Paper 228-2012

Is Your Failed Macro Due To Misjudged "Timing"?

Arthur Li, City of Hope Comprehensive Cancer Center, Duarte, CA

ABSTRACT

The SAS® macro facility, which includes macro variables and macro programs, is the most useful tool to develop your own applications. Beginning SAS programmers often don't realize that the most important function in learning a macro program is understanding the macro language processing rather than just learning the syntax. The lack of understanding includes how SAS statements are transferred from the input stack to the macro processor and the DATA step compiler, what role the macro processor plays during this process, and when best to utilize the interface to interact with the macro facility during the DATA step execution. In this talk, these issues are addressed via creating simple macro applications step-by-step.

INTRODUCTION

The SAS macro facility includes macro variables and macro programs. One can use SAS macro variables and macro programming statements in a SAS macro program to generate SAS codes. In order to utilize the SAS macro facility, one needs to learn the SAS macro language. Although the convention of the macro language is similar to the SAS language, macro language is indeed a separate language that is used in the SAS macro program. The most essential aspect of learning the SAS macro language is understanding the mechanism of macro processing and utilizing the interface to interact with the macro facility during the DATA step execution.

AN APPLICATION

Ht.sas7dat contains three variables: RACE, SEX, and HEIGHT. Three unique values for the RACE variable are 'A' for Asian, 'B' for Black, and 'W' for White students. The SEX variable is coded as either 'M' for male or 'F' for female students. HEIGHT is a numerical variable that contains the height in inches for each student.

You can use the two-sample t-test to study whether student's height is varied by their gender. In order to perform the two-sample t-test, the students' height for males and females must follow normal distribution. If the normality assumption is violated, the Wilcoxon rank sum test can be used. You can also use the chi-square test to evaluate the association between students' gender and their heights. But students' heights need to be recoded as a categorical or an indicator variable. For example, you can create an indicator variable, say HEIGHT_CAT, based on the HEIGHT variable; HEIGHT_CAT is set to 1 if HEIGHT is greater than a threshold value or the mean height of all the students; otherwise HEIGHT_CAT is set to 0. Once the indicator variable is created, the distribution of SEX and HEIGHT_CAT can be categorized into a two-by-two table like the one below:

	HEIGHT_CAT = 1	**HEIGHT_CAT = 0**
SEX = 'M'	A	B
SEX= 'F'	C	D

You can use the Pearson chi-square test to evaluate the association between SEX and HEIGHT_CAT. To perform a valid chi-square test, all expected counts in each cell of the two-by-two table must be greater than or equal to 5.

Suppose that you are asked to generate a table that contains the chi-square statistics between the SEX and HEIGHT_CAT variables for all students and the students within each ethnic group. The HEIGHT_CAT is set to 1 if the student's height is greater than the mean height of their group; otherwise HEIGHT_CAT is set to 0. The table should look like the one below that consists of four columns. The TEST column indicates whether the chi-square test is valid or not. If one of the expected counts is less than 5, the TEST column should have the value of "not valid;" otherwise, it should be "valid." CHISQ column contains the chi-square statistics and P column contains the corresponding P values.

Group	**Test**	**Chisq**	**P**
All	valid	8.97231	0.00274
A	valid	0.07529	0.78378
B	not valid	6.74074	0.00942
W	valid	9.39323	0.00218

CREATING AND REFERENCING MACRO VARIABLES

Macro variables are either automatic, which is provided by SAS, or user-defined, which is created by SAS users. One way to create a macro variable is to use the %LET statement, which has the following form:

%LET MACRO-VARIABLE = TEXT;

TEXT is stored as character strings that can be ranged from 0 to 65,534 characters. Mathematical expressions that are stored as TEXT are not evaluated. Furthermore, the case of TEXT is also preserved. If TEXT contains quotation marks that include literals, then the quotation marks will be part of TEXT. Before assignment is made, any of the leading and trailing blanks will be removed from TEXT. If MACRO-VARIABLE and/or TEXT contain references to another macro variable, the reference will be evaluated first before the assignment. Also, if the MACRO-VARIABLE has already been previously-defined in the program, the new value will replace the most current value of the MACRO-VARIABLE. Here are some examples of defining macro variables by using the %LET statement::

%LET Statement	Variable Name	Value	Length
`%let var1 = 4 + 3;`	var1	4 + 3	5
`%let var2 = hello;`	var2	hello	5
`%let var3 =    leading blank;`	var3	leading blank	13
`%let var4 = " quotations ";`	var4	“ quotations “	14
`%let var5 =;`	var5		0
`%let var6 = var7;`	var6	var7	4
`%let &var6 = &var1 + 2;`	Var7	4 + 3 + 2	9

Once a macro variable is defined, the value of the macro variable is stored in the global symbol table. In order to substitute a macro variable in the SAS program, you must reference the macro variable by preceding the macro variable name with an ampersand (&). This reference causes the macro processor to search the macro variable in the global symbol table. Once the macro variable is found, the value that corresponds to the macro variable will be substituted into the SAS program. If the reference of a macro variable is within quotations, then double quotation marks must be used. Program 1 creates a macro variable, VALUE, which has a value of 63. The macro variable VALUE is then referenced in the DATA step.

Program 1:
```
%let value = 63;
data new_ht;
    set ht;
    height_cat = height > &value;
run;
```

UNDERSTANDING SAS AND MACRO PROCESSING

SAS PROCESSING

In order to understand how macro variables are processed and stored, one needs to understand how SAS processing works. Once a sequence of SAS codes is submitted, it is processed in two-phase sequences: the compilation and execution phases. The compilation phase is performed by the compiler. Before the SAS code is transferred to the compiler, the codes are placed in a memory area, which is called the input stack. Next, the word scanner takes the SAS code from the input stack and breaks that code into words or symbols, which are called tokens, and directs the tokens to the correct destination, such as the compiler or the macro processor. When the compiler receives a semicolon following the RUN statement, it stops accepting tokens from the word scanner. The compiler then compiles the received tokens, checking for syntax errors. If there are no syntax errors, the execution phase begins.   The types of tokens that the compiler recognizes are illustrated in following table:

Types of Token	Contains…	Examples
Literal	Characters enclosed in quotation marks	“John”, ‘John’
Number	Numerals including decimals, E-notation, date, time, datetime constants, and hexadecimal constants	555, ‘01mar2010’d, 30e4, 2.5
Name	Characters that begin with a letter or underscore and that continue with underscores, letters, or numbers. A period can sometimes be part of a name	_n_, means, dollar9.2 Descending
Special character	Characters other than a letter, number, or underscore that have a special meaning to the SAS system	*, /, +, %, &, ., ;

MACRO PROCESSING

The macro processor is responsible for processing all the macro languages, such as the %LET statement and the macro variable references. So that the macro processor can process the macro language, the word scanner has to be able to recognize the macro language and direct the macro language to the macro processor. The tokens that prompt the word scanner that the subsequent codes are macro languages are called macro triggers, such as %LET followed by a name token and "&" followed by a name token. Once the macro triggers are detected by the word scanner, the word scanner passes the tokens to the macro processor. Then the macro processor performs its actions. For macro variables, the macro processor will either create a new macro variable or modify an existing macro variable in a symbol table. The macro processor also retrieves the value from the existing macro variable and returns it to the input stack where it originally contains the macro reference.

We can use Program 1 as an example to illustrate how macro processing works. First, Program 1 is pushed into the input stack (Figure 1a). The word scanner recognizes that %LET followed by VALUE (a name token) is a macro trigger; it directs the %LET statement to the macro processor (Figure 1b). The macro processor will keep requesting tokens until it reaches the semicolon. The macro processor creates the macro variable VALUE and assigns the value 63 in the symbol table (Figure 1c). After the macro processor receives the semicolon, the word scanner begins to transfer the subsequent statements to the compiler until it reaches the next macro trigger, which is ampersand (&) followed by a name token, VALUE. Then the word scanner directs &VALUE to the macro processor (Figure 1d). The macro processor retrieves the value that corresponds to VALUE in the symbol table, which is 63, and returns it to the input stack (Figure 1e). The word scanner continues scanning. Since there are no more macro triggers, the remaining tokens are passed-on to the compiler. When the compiler receives the semicolon following the RUN statement, it stops accepting tokens from the word scanner. The compilation phase begins.

