Paper 128-29

SAS® Macro: Symbols of Frustration? %Let us help! A Guide to Debugging Macros

Kevin P. Delaney, Centers for Disease Control and Prevention, Atlanta, GA
Arthur L. Carpenter, California Occidental Consultants, Oceanside, CA

ABSTRACT

Are you just learning to write MACROS? Are you baffled by when to use single quotes and double quotes - or one inside the other? Are you confused by when to use statements such as DO versus %DO? Want to know the difference between a GLOBAL and LOCAL Macro variable? Do you have macros that don't give the right results yet don't give error messages either? This workshop will introduce a few MACRO concepts that are often misunderstood. You'll also learn how to use options such as MPRINT and SYMBOLGEN, and statements such as %PUT, to help you determine why your macro doesn't run, or why it does run but gives you the wrong results.

INTRODUCTION

This paper is designed both for users who are new to the SAS Macro language and experienced users who are seeking efficient debugging techniques. We will describe both HOW TO debug a SAS Macro, and some actual bugs that people often trip over when working in the MACRO language. The first section of the paper provides a review of the various SAS SYSTEM OPTIONS that you might want to use to debug a MACRO. This is followed by several examples of problems routinely encountered in the MACRO language, and techniques for identifying and fixing those problems. We assume a basic understanding of the MACRO language, but will take the time to explain many of its intricacies that are often missed when you start to learn MACRO. Except where noted, all examples in the text represent working SAS code. When the code doesn't work, the reasons why it doesn't work and a way to make it work are usually provided (you will have to do some independent thinking, just not too much…). Excerpts from the SAS LOG that present the results of SAS Code are presented in text boxes either next to or following the example code, depending on the space required. This paper is meant to compliment examples available for download from the SUGI 2004 Hands on Workshop website; information on the location of these materials will be available at and after the conference.

HOW TO DEBUG A MACRO

HOUSTON, WE (MAY OR MAY NOT) HAVE A PROBLEM?!?

The first challenge in debugging a macro is figuring out that you have a problem. Very often this will be readily apparent, the macro will generate a warning or an error that at least let's you know there is a problem, and at best tells you exactly what that problem was. Unfortunately, the macro language will not always be able to tell you when "an error" has been made. Often, your only clue is that you don't get the results you want.

For example the following code produces the LOG shown in the box:

```
*1);
%macro wrttxt(text=Something we really want to write to the log!);
      %put text;
%mend wrttxt;
%wrttxt
```

```
text
```

In example 1 above, we have accidentally forgotten to include the '&.' Without the & to 'trigger' the macro facility 'text' is not seen as a macro variable. So SAS instead writes the letters 'text' out to the LOG, and doesn't see anything wrong with doing that!

```
*2);
%macro wrttxt(text=Something we really want to write to the log!);
      %put &txt;
%mend wrttxt;
%wrttxt
```

```
WARNING: Apparent symbolic reference TXT not resolved.
&txt
```

In this second example we remembered the '&' but misspelled the MACRO parameter name as "TXT" instead of "TEXT." SAS recognizes that we want it to write out the parameter value, but it's looking for the value of &TXT, and can't find it, so we get an error to that effect.

```
*3);
%macro wrttxt(text=Something we really want to write to the log!);
%put &text;
%mend wrttxt;
%wrttxt
```

```
Something we really want to write to the log!
```

Example 3 shows %WRTTXT doing what we wanted it to, with no errors. The purpose of this silly little example is to remind you that the only way that most errors will be detected is to remember to look for them, in both your LOG, and your output.

A LITTLE HELP FROM SOME SAS SYSTEM OPTIONS

Now that you have been reminded to check your LOG for warnings and errors, let's explore some of the SAS System options that can help you shake the errors out of your MACRO code. There are four system options that will be most helpful to you in debugging your SAS Macros: SYMBOLGEN, MPRINT, MLOGIC, and MFILE. Note that each of these options adds some (different) information to your SAS LOG; sometimes they can add A LOT of information to your LOG. Because of this it is probably a good idea to only turn them on when you are debugging a MACRO, and to turn them off when you are through with your debugging. Like any other SYSTEM OPTIONS, you turn these on using the OPTIONS statement:

```
Options symbolgen mlogic mprint mfile;
```

And turning them off is just as simple:

```
Options NoSymbolgen nomlogic nomprint nomfile;
```

Let's look at each option, and the output they produce in more detail

SYMBOLGEN

This system option tells you what each Macro variable called by your program resolves to. It can be very helpful in diagnosing problems with MACRO quoting, indirect references to MACRO variables, and problems with your code (logic, typos etc.) that lead to a macro variable with a value that is different than what you expected it to be. The example below creates combinations of different pet types, a very useful little utility! It is designed so show you nothing more than what SYMBOLGEN will print to the LOG when you turn it on.

```
*4);

options symbolgen;
%macro bighouse(Pet=Cat Dog,
                    Type=Fat Fuzzy,
                    Npets=2,
                    Ntypes=2);
      %do i=1 %to &npets;
            %do j=1 %to &ntypes;
                  %let allpets=%scan(&type,&i) %scan(&pet,&j);
                  %put &allpets;
            %end;
      %end;
%mend bighouse;

%bighouse
```

```
SYMBOLGEN:  Macro variable NPETS resolves to 2
SYMBOLGEN:  Macro variable NTYPES resolves to 2
SYMBOLGEN:  Macro variable TYPE resolves to Fat Fuzzy
SYMBOLGEN:  Macro variable I resolves to 1
SYMBOLGEN:  Macro variable PET resolves to Cat Dog
SYMBOLGEN:  Macro variable J resolves to 1
SYMBOLGEN:  Macro variable ALLPETS resolves to Fat Cat
Fat Cat
SYMBOLGEN:  Macro variable TYPE resolves to Fat Fuzzy
SYMBOLGEN:  Macro variable I resolves to 1
SYMBOLGEN:  Macro variable PET resolves to Cat Dog
SYMBOLGEN:  Macro variable J resolves to 2
SYMBOLGEN:  Macro variable ALLPETS resolves to Fat Dog
Fat Dog
...More LOG snipped for brevity...
```

MPRINT

MPRINT writes each statement generated by a Macro to the LOG. Remember, all the Macro language does is store and write text, you have to make sure that the text that is being written is valid SAS code. MPRINT is the option that will help you figure this out, by showing each line of code generated by the MACRO. The following example is taken directly from the online documentation, and represents the first example of a macro that needs debugging:

```
*5);

options mprint nosymbolgen;
%macro second(param);
   %let a = %eval(&param);a
%mend second;

%macro first(exp);
   data _null_;
      var=%second(&exp);
      put var=;
   run;
%mend first;

%first(1+2);
```

```
MPRINT(FIRST):   data _null_;
MPRINT(FIRST):   var=
MPRINT(SECOND):  a
MPRINT(FIRST):  ;
MPRINT(FIRST):   put var=;
MPRINT(FIRST):   run;

NOTE: Variable a is uninitialized.
var=.
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Notice that MPRINT wrote only the text that was generated by the Macros to the LOG, and that it provided information (in parentheses) about which MACRO generated which piece of text. The Macro %Second was supposed to evaluate the numeric expression 1+2, and pass that value to the variable VAR in the DATA step created by the Macro %First. So what went wrong? Instead of passing 3 to VAR, according to our MPRINT output, **a** was passed to the %First macro, and SAS tried to assign the value of an unknown variable A to VAR. We could turn SYMBOLGEN back on to see if it adds any more useful information:

