print title. After the sort the top selling book will be the first book in the data set. But how do we pass this information to the proc print? One solution is to create a macro variable containing the name of the top selling book (Top_Selling_Book) and its author (Top_Author) using Call Symput:

```
data _null_;
/* A _null_ processes data without
   creating a data set.*/
   Set sort&month (obs = 1);
   /* uses first obs only */
   Call Symput('Top_Selling_Book',
               trim(book) );
   Call Symput('Top_Author',
               trim(author) );
   /* trim trailing blanks */
run;
```

Not only are we able to assign the value of the character variable "BOOK" to the macro variable "TOP_SELLING_BOOK", but we are also able to trim the trailing blanks from the variable at the same time (using the TRIM function). Macro variable names can exceed 8 characters. Re-issuing a "%put _user_" statement reveals our two new macro variables:

```
GLOBAL MONTH 06
GLOBAL TOPNUM 10
GLOBAL TOP_AUTHOR J.K. Rowling
GLOBAL TYPE Fiction
GLOBAL CRITERIA Volume
GLOBAL TOP_SELLING_BOOK Harry Potter
and the Goblet of Fire
```

We can now use these macro variables throughout our program, including our proc print title:

```
title1 "Our #1 seller was
<<&TOP_SELLING_BOOK>> by
&TOP_AUTHOR!";
```

Which yields the title:

Our #1 seller was <<Harry Potter and the Goblet of Fire>> by J.K. Rowling!

User-defined Macro Variables At-A-Glance

Assigning:

- Assign using a:
  `%Let <macro var name > = <unquoted text>;`
  or
  `Call Symput(<macro varname>, <expression>);`
- With Call Symput you can assign an expression, variable name, or quoted text to a macro variable.
- Macro variable names must start with a letter or underbar but can also contain numbers. Macro variable names are NOT limited to 8 characters.
- Values assigned to macro variables can be up to 32K characters.
- Macros created using %Let and Call Symput (outside of a macro execution) are by default global. These values are in effect until the end of the SAS session or until the user assigns a new value to the macro variable.

Using:

- Same syntax as for automatic variables (i.e., &MYVAR).

Displaying values:

- %put _user_;

## 4. PERFORMING REPETITIVE TASKS BY WRITING A MACRO

The program in Section 2 is fairly flexible. But if we wanted to repeat the task five times (one time for each of five types of books), we would have to do a lot of cutting and pasting. We could accomplish the task but our code would be awkward and difficult to maintain. A better solution is to use macros rather than just macro variables. Macros substitute whole sections of code and can be used to simplify repetitive tasks.

The format for defining a macro is:

```
%MACRO <macro name>;
...
SAS code
....
%MEND;
```

The format calling a macro is:

```
%<macro name>
```

(Note that we do not need a semicolon after a macro call.) Below is an example of a simple macro called "PRINT_IT" which prints out the last data set created.

```
%MACRO PRINT_IT;

proc print data = &SYSLAST;
title "Last Data:  &syslast";
run;

%MEND;
```

Anywhere in our program where we type:

```
%PRINT_IT
```

the following code will be substituted:

```
proc print data = &SYSLAST;
title "Last Data:  &syslast";
run;
```

Looking at our Output after submitting the "%PRINT_IT" we will see a printout of the data set.

However if we look at our Log, we will only see:

```
588     %PRINT_IT

NOTE: PROCEDURE PRINT used:
   real time           0.71 seconds
```

By default the code generated by a macro is not shown in the Log. Only NOTES and ERROR messages appear in the Log.