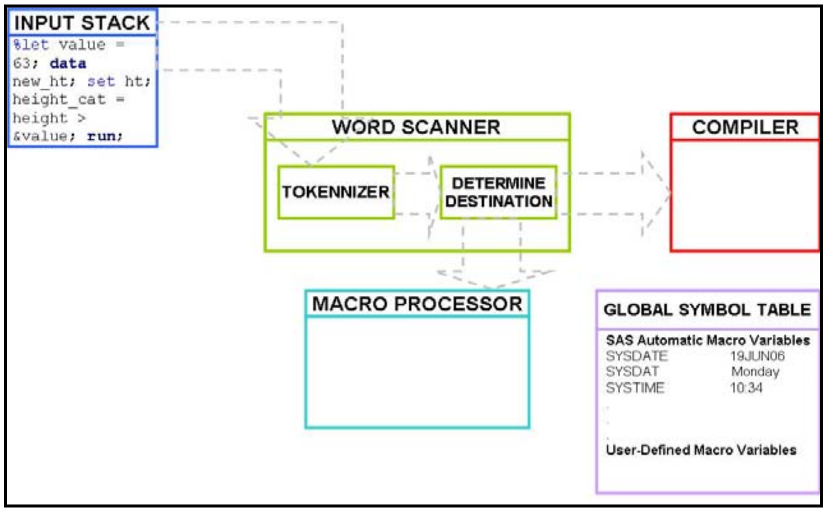


Figure 1a. Program 1 is pushed into the input stack.

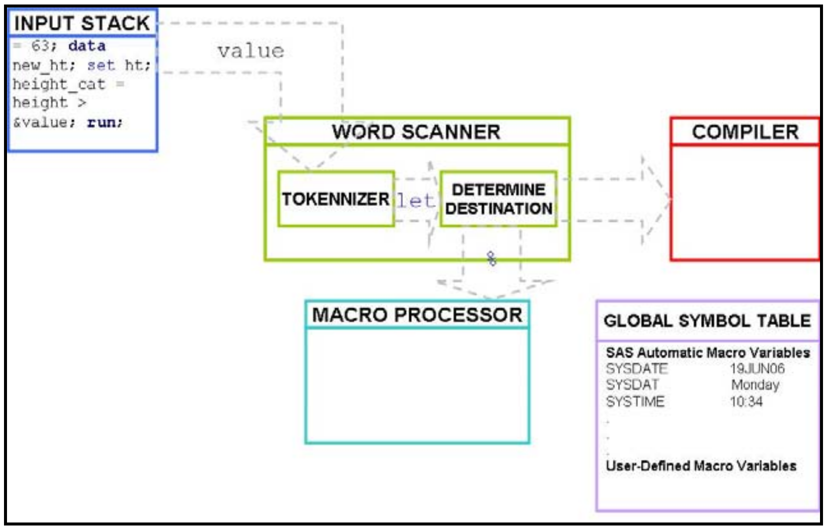


Figure 1b. The word scanner recognizes that %LET followed by "value" (a name token) is a macro trigger, directing %LET to the macro processor. The macro processor requests additional tokens until it receives a semicolon.

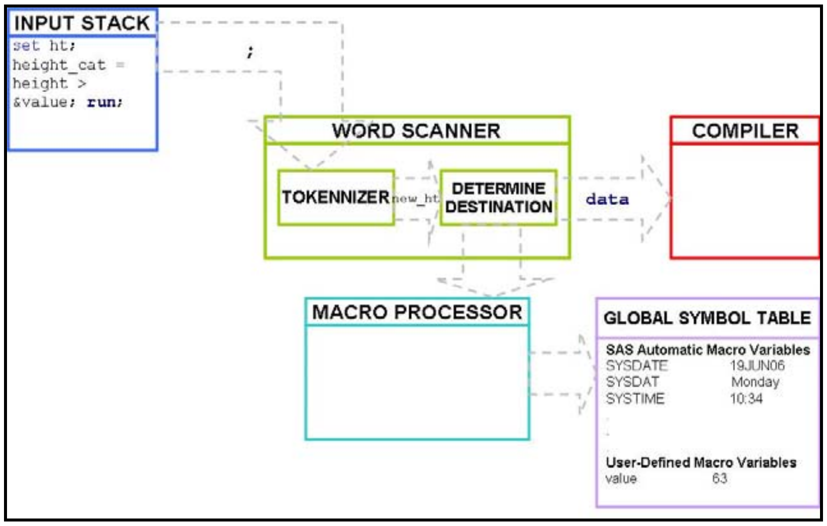


Figure 1c. The macro processor creates the macro variable VALUE and assigns the value 63 in the global symbol table.

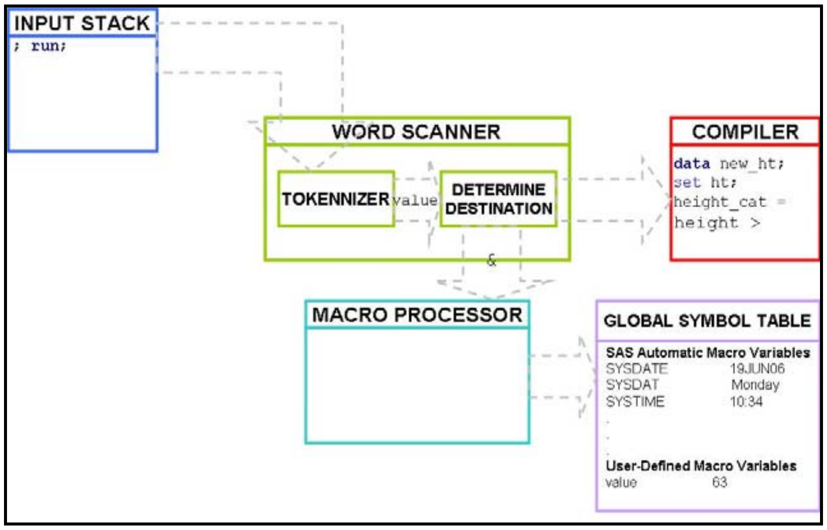


Figure 1d. After the macro processor receives the semicolon, the word scanner begins to transfer the subsequent statements to the compiler until it reaches the next macro trigger, which is ampersand (&) followed by a name token (VALUE). The word scanner directs &VALUE to the macro processor.

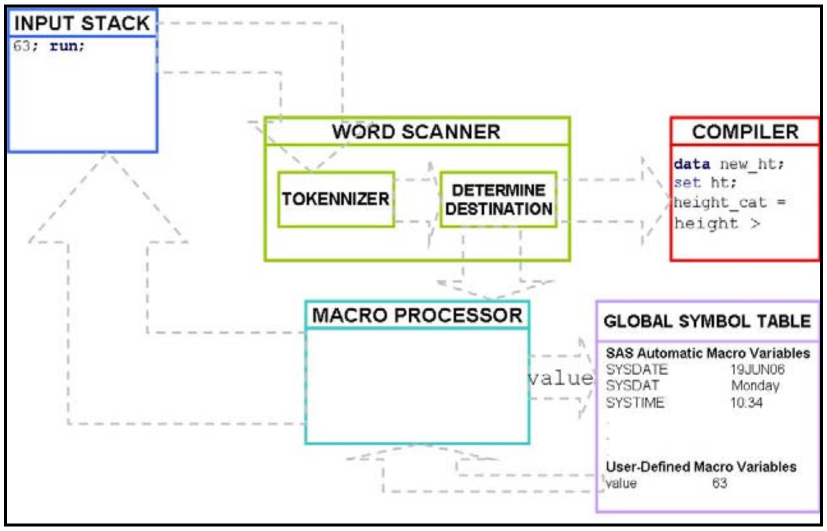


Figure 1e. The macro processor retrieves the value that corresponds to VALUE in the symbol table, which is 63, returning it to the input stack.

CREATING MACRO VARIABLES DURING THE DATA STEP EXECUTION

PROBLEMS WITH CREATING MACRO VARIABLES DURING THE DATA STEP EXECUTION

Sometimes creating a macro variable during the DATA step execution, based on the data value or programming logic, is necessary. Using the %LET statement will not work since the macro variable created by %LET occurs before the execution begins. For example, to calculate the chi-square statistics between SEX and HEIGHT_CAT among Blacks, you can use PROC FREQ with the CHISQ option in the TABLES statement, like below:

```
proc freq data=new_ht;
    where race = 'B';
    tables sex*height_cat/chisq;
run;
```

Output:

```
Statistics for Table of sex by height_cat
Statistic                     DF       Value      Prob
------------------------------------------------------
Chi-Square                     1      3.2590    0.0710
Likelihood Ratio Chi-Square    1      3.2898    0.0697
Continuity Adj. Chi-Square     1      1.4106    0.2350
Mantel-Haenszel Chi-Square     1      3.0083    0.0828
Phi Coefficient                       0.5007
Contingency Coefficient               0.4477
Cramer's V                            0.5007
 WARNING: 75% of the cells have expected counts less
          than 5. Chi-Square may not be a valid test.
```

The warning message in the output window tells us some of the cells from the 2-by-2 table have expected counts < 5, which violates the assumption for performing the chi-square statistics. Suppose that you would like to create a macro variable, TEST, which contains either a VALID or NOT VALID value. If any of the expected values in the 2-by-2 table are less then 5, then TEST will be assigned with “not valid;” otherwise, it will assigned with the VALID value. To calculate the expected counts for each cell, you can use the EXPECTED option in the TABLES statement from PROC FREQ. You can then output the expected counts to an output data set by using the ODS OUTPUT statement.