```
*6);

options mprint symbolgen;
%macro second(param);
   %let a = %eval(&param);a
%mend second;

%macro first(exp);
   data _null_;
      var=%second(&exp);
      put var=;
   run;
%mend first;
%first(1+2)
```

```
MPRINT(FIRST):   data _null_;
MPRINT(FIRST):   var=
SYMBOLGEN:  Macro variable EXP resolves to 1+2
SYMBOLGEN:  Macro variable PARAM resolves to 1+2
MPRINT(SECOND):  a
MPRINT(FIRST):  ;
MPRINT(FIRST):   put var=;
MPRINT(FIRST):   run;

NOTE: Variable a is uninitialized.
var=.
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds
```

What is important to notice here, is not what is printed by SYMBOLGEN, but what is not printed by SYMBOLGEN. Remember, the value we wanted to be passed from %SECOND to %FIRST was 3, not a. Well, where is 3 being stored as text in %SECOND? In the Macro variable &a! Why doesn't SYMBOLGEN ever mention &a? Because whoever came up with this example forgot to ask for it! Let's try this one more time:

```
*7);

%macro second(param);
      %*Change a to &a!!!;
   %let a = %eval(&param);&a
%mend second;

%macro first(exp);
   data _null_;
      var=%second(&exp);
      put var=;
   run;
%mend first;


%FIRST(1+2)
```

```
MPRINT(FIRST):   data _null_;
MPRINT(FIRST):   var=
SYMBOLGEN:  Macro variable EXP resolves
to 1+2
SYMBOLGEN:  Macro variable PARAM resolves
to 1+2
SYMBOLGEN:  Macro variable A resolves to
3
MPRINT(SECOND):  3
MPRINT(FIRST):  ;
MPRINT(FIRST):   put var=;
MPRINT(FIRST):   run;

var=3
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds
```

MLOGIC

The MLOGIC system option tracks the flow of your MACRO code. It keeps track of the parameter values, and tracks the logic that drives %DO loops and %IF logic checks. It is useful for diagnosing problems with the logic and flow of your Macro code, when you think that the problem with your Macro lies in the way it was written or is executing, rather than problems with the code it generates. MLOGIC can write an awful lot of information to the LOG, so it should only be used for debugging, and turned off when you don't really need it. For an example, let's make a slight modification to the %BIGHOUSE Macro (it's proving more useful than I thought), adding one more logic check, to see the type of information that MLOGIC can add to the LOG. *(Logistical issues have made it impossible to keep the MLOGIC LOG output for this example on one page, or to put it in a text box!)*

```
*8);
options mlogic;
%macro bighouse(Pet=Cat Dog,
                Type=Fat Fuzzy,
                Npets=2,
                Ntypes=2);
      %do i=1 %to &npets;
            %do j=1 %to &ntypes;
                  %let allpets=%scan(&type,&i) %scan(&pet,&j);
                  %if %scan(&allpets,2)=Cat %then %put &allpets;
            %end;
      %end;
%mend bighouse;

%bighouse
```

The LOG shows:

```
MLOGIC(BIGHOUSE):  Beginning execution.1)
MLOGIC(BIGHOUSE):  Parameter PET has value Cat Dog
MLOGIC(BIGHOUSE):  Parameter TYPE has value Fat Fuzzy   2)
MLOGIC(BIGHOUSE):  Parameter NPETS has value 2
MLOGIC(BIGHOUSE):  Parameter NTYPES has value 2
MLOGIC(BIGHOUSE):  %DO loop beginning; index variable I; start value is 1; stop value is 2; by
                   value is 1.                                                          3)
MLOGIC(BIGHOUSE):  %DO loop beginning; index variable J; start value is 1; stop value is 2; by
                   value is 1.
```

```
MLOGIC(BIGHOUSE):  %LET (variable name is ALLPETS)
MLOGIC(BIGHOUSE):  %IF condition %scan(&allpets,2)=Cat is TRUE 4)
MLOGIC(BIGHOUSE):  %PUT &allpets
Fat Cat
MLOGIC(BIGHOUSE):  %DO loop index variable J is now 2; loop will  iterate again.
MLOGIC(BIGHOUSE):  %LET (variable name is ALLPETS)
MLOGIC(BIGHOUSE):  %IF condition %scan(&allpets,2)=Cat is FALSE 5)
MLOGIC(BIGHOUSE):  %DO loop index variable J is now 3; loop will not iterate again.
MLOGIC(BIGHOUSE):  %DO loop index variable I is now 2; loop will  iterate again.      6)
MLOGIC(BIGHOUSE):  %DO loop beginning; index variable J; start value is 1; stop value is 2; by
                   value is 1.
MLOGIC(BIGHOUSE):  %LET (variable name is ALLPETS)
MLOGIC(BIGHOUSE):  %IF condition %scan(&allpets,2)=Cat is TRUE
MLOGIC(BIGHOUSE):  %PUT &allpets
Fuzzy Cat
MLOGIC(BIGHOUSE):  %DO loop index variable J is now 2; loop will  iterate again.
MLOGIC(BIGHOUSE):  %LET (variable name is ALLPETS)
MLOGIC(BIGHOUSE):  %IF condition %scan(&allpets,2)=Cat is FALSE
MLOGIC(BIGHOUSE):  %DO loop index variable J is now 3; loop will not iterate again.
MLOGIC(BIGHOUSE):  %DO loop index variable I is now 3; loop will not iterate again.
MLOGIC(BIGHOUSE):  Ending execution.7)
```

1) MLOGIC keeps track of the beginning and 7) end of each MACRO that is called.

2) MLOGIC also keeps track of the parameter values

3) MLOGIC keeps track of the DO loop values, at the start (3) and as they iterate and end (6)

4) MLOGIC also keeps track of the evaluation of MACRO expressions that control program flow, such as %IF, and tells you whether the evaluation is true (4) or false (5)

We saw before what would have been added to this LOG, if we had the SYMBOLGEN option turned on for this program. What would have been added if we had used MPRINT? Why?

MFILE

Similar to MPRINT, the MFILE option can be used to write out the resolved macro code to a file. To take advantage of this option, use a FILENAME statement with a *fileref* of MPRINT, and turn on both the MPRINT and MFILE options.

