PhUSE 2009

Paper TU02

MACRO PROCESSOR – The Masked Warrior

Salaja Akkadan, Quanticate, Manchester, United Kingdom

ABSTRACT

In the current pharmaceutical world, everyone is aiming to achieve a targeted speed goal and reduce the overall time and cost involved in the drug development process. The entire industry is continuously monitoring and improving their processes and procedures to eventually speed up the overall process of clinical development. SAS® programmers are making their own contribution by creating efficient, re-usable and robust SAS programs. When one refers to efficient, re-usable and robust programs, one of the SAS facilities that immediately pop up in your mind is the SAS MACRO facility.

MACROS play a vital role in creating standardized programs that can work across various clinical studies. It presents a learning experience everyday, which makes us realise how vast and powerful the SAS MACRO facility is. One very important but simple aspect makes the process of writing and debugging macros a lot easier - understanding the function of the MACRO PROCESSOR. This is the primary focus of the paper. The paper takes you through the behaviour of the Macro processor in various program scenarios. We will look at the functioning of the macro process step by step, beginning from the definition of a macro, to compilation and eventually execution, including the processing of macro statements and variables.

This paper will prove beneficial to any SAS programmer dealing with MACROS and will provide useful insights to the technical and software interaction features of macros. The programmer will be able to "think" and interpret like the MACRO PROCESSOR which, I believe, will surely increase the efficiency and re-usability, in addition to faster debugging of macros.

INTRODUCTION

The Macro processor is that part of the SAS macro facility which does the work of translating macro language to SAS statements and commands. In simple terms, when macros are used within a SAS program, the control is passed over to the macro processor, which in turn, converts or replaces the macro statements to SAS statements (text substitution). Once done, the control is passed back to the word scanner. This ensures that the macro substitution is completed before the rest of the program text is compiled and executed. The macro processor's job is not limited to just text substitution. In addition to this, the macro processor does the following:

- Checks all macro language statements for syntax errors
- Writes error messages to the log
- Creates a dummy, non-executable macro if any syntax errors are found
- Else, stores the checked macro language statements and constants in a SAS catalog entry.

WHAT ARE THE PHASES

Each macro has to go through the three stages, starting from definition, progressing to compilation and eventually ending with execution. The macro processor has a different role to play during each of these stages. The rest of the paper will explain in further detail, the behaviour of the macro processor in these phases. A brief outline of these three stages is as follows:

In the macro definition phase, a new macro is created by wrapping the macro content within a %macro and a %mend statement.
In the macro compilation phase, the newly defined macro is checked for syntax errors and stored in the macro catalog ready for use.
In the macro execution phase, the compiled macro is retrieved from the catalog and executed as part of the SAS program.

Now let's dig a little more deep into what the macro processor does to macros during definition, compilation and execution.

MACRO DEFINITION

To enable us to use Macros in our SAS programs, they must first be defined i.e. we must tell SAS that the following set of SAS statements or text statements are part of a macro. A macro is defined as follows:

```
%macro macro_name;
<macro body>
%mend <macro_name>;
```

The *macro_name* is a SAS name that identifies the macro. The *macro_text* can be any combination of SAS or MACRO statements, calls to other MACROS, text expressions or constants.

When a SAS program is submitted, it is stored in an area of memory called the input stack. A SAS component called the word scanner reads the contents of the input stack and splits them into tokens. This process is called tokenization. When the word scanner sees a % followed a character it causes to trigger the macro processor which then takes charge. The macro processor detects this token as the beginning of a macro definition. It continues to ask the word scanner for more tokens until it receives a %mend token. The %mend tells the macro processor that this is the end of the macro definition.

Once the definition is done, the macro COMPILES.

MACRO COMPILATION

When a macro definition is submitted, the macro processor starts compiling the definition. In this phase, the macro processor checks the entire macro for any syntax errors. If no syntax errors are found, the macro processor makes an entry in the SAS session catalog and stores the compiled macro statements with the name as in the macro definition. On the other hand, if syntax errors are found, the macro processor continues to check the rest of the macro and writes out the error messages and warnings to the SAS log. This macro is not stored in the session catalog. Instead, the macro processor creates a non-executable version of this macro which is called as the 'dummy macro'.