Program 2a:
```
proc freq data=new_ht;
    where race = 'B';
    tables sex*height_cat/expected;
    ods output CrossTabFreqs =CrossTabFreqs1;
run;

proc print data=CrossTabFreqs1;
run;
```

```
The SAS System
                                h                                       R         C
                                e           F                           o         o
                                i           r    E                      w         l
                                g       _   e    x         P            P         P    M
                                h   _   T   q    p         e            e         e    i
              T                 t   T   A   u    e         r            r         r    s
              a                 _   Y   B   e    c         c            c         c    s
O             b             s   c   P   L   n    t         e            e         e    i
b             l             e   a   E   E   c    e         n            n         n    n
s             e             x   t   _   _   y    d         t            t         t    g

1   Table sex * height_cat  F   0   11  1   7   5.53846   53.846   77.7778   87.5    .
2   Table sex * height_cat  F   1   11  1   2   3.46154   15.385   22.2222   40.0    .
3   Table sex * height_cat  F   .   10  1   9    .        69.231     .         .     .
4   Table sex * height_cat  M   0   11  1   1   2.46154    7.692   25.0000   12.5    .
5   Table sex * height_cat  M   1   11  1   3   1.53846   23.077   75.0000   60.0    .
6   Table sex * height_cat  M   .   10  1   4    .        30.769     .         .     .
7   Table sex * height_cat      0   01  1   8    .        61.538     .         .     .
8   Table sex * height_cat      1   01  1   5    .        38.462     .         .     .
9   Table sex * height_cat      .   00  1  13    .       100.000     .         .     0
```

Program 2b uses the %LET statement to create the macro variable TEST. To verify whether TEST is created correctly, you can use the %PUT statement[1]:

Program 2b:
```
data _null_;
    set CrossTabFreqs1 end=last;
    if not missing(expected) and expected < 5 then count +1;
    if last then do;
        if count then do;
            %let test = not valid;
        end;
        else do;
            %let test = valid;
        end;
    end;
run;
%put test: &test;
```

SAS log:
```
125  %put test: &test;
test: valid
```

Program 2b did not create the macro variable correctly. There are three cells with expected counts less than 5, but the TEST macro variable is assigned with a "valid" value. Let's exam this program in more detail. At first, Program 2b is pushed into the input stack. The word scanner directs the SAS code to the compiler until it reaches the macro trigger (%LET). The word scanner directs the %LET statement to the macro processor. The macro processor creates the macro variable TEST with the value of NOT VALID in the symbol table. After directing the %LET statement to the macro processor, the word scanner directs subsequent tokens to the compiler until it reads the second %LET statement. The word scanner directs the second %LET statement to the macro processor. The macro processor reassigns VALID to the macro variable TEST. The word scanner continues to send the remaining tokens to the compiler. When the compiler receives the semicolon following the RUN statement, it stops accepting tokens from the word scanner. When the compilation begins, there will be no more %LET statements in the DATA step.

THE SYMPUT(X) ROUTINES

To fix the problem in Program 2b, you need to be able to create a macro variable during the DATA step execution, which can be accomplished by using the SYMPUT routine. Since the macro variable is assigned with the value during the DATA step execution, you can only reference the macro variable after the DATA step in which it is created. The SYMPUT routine has the following form:

CALL SYMPUT (MACRO-VARIABLE, VALUE);

In the SYMPUT routine, both MACRO-VARIABLE and VALUE can be specified as literal (text in quotations), a DATA step variable, or a DATA step expression. When both MACRO-VARIABLE and VALUE are literal, they are enclosed in quotation marks. The text enclosed in quotation marks for the first argument is the exact macro variable name. The second argument enclosed in quotation marks is the exact value that is assigned to the macro variable. In Program 2, the macro variable we attempted to create is based on a calculated value in the DATA step. This is a perfect situation to utilize the SYMPUT routine.

Program 3:
```
data _null_;
    set CrossTabFreqs1 end=last;
    if not missing(expected) and expected < 5 then count +1;
    if last then do;
        if count then call symput ('test', 'not valid');
        else call symput ('test', 'valid');
    end;
run;
%put test: &test;
```

[1] Following is the general form of the %PUT statement: **%PUT *text*** , where *text* is any text string. The %PUT statement writes only to the SAS log. If the text is not specified, the %PUT statement writes a blank line. The %PUT statement resolves macro triggers in *text* before *text* is written. For more information, check SAS documentations.

SAS log:

```
139   %put test: &test;
test: not valid
```

In Program 3, both arguments in the SYMPUT routine are enclosed in quotation marks. The first argument in the SYMPUT routine, TEST, is the name of the macro variable. The second argument, NOT VALID or VALID, is the value that is assigned to the macro variable TEST. Since the variable COUNT is greater than 0, macro variable TEST is assigned with NOT VALID through the SYMPUT routine.

When the second argument in the SYMPUT routine is not in quotation marks, you are assigning the *value* of a DATA step variable to the macro variable. Any leading or trailing blanks that are part of the values of a DATA step variable will be part of the macro variables. If the DATA step variable is a numeric variable, the values will be converted to the character variables automatically by using the BEST12. format. When the first argument in the SYMPUT routine is not in quotation marks, you are creating multiple macro variables by using only one SYMPUT routine. The names of the macro variables that you are creating are the value of a DATA step variable. You can use a DATA step expression in either or both arguments in the SYMPUT routine.

The SYMPUTX routine is an improved version of the SYMPUT routine and becomes available beginning with SAS®9. Besides creating macro variables like the SYMPUT routine, the SYMPUTX routine can remove leading and trailing blanks from both arguments. Also, when the second argument is numeric, the SYMPUTX routine converts it to characters by using the BEST32.format instead of using the BEST12. format. The SYMPUTX routine has an optional third argument that enables you to specify the symbol table in which to store the macro variables, but the SYMPUT routine does not. Further details can be found in SAS documentations.

USING THE EXECUTE ROUTINE TO CREATE MACRO VARIABLES DURING THE DATA STEP EXECUTION

In addition to using SYMPUT(X) routines, you can also use the EXECUTE routine to create a macro variable during the DATA step execution. CALL EXECUTE has the following form:

CALL EXECUTE (ARGUMENT);

ARGUMENT in CALL EXECUTE can be a text expression that is enclosed in single or double quotation marks, the name of a character variable, or a character expression that is resolved by the DATA step to a macro text expression or a SAS statement.

If ARGUMENT in CALL EXECUTE produces macro language elements that are enclosed in single quotes, those elements execute immediately during the DATA step execution phase. On the other hand, if the generated macro language elements that are enclosed in double quotes, those elements will execute immediately during the DATA step compilation phase. Single quote notation is commonly used in most applications.

If ARGUMENT in CALL EXECUTE produces SAS language statements, or if the macro language elements from ARGUMENT generate SAS language statements, the SAS language statements will be placed into the input stack as additional program code and execute after the end of the current DATA step's execution. Program 4 creates the TEST macro variable by enclosing the %LET statement in quotations as the argument of CALL EXECUTE. More examples on CALL EXECUTE will be shown in the section below.

Program 4:

```
data _null_;
    set CrossTabFreqs1 end=last;
    if not missing(expected) and expected < 5 then count +1;
    if last then do;
        if count then call execute ('%let test = not valid;');
        else call execute ('%let test = valid;');
    end;
run;
%put test: &test;
```

SAS log:

```
153   %put test: &test;
test: not valid
```

REFERENCING MACRO VARIABLES INDIRECTLY

The method of referencing macro variables in previous examples is called direct macro variable references, which is placing an ampersand preceding the name of the macro variable. The SAS macro facility provides indirect macro variable references, which enables you to use an expression to generate a reference to one or a series of macro variables. In indirect referencing, you need to use more than one ampersand to precede the name of the macro variable. To resolve indirect references, the macro processor follows the rules below:

- Two ampersands (&&) are resolved to one ampersand.
- Macro variable references are resolved from left to right.
- More than one leading ampersand causes the macro processor to re-scan the reference until no more ampersands can be resolved.