```
*9);
FILENAME MPRINT 'C:\MYMACROS\MACCODE.SAS';
OPTIONS MPRINT MFILE;
```

As each subsequent macro is called, the resultant code is written to the file MACCODE.SAS. You can turn off the writing of these statements by issuing either a NOMPRINT or a NOMFILE option. Since only the resolved code is written, macro statements like %LET and %PUT will not be shown.

WHEN AND WHY TO DEBUG A MACRO, OR, HOW DID I GET MYSELF INTO THIS

Now that we have some basic tools to use to debug a Macro program, let's explore some of the problems that are often encountered when using the MACRO language.

IS IT THE DATA STEP, OR IS IT MACRO?

One of the main problems experienced by people who know some SAS and want to learn to write Macros is distinguishing between when to write BASE SAS code, and when to write MACRO code. There are many similarities between BASE SAS and MACRO code, but there are enough important differences to easily trick or confuse a MACRO novice. Perhaps the most important thing to remember when mixing DATA step and MACRO code is that MACRO statements and MACRO calls execute before the DATA step. This is a MACRO fundamental that sometimes gets overlooked. Exploring the following silly example (of course none of YOU would ever write code like this…) will provide a demonstration:

```
*9);

data mixmac;
set sashelp.class;
  if _N_=1 then do;
      put age=;
      if age < 14 then %let group=JRHIGH;
      else %let group=HIGH;
  end;
run;

%put &group;
```

```
31   data mixmac;
32   set sashelp.class;
33   if _N_=1 then do;
34   put age=;
35   if age < 14 then %let group=JRHIGH;
36   else %let group=HIGH;
37   end;
ERROR: No matching IF-THEN clause.
38   run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.MIXMAC may be incomplete.  When this step was stopped there were 0
         observations and 5 variables.
WARNING: Data set WORK.MIXMAC was not replaced because this step was stopped.
NOTE: DATA statement used:
      real time           0.03 seconds
      cpu time            0.03 seconds


39
40   %put &group;
HIGH
```

Let's see what happened here, in "slow motion." SAS scans the entire submitted piece of code for tokens that tell it that there are MACRO statements to execute, it is looking for MACRO variables (that start with &) or MACRO statements/calls that start with %. In the code above there are two such tokens, which indicate the %LET statements. Finding these, SAS executes the MACRO statements, the first %LET assigns the Macro variable &GROUP the value JRHIGH, the second %LET assigns the same MACRO variable the value HIGH. After taking care of it's MACRO responsibilities, SAS returns to do a syntax check on the remaining SAS DATA step code. But what code is left for SAS to run?

```
data mixmac;
set sashelp.class;
if _N_=1 then do;
put age=;
if age < 14 then
else
end;
run;

%put &group;
```

Obviously the DATA step statement `if age < 14 then else end;` is invalid, so the ERROR: No Matching IF-THEN clause is produced, and the DATA step doesn't compile or execute. Notice that, despite the fact that the

DATA step didn't execute, the MACRO variable &GROUP was assigned a value, HIGH, which would have been incorrect given the actual data in the first observation of SASHELP.CLASS. Also notice that there is no error, warning or anything other than our friendly %PUT statement that would allow us to know that &GROUP had been assigned a value, let alone the wrong value.

The problem is that, because of the timing issues mentioned above (*e.g.* macro code is executed first), you can **NEVER** use DATA step logic to conditionally execute macro statements.

IF OR %IF

This is a distinction that is confused more often than one might think, especially when people are first learning Macros. It is fairly common to use an IF statement that involves the evaluation of one or more Macro variables, but it is almost impossible to conceive of a %IF statement that could be used to evaluate an expression involving data step variables. %IF can only be used within a MACRO declaration, to control what code is written or how the logic is evaluated within the MACRO. An IF STATEMENT can be used in a MACRO, but will be executed as part of DATA step code that is written out by the MACRO. These distinctions are illustrated below.

```
*10);

%macro whatif(condition=gt 14);
 Data subset;
   set sashelp.class;
     if age &condition then output;
 run;
%mend;
```

Could we use a %IF instead of an IF? Notice we added a second semi-colon in the next example.

```
*11);

%macro whatif(condition=gt 14);
 Data subset;
   set sashelp.class;
%if age &condition %then output;;
 run;
%mend;
```

There are no macro or DATA step errors. The %IF executes successfully, and under Windows results in a TRUE comparison e.g. age is gt 14. However there are logic errors, and every single observation from SASHELP.CLASS, regardless of the value of the variable AGE, will be written to the data set SUBSET!

Since macro code is executed before the DATA step is even compiled AGE in the %IF is not seen as a DATA step variable but rather as the letters a-g-e. Since numbers are smaller than letters alphabetically, the letter 'a' comes after 14. The DATA step code becomes:

```
Data subset;
set sashelp.class;
output;
run;
```

Ok, we couldn't use %IF in the last example, so when should we use it? The ability to change the subsetting criteria in the IF statement is the only thing that the Macro from example 10 is designed to accomplish. This is in contrast to the Macro in example 12, which takes the %WHATIF Macro a step further. In addition to parameterizing the subsetting within the DATA step, this MACRO uses %IF logic to control whether only the subset, or the subset and a bootstrap sample of the subset are output. *(NOTE: This is not the most efficient way to create a bootstrap sample, and is intended only for demonstration purposes.)* In this case the %IF is necessary to control the logic flow in the MACRO, and it has nothing to do with the logic of the DATA step that is written by the MACRO.

```
*12);

%macro Ifagain(condition=gt 14,
                        subsuper=1);
```

```
%if &subsuper=1 %then
      %do;
      data subset;
      set sashelp.class;
      if age &condition then output;
      run;
      %end;
%else %if &subsuper=2 %then
      %do;

      data subset;
      set sashelp.class;
      if age &condition then do;
      output;
      end;
      run;

      DATA subset;                              /* count size of subsample */
      SET subset END=lastobs;
      IF lastobs THEN CALL SYMPUT('ngp',_n_);
      RUN;

      data bootstrap;
      seed=123456789;
      DO sample=1 to 10000;
     DO j = 1 TO &ngp;                    /* generate bootstrap sample */
        CALL RANUNI(seed,r);
        point = CEIL(r*&ngp);
      set subset point=point;
      if _error_ then abort;
      output;
       end;
      end;
      drop j r seed;
      stop;
  RUN;

 %end;
%mend;
```

```
MLOGIC(IFAGAIN):  Beginning execution.
MLOGIC(IFAGAIN):  Parameter CONDITION has value gt 12
MLOGIC(IFAGAIN):  Parameter SUBSUPER has value 2
SYMBOLGEN:  Macro variable SUBSUPER resolves to 2
MLOGIC(IFAGAIN):  %IF condition &subsuper=1 is FALSE
SYMBOLGEN:  Macro variable SUBSUPER resolves to 2
MLOGIC(IFAGAIN):  %IF condition &subsuper=2 is TRUE

... Log snipped since the point was made...
```

```
%ifagain(condition=gt 12, subsuper=2);
```

Notice the bolded section of the LOG, part of the MLOGIC output that describes what %IF adds to this Macro. It only allows for branching logic in the Macro, but does not (**can not**) be used to evaluate a DATA step variable.

MISSING VS NULL VALUE COMPARISONS

Another big difference between DATA step logic and Macro logic is the treatment of missing values. In the DATA step, there is no such thing as a truly NULL value. All Character variables have a value, if they are missing, SAS uses a single blank space ( ) for the variable value. Numeric variables are treated similarly, with a period (.) as the value most often used to represent a missing value. In the MACRO language there are no characters used to represent a missing value, if a MACRO variable is NULL, it really is NULL, that is, it truly has no value.

For example in the DATA step we might check to see if the variable AGE is missing with:

```
if gender = "" then do;
```

but, if we tried something similar within the MACRO language using %IF logic, e.g.