MACRO EXECUTION: MACRO CALL

Once a macro has been compiled, it is available to use in our SAS programs. To be able to use a macro, we have to call it in our program. A macro would be called in the following manner: %macro_name. As soon as the word scanner comes across this token, it triggers the macro processor. The macro processor now searches the session catalog for an entry associated with this macro name and retrieves the stored statements within this macro. The macro language statements are then executed. Any use of a macro variables, functions or statements is executed by the macro processor. When it comes across a non-macro language token, say, a data step for example, it suspends its activity and puts the data step back into the input stack for further word scanning. The SAS compiler now takes charge and proceeds with the compilation and execution of the data step. Once the SAS code has executed, the macro processor resumes its function. As soon as it executes the %mend statement, the macro execution ceases.

MACRO VARIABLES: LOCAL AND GLOBAL

Macro variables in SAS allow you to use text values and variables dynamically during the execution of a program. The value of the macro variable remains constant unless it is explicitly changed. Macro variables can be AUTOMATIC (already defined in the SAS system) or they can be USER-DEFINED (created and specified by the user). Each macro variable has a life span or scope.

When a macro variable is available to be used by any macro or any part of the SAS session while the session is running, it is called as a global macro variable. These are generally the SAS system generated macro variables or any user defined macro variable explicitly defined as global.

When a macro variable is created within a macro and is not specified as global, it is created as a local macro variable. The local macro variable can only be used by the macro in which it has been defined and is available only until this macro stops executing.

A Global macro variable is stored in a logical entity called the Global Symbol Table, whereas a Local macro variable is stored in a logical entity called the local symbol table. In an active SAS session, there is just one Global symbol table. The number of local symbol tables depends on the number of macros executed in the current active SAS session.

The most common ways of creating a macro variable is by using either of the following:

- %LET statement,
- %GLOBAL, %LOCAL statement
- CALL SYMPUT routine
- Using a :INTO clause in SQL
- Using macro parameters

WHAT TRIGGERS IT

When does the Macro processor know that it needs to start functioning?
There are two delimiters & and %, when followed by a non-blank character trigger the macro processor activity due to which these are known as macro triggers.

When the word scanner encounters a macro definition (a % sign followed by the word 'macro') or a macro call (a % sign followed by a SAS name), it passes control over to the macro processor. In the former case, the macro processor detects %macro to be the start of a macro definition and proceeds to compile the macro making it ready for use in our SAS programs. In the latter case, the macro processor looks for the macro name in the session catalog and if present, proceeds to execute the macro. Other statements that start with a % sign also trigger the macro processor for eg. %LET, %IF-%THEN-%ELSE, %DO etc.

When the word scanner encounters a macro variable reference like, *&varname*, it passes control to the macro processor. Here it looks up this macro variable name in the global or the local symbol table as appropriate and substitutes &test with the equivalent text. This refers to macro variable resolution.

FUNCTIONING OF THE MACRO PROCESSOR

Having familiarised ourselves, with the various phases of a macro, macro triggers and the creation of macro variables, let us now look at how the macro processor functions, with the help of some simple real-time examples.

EXECUTION OF A SIMPLE MACRO

Let us define and call a simple macro first.

```
%macro sortit;
    proc sort data=&syslast;
    by study subject gender;
    run;
%mend sortit;

%sortit;
```

When the above piece of code is submitted, the following is what happens.

1. The whole of the code is dumped into the input stack and the word scanner starts tokenizing the program. As soon as the word scanner encounters the % sign followed by a character, it passes the token to the macro processor.
2. The macro processor recognises the token to be the beginning of a macro definition and keeps accepting tokens from the word scanner and compiling till it encounters a %mend statement.