Suppose that you have the following macro variables:

MACRO-VARIABLE NAME	MACRO-VARIABLE VALUE
I	1
VALUE1	A
VALUE2	B
VALUE3	W
VARNAME	VALUE
DAT	ht

Based on the macro variables above,

- &&VALUE&I resolves to A:

At beginning	First Pass	Second Pass
&&VALUE&I	&& → & VALUE → VALUE &I →1 **Result**: &VALUE1	&VALUE1 → A **Result**: A

- &&&VARNAME&I..TEST resolves to ATEST

At beginning	First Pass	Second Pass
&&&VARNAME&I..TEST	&& → & &VARNAME → VALUE &I. →1 .TEST → .TEST **Result**: &VALUE1.TEST	&VALUE1. → A TEST → TEST Result: ATEST

Sometimes you need to be able to create indirect references based on a given token (i.e.how to create a macro reference based on the given macro variables to generate a token 'A'). In this situation, you need to think "backward." Since 'A' is the value of macro variable VALUE1, you can write &VALUE1. Since 1 is a component of the macro variable VALUE1, you need to utilize indirect referencing, e.g. &&VALUE&I.

Let's look at a more complicated example: how to generate a token 'new_htA_chisq'?

At beginning	First Pass	Second Pass
new_htA_chisq	new_ →new_ ht → &DAT A →&VALUE1. (A period is necessary to separate the trailing text) _chisq → _chisq **Result**: new_&DAT&VALUE1.chisq	new_&DAT → new_&DAT & → && VALUE → VALUE 1 → &I. (A period is necessary to separate the tailing text) .chisq →.chisq **Result**: new_&DAT&&VALUE&I..chisq

MACRO PROGRAMS

Macro programs (or Macros) are compiled programs that enable you to substitute text in a program. Macros can utilize conditional logic to make decisions about the text that you want to substitute in your programs. Macros can also accept parameters, which enables you to pass information into your macro. When you include parameters in a macro definition, a macro variable is automatically created for each parameter each time you call the macro. You can include parameters in the macro definition in one of the following methods: Positional, Keyword, and Mixed. We will only focus on the Positional parameters in this paper, which has the following form:

```
%MACRO MACRO-NAME(PARAMETER1 <, …, PARAMETERN>);
TEXT
%MEND <MACRO-NAME>;
```

The MACRO-NAME is the name of the macro. TEXT can be constant text, SAS data set names, SAS variable names, SAS statements, macro variables, macro functions, or macro statements. If more than one parameter is being specified, you need to separate them with a comma. Each parameter must have a valid SAS name. You cannot use a SAS expression to generate the name of the parameter.

Executing a macro with positional parameters has the following form:

```
%MACRO-NAME(VALUE1 <, …, VALUEN>)
```

The VALUE(S) can be null, text, macro variable references, or macro calls. For positional parameters, the VALUES are assigned to the PARAMETERS via one-to-one correspondence. You don't need to place a semicolon at the end of the macro call; as a matter of fact, in some situations, placing a semicolon at the end of the macro call will cause an error. For example, Program 5 creates the macro CREATE_CAT1. CREATE_CAT1 takes only one argument, VALUE. This macro creates a new data set, *new_ht*, by adding an additional indicator variable, HEIGHT_CAT. HEIGHT_CAT is defined by comparing variable HEIGHT with VALUE.

Program 5:
```
%macro create_cat1(value);
    data new_ht;
        set ht;
        height_cat = height > &value;
    run;
%mend;

%create_cat1(63)
```

MACRO COMPILATION AND EXECUTION

MACRO COMPILATION

Before you execute your macro, you need to compile the macro first by submitting the macro definition. For example, when you submit Program 5, the code is pushed into the input stack (Figure2a). The word scanner begins tokenizing the program. When the word scanner detects % followed by a non-blank character in the first token, it starts to send the tokens to the macro processor. The macro processor examines the token and recognizes the beginning of a macro definition. It then starts to pull tokens from the input stack and compiles them until the %MEND statement terminates the macro definition. During macro compilation, the macro processor creates an entry in the session catalog. By default, macro programs are stored in a catalog in the WORK library (Figure 2b). The name of the catalog is SASMACR. The macro processor stores all macro program statements for that macro as macro instructions and stores all non-compiled items in the macro as text.

MACRO EXECUTION

After the macro is compiled, you can use it in your SAS programs for the duration of your SAS session without re-submitting the macro definition. To execute the compiled macro program, you must call the macro program and place a macro call anywhere in your program (except in the data lines of the DATALINES statement).

Once you submit the macro call, it is placed into the input stack. The word scanner examines the input stack and detects the percent sign (%) followed by a non-blank character in the first token. The word scanner directs the tokens to the macro processor to examine the token. The macro processor recognizes a macro call and begins to execute macro CREATE_CAT1(Figure2c). The macro processor creates a local symbol table for the macro and adds entries for the parameter and variable to the newly-created local symbol table. For this example, one macro variable, VALUE, is created and stored in the local symbol table with the corresponding value of 63 (Figure 2d). The macro processor begins to execute the compiled instructions of the macro and places SAS language statements on the input stack (Figure 2e). The word scanner starts to pass the tokens to the compiler until it encounters a macro variable reference (&VALUE). The word scanner directs &VALUE to the macro processor. The macro processor retrieves the value that corresponds to &VALUE in the local symbol table, which is 63, and returns it to the input stack (Figure 2f). The word scanner continues to pass tokens to the compiler. After the compiler receives the RUN token and a semicolon, the DATA step compilation begins and is immediately followed by the execution. The data set *new_ht* is then created.

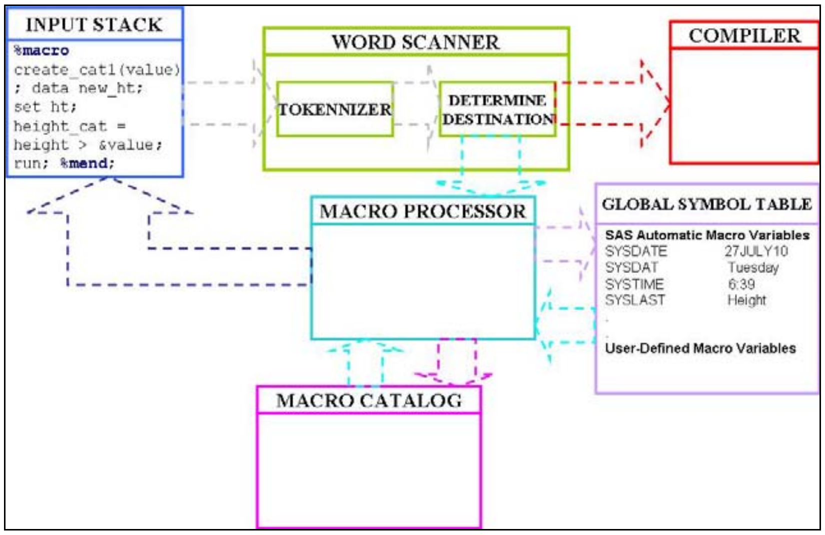


Figure2a. The macro definition is pushed into the input stack.

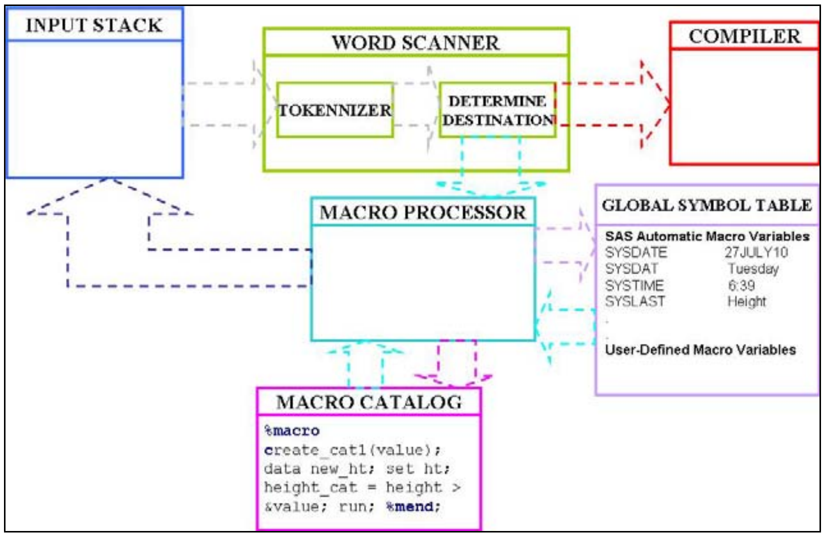


Figure2b. The compiled macro programs are stored in a catalog.