```
%if &age = "" %then %do;
```

we end up with a statement that is only evaluated as true when the value of &AGE is a pair of quotes separated by a space (“ “) which is not the same as a null value. To test a null value in the MACRO language you could just leave a single blank space after the equals sign

```
%if &age = %then %do;
```

Many programmers who are used to making comparisons in the DATA step have reservations about this type of statement; the lack of a value on the right side of the equal sign makes them uncomfortable. One common solution is to use quote marks on both sides of the equal sign, to mark the comparison operators.

```
*13a);

%macro sillytest(age=);
%if "&age" = ""  %then
      %do;
      %put This comparison worked;
      %end;
%else
      %do ;
      %put I guess it did not work;
%end;
%mend;
%sillytest;
```

A better solution, which stays within the Macro language and still satisfies the need to place something on the right hand side of the comparison operator, is to use a Macro quoting function with no argument or spaces between the parentheses:

```
*13b);

%macro sillytest(age=);
%if &age = %str()  %then
      %do;
      %put This comparison worked;
      %end;
%else
      %do ;
      %put I guess it did not work;
%end;
%mend;
%sillytest;
```

SCAN VS. %SCAN (AND %QSCAN)

The %SCAN function is a particularly useful Macro function, that is frequently used with %DO loop processing to act on a Macro variable that acts as an array, with multiple Macro values saved in one long string separated by spaces or commas. Such a string can easily be constructed using the INTO command in PROC SQL:

```
*14);

proc sql;
select name into: names separated by " "
from SASHelp.class;quit;

%put &Sqlobs Names added to Macro
      variable (array) %nrstr(&NAMES) ;
```

```
proc sql;
select name into: names separated by " "
from SASHelp.class; quit;
NOTE: PROCEDURE SQL used:
      real time           0.02 seconds
      cpu time            0.02 seconds
%put &Sqlobs Names added to Macro variable
(array) %nrstr(&NAMES) ;
19 Names added to Macro variable (array) &NAMES
```

Lists can also be specified as a macro parameter that is submitted to %SCAN (this technique will be illustrated in the example code below). The problem with %SCAN is that it looks a lot like it's DATA step cousin the SCAN function, but it differs in one important respect, how you define the delimiters in the list that is going to be scanned. Programmers who are familiar with the DATA step SCAN function may know that the 3rd argument to the function, the delimiter of the list is optional, and that the default is to look for any of the following characters **blank . < (+ & ! $ *) ; ^ - / , %** | as potential delimiters. However, if you only want a blank space or a comma and not both to be considered delimiters, then the DATA step SCAN function asks that you put the delimiter you want in quotes. MOST of this is also true of the Macro %SCAN function, except the part about specifying a single delimiter in quotes. This distinction is illustrated below:

DATA step

```
*15a);
data _null_;
test="AA B'C D'C";
i=1;
do until(scan(test,i,' ')=" ");
word=scan(test,i,' ');
put word;
i=i+1;
end;
run;
```

Gives Us:

```
AA
B'C
D'C
NOTE: DATA statement used:
      real time           0.83 seconds
      cpu time            0.08 seconds
```

MACRO

```
*15b);
%macro breakstrg(string=);
%let i=1;
%do %until (%qscan(&string,&i,' ')=
%str());
%let word=%qscan(&string,&i,' ');
%put &word;
%let i=%eval(&i+1);
%end;
%mend;

%breakstrg(string=AA B'C D'C);
```

Gives something different (and wrong) in the log:

```
AA
B
C
D
C
```

Why did that happen? In the DATA step, we use quotes in the delimiter (3rd argument of the SCAN function) because we need to tell SAS what **tex**t to look for to break up the list. In the MACRO language, everything is text, so when we include the quotes in the third argument we are actually adding an extra delimiter to the %SCAN function, so that it will be looking for either a quote mark (') or a blank space ( ) as delimiters. In order to preserve the space as a single delimiter in the %SCAN function, we should use a Macro quoting function to mark it, rather than the quote marks we would use in the DATA step:

*15c);

```
%macro breakstrg(string=);
%let i=1;
  %do %until (%qscan(&string,&i,%str( ))= %str());
    %let word=%qscan(&string,&i,%str( ));
    %put &word;
    %let i=%eval(&i+1);
  %end;
%mend;

options mlogic symbolgen;

%breakstrg(string=AA B'C D'C)
```

```
AA
B'C
D'C
```

If your delimiter for the %SCAN function was a comma (,) how do you think you would list that in a %SCAN function, just to be safe?

If I were you, my next question would be, OK, so you just talked for almost a whole page about %SCAN, but in your examples, you used %QSCAN, what gives? Very good question, so glad you asked! The easiest way to find the answer would be to try changing the %QSCAN functions to %SCANs in the %BREAKSTRG macro. What happens?

Because the result of the %SCAN function was anticipated to contain a special character, in this case a single unmatched quote (‘), we have to use one of the Macro quoting functions to mask the single quote, so that the code will run without errors. A similar situation arose in the PROC SQL example (#14) that started this section, when we had to use the Macro quoting function %NRSTR to mask the & sign and prevent &NAMES from being resolved in our %PUT statement. Macro quoting is a nightmare unique to the MACRO language. Entire SUGI papers and a Section in “the” book written by one of the authors have been devoted to this topic; these resources provide eloquent explanations of some of the problems that can arise due to macro quoting, and how to fix them. The short answer is, if you have Macro variable values or results of Macro functions that MIGHT contain any of the following special characters: **& % ' " ( ) + - * / < > = ¬ ^ ~ ; , blank AND OR NOT EQ NE LE LT GE GT** (or in SAS 9 **IN**) then you may well need at least one Macro quoting function. Which one(s) you will need, and where you need them will depend on the specific situation you are dealing with. Thankfully, situations in which you have to deal with Macro quoting problems are fairly infrequent, and if you are writing a Macro that will have a lot of Macro quoting issues, you hopefully have gained the experience necessary to work through them. I refer the reader who wants or needs to explore this area of the Macro language in more detail to one or more of the aforementioned references.

MACRO MATH – HANDLING OF ARITHMATIC OPERATIONS IN THE MACRO LANGUAGE

Hmm, seeing Macro and Math in the same sentence is sort of counter intuitive to me. The MACRO language is designed to process TEXT, that’s really all it knows how to do. As a result, all kinds of interesting issues can arise when we ask the MACRO language to do math. There are two functions that allow the language to handle arithmetic operations, %EVAL and %SYSEVALF. %EVAL can only do integer math, anything that contains a decimal place (e.g. 1.0, .3, even 2.) will need to be handled by %SYSEVALF. This is very important to remember when coding an arithmetic operation (explicitly or implicitly) in your Macro code. This is one instance where the error generated by the Macro facility is a dead give away. In a situation where you need to use %SYSEVALF and you have used %EVAL, you will get the error:

```
ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric
operand is required. The condition was: _________
```

Another important thing to remember when doing “math” in the Macro language is that, unlike the DATA step, an iterative %DO loop in the Macro language must have integers for its START, END and INCREMENT values, this can lead you down some pretty extreme roads when you get into a situation that is best suited for a %DO loop with a decimal increment. Example 16 shows a Macro that creates data sets with different numbers of observations as a result of varying the cutoff (in some cases by .01!!) of the inclusion criteria. It is designed to demonstrate the extreme measures that need to be taken if you want your %DO loop to iterate with a non-integer value. The Macro checks for a decimal in the START value for the %DO loop, and uses %SYSEVALF to handle the non-integer math where necessary. It may be a bit of overkill, but did you really expect doing math with TEXT to be easy? *Fortunately for the kids in the SASHELP.CLASS data set, but not great for this example, is the fact that they are all pretty skinny and healthy for their age, which resulted in my having to use a pretty extreme cutoff to classify any of them as overweight.*

```
*16);

%macro chgcutpt(Start=,End=,inc=);
%if %index(&start,.) gt 0 %then
      %do;
      %let decnum=%eval(%length(&start)-%index(&start,.));
      %put &decnum;
      %let ints=%sysevalf(&start*10**&decnum);
      %let inte=%sysevalf(&end*10**&decnum);
      %let inti=%sysevalf(&inc*10**&decnum);

            %do i=&ints %to &inte %by &inti;
            %let name=%substr(&i,1,%eval(%length(&start)-&decnum-
                        1))_%substr(&i,%eval(%length(&start)-&decnum));
            data fat_&name;
            set sashelp.class;
            bmi=(weight/2.2)/(height/39)**2;
            if bmi gt %sysevalf(&i/10**&decnum) then output ;
```

```
            run;
            %end;
      %end;
%else
      %do i=&start %to &end %by &inc;
            data fat_&i;
            set sashelp.class;
            bmi=(weight/2.2)/(height/39)**2;
            if bmi gt &i then output ;
            run;
      %end;

%mend;
%chgcutpt(start=19,end=21,inc=1);
%chgcutpt(start=20.5,end=21.5,inc=.2);
%chgcutpt(start=21.05,end=21.10,inc=.01);
```

There is one more gotcha in the world of Macro Math. It lies in how the Macro language handles range comparisons. The Macro %CHECKIT shown below is supposed categorize values that are either in the range of –5 to 0 (Negative) or in the range of 1 to 5 (positive) and flag values that are outside both of these ranges. That should be easy to do, right?

```
*17a);
%Macro checkit(val=);
%if -5 le &val le 0 %then %put &val is in neg range (-5 to 0);
%else
      %if 1 le &val le 5 %then %put &val is in pos range (1 to 5);
%else
      %put &Val is WAY out of Range;
%mend checkit;

%checkit(val=-10);
%checkit(val=-2);
%checkit(val=2);
%checkit(val=10);
```

```
831  %checkit(val=-10);
-10 is in neg range (-5 to 0)
832  %checkit(val=-2);
-2 is in pos range (1 to 5)
833  %checkit(val=2);
2 is in pos range (1 to 5)
834  %checkit(val=10);
10 is in pos range (1 to 5)
```

Ok, so maybe it's not that easy! What happened here? In the DATA step this type of range check is used by programmers all the time, the code below works correctly:

```
*17b);
data _null_;
do val=-10,-2,2,10;
if -5 le val le 0 then do;
put Val " is in the negative range";
end;
else if 1 le val le 5 then do;
put Val " is in the positive range";
end;
else do;
put Val " is WAY out of range";
end;
end;
run;
```

```
-10  is WAY out of range
-2  is in the negative range
2  is in the positive range
10  is WAY out of range
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.00 seconds
```

The DATA step interprets the range comparison as if it had been coded as

```
if -5 le val and val le 0 then do;.
```

The Macro Language does not interpret the expression the same way. The first %IF in %CHECKIT is interpreted as

```
%if (-5 le &val) le 0 %then %put &val is in neg range (-5 to 0);
```

and when &val is –10 the %IF becomes

```
%if (-5 le -10) le 0 %then %put &val is in neg range (-5 to 0);
```

Since the comparison which checks to see if –5 is less than or equal to –10 is false, a zero is returned and the expression becomes

```
%if 0 le 0 %then %put &val is in neg range (-5 to 0);
```

which is of course evaluated as true. The moral of the story here is that when you want to do a range check in the Macro Language, you **MUST** use a compound expression. In order for %CHECKIT to work correctly it needs to be:

```
*17c);
%Macro checkit(val=);
%if -5 le &val and &val le 0 %then %put &val is in neg range (-5 to 0);
%else
      %if 1 le &val and &val le 5 %then %put &val is in pos range (1 to 5);
%else
      %put &Val is WAY out of Range;
%mend checkit;

%checkit(val=-10);
%checkit(val=-2);
%checkit(val=2);
%checkit(val=10);
```

```
%checkit(val=-10);
-10 is WAY out of Range
%checkit(val=-2);
-2 is in neg range (-5 to 0)
%checkit(val=2);
2 is in pos range (1 to 5)
871  %checkit(val=10);
10 is WAY out of Range
```

DEROGATORY COMMENTS

In the DATA step and in open code you can use two different types of comments. Comments of the type

```
*This code will not run;
```

comment out one line of text, from the * to the Semi-colon. Comments denoted by /*…*/ can be used to comment out part of a line, or multiple lines of SAS code:

```
*18a);
Data _Null_;
*this whole line won't run;
put "Some" /*don't print*/
 " of this line will print";
/*do until (comment=ended);
put comments like these can also extend for
multiple lines;
*/
run;
```

```
Some of this line will print
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The second type of comment with /**/ will work equally well in the Macro language, but sometimes using the first type of comment **inside a Macro definition** can cause problems.

```
*18b);
%macro tryit;
*%let dattype=ae;
%let dattype=demog;
*%let dattype=meds;
**********************;

%put data type is &dattype;
%mend tryit;

%tryit
```

```
11   %macro tryit;
12   *%let dattype=ae;
13   %let dattype=demog;
14   *%let dattype=meds;
15   **********************;
17   %put data type is &dattype;
18   %mend tryit;
20   %tryit
data type is meds
```

To understand what happened here, we need to remember how the code is being parsed by SAS. The parser breaks down the code into pieces known as tokens. All SAS statements start with an identifying set of characters known as a KEYWORD. The keyword tells SAS how to interpret the remaining characters up until the next semi-colon, which signifies the end of the statement. In open code (including the DATA step) the asterisk (*) is recognized as a KEYWORD that begins a SAS statement, and tells SAS to comment out all the characters between it and the next semi-colon. This is even true for %Let or %*Macro-name* statements within the comment. Thus in open code it is possible to comment out Macro statements and Macro calls with the asterisk style comments.