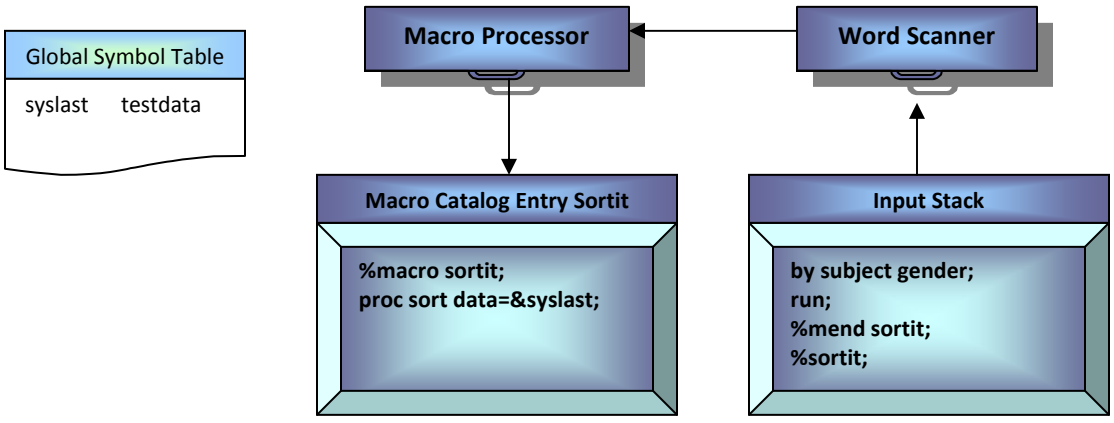


Figure 1: Macro Compilation

3. During compilation, the macro processor checks the macro for any syntax errors. If the macro compiles without any errors, it stores all the compiled statements in the SAS session catalog under the macro name sortit. By default the work catalog is opened, work.sasmacr and with an entry called sortit.macro. If errors are found during compilation the macro processor writes out error messages to the log as appropriate and does not store the compiled macro. Instead, it creates a non-executable (dummy) version of the macro which cannot be used. The macro without errors is now compiled and ready to use. Part of the compilation phase the status of the various components can be seen in Figure 1 above.
4. The word scanner continues tokenization and then comes across the % sign followed by a non-blank character. At this stage, the control is passed back to the macro processor.
5. The macro processor recognizes the *%sortit* as a macro call and begins executing the macro. It searches the session catalog to search for an entry matching the macro name. It also initialises a local symbol table for the macro to store any new macro variables.
6. When the macro processor detects the next item as text, it puts the rest of the code on the input stack and waits for the word scanner to continue tokenization.
7. The word scanner detects the beginning of a PROC step and signals the SORT procedure and passes the tokens to the SAS compiler. When the word scanner encounters the '&' sign followed by a character, it transfers the control back to the macro processor. The compiler suspends compilation of the PROC step.
8. When the macro processor encounters the macro variable *&syslast*, it first searches the SORTIT local symbol table for a match. When it does not find an entry for SYSLAST in the local symbol table, it searches the global symbol table. On finding the entry, it replaces &syslast with the value *testdata* and places it back on the input stack.
9. The word scanner continues with tokenization. When the word scanner encounters the RUN; statement it triggers the SAS compiler to execute the compiled PROC sort.
10. %mend triggers the macro processor, which when executed, removes the SORTIT local symbol table and the macro execution completes.

SIMPLE MACRO WITH PARAMETERS

Now let us try amending our noble macro to include a few macro parameters and see if the macro processor behaves any differently.

```
%macro sortit (var1);
    proc sort data=&syslast;
    by study subject &var1;
    run;
%mend sortit;
%sortit (gender);
```

The compilation phase of this macro remains the same. During the execution phase, i.e. when the macro processor receives the %sortit token from the word scanner, it opens the session catalog for the macro and initialises the local symbol table with an entry for the macro variable VAR1 with the value of GENDER. The rest of the execution remains exactly the same. Part of the execution phase is depicted in Figure 2 below.

Note: Macro substitution happens before any SAS steps are executed.