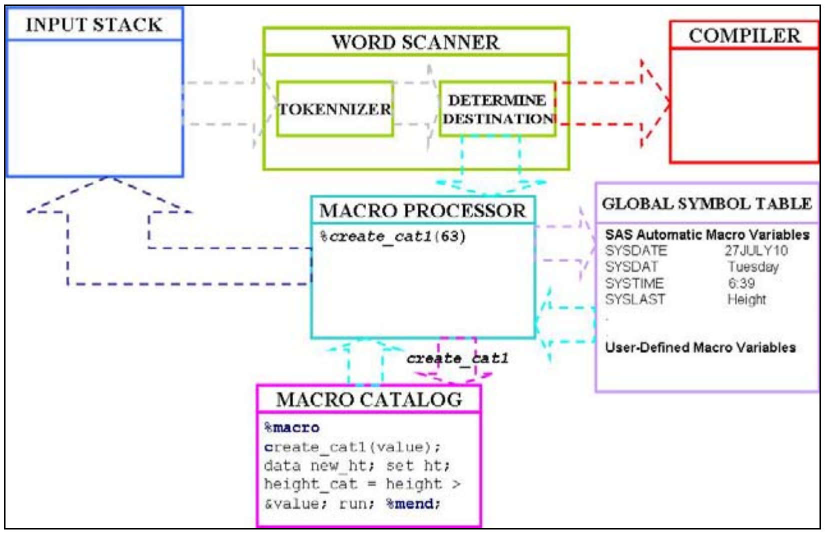
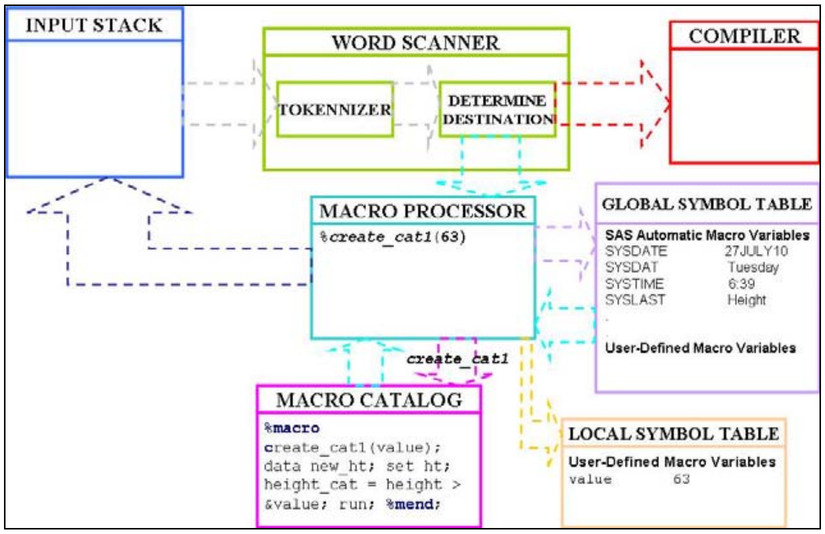
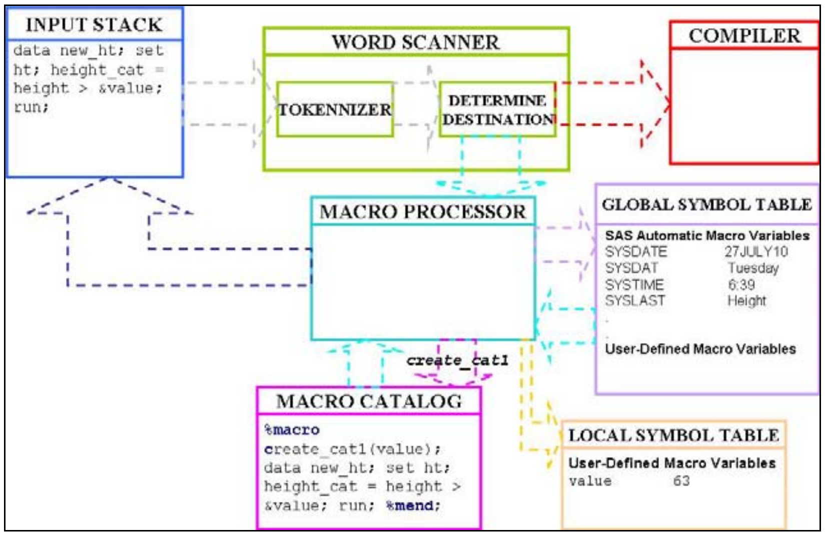


Figure2c. The macro processor recognizes a macro call and begins to execute macro CREATE_CAT1.



```
%macro
create_cat1(value);
data new_ht; set ht;
height_cat = height >
&value; run; %mend;
```

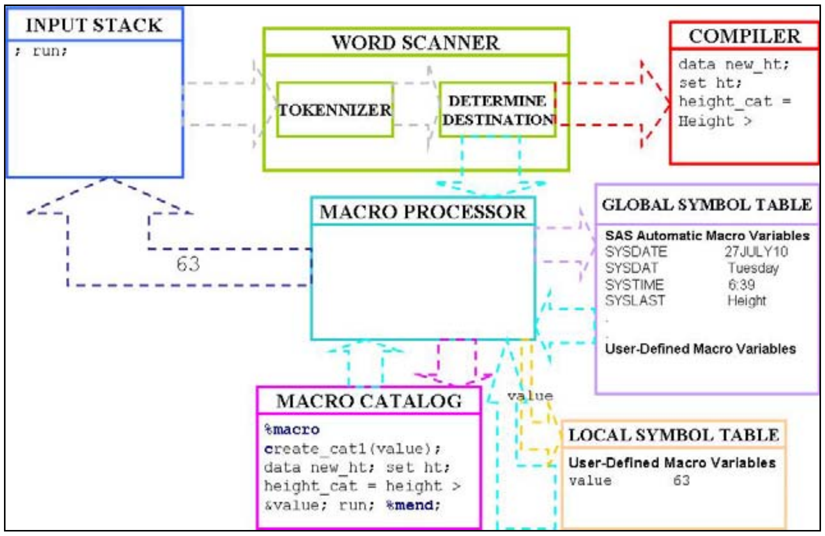
Figure2d. The macro variable VALUE is created and stored in the local symbol table with the corresponding value of 63.



```
data new_ht; set
ht; height_cat =
height > &value;
run;
```

```
%macro
create_cat1(value);
data new_ht; set ht;
height_cat = height >
&value; run; %mend;
```

Figure2e. The macro processor begins to execute the compiled instructions of the macro and places SAS language statements on the input stack.



```
; run;
```

```
data new_ht;
set ht;
height_cat =
Height >
```

```
%macro
create_cat1(value);
data new_ht; set ht;
height_cat = height >
&value; run; %mend;
```

Figure2f. The macro processor retrieves the value that corresponds to &VALUE in the local symbol table, which is 63, and returns it to the input stack.

THE GLOBAL AND LOCAL SYMBOL TABLES

THE GLOBAL SYMBOL TABLE

The global symbol table is automatically created when you start your SAS session and is deleted at the end of the session. All the automatic macro variables are stored in the global symbol table. You can create the macro variables that are stored in the global symbol table with a %LET statement in open code, a SYMPUT or SYMPUTX routine in the DATA step in open code, an INTO clause of a SELECT statement in PROC SQL or a %GLOBAL statement. To delete macro variable(s) from the global symbol table, you can use the %SYMDEL statement.

THE LOCAL SYMBOL TABLE

A local symbol table is automatically created when you call a macro that includes parameter(s) and it is deleted when the macro execution is finished. You can create macro variables that are stored in the local symbol table by the following method: parameters that are included in the macro definition, a %LET statement within a macro definition, the SYMPUT or SYMPUTX routines in the DATA step in a macro definition, an INTO clause of a SELECT statement in PROC SQL within a macro definition, or a %LOCAL statement.