Inside a macro things work differently! Once the %Macro statement is encountered all the code between the %MACRO and %MEND is passed to the macro facility. Within the macro facility, when the code is parsed SAS looks for macro statements and generally ignores most of the things that have meaning in the BASE language, thinking they are part of the code to be written out by the Macro. Thus, asterisks are not seen as tokens and are not interpreted as a keyword that starts a comment. In our %TRYIT macro shown again below, the "big and bold" statements are the ones executed by the Macro:

```
%macro tryit;
  *%let dattype=AE;
   %let dattype=DEMOG;
  *%let dattype=MEDS;
  **********************;
   %put data type is &dattype;
%mend tryit;
```

All three %LET statements are executed; the last one assigns MEDS as the value of &DATTYPE.

There is of course a way to comment Macro statements inside a Macro declaration, actually there are two ways. As I stated above, you could use the /**/ style comments, or you could use a Macro comment %*. Macro comments are the equivalent of the Asterisk style comment for Macro statements. The %* is the KEYWORD within a Macro declaration that tells the Macro facility, 'all characters up to the next semi-colon are to be treated as a comment'. Thus by simply reversing the order of the %* in the example above, we use Macro comments to comment out the two %LET statements we don't want to run:

```
%macro tryit;
%*let dattype=ae;
%let dattype=demog;
%*let dattype=meds;
**********************;

%put data type is &dattype;
%mend tryit;

%tryit
```

```
1    %macro tryit;
2    %*let dattype=ae;
3    %let dattype=demog;
4    %*let dattype=meds;
5    **********************;
6
7    %put data type is &dattype;
8    %mend tryit;
9
10   %tryit
data type is demog
```

Two more important notes about our Asterisk style comments; after the Macro statements have executed, the asterisks remain, so the code that is passed out of %TRYIT becomes:

```
%macro tryit;
*

*
**********************;


%mend tryit;
```

Lucky for us, in this case this is just one long comment with two extra asterisks added to the front of the string of asterisks at the end of the Macro, but there could easily be a situation where the left over Asterisks (without a semi-colon) create a comment where it was not wanted.
The final point here is that, although it might seem ideal, our solution is not as simple as "Never use asterisk style

comments within a Macro definition." Neither Macro comments (started with %*) nor /**/ style comments are passed out of a Macro. There may be a situation where you do want your comment to continue in the code that is passed out of a macro, or to at least show up in your MPRINT. For example, when we turn on MPRINT to show that the two extra asterisks are passed out of %TRYIT, we could also add real Asterisk style comments to keep track of the extras:

```
*18c);

options Mprint;
%macro tryit;
*Look that asterisk sticks around, and messes up my next comment!!;
*%let dattype=ae;
%let dattype=demog;
*So does this one, and it is added to the other comment below it;
*%let dattype=meds;
**********************;
/*Too bad, but these comments do not show up in MPRINT*/
%*Which means they are not in the generated code;
%*Or Passed out of the Macro :-(;
%put data type is &dattype;
%mend tryit;

%tryit
```

```
MPRINT(TRYIT):   *Look that asterisk sticks around, and messes up my next comment!!;
MPRINT(TRYIT):   * *So does this one, and it is added to the other comment below it;
MPRINT(TRYIT):   * **********************;
data type is meds
```

So if you want to use your comments to Mark up your LOG while you are debugging a Macro (while you are using MPRINT), careful use of the Asterisk style comment may prove helpful.

STRICTLY MACRO PROBLEMS

To this point we have pretty much focused on problems that arise when programmers confuse Macro language statements and programming with DATA step statements and programming. There are of course, several other 'traps' that you can fall into without ever leaving the Macro language.

WHERE DOES YOUR MACRO VARIABLE LIVE – GLOBAL VS. LOCAL MACRO VARIABLES

There are several ways to create a Macro variable. How and when your variable is created determines whether the Macro variable is a GLOBAL or LOCAL macro variable. What does that mean? When a Macro variable is created, the Text assigned to the Macro variable value is stored, remember, that's all a Macro variable is, stored text. WHERE is it stored, that's the tricky part! Global macro variables are typically defined in **open code** with either a %LET statement, a Call SYMPUT statement or with the PROC SQL; SELECT INTO: syntax that was shown in example 14. Local variables can be defined using any of the same methods inside a Macro, as parameters to a macro, or as an index for an iterative %DO loop. Global macro variables can be used anywhere, local Macro variable can only be used inside the Macro in whose symbol table they reside. It is not uncommon to lose track of which symbol table your Macro variable lives in, but it is just as easy to track them down. An indication of a typical problem that arises when you lose a Macro variable in the wrong symbol table is the message:

```
WARNING: APPARENT SYMBOLIC REFERENCE AMVAR NOT RESOLVED.
```

This warning appears in your LOG when SAS can't find the Macro variable you asked for in the symbol table that you asked it to look in. When you get this message and you KNOW that the variable is defined somewhere, either the variable was placed in the wrong symbol table, or you unwittingly asked SAS to look for the Macro variable in the wrong place. Assuming that the Macro variable does exist, you may want to look for it in a symbol table that was not available when you got the warning message. You can use a %PUT statement to identify the macro variables in various symbol tables, to help you track down your lost Macro variable. Example 19 creates three macro variables with three different scopes,

1) The Macro Variable &A is created in open code by a %LET statement, it is a **Global Macro variable**

2) The Macro variable &AT is created by a %LET Statement in the macro %TRYIT, it is a **Local macro variable** in the symbol table for %TRYIT

3) The Macro variable &B is created by a %LET statement in the macro %INSIDE, it is a **Local Macro variable** in the symbol table for %INSIDE

and illustrates which macro variables are available at several points in the program.

```
*19);
%let a= global_var;

%Macro tryit;
      %let at= var_local_to_tryit;
      %put **1)Can we get to B, in the inside symbol table?;
      %put &B;
      %put;
      %inside
      %put;
      %put **4)How about now, after running the Inside Macro?;
      %put &B;
%mend tryit;

%Macro inside;
      %let b=var_on_the_inside_table;
      %put ** 2)We can call to TRYIT from here, it is a higher table;
      %Put &AT;
      %put;
      %put **3)All Current Vars;
      %put _user_;
%mend inside;

%tryit
%put *** 5)Open Code Macro
            Vars;

%PUT _USER_;
```

```
160  %TRYIT
**1)Can we get to B, in the inside symbol table?
WARNING: Apparent symbolic reference B not resolved.
&B

**2) We can call to TRYIT from here, it is a higher table
var_local_to_tryit

**3)All Current Vars
INSIDE B var_on_the_inside_table
TRYIT AT var_local_to_tryit
GLOBAL A global_var

**4)How about now, after running the Inside Macro?
WARNING: Apparent symbolic reference B not resolved.
&B
161  %put ***5)Open Code Macro Vars;
***5)Open Code Macro Vars
162  %put _User_;
GLOBAL A global_var
```