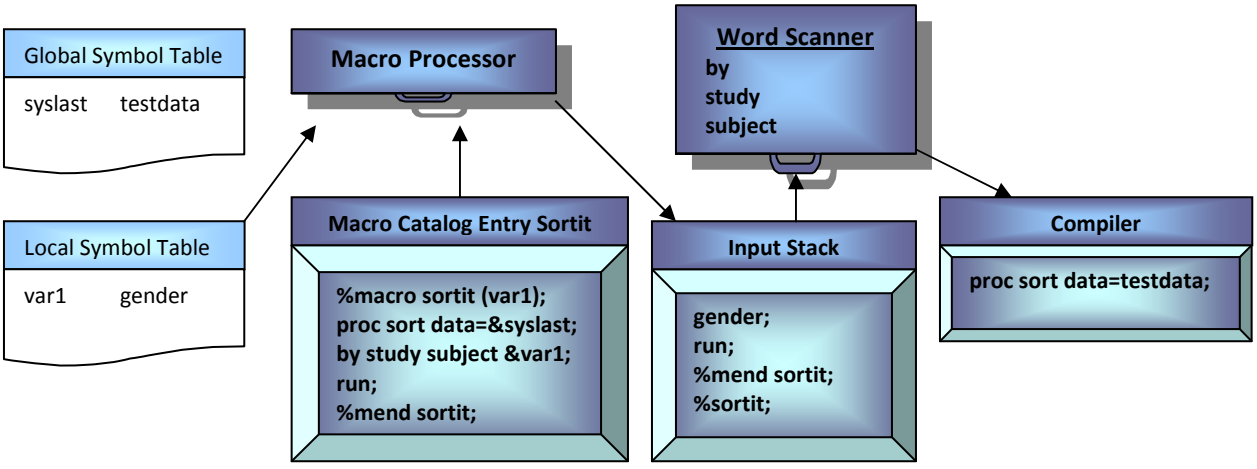


Figure 2: Macro Execution

THE %LET STATEMENT

Now instead of using a macro parameter, let us use the %LET statement to create a macro variable and then use this variable in our macro. Please take note that %let statement is placed outside of the macro. The scene so far is that the below code has been submitted. So this piece of code has already been dumped into the input stack.

```
%let var1=gender;

%macro sortit;
    proc sort data=&syslast;
    by study subject &var1;
    run;
%mend sortit;

%sortit;
```

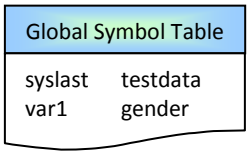


Figure 3: Global Symbol table

When the word scanner encounters the % sign the macro processor is triggered. When it sees the %let sign, it notices that this statement has been defined outside of any macro. Due to this, it creates an entry of this macro variable VAR1 with the value of GENDER in the GLOBAL symbol table. A virtual picture of the global symbol table is shown in figure 3 above. When the macro processor looks at resolving the macro variable value, it first checks if the macro variable exists in the local symbol table of the macro sortit. It then checks the global symbol table and substitutes var1 with 'gender'. The rest of the compilation and execution of the macro remains the same.

In the above example, if the %let statement would have been within the macro, the macro processor would have made an entry for the macro variable var1 in the local symbol table.

GLOBAL AND LOCAL SYMBOL TABLES

While we are at it, let us explore a few more examples which will provide useful insights on the GLOBAL and LOCAL symbol tables.

```
%let datname=outside;
%macro first;
    %let datname=inside;
%mend;
%first;

proc print data=&datname;
run;
```

When the above piece of code is run, the resolved proc print appears as follows:

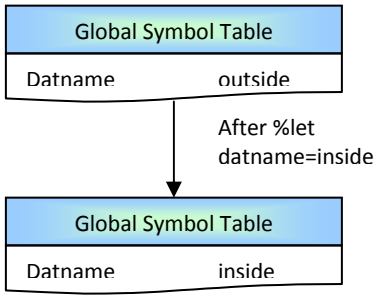
```
proc print data=inside;
run;
```

One would imagine that since the value of 'inside' is defined within the macro, the variable scope is limited to be local. But from the above, it appears that the global value was replaced with the value defined inside the macro. Let us check the status of the global and local symbol tables when we submit the following piece of code.

```
%let datname=outside;

%macro first;
    %let datname=inside;
    %let varname=age;

    proc print data=&datname;
    var &varname;
    run;
%mend;
%first;
```