RULES FOR CREATING/UPDATING MACRO VARIABLES

When creating a macro variable by using a %LET statement during the macro execution, the macro processor follows certain rules. For example, suppose that you created the following macro variable *within* the macro definition:

```
%let value = 63;
```

The macro processor will adhere the following rules when creating the macro variable VALUE:

1. The macro processor checks to see if VALUE already exists in the local symbol table. If it does, the macro processor updates VALUE in the local symbol table. If not, the macro processor goes to STEP 2.
2. The macro processor checks to see if VALUE already exists in the global symbol table. If it does, the macro processor updates VALUE in the global symbol table. If not, the macro processor goes to STEP 3.
3. The macro processor creates a macro variable, VALUE, in the local symbol table and assigns 63 to it.

When referencing a macro during the macro execution (for example, &VALUE), the macro processor follows the rules below:

1. The macro processor checks to see whether macro variable VALUE exists in the local symbol table. If it does, the macro processor retrieves the value of VALUE. If it does not, the macro processor goes on to STEP 2.
2. The macro processor checks to see whether macro variable VALUE exists in the global symbol table. If it does, the macro processor retrieves the value of VALUE. If it does not, the macro processor goes on to STEP 3.
3. The macro processor returns the tokens (&VALUE) to the word scanner. A warning message is written to the SAS log: '`WARNING: Apparent symbolic reference VALUE not resolved`'.

DIVIDE AND CONQUOR

To solve a difficult problem, it is better to break the problem down into a few small and easy-to-solve problems then to connect them together in the end. This strategy is known as Divide and Conquer. For example, we can divide our problem into the following components:

1. Subsetting the whole data set to a smaller data set for each ethnic group.
2. Creating the indicator variable HEIGHT_CAT.
3. Calculating the chi-square statistics for each ethnic group.
4. Combing the results together into one final table.

STEP 1: SUBSETTING THE DATA SET

To subset a data set, you need to write a macro with the following parameters:

- DAT: the input data set, which is the data set that you would like to subset, such as HT.
- VARNAME: the name of the variable, such as RACE.
- VAL: the value of the variable, such as W.

Before you write your macro, you should hardcode your program first to make sure it is working. Since this program requires three parameters, you can create three macro variables first by using the %LET statement. Then you can write the DATA step and use the macro variables that you created.

```
%let dat = ht;
%let varname = race;
%let val = W;
data &dat&val;
    set &dat;
    if &varname = "&val";
run;
```

Once you have confirmed that your code works, you can then write your macro by including the macro variable names in the parameter lists of the macro definition.

Program 6:

```
%macro subset(dat, varname, val);
    data &dat&val;
        set &dat;
        if &varname = "&val";
    run;
%mend;
%subset(ht, race, W)
```

The macro in Program 6 creates a data set (*htW)* by selecting White students from the data set *ht.* The name of the resulting data set is created by concatenating two parameters, DAT and VAL, from the macro definition.

STEP 2: CREATING THE HEIGHT_CAT VARIABLE

In Program 5, the macro CREATE_CAT1 only contains one parameter, VALUE. You can make this macro more flexible by including more parameters:

- o DAT: the input data set, such as HTW, which is the one you just created in the previous step
- o VARNAME: the name of a continuous variable that you used to create the indicator variable
- o VALUE: either null or a given value. If VALUE is null (or not provided), then VALUE can be calculated within the macro by using the mean of the continuous variable

One of the advantages of creating a macro program is that you will be able to use conditional logic to make decisions. You can conditionally process a portion of a macro by using the %IF-%THEN/%ELSE statement, which can only be used inside a macro program, not in open code. The %IF-%THEN/%ELSE statement has the following form:

```
%IF EXPRESSION %THEN ACTION;
<%ELSE ACTION;>
```

EXPRESSION is any macro expression that resolves to an integer. If the EXPRESSION resolves to a non-zero integer (TRUE), then the %THEN clause is processed. If the EXPRESSION resolves to zero (FALSE), then the %ELSE statement, if one is present, is processed. If the EXPRESSION resolves to a null value or a value containing nonnumeric characters, the macro processor issues an error message. ACTION is either constant text, a text expression, or a macro statement. If ACTION contains semicolons, then the first semicolon after %THEN ends the %THEN clause. Use a %DO group or a quoting function, such as %STR, to prevent semicolons in ACTION from ending the %IF-%THEN statement. Often you will use the %DO and %END statements in conjunction with the %IF-%THEN/%ELSE statement:

```
%IF EXPRESSION %THEN %DO;
ACTION
%END;
%ELSE %DO;
 ACTION
%END;
```

Programmers often get confused about the difference between the %IF-%THEN/%ELSE statement and the IF-THEN/ELSE statement. These two statements belong to two different languages. The %IF-%THEN/%ELSE statement is part of the SAS macro language that conditionally generates text. On the other hand, The IF-THEN/ELSE statement is part of the SAS language to conditionally execute SAS statements during the DATA step execution. The EXPRESSION that is the condition for the %IF-%THEN/%ELSE and the IF-THEN/ELSE statements are also different because the EXPRESSION in the %IF-%THEN/%ELSE statement can contain only operands that are constant text or text expressions that generate text; while the EXPRESSION in the IF-THEN/ELSE statement can contain operands that are DATA step variables, character constants, numeric constants, or date and time constants. When the %IF-%THEN/%ELSE statement generates text that is part of a DATA step, it is compiled by the DATA step compiler and executed. On the other hand, when the IF-THEN/ELSE statement executes in a DATA step, any text generated by the macro facility has been resolved, tokenized, and compiled. No macro language elements exists in the compiled code.

Similar to the previous example, you should hardcode your program first to make sure it is working. If the VALUE is a null value, you can write the following code:

```
%let dat = htW;
%let varname = height;
%let value =;

proc means data=&dat mean;
    var &varname;
    ods output summary = summary1;
run;
data _null_;
    set summary1;
    call symputx('value', &varname._Mean);
run;

data new_&dat;
    set &dat;
    &varname._cat = &varname > &value;
run;
```

If the VALUE is given, you can write the following code:

```
%let value =63;
data new_&dat;
    set &dat;
    &varname._cat = &varname > &value;
run;
```

Based on the code above, you can see that PROC MEANS and the first DATA step can be enclosed in the %IF-%THEN statement. That is, if VALUE is equal to the null value, then run the PROC MEANS and the DATA step to create the macro variable VALUE.

Program 7:
```
%macro create_cat(dat, varname, value);
    %if &value = %then %do;
        proc means data=&dat mean;
            var &varname;
            ods output summary = summary1;
        run;

        data _null_;
            set summary1;
            call symputx('value', &varname._Mean);
        run;
    %end;
    data new_&dat;
        set &dat;
        &varname._cat = &varname > &value;
    run;
%mend;

%create_cat(htW, height,)
```

The macro CREATE_CAT in Program 7 creates a data set (*new_htW)* that contains the newly-created indicator variable HEIGHT_CAT. The name of the newly-created variable is created by concatenating the parameter VARNAME and the text string "_cat". The name of the data set is created by concatenating the text string "new_" and the parameter DAT.

STEP 3: CALCULATE THE CHI-SQUARE STATISTICS

For this step, you need to create the following data set with four variables and one observation.

Group	Test	Chisq	P
W	valid	9.39323	0.00218

To generate this data set, four parameters are needed:

- o DAT: input data set, such as NEW_HTW, which is created from the previous step.
- o VAR1: one of the categorical or indicator variables that is used to calculate the chi-square statistics, such as HEIGHT_CAT, which is created from the previous step.
- o VAR2: same as VAR1, such as SEX.
- o GROUP_LABEL: used to indicate the group, such as W.

To hardcode your program, you can test the following code first:

```
%let dat = new_htw;
%let var1 = height_cat;
%let var2 = sex;
%let group_label = W;

proc freq data=&dat;
    tables &var1*&var2/chisq expected;
    ods output chiSq = chiSq1 CrossTabFreqs =CrossTabFreqs1;

data _null_;
    set chiSq1;
    if Statistic = 'Chi-Square';
    call symputx ('chisq', value);
    call symputx ('p', Prob);

data _null_;
    set CrossTabFreqs1 end=last;
    if not missing(expected) and expected < 5 then count +1;
    if last then do;
        if count then call symputx ('test', 'not valid');
        else call symputx ('test', 'valid');
    end;

data &dat._chisq;
    length group $ 3 test $ 9;
    group = "&group_label";
    chisq = &chisq;
    p = &p;
    test = "&test";
run;
```

The two CALL SYMPUTX in the first DATA step above creates two macro variables, CHISQ and P. Notice that the second arguments are not enclosed in quotation marks because the values of the VALUE and P DATA step variables are assigned to the macro variables CHISQ and P, respectively. The second DATA step above is taken directly from Program3, which is used to create the macro variable TEST. Now, you can write your macro like following:

Program 8:
```
%macro chi_sq(dat, var1, var2, group_label);
    proc freq data=&dat;
        tables &var1*&var2/chisq expected;
        ods output chiSq = chiSq1 CrossTabFreqs =CrossTabFreqs1;

    data _null_;
        set chiSq1;
        if Statistic = 'Chi-Square';
        call symputx ('chisq', value);
        call symputx ('p', Prob);

    data _null_;
        set CrossTabFreqs1 end=last;
        if not missing(expected) and expected < 5 then count +1;
        if last then do;
            if count then call symputx ('test', 'not valid');
            else call symputx ('test', 'valid');
        end;

    data &dat._chisq;
        length group $ 3 test $ 9;
        group = "&group_label";
        chisq = &chisq;
        p = &p;
        test = "&test";
    run;
%mend;

%chi_sq(new_htW, height_cat, sex, W)

proc print data=new_htW_chisq;
run;
```

```
The SAS System

Obs    group    test      chisq          p


 1       W      valid    9.39323    .002177878
```

The name of the data set that is created from the CHI_SQ macro from Program 8 is created by concatenating the DAT parameter with the “chisq” text string.