Although this is more efficient when you call the same macro many times, it can make debugging difficult. There are several global options which you can use to send more information to the log when you call a macro:

| OPTION | DESCRIPTION |
|---|---|
| MLOGIC | traces the beginning of macro execution and any parameter values assigned |
| MPRINT | displays the SAS statements generated by macros |
| SYMBOLGEN | writes a message for the resolution of each macro variable |

To see what is really happening when we submit "%PRINT_IT", we can first turn on all three options:

```
options mlogic symbolgen mprint;
%PRINT_IT
```

This produces the following messages in our Log:

```
MLOGIC(PRINT_IT):  Ending execution.
653     options mlogic symbolgen
mprint;
654     %PRINT_IT
MLOGIC(PRINT_IT):  Beginning execution.
SYMBOLGEN:  Macro variable SYSLAST
resolves to WORK.SORT06
MPRINT(PRINT_IT):   proc print data =
WORK.SORT06 ;
SYMBOLGEN:  Macro variable SYSLAST
resolves to WORK.SORT06
MPRINT(PRINT_IT):   title "Last Data
Set Created:  WORK.SORT06
";
MPRINT(PRINT_IT):   run;
NOTE: PROCEDURE PRINT used:
      real time           0.71 seconds
```

Part of the power of macros stems from our ability to "parameterize" macros. Macro parameters are the macro variables associated with a macro. When we call a macro we can "feed" it different parameters each time. Macro parameters need to be defined when you define a macro. Below is an example of how we could parameterize our %PRINT_IT. These parameters allow us to specify the data set name and the number of observations to print.

```
/* Positional Parameter Example */
%MACRO PRINT_IT(dataname, num_obs);

proc print data =&dataname
                (obs = &num_obs);
run;

%MEND;
```

The example above uses Positional Parameters. Positional Parameters match the values you specify to the names of the macro variables based on their position in the list: the first value is assigned to the first variable, etc. After defining the revised macro, we could print out the first 5 observations of WORK.SALES06 and the first 10 observations of WORK.SORT06 using:

```
%PRINT_IT(work.sales06, 5)
%PRINT_IT(work.sort06, 10)
```

Alternatively, we could have used Keyword Parameters to define the macro parameters:

```
/* Keyword Parameter Example */
%MACRO PRINT_IT(dataname=, num_obs=10);

proc print data =&dataname
          (obs = &num_obs);
run;

%put Within the Macro;
%put _user_ ;

%MEND;

%PRINT_IT(num_obs =5,
          dataname= work.sales06)
%PRINT_IT(dataname = work.sort06)

%put Outside of the Macro;
%put _user_ ;
```

With Keyword Parameters, values and macro names are explicitly linked using an equals sign to link the two (rather than rely on the order of the values). Notice the "num_obs = 10" in the DEFINITION of the macro. This assigns a default value of "10" to the macro variable "num_obs." In our second %PRINT_IT call we did not specify a "num_obs =" so the default value of 10 will be assigned.

Our %put statements allow us to look at the macro assignments within the macro (while the macro was running) and outside of the macros. Our log shows:

```
Within the Macro
PRINT_IT DATANAME work.sales06
PRINT_IT NUM_OBS 5
GLOBAL MONTH 06
...
Within the Macro
PRINT_IT DATANAME work.sort06
PRINT_IT NUM_OBS 10
GLOBAL MONTH 06
...
Outside of the Macro
GLOBAL MONTH 06
```

Notice that in the first execution of the macro the NUM_OBS had a value of 5 and in the second it had a value of 10. Also note that after the macro has finished running, NUM_OBS and DATANAME no longer exist. By default, macro parameters are in effect only during the execution of the macro.

Below are the suggested Steps for moving from a program with macro variables to a full macro. The Steps integrate debugging with the writing process. This is to avoid one of the most frustrating aspects of writing macros: trying to figure out what is wrong with your macro. By making sure your code is working properly at each step, you can help reduce frustrating bugs.

STEPS FROM MACRO VARIABLES TO MACROS

| |
|---|
| 1. Write your program and assign macro variables using a %Let. |
| 2. Test run, debug and fix your code as necessary. |
| 3. Enclose your code between a "%MACRO <macro name>" and a "%MEND." |
| 4. Convert your "%Let" statements to Keyword Parameters and retain the values as defaults. |
| 5. Turn on the debugging options:<br>`options mlogic symbolgen mprint;` |
| 6. Test run your new macro without specifying any parameters (i.e., the default parameters will be used):<br>`%<macro name>()` |
| 7. Debug and fix your code as necessary. Use "%put" to trace values if necessary. |
| 8. Once the code is working, add any repetitions of the macro call and more advanced macro statements as needed. |
| 9. Turn off the debugging options (to avoid producing a massive log):<br>`options nomlogic nosymbolgen nomprint;` |
| 10. Run your full program. |