The SAS code resolves to the following when %first is executed:

```
proc print data=inside;
var age;
run;
```

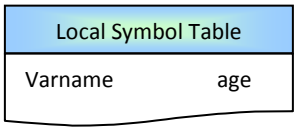


Figure 4: Symbol tables for macro first

Here, the datname resolves as in the previous example, but the varname is picked up from the local symbol table of the macro first. Thus, the value of varname will not be accessible outside of the macro first. This behaviour of the macro processor is clearly defined in the following flowchart. It explains clearly, how the macro processor goes about creating a new variable in either the local or the global symbol table.

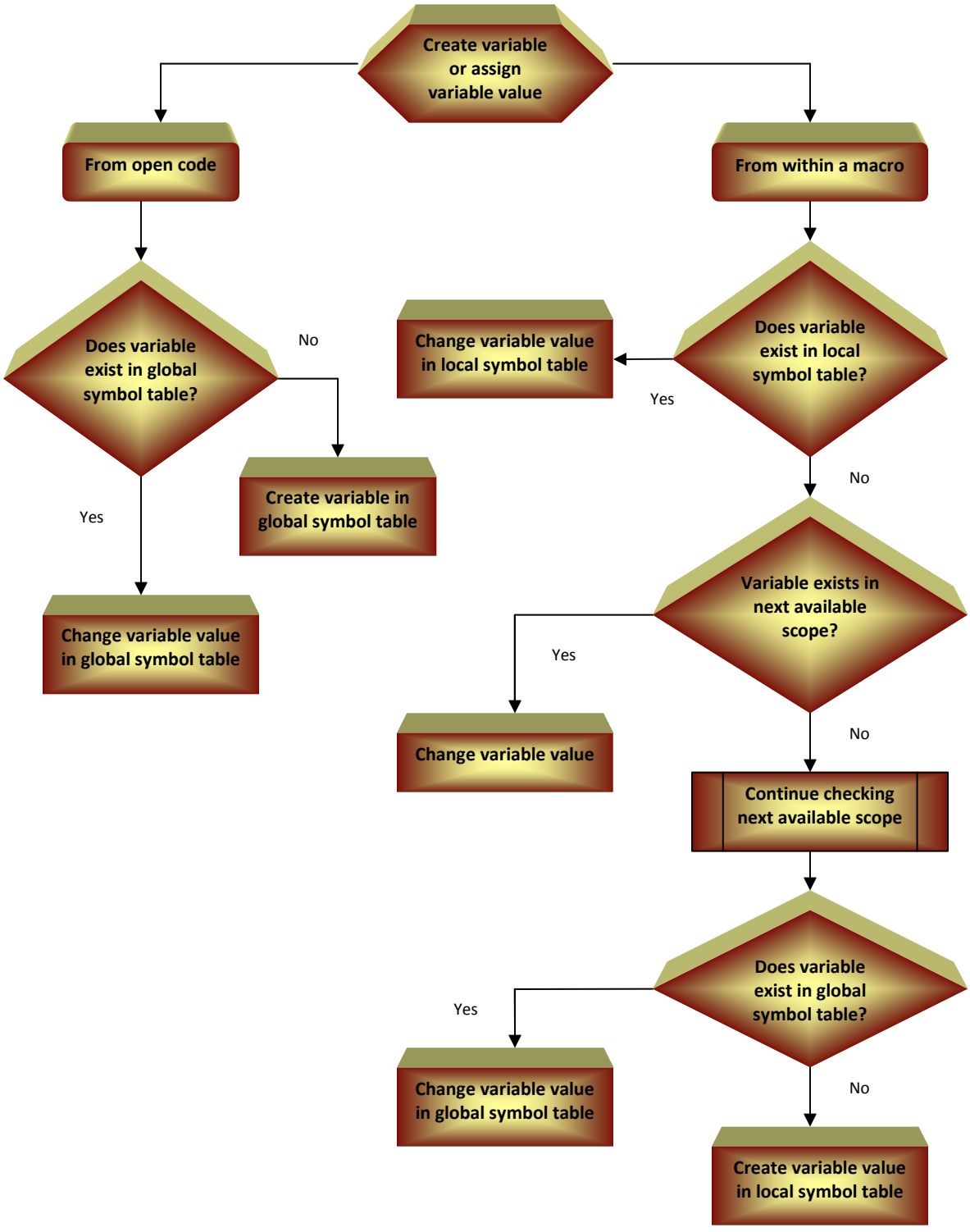


Figure 5: Flowchart showing the search order when assigning or creating macro variables

MACRO NESTING

We will now have a look at nested macros, i.e one macro being defined or called within another macro.

```
%macro inner;
    %local var1;
    %let var1=&var2;
%mend inner;

%macro outer;
    %local var2;
    %let var2=one;
    %inner;
%mend outer;
```

When we execute the following piece of code, let us see what happens to the local and global symbol tables.

```
%let var1=ten;
%outer;
%put &var1;
```

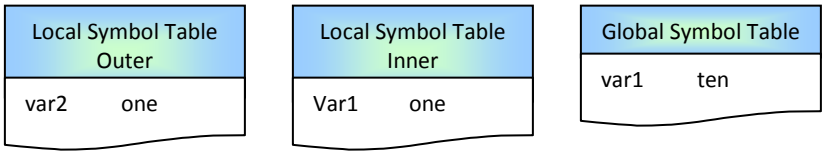


Figure 6: Local and Global symbol tables
during the execution of the nested macro.

1. The macro processor receives the %let var1=ten; statement and since this is placed outside of any macro, it searches for this variable in the global symbol table. Since there is no entry already, it will make an entry for var1. The local and the global symbol tables are shown in figure 6 above.
2. The macro processor then receives the %outer. It looks for this macro in the session catalog and receives the statements. On receiving the statement %local var2; it creates a local symbol table and makes a blank entry for var2 with value 'one' in the local symbol table for the macro outer.