STEP4: CREATING THE FINAL TABLE

In the final step, you need to create the chi-square statistics for each and all races separately within one macro. Once the chi-square statistics is generated for each and all subgroups, you need to stack them all to create a final table. The final macro can consist of the following parameters:

- DAT: the input data set; for example, HT.
- NUM_VAR: the numerical variable; for example, HEIGHT. This is the variable that is used to create the indicator variable HEIGHT_CAT. Then you will use HEIGHT_CAT with CAT_VAR (next parameter) to calculate the chi-square statistics.
- CAT_VAR: the categorical variable, such as SEX. You use this variable to calculate the chi-square statistics.
- GROUP_VAR: the variable that is used to separate the input data set; for example, RACE.
- GROUP_VALUE: the value of GROUP_VAR, such as W, A, or B.
- CUTOFF: a threshold value that is used to create the indicator variable. If CUTOFF is not given, the mean of the NUM_VAR is used.

To calculate the chi-square statistics for each group, you can utilize the iterative %DO statement, which has the following form:

%DO MACRO-VARIABLE = START **%TO** STOP <**%BY** INCREMENT>;
TEXT AND MACRO LANGUAGE STATEMENTS
%END;

MACRO-VARIABLE is used to name a macro variable or a text expression that generates a macro variable name. Its value functions as an index that determines the number of times the %DO loop iterates. If the macro variable specified as the index does not exist, the macro processor creates it in the local symbol table.

You can change the value of the index variable during processing. START and STOP are integers or macro expressions that generate integers to control the number of times the portion of the macro between the iterative %DO and %END statements is processed. INCREMENT is an integer (other than 0) or a macro expression that generates an integer to be added to the value of the index variable in each iteration of the loop. By default, the increment is 1. INCREMENT is evaluated before the first iteration of the loop. Therefore, you cannot change it as the loop iterates. The iterative %DO statements are macro language statements that can be used only inside a macro program.

Just like before, you need to hardcode your program first. You can first create the macro variables by using the %LET statement. These macro variables will become the parameters of the macro.

```
%let dat = ht;
%let num_var = height;
%let cat_var = sex;
%let group_var = race;
%let group_value = A B W;
%let cutoff =;
```

The first step of the macro will be calculating the chi-square statistics for everyone in the data set. To calculate the chi-square statistics, you need to create the indicator variable, HEIGHT_CAT, first.

```
%create_cat(&dat, &num_var, &cutoff)
```

The macro above will create the variable HEIGHT_CAT, which will be stored in the data set *new_ht*. HEIGHT_CAT and NEW_HT will be used when calling the %CHI_SQ macro. To write a dynamic macro, you cannot use these two names directly; you must change it to macro references based on the macro variables above. The macro reference for HEIGHT_CAT will be &NUM_VAR._CAT and the macro reference for NEW_HT will be NEW_&DAT. Now you can calculate the chi-square statistics for all subjects by calling the macro below:

```
%chi_sq(new_&dat, &cat_var, &num_var._cat, all)
```

The chi-square statistics that are generated from the macro above will be stored in the *new_ht_chisq* and NEW_&DAT._CHISQ is the corresponding macro reference.

The next step will be calculating the chi-square statistics for each ethnic group. One additional step to calculate the chi-square statistics for each ethnic group is subsetting the entire data set into a smaller data set based on a given ethnic group. To simplify the lengthy code, you can use the iterative %DO statement to loop along each ethnic group. The values of the ethnic group are given in the GROUP_VALUE parameter. To use the iterative %DO loop, you also need to know the number of elements in the GROUP_VALUE, which will be the STOP value in the loop. There is a DATA step function, COUNTW, that can be used to count the number of elements in a given string; however, there is no corresponding macro function. In this situation, you can use the %SYSFUNC function to execute SAS functions, which has the following form:

%SYSFUNC (FUNCTION(ARGUMENT1 <, ARGUMENTN>))

To create a macro variable GROUP_NUM that contains the number of elements in the GROUP_VALUE, you can write the following:

```
%let group_num = %sysfunc(countw(&group_value));
```

To subset the whole data set into one that only contains one ethnic group, you only need one value from the GROUP_VALUE parameter. Thus, you can use the %SCAN function, which can be used to search for a word that is specified by its position in a string.

The %SCAN function is part of the macro character functions that have the same basic syntax as the corresponding DATA step functions. But unlike DATA step character functions, macro character functions enable you to communicate with the macro processor to manipulate the text that you insert into your program. The %SCAN function has the following form:

%SCAN (ARGUMENT, N <, CHARLIST>)

ARGUMENT is a character string or a text expression. N is an integer or a text expression that yields an integer, which specifies the position of the word to return. If N is greater than the number of words in ARGUMENT, the function returns a null string. If N is negative, %SCAN scans the character string and selects the word starting from the end of the string and searches backward. CHARLIST is used to specify an optional character expression that initializes a list of characters. The default CHARLIST is blank ! $ % & ( ) * + , - . / ; < ^ |

The iterative %DO loop will loop three times within the macro program. You can create a macro variable that serves as an index to test your program first before writing them into a loop. Suppose that you are using I as the index macro variable.

```
%let i = 1;
```

Next, create a macro variable VALUE1 or VALUE&I that contain the value A by using the %SCAN function.

```
%let value&i = %scan(&group_value, &i);
```

Next, subset the data set by calling the %SUBSET macro. To reference the macro variable VALUE1, you need to use indirect referencing, &&VALUE&I.

```
%subset(&dat, &group_var, &&value&i)
```

After subsetting the data set, the name of the resulting data set is *htA* and the corresponding macro reference is &DAT&&VALUE&I. Now you can create the HEIGHT_CAT variable by calling the %CREATE_CAT macro.

```
%create_cat(&dat&&value&i, &num_var, &cutoff)
```

The resulting data set will be *new_htA* or NEW_&DAT&&VALUE&I. Next, you can call the %CHI-SQ macro.

```
%chi_sq(new_&dat&&value&i, &cat_var, &num_var._cat, &&value&i)
```

The chi-square statistics will be stored in *new_htA_chisq*, or NEW_&DAT&&VALUE&I.._CHISQ. To calculate the chi-square statistics for Blacks, you can just assign the macro variable I to 2. Similarly, to calculate the chi-square statistics for Whites, you can assign I to 3. These repetitions can be enclosed in an iterative %DO loop like below:

```
%do i = 1 %to &group_num;
    %let value&i = %scan(&group_value, &i);
    %subset(&dat, &group_var, &&value&i)
    %create_cat(&dat&&value&i, &num_var, &cutoff)
    %chi_sq(new_&dat&&value&i, &cat_var, &num_var._cat, &&value&i)
%end;
```

Once you are done calculating the chi-square statistics for each and all ethnic groups, you can stack them all by using a DATA step.

```
data final;
    set new_ht_chisq
        new_htA_chisq
        new_htB_chisq
        new_htW_chisq;
run;
```

If you need to write the DATA step above within the macro, you have to use macro references like we did before. For example,

```
data final;
    set new_&dat._chisq
    %do i = 1 %to &group_num;
        new_&dat&&value&i.._chisq
    %end;
    ;
run;
```

Program 9 is the final version of the macro by putting all the codes above into one single macro.