At the start of this section, we set out to print our top book list for each of five different types of books. Applying the steps above, we can create a TOPLIST macro and easily call it five times:

```
%MACRO TOPLIST(month = 06,
     type =  Fiction,
     criteria=Volume,
     topnum= 10);

proc sort data = sales&month
      (where = (type = "&type") )
      out = sort&month;
```

```
   by descending &criteria;
run;

proc print data = sort&month
      (obs = &topnum) noobs;
   var book author &criteria;
   title1 "Top &topnum &type
Books for &month.00";
   title2 "Based on &criteria";
run;
%MEND;

%TOPLIST (type = Fiction)
%TOPLIST (type = Sci-Fi)
%TOPLIST (type = Reference)
%TOPLIST (type = Mystery)
%TOPLIST (type = Non-Fiction)
```

Clearly, this is a lot neater than if we had cut and pasted the program five times! Note that the "%TOPLIST" calls must be placed in the program AFTER (but not necessarily immediately after) the definition of the macro ("%MACRO.. %MEND"). A macro must be compiled (submitted) before it is called in your program.

We could also use the macro calls to specify different numbers of "top" books to print depending on the type of book:

```
%TOPLIST (type = Fiction, topnum=25)
%TOPLIST (type = Sci-Fi, topnum = 3)
```

Macros and Macro Parameters At-A-Glance

Defining Macros:

- Begin with:
  `%MACRO <macro name> (<parameter definition>);`
- End with:
  `%MEND;`

Defining Macro Parameters:

- Specified when defining the macro. This can be done with either Positional Parameters (variables separated by commas) or Keyword Parameters (variables separated by an equal sign and a comma).
- Macro parameters are, by default, only in effect while the macro is running. (Note that this rule also applies to macro variables created using "%Let" within a macro.)

Using Macro Parameters:

- Same syntax as for automatic variables and user-defined macro variables (i.e., &DATANAME).

To display values within the macro execution:

- `%put _user_;`
- `option symbolgen;`
- `option mlogic;`

## 5. CONDITIONALLY EXECUTING DATA STEPS

Macros allow you to conditionally execute entire data steps or sections of code. One way of conditionally executing code within a macro is to use a %IF statement. Suppose that some of the time we want to create the data set (i.e., create SORT06.SD2), but not print out the list of books. We could add a parameter to our macro called "PRINTLIST" to indicate whether or not we want to print. We would also need to enclose our proc print within an %IF block. The format for %IF is:

```
%IF <condition> %THEN %DO;
   ...
   SAS Code
   ...
%END;
```

Adding %IF to our code:

```
%MACRO TOPLIST(month = 06,
     type =  Fiction,
     criteria=Volume,
     topnum= 10,
     printlist=Y);

proc sort data = sales&month
      (where = (type = "&type") )
      out = sort&month;
   by descending &criteria;
run;

%IF "PRINTLIST" = "Y" %THEN %DO;
proc print data = sort&month
      (obs = &topnum) noobs;
   var book author &criteria;
   title1 "Top &topnum &type Books
for &month.00";
   title2 "Based on &criteria";
run;
%END;

%MEND;
```

The default parameter for PRINTLIST is "Y" (i.e., print the list). But we can easily turn off the printing by calling the macro and giving "PRINTLIST" another value:

```
   %TOPLIST (type = Sci-Fi,
printlist=N)
```

The result of the above call will be to create SORT06.SD2 but not print anything. With

MLOGIC turned on, SAS will write a message to the Log indicating whether or not the %IF condition was true:

```
MLOGIC(TOPLIST):  %IF condition "PRINTLIST" =
"Y" is FALSE
```

In this case it was false.

## 6. PERFORMING HIGHLY REPETITIVE TASKS WITH LOOPS AND NESTING

If you need to repeat a task under a dozen times, then repeatedly calling a macro works fine. However, for repeating a task 1000 times this will not work.