3. It then executes the %inner macro by looking it up in the session catalog. On receiving the %let var1=&var2; the macro processor creates an entry for the variable var1 in the local symbol table for the inner macro.
4. The macro processor checks the most recently created local symbol table for the variable var2. Since there is no such variable in the inner local symbol table, the macro processor then checks the next available local symbol table one level up.
5. It finds the variable var2 in the local symbol table of the outer macro and assigns the value of one to var1 as per the assignment statement.
6. When the %mend inner is encountered, the local symbol table for the inner macro is removed by the macro processor.
7. Consequently the %mend outer causes the local symbol table for the outer macro also to be removed.
8. Next, the %put outputs a value of 'ten' to the log on execution. This is due to the fact the global symbol table will still have the value from the %let statement in step 1. This symbol table remains till the end of the SAS session. The value will remain as 'ten' until an explicit change of value is executed by another macro statement.

CONDITIONAL MACRO PROCESSING

Macros can be used to perform conditional execution of a program. We can use macro statements to execute a program only if a certain condition or set of conditions are satisfied. Conditional execution is possible by using the %IF-%THEN and %ELSE statements.

The general form of these statements is as given below:

```
%IF expression %THEN text;
<%ELSE text;>
```

Where expression is any macro expression that resolves to an integer and text is any combination of constant text, text expression, a macro variable reference, a macro call or a macro program statement.