Program 9:
```
%macro chisq_table(dat, num_var, cat_var, group_var, group_value, cutoff);

    %local i;
    %create_cat(&dat, &num_var, &cutoff)
    %chi_sq(new_&dat, &cat_var, &num_var._cat, all)

    %let group_num = %sysfunc(countw(&group_value));

    %do i = 1 %to &group_num;
        %let value&i = %scan(&group_value, &i);
        %subset(&dat, &group_var, &&value&i)
        %create_cat(&dat&&value&i, &num_var, &cutoff)
        %chi_sq(new_&dat&&value&i, &cat_var, &num_var._cat, &&value&i)
    %end;

    data final;
        set new_&dat._chisq
        %do i = 1 %to &group_num;
            new_&dat&&value&i.._chisq
        %end;
        ;
    run;
%mend;

%chisq_table(ht, height, sex, race, A B W, )
```

ALTERNATIVE WAY TO SOLVE THE PROBLEM BY USING CALL EXECUTE

If you invoke a macro by enclosing the macro call as the ARGUMENT of CALL EXECUTE, the macro call will execute immediately. If the macro call generates any macro language element, such as the %IF-%THEN statement or macro references, these macro language elements execute immediately. However, any of the SAS language statements that are generated by the macro call will be pushed to the input stack and executed after the end of the current DATA step, which contains CALL EXECUTE. This will create problems if you invoke a macro that contains references for macro variables that are created by CALL SYMPUT(X). CALL SYMPUT(X) is not considered a part of the macro language; instead, it is just DATA step CALL routines. That is to say the macro references to the macro variables created by CALL SYMPUT(X) will execute before they are even created.

In Program 10, macro FOO creates a macro variable VALUE by using CALL SYMPUTX. This program illustrates the differences in invoking FOO between using CALL EXECUTE and without using CALL EXECUTE.

Program 10:
```
option mprint mlogic symbolgen;
%macro foo;
    %local value;
    data bar;
        a = 5;
        call symputx('value', a);
    run;
    %put value inside macro foo: &value;
%mend;

%foo
%put value outside macro foo &value;

data _null_;
    call execute('%foo');
run;
%put value outside macro foo &value;
```

SAS Log from Program 10:

```
792  %foo
MLOGIC(FOO):  Beginning execution.
MLOGIC(FOO):  %LOCAL  VALUE
MPRINT(FOO):   data bar;
MPRINT(FOO):   a = 5;
MPRINT(FOO):   call symputx('value', a);
MPRINT(FOO):   run;

NOTE: The data set WORK.BAR has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

MLOGIC(FOO):  %PUT value inside macro foo: &value
value inside macro foo: 5
MLOGIC(FOO):  Ending execution.
WARNING: Apparent symbolic reference VALUE not resolved.
793  %put value outside macro foo &value;
value outside macro foo &value
794
795  data _null_;
796      call execute('%foo');
797  run;

MLOGIC(FOO):  Beginning execution.
MLOGIC(FOO):  %LOCAL  VALUE
MPRINT(FOO):   data bar;
MPRINT(FOO):   a = 5;
MPRINT(FOO):   call symputx('value', a);
MPRINT(FOO):   run;
MLOGIC(FOO):  %PUT value inside macro foo: &value
value inside macro foo:
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

MLOGIC(FOO):  Ending execution.

NOTE: CALL EXECUTE generated line.
1   + data bar;         a = 5;         call symputx('value', a);     run;

NOTE: The data set WORK.BAR has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


798  %put value outside macro foo &value;
value outside macro foo 5
```

When invoking FOO the first time without using CALL EXECUTE, the macro variable VALUE is created and stored in the local symbol table. The VALUE is then deleted from the local symbol table at the end of the macro execution. When using CALL EXECUTE to invoke FOO, the %LOCAL statement executes immediately, which assigns the macro variable VALUE to a NULL value. The DATA step within the macro FOO was pushed to the input stack. The %PUT statement executes next; notice that at this point, VALUE contains a null value. The DATA step that creates BAR executes after the execution of FOO, which creates the macro variable VALUE. Since the macro execution has already ended, the VALUE is then stored in the global symbol table, which is not what you intended.

In program 9, the CHISQ_TABLE macro calls two macros (CREATE_CAT and CHI_SQ) that contain the SYMPUT routine. Thus, to modify the CHISQ_TABLE macro, you need to also modify the CREATE_CAT and CHI_SQ macros by not including the SYMPUT routines.

Program 11 contains three macros. The first two macros (CREATE_CAT_NEW and CHI_SQ_NEW) are the modifications of macros CREATE_CAT and CHI_SQ without using the SYMPUT routine. The last macro (CHISQ_TABLE_NEW) is the modification of CHISQ_TABLE by utilizing a series of CALL EXECUTE. The CHISQ_TABLE_NEW macro also eliminates the GROUP_VALUE parameters, which makes the macro simpler. In the CHISQ_TABLE_NEW macro, after calculating the chi-square statistics for all the subjects combined, it creates data set LIST that contains one variable RACE (&GROUP_VAR) with three unique values (A, W, B). The first DATA _NULL_ utilizes CALL EXECUTE three times to subset the data by each ethnic group, creates indicator variables, and calculates the chi-square statistics for each ethnic group. The CATT function within the CALL EXECUTE is used to generate macro expressions. This approach also eliminates the need for indirect referencing. CALL EXECUTE in the second DATA _NULL_ is used to stack all the chi-square statistics into one table.

Program 11:
```
%macro create_cat_new(dat, varname, value);
    %if &value = %then %do;
        proc means data=&dat mean;
            var &varname;
            ods output summary = summary1;

        data new_&dat (drop=height_Mean foo);
            retain foo;
            merge &dat summary1;
            if _N_ = 1 then foo = height_Mean;
            &varname._cat = &varname > foo;
        run;
    %end;
    %else %do;
        data new_&dat;
            set &dat;
            &varname._cat = &varname > &value;
        run;
    %end;
%mend;

%macro chi_sq_new(dat, var1, var2, group_label);
    proc freq data=&dat;
        tables &var1*&var2/chisq expected;
        ods output chiSq = chiSq1 CrossTabFreqs =CrossTabFreqs1;

    data chiSq2 (rename =(Value = chisq Prob = p) keep = Value Prob);
         set chiSq1;
         if Statistic = 'Chi-Square';

    data CrossTabFreqs2 (keep=test);
        retain test;
        length test $ 9.;
        set CrossTabFreqs1 end=last;
        if not missing(expected) and expected < 5 then count +1;
        if last then do;
            if count then test = 'not valid';
            else test ='valid';
            output;
        end;

    data &dat._chisq;
        length group $ 3.;
        group = "&group_label";
        merge CrossTabFreqs2 chiSq2;
    run;
%mend;
```

```
%macro chisq_table_new(dat, num_var, cat_var, group_var, cutoff);
    %create_cat_new(&dat, &num_var, &cutoff)
    %chi_sq_new(new_&dat, &cat_var, &num_var._cat, all)  *new_&dat._chisq;

    proc sort data=&dat out= list (keep =race) nodupkey;
        by &group_var;
    run;

    data _null_;
        set list;
       call execute(catt('%subset(&dat, &group_var,', &group_var, ')'));
       call execute(catt('%create_cat_new(&dat.', &group_var, ',&num_var, &cutoff)'));
       call execute(catt('%chi_sq_new(new_&dat.', &group_var, ',&cat_var,
            &num_var._cat,', &group_var,')'));
    run;

    data _null_;
        set list end=last;
       if _N_ = 1 then do;
           call execute('data final;');
           call execute('set new_&dat._chisq');
       end;
       call execute(catt('new_&dat.', &group_var, '_chisq'));
       if last then call execute('; run;');
    run;
%mend;

%chisq_table_new(ht, height, sex, race, )
```

CONCLUSION

The SAS language and the SAS macro language look similar but are completely different languages. We use the macro language to write macro programs to generate SAS code. Understanding the mechanisms of macro processing is essential for creating macro variables precisely.

REFERENCES

Burlew, Michele M. SAS® Macro Programming Made Easy, 2nd Edition
SAS Institute Inc. 2006. SAS OnlineDoc® 9.1.3. Cary, NC: SAS Institute Inc
Whitlock, Ian, CALL EXECUTE: How and Why, Proceedings of the 22nd Annual SAS Users Group International Conference, 1997

ACKNOWLEDGMENTS

I would like to thank Russ Tyndall, Technical Support Analyst from SAS Technical Support, for his valuable programming suggestions and insight.

CONTACT INFORMATION

Arthur Li
City of Hope Comprehensive Cancer Center
Department of Information Science
1500 East Duarte Road
Duarte, CA 91010 - 3000
Work Phone: (626) 256-4673 ext. 65121
Fax: (626) 471-7106
E-mail: arthurli@coh.org