As you can see it can be very easy to lose track of a macro variable, if you don't know which symbol table it was written to, or how to gain access to it. Number 1 from example 19 shows that we can not call to a Macro variable in a lower symbol table from a higher level macro. Number 2 shows however, that we CAN call to a higher level table from a lower level one. When you call one Macro from inside another, all the Macro variables from the first Macro are then available to the second Macro that will execute "inside" the first. This is also shown by the %PUT statement in Number 3, which shows us that the Inside Macro has access to the Global variable &A, the variable &AT from %TRYIT, and, of course, the variable &B that was created in the %INSIDE macro. Number 4 shows us that even though &B has been defined in %INSIDE, which was already called by %TRYIT and has already executed, because it was LOCAL to %INSIDE we still can't get to it from the %TRYIT macro. Finally, Number 5 shows us that the only Macro variable that we can use in open code is the GLOBAL macro variable that had been defined in open code. In the next section we will look at some of the more catastrophic consequences of losing track of which symbol table a macro variable lives in, and describe how to change the symbol table that a given Macro variable is written to, using the %GLOBAL and %LOCAL statements.

A TRAIN WRECK WAITING TO HAPPEN – WHEN MACRO VARIABLES COLLIDE

As we saw in the last section, keeping track of where Macro variable values are written to can be rather tricky. If a programmer loses track of where Macro variable values are being written to, a Macro variable collision can occur. A Macro variable collision occurs when the value of one Macro variable interferes with (replaces or changes) the value

of a similarly named macro variable that should be in a separate symbol table. There are several ways that this could happen but I will present two common ones that used to sneak up on me quite frequently.

The Macro %GETIT in example 20a uses PROC SQL and SELECT INTO: to create a Macro variable list out of all the values of a given variable. For our example it is going to select all the values of the data set variable NAME from SASHELP.CLASS and put them into the Macro variable &ALLNAMES. Those of you familiar with this type of syntax may know that when you select information with PROC SQL, the procedure keeps track of the number of observations that it selected. This information is 'automagically' stored in a Macro variable called &SQLOBS. In %GETIT, I have taken the value from &SQLOBS and saved it to the Macro variable &N (Why I did this should become apparent shortly). I also used a %PUT statement to write out the variables available to me in %GETIT.

```
*20a);

%macro getit(dset=SASHELP.CLASS,var=Name);
proc sql;
select &var into: All&var.s separated by ","
from &dset;
quit;
%let n=&sqlobs;
%put _user_ ;
%mend getit;


%GETIT
```

What does the `%PUT _USER_ ;` show in the LOG??

```
GETIT SQLOBS 19
GETIT SQLOOPS 94
GETIT ALLNAMES
Alfred,Alice,Barbara,Carol,Henry,Jame,
Jane,Janet,Jeffrey,John,Joyce,Judy,Lou
ise,Mary,Philip,Robert,Ronald,Thomas,W
illiam
GETIT VAR Name
GETIT DSET SASHELP.CLASS
GETIT N 19
GETIT SQLXOBS 0
GETIT SQLRC 0
```

Not surprizingly, it shows that all of the Macro variables created in %GETIT are LOCAL to that Macro. Well maybe it is a little surprizing? The fact that &SQLOBS (and several other Macro variables created by PROC SQL) are local to %GETIT because they were created in %GETIT could catch you off guard, if you go to use &SQLOBS outside %GETIT. Such as:

```
173  %put &sqlobs &N &ALLNAMES;
WARNING: APPARENT SYMBOLIC REFERENCE SQLOBS NOT RESOLVED.
WARNING: APPARENT SYMBOLIC REFERENCE N NOT RESOLVED.
WARNING: APPARENT SYMBOLIC REFERENCE ALLNAMES NOT RESOLVED.
```

But, we can fix our little problem, we just need to use a %GLOBAL statement to make &N and &ALLNAMES available outside of %GETIT!

```
*20b);

%macro getit(dset=SASHELP.CLASS,var=Name);
%Global All&var.s N;
proc sql;
select &var into: All&var.s separated by ","
from &dset;
quit;
%let n=&sqlobs;
%put _user_ ;
%mend getit;

%getit
%PUT _USER_ ;%PUT &SQLOBS;
```

```
GLOBAL ALLNAMES
Alfred,Alice,Barbara,Carol,Henry,James,
Jane,Janet,Jeffrey,John,Joyce,Judy,Loui
se,Mary,Philip,Ro
bert,Ronald,Thomas,William
GLOBAL N 19
WARNING: Apparent symbolic reference
SQLOBS not resolved.
&sqlobs
```

Ok, so now &N and &ALLNAMES can be used outside of %GETIT. &SQLOBS is still not found outside of %GETIT because we haven't made it GLOBAL. Now comes the fun part!! Suppose you are running %GETIT when you have already used PROC SQL for something else, anything else, where will the value of &SQLOBS be written? NOT to the %GETIT symbol table! &SQLOBS is a somewhat unique Macro variable in that although it is not considered an AUTOMATIC Macro variable, it is 'AUTOMAGICALLY' recreated with every Select statement that PROC SQL executes. So if you have run PROC SQL **in open code** at any point before running %GETIT the value of &SQLOBS will be written to the GLOBAL symbol table (See example 21a below). Hence it is good practice to grab the value of &SQLOBS right after it is created and set it to something (either GLOBAL or LOCAL) that will not get overwritten. It is

also good practice to make sure values that you want to be LOCAL (whether you ask for them to be set or not) are in fact LOCAL, by defining them with the %LOCAL statement, this practice is demonstrated in example 21b below.

```
*21a);

Proc sql;
select * from sashelp.class
where age gt 14;
quit;

Options MPRINT;
%GETIT(Dset=Sashelp.class(where=(Sex="M" and Age gt 14)),var=Name)

Proc print data=Sashelp.class;
Title "There are &N boys and &SQLOBS total Observations from SASHELP.CLASS
with AGE gt 14";
where age gt 14;

RUN;
```

Produces (in the OUTPUT window):

There are **3 boys and 3** total Observations from SASHELP.CLASS with AGE gt 14

Obs	Name	Sex	Age	Height	Weight
8	Janet	F	15	62.5	112.5
14	Mary	F	15	66.5	112.0
15	Philip	M	16	72.0	150.0
17	Ronald	M	15	67.0	133.0
19	William	M	15	66.5	112.0

If we modify the %GETIT Macro to make all the SQL associated Macro variables %LOCAL, then we don't have to worry about resetting whatever the GLOBAL values are (although in this case it could be argued that the real problem is relying on the Global value of &SQLOBS rather than a more stable MACRO variable):

```
*21b);

%macro getit(dset=SASHELP.CLASS,var=Name);
%Global All&var.s N;
%Local SQLOBS SQLRC SQLXOBS SQLOOPS;
proc sql;
select &var into: All&var.s separated by ","
from &dset;
quit;
%let n=&sqlobs;
%put _user_ ;
%mend getit;

%GETIT
```