If the expression resolves to zero, then the statement following %then is no executed and the %else is executed instead. Please note that %ELSE is optional. The %then is executed for any non-zero integer value resulting out of the expression. If multiple statements need to be executed when a condition is satisfied, we can add a %DO - %END clause to the %THEN or %ELSE statements;

The below example explains this better.

```
%macro demo(opt=);
data first;
    set first;
    %if &opt=1 %then %do;
          where var=1;
          put 'This is the if section';
    %end;
    %else %do;
          where var ne 1;
          put 'This is the else section';
    %end;
run;
%mend demo;

%demo (opt=1);
%demo (opt=3);
```

After compilation, when the first macro call is executed, the %if returns a non-zero integer and thus the statements following the %THEN %DO are executed. Thus, the dataset is filtered on the value of var=1 and the put statement writes the following to the log: This is the if section

Whereas, in the second call the %IF returns a false value and thus the %ELSE %DO is executed resulting in the dataset being filtered on var ne 1 and 'This is the else section' being written to the log.

The screenshot of the log in both the macro calls is shown below. The macro has been run with the options statement:
OPTIONS MPRINT MLOGIC SYMBOLGEN;

```
83   %demo (opt=1);
MLOGIC(DEMO):  Beginning execution.
MLOGIC(DEMO):  Parameter OPT has value 1
MPRINT(DEMO):   data first;
MPRINT(DEMO):   set first;
SYMBOLGEN:  Macro variable OPT resolves to 1
MLOGIC(DEMO):  %IF condition &opt=1 is TRUE
MPRINT(DEMO):   where var=1;
MPRINT(DEMO):   put 'This is the if section';
MPRINT(DEMO):   run;

This is the if section
NOTE: There were 1 observations read from the data set WORK.FIRST.
      WHERE var=1;
NOTE: The data set WORK.FIRST has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds


MLOGIC(DEMO):  Ending execution.
```

```
94   %demo (opt=3);
MLOGIC(DEMO):  Beginning execution.
MLOGIC(DEMO):  Parameter OPT has value 3
MPRINT(DEMO):   data first;
MPRINT(DEMO):   set first;
SYMBOLGEN:  Macro variable OPT resolves to 3
MLOGIC(DEMO):  %IF condition &opt=1 is FALSE
MPRINT(DEMO):   where var ne 1;
MPRINT(DEMO):   put 'This is the else section';
MPRINT(DEMO):   run;

This is the else section
This is the else section
This is the else section
This is the else section
NOTE: There were 4 observations read from the data set WORK.FIRST.
      WHERE var not = 1;
NOTE: The data set WORK.FIRST has 4 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds


MLOGIC(DEMO):  Ending execution.
```

Note: The OPTIONS MPRINT MLOGIC SYMBOLGEN; are macro debugging options which enable you to see which part of the code is being executed and display the text that is being sent to the compiler by the macro processor. The symbolgen option tells you what a particular macro variable has resolved to. As you can see, the use of these options while developing code, makes macro debugging a lot easier.
Also, this would be a good stage to remember that the macro substitution takes place before the SAS statements are executed. In the above case, the decision about which section of the %IF-%THEN-%ELSE would be executed is made by the macro processor before the execution of the data step begins.

ITERATIVE MACRO PROCESSING

Many times, it is necessary to execute program code repetitively. This can be accomplished by using the iterative %DO loop. The general form of the iterative %DO statement is as below. Please note that event %DO should have a corresponding %END statement.

```
%DO index-variable=start %TO stop <%BY increment>;
    Text
%END;
```

Let us take a look at this with the help of an example.

```
%macro dotest;
%do i=1 %to 10;
    data dat&i;
    set test;
    where var=&i;
    run;
%end;
%mend dotest;

%dotest;
```

1. When the macro %dotest is executed, the index variable i is created as a macro variable in the local symbol table initially with a value of 1.
2. The macro processor then replaces the &i with the value of 1 and the data step executes with the where clause of where var=1.
3. Once the data step is executed, the index variable i is incremented by 1 and the data step execution is repeated. This continues till the value of the index variable increments by 1 to 10.
4. After the data step is executed for the 10^{th} time, the value of i is incremented to 11. At this stage the iteration stops as the condition is no longer true and the control moves to the next statement in the macro, which is the %mend statement.
5. After this, the macro processor removes the local symbol table and the macro stops execution.