Luckily, we can use loops and nesting of macros to perform highly repetitive tasks. Imagine for example that we wanted to print out the sales figures for each of 500 stores across the country. The sales data for each store could be contained in a data set of the form: STORE<store number>. Also imagine that while we currently have 500 stores, next month they may add more stores. Below is the format for creating a DO loop within a macro:

```
%local i;
/*creates a local macro var i*/
%DO i  = <start number> %TO <stop number>;
  ...
  SAS Code
  ...
%END;
```

Using this format we could print out the sales files for all 500 stores:

```
%MACRO PRINTSALES(first_store = ,
     last_store= );
   %local i;
   %DO i  = &first_store %TO
         &last_store;

       proc print data = store&i;
       run;

   %END;
%MEND;

%PRINTSALES(first_store=01,
            last_store=500);
```

If we wanted to print out five different reports (one for each type of book), we could "nest" one macro inside of another. For this example, we need to modify TOPLIST so that the name of the sales data is a parameter (SALEDATA). Our PRINTSALES macro would now call our TOPLIST macro for each type of book:

```
%MACRO TOPLIST(saledata=,
     month = 06,
     type =  Fiction,
     criteria=Volume,
     topnum= 10);

     proc sort data=&saledata....
     ....
     SAS CODE
     .....
 %MEND;


%MACRO PRINTSALES(first_store = ,
     last_store= );

   %local i;
   %DO i  = &first_store %TO
         &last_store;

     %TOPLIST (type = Fiction,
           saledata= store&i)
     %TOPLIST (type = Sci-Fi,
           saledata= store&i)

     %TOPLIST (type = Reference,
           saledata= store&i)
     %TOPLIST (type = Mystery,
           saledata= store&i)
     %TOPLIST(type=Non-Fiction,
           saledata= store&i)

   %END;
%MEND;

%PRINTSALES(first_store=01,
            last_store=500);
```

This code will print out 2,500 hundred reports on book sales. It is hard how this could have been accomplished without macros!

## 7. GENERATING DATA DEPENDENT CODE

Generating data dependent code is a powerful capability of the macro facility but well beyond the scope of an introduction to macros. For examples of how to generate data dependent code, please see the References.

One caveat: in examples of how to generate data dependent code, you will probably encounter double ampersands. To understand what a double ampersand is you need to understand a little bit about how the macro facility works. On each pass, the macro facility resolves one level of ampersands. Using double or even triple ampersands is a way of embedding one macro variable name in another. For example:

```
%let i=3;
%let type1=Fiction;
```

```
%let type2=Sci-Fi;
%let type3=Mystery;

%put with one ampersand:  &type&i;
%put with two ampersands:  &&type&i;
```

Will generate the following messages to the Log:

```
WARNING: Apparent symbolic reference TYPE not
resolved.
with one ampersand:  &type3
106  %put with two ampersands:  &&type&i;
with two ampersands:  Mystery
```

In the double ampersand case, “&&type&i” resolves to “&type3” on the first pass and then “&type3” resolves to “Mystery” on the second pass.

**CONCLUSION**

The macro facility is a text substitution mechanism. Through macro variables and macros, the macro facility offers many opportunities for flexible programming of complex and repetitive tasks. Macro variables are easy to use. With macro variables, even a beginning programmer can add flexibility to his or her programs. Once macro variables have been mastered, it is a short step to writing actual macros. Moving from writing basic macros to writing complex macros, however, takes more time.

**REFERENCES**


Carpenter, Art. **Carpenter's Complete Guide to the SAS Macro Language**, Cary, NC, SAS Institute., 1998.

SAS Institute, Inc. **SAS Macro Language: Reference, Version 8**, Cary, NC, SAS Institute., 1999.


**CONTACT INFORMATION**


Lisa Sanbonmatsu
Harvard University
Malcolm Wiener Center for Social Policy
79 JFK Street
Cambridge, MA 02138
Work Phone: 617-495-5131
Email: Lisa_Sanbonmatsu@usa.com