There are **3 boys and 5** total Observations from SASHELP.CLASS with AGE gt 14

Obs	Name	Sex	Age	Height	Weight
8	Janet	F	15	62.5	112.5
14	Mary	F	15	66.5	112.0
15	Philip	M	16	72.0	150.0
17	Ronald	M	15	67.0	133.0
19	William	M	15	66.5	112.0

There is another subtle example of a macro variable collision that can cause horrible errors and leave no indication that anything has gone wrong! In this case a secondary Macro is called from within a %DO loop, but there is a Macro collision between the %DO loop iterator and a variable in the Macro inside the %DO loop. For instance, suppose we wanted to run the %GETIT Macro to create several Macro variable lists for several variables in the SASHELP.CLASS data set, we could code something like:

```
*22);
options nomprint nosymbolgen nomlogic;
%Macro trap(vars=Name Age Weight,
                  tot=3);
      %do n=1 %to &tot;
            %let var&n=%scan(&vars,&n,%str( ));
            %getit(var=&&var&n)
      %end;
%mend;

%trap
%put &allnames &allages &allweights;
```

Which gives **no errors**, but:

```
22   %put &allnames &allages &allweights;
WARNING: Apparent symbolic reference ALLAGES
not resolved.
WARNING: Apparent symbolic reference
ALLWEIGHTS not resolved.
Alfred,Alice,Barbara,Carol,Henry,James,Jane,J
anet,Jeffrey,John,Joyce,Judy,Louise,Mary,Phil
ip,Robert,Ronald,Thomas,William &allages
&allweights
```

If it isn't obvious what the problem is here, MLOGIC will sort it out for us:

```
Log trimmed for brevity's sake...

GLOBAL N 19
MLOGIC(GETIT):  Ending execution.
MLOGIC(TRAP):  %DO loop index variable N is now 20; loop will not iterate again.

MLOGIC(TRAP):  ENDING EXECUTION.
```

The Global value for &N from %GETIT is used as the value of &N that is iterating the %DO loop in %TRAP, and since the iterator has exceeded the Stop value of the loop, the loop ends. If the Global &N had been less than the Stop value of the %DO loop, the iteration of the loop would have gone on forever, but at least we might have recognized that problem. In this case making the &N in %TRAP LOCAL to %TRAP won't help, the &N in %GETIT would then be Local to %GETIT, and %TRAP. It's LOCAL value would be the value of &SQLOBS in %GETIT, and the Global &N would not be set (The %GLOBAL statement only works on variable names that have not already been declared with a %LOCAL, even in another higher scope)! To get this code to work, you need to make the &N in %GETIT Local to %GETIT, or change the %DO Loop iterator in %TRAP to another Macro variable name.

In summary be very careful when you pick the name of your %DO loop iterators, and when mixing Global and Local macro variables with the same name! **As a general rule most of these types of collisions can be avoided by minimizing the use of GLOBAL macro variables and by making other macro variables LOCAL whenever possible**.

OOPS – MISSING MACRO STATEMENTS AND OTHER ASSORTED DISASTERS

As we have already seen, debugging Macros is not easy. The error messages produced, even when you are using MLOGIC, MPRINT and SYMBOLGEN, can be hard to decipher. Often errors only come to light through careful study of the LOG and output. The problems with Macro debugging can be made worse when the error arises from the absence of a piece of Macro code.

A simple example occurs when a programmer forgets to end the Macro declaration with a %MEND statement. When programming interactively, this is an especially irritating problem as the system seems to freeze, no errors are produced, but nothing really happens at all! In truth, SAS is waiting patiently for the %MEND, continuing to add any and all code you submit to the Macro. If you resubmit the code things only get worse, now SAS is waiting on two %MEND statements. You can clear these pending Macro definitions by submitting a series of %MEND statements. You will know you have cleared SAS's need for a %MEND when you get the error:

```
ERROR: NO MATCHING %MACRO STATEMENT FOR THIS %MEND STATEMENT.
```

I know what you are thinking, "I would never forget to put in the %MEND statement at the end of the Macro." That may or may not be true, but often, forgetting the %MEND is the least of your problems. Take the 'simple' (pun intended) Macro below:

```
*23);
%macro quoted(Text=A Problem with Kevin's Macro);
%put &text;
%mend;

%QUOTED
```

The pretty purples provided by the Enhanced Editor show us the problem, our %MEND is there, it's just being swallowed up by an unmatched single quote! You can submit the program all day and only get a warning about a Missing %MEND, which of course isn't your real problem. Hopefully you notice there is a problem before you submit too much code! The same is true for missing parentheses for instance,

```
*24);
%Macro wrdcnt(string);
%let cnt=0;
%do %while(%scan(&String,&cnt+1,%str( ) ne %str());
      %let cnt=%eval(&cnt+1);
%end;
&cnt;
%MEND WRDCNT;
```

There should be an extra Close Paren after the super sized %STR( ). When this code is executed, the LOG shows multiple errors, none of which come close to identifying the missing parenthesis as the culprit.

CONCLUSION

We have tried to highlight some of the big GOTCHAS that can sneak up on even the most experienced Macro programmer. The differences between how the DATA step works and how the Macro language works, and several sneaky Macro problems that we have discussed in these pages should act as a guide and provide things to look for when your Macros start to go wrong. The SAS system options SYMBOLGEN MLOGIC MPRINT and MFILE and the Macro Statement % PUT are your friends in the debugging process. The examples that accompany this Hands on Workshop will provide step by step walk throughs of some bugged out Macro Programs, they can serve as another reference of things to look for and how to fix them. However, most of the time the real BUG in your program is a little typo, a missing &, quote mark or parenthesis, and only some trial and error and experience reading the SAS LOG will help you find them. Like everything else in SAS, with time and practice you will be able to figure out which little hang up or typo has caught you, and how to fix it. Happy hunting!

REFERENCES

Carpenter, Arthur L. 2004, Carpenter's Complete Guide to the SAS® Macro Language, 2nd Edition, SAS Institute Inc. Cary, NC, USA.

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

Whitlock, Ian. 2003, A Serious Look at Macro Quoting. Proceedings of the 28th SAS Users Group International Meeting. SAS Institute Inc. Cary, NC USA

Other Macro papers available at http://www.lexjansen.com/sugi/
Other Macro Help available from http://listserv.uga.edu/archives/sas-l.html

CONTACT INFORMATION

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

Kevin P. Delaney
Epidemiologist
Centers For Disease Control and Prevention
1600 Clifton Rd NE
MS E-46
Atlanta, GA 30333
KDelaney@cdc.gov

Arthur L. Carpenter
California Occidental Consultants
PO Box 586199
Oceanside, CA 92058-6199
(760) 945-0613
art@caloxy.com
www.caloxy.com