The screenshot of the log after running the above %DO loop is shown below.

This figure shows the text substitution done by the macro processor and the interpretation done by the compiler in the first two iterations.

```
MLOGIC(DOTEST):  Beginning execution.
MLOGIC(DOTEST):  %DO loop beginning; index variable I; start value is 1; stop value is 10; by value
                 is 1.
SYMBOLGEN:  Macro variable I resolves to 1
MPRINT(DOTEST):   data dat1;
MPRINT(DOTEST):   set test;
SYMBOLGEN:  Macro variable I resolves to 1
MPRINT(DOTEST):   where var=1;
MPRINT(DOTEST):   run;

NOTE: There were 1 observations read from the data set WORK.TEST.
      WHERE var=1;
NOTE: The data set WORK.DAT1 has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds


MLOGIC(DOTEST):  %DO loop index variable I is now 2; loop will  iterate again.
SYMBOLGEN:  Macro variable I resolves to 2
MPRINT(DOTEST):   data dat2;
MPRINT(DOTEST):   set test;
SYMBOLGEN:  Macro variable I resolves to 2
MPRINT(DOTEST):   where var=2;
MPRINT(DOTEST):   run;

NOTE: There were 1 observations read from the data set WORK.TEST.
      WHERE var=2;
NOTE: The data set WORK.DAT2 has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The figure below shows the text substitution done by the macro processor and the interpretation done by the compiler in the last two iterations. The last two lines indicate why the DO loop has come to an end.

```
MLOGIC(DOTEST):  %DO loop index variable I is now 9; loop will  iterate again.
SYMBOLGEN:  Macro variable I resolves to 9
MPRINT(DOTEST):   data dat9;
MPRINT(DOTEST):   set test;
SYMBOLGEN:  Macro variable I resolves to 9
MPRINT(DOTEST):   where var=9;
MPRINT(DOTEST):   run;

NOTE: There were 1 observations read from the data set WORK.TEST.
      WHERE var=9;
NOTE: The data set WORK.DAT9 has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds


MLOGIC(DOTEST):  %DO loop index variable I is now 10; loop will  iterate again.
SYMBOLGEN:  Macro variable I resolves to 10
MPRINT(DOTEST):   data dat10;
MPRINT(DOTEST):   set test;
SYMBOLGEN:  Macro variable I resolves to 10
MPRINT(DOTEST):   where var=10;
MPRINT(DOTEST):   run;

NOTE: There were 1 observations read from the data set WORK.TEST.
      WHERE var=10;
NOTE: The data set WORK.DAT10 has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds


MLOGIC(DOTEST):  %DO loop index variable I is now 11; loop will not iterate again.
MLOGIC(DOTEST):  Ending execution.
```

CONCLUSION

The MACRO facility in SAS is a very powerful mechanism for efficient, portable and robust programming. With the ongoing standardization of the overall clinical trial process, the potential use of macros is very high and extensive. Once we understand how the Macro Processor behaves and how it interacts with the rest of the SAS components, it becomes far easier to write a macro. Macro debugging is far easier with the SAS options available for the same and it shows us to an extent, how the interaction works between the macro processor and the SAS compiler. Once a macro is written, there is a lot going on with the interpretation of the program. All we can see is the log and the output, whereas the internal SAS components have a lot of 'hard work' to do in the background. The Macro Processor is one of the key components for a macro to work effectively and if we are able to 'think' the way it does, macro writing will be a piece of cake.

REFERENCES

SAS Macro Language Reference: http://support.sas.com/onlinedoc/913/docMainpage.jsp
SAS Certification Prep Guide for Advanced Programming in SAS 9

CONTACT INFORMATION

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

Name	:	Salaja S. Akkadan
Company	:	Quanticate
Address	:	Black Box Business Centre Beech Lane Wilmslow, United Kingdom SK9 5ER
Work Phone	:	+44 (0)1625 544 822
Email	:	salaja.akkadan@quanticate.com
Web	:	www.quanticate.com