SAS Macros Workshop

I. Why use SAS Macros?

A SAS macro is way of defining parts of or collections of SAS statements which can be carried out repeatedly or which can substitute names of datasets or variables with symbolic names. SAS macro language also gives you the opportunity to insert programming steps inside a SAS code.

SAS Macro Advantages:
- Reduce coding time
- Reduce programming errors by not having to edit so many lines of code
- In some cases, it could be even more time efficient in execution

SAS Macro Disadvantages:
- Harder to understand if multiple people work on it
- Harder to debug
- Some of the macro features (call symput) do not resolve in execution in the log file until the very end of a data step
- Not easily adaptable

All the usual programming elements: if-then-else statements, loops for I=1 to N do, and other similar operators can be used in SAS macros.

II. Overview of elements of the macro language

The three different main elements of the macro language are:

- macro variables;
- macro program statements;
- macro functions.

III. Macro variables

Macro variables are tools that enable you to modify text in a SAS program through a symbolic substitution. It assigns a value (either string or integer) to a variable that you can invoke several times in a program after that.

To define a macro variable you can use the %let statement.

EXAMPLE 1:

For example to assign the value 24 to the variable named sto, you do:
%let sto=24;

SYNTAX:
%<Name of MACRO Variable> = Macro variable value;

The variable sto will take this value every time it gets invoked until another macro variable statement changes it. This code can be placed anywhere in the program except within data lines.

To invoke a macro variable place a & before its name. The macro processor resolves this reference to the variable by replacing &sto by the value of sto.

%let sto=year;
proc means data=&sto;

So, for example,

%let sto=1;
proc means data = temp&sto;
Run;

Will really resolve to

Proc means data = temp1;
Run;

EXAMPLE 2:

Macro variables that contain entire sections of a SAS program can also be created (with the use of the str macro function):

%let sto2=%str(proc means data=year;
  var rainfall;
  run;
);

Will resolve to the proc means step running every time the sto2 macro variable is invoked.

IV. Macro program statements

A macro program is a text identified by a name. These lines of codes can be invoked several times by using this name. So, in its most simple form, a macro program is
equivalent to a macro variable. However, unlike a macro variable, a macro program can include more complex functionalities, such as if-then loops, etc.

The next example is one of a simple macro that could also be defined by a macro variable:

**EXAMPLE 3 (Simple macro template)**

```sas
%macro example;
    proc means data=year;
    var rainfall;
%mend example;
```

The definition of a macro must begin by a `%macro` statement followed by the name of the macro. Here, our macro has been named example. The statement must end by a `%mend` statement. When it is invoked, the macro processor will run the lines of text between the `%macro` and `%mend` statements.

To invoke a macro program, place a `%` sign before its name. The macro example will be invoked in the following way:

```sas
%example;
```

This line of text can be included at any time in the program. For example:

**EXAMPLE 4:**

```sas
Proc plot data=year;
    plot rainfall*temperature;
%example;
Proc print data=year;
```

The processor will execute the following lines:

```sas
Proc plot data=year;
    plot rainfall*temperature;
Proc means data=year;
    var rainfall;
Proc print data=year;
```

**MACROS WITHIN MACROS**

Macro variables can be invoked inside a macro program. In fact, a macro program can even be invoked inside another macro program:

**EXAMPLE 5:**

```sas
%let var1=rainfall;
```
%let var2=temperature;

%macro design;
   proc plot data=result;
   plot &var1*&var2;
%mend design;

%macro compile;
   data result;
      set year;
      keep &var1 &var2;
      %design;
%mend compile;

If elsewhere in the program, compile is invoked by the statement %compile, the processor will replace these lines by:

data result;
   set year;
      keep rainfall temperature;

   proc plot data=result;
   plot rainfall*temperature;

MACRO WITH PARAMETERS

There is an easier solution than invoking macro variables inside the definition of a macro program: you can use parameters in the definition of your macro.

This is perhaps the most widely used form of the macro features.

For example, to redefine the macro design with parameters, type the following lines:

EXAMPLE 6:
%macro design1(para1,para2);

proc plot data=year;
   plot &para1*&para2;

%mend design1;

To invoke it, we will type: %design1(rainfall,temperature);
This gives the parameter para1 the value rainfall and para2 the value temperature (the processor assigns the values to the parameters in the order entered). The result of this line of code will be equivalent to the %design used previously, without having to define the macro variables var1 and var2.

MACROS WITHIN DATA STEPS

%DO Statement

If you want to perform procedures on a range of consecutive integer values for a certain variable or a set of variables, you can use a do loop. The following example introduces the use of do-loops with a macro.

EXAMPLE 7:
%macro year;
%do i=75 %to 99;
    proc means data=year&i;
    var rainfall;
%end;
%mend year;

A %do statement will always be terminated by a %end statement. The counter is then referenced by a &i inside the macro.

%IF %THEN %DO STATEMENT

This tool is used to insert conditional statements in a macro program. To illustrate this new tool, an extension of the previous example is used.

Suppose that it is necessary to obtain the average rainfall for every year, but that you need graphs only for the years after 1990, then type the following macro:

EXAMPLE 8:
%macro analyze;
%do i=75 %to 99;
    proc means data=year&i;
    var rainfall;
    %if i>=90 %then %do;
        proc plot data= year&i;
        plot rainfall*temperature;
    %end;
%end;
%mend analyze;

%do %while and %do %until statements exist.
Suppose a series of macro variables m1, m2, m3…m25 has already been defined. If they represent the rainfall predictions produced by a model, it is interesting to compute the difference between the real value and the prediction. This can be done using a loop on these macro variables:

**EXAMPLE 9:**
```
%do j=1 %to 25;
   data compare&j;
      set year&j;
      dif=rainfall-&m&j;
%end;
```

You can see that this can be done by placing two & before the name of the macro variable. The processor resolves it in the following way: If for example j=3, it replaces first &&m&j by &m3, and then scans it again to replace &m3 by the value it has been previously assigned.

## V. Macro functions

Macro functions process macro expressions, called arguments, to produce a new expression. While DATA step functions are applied to values on a vector of variables, macro functions are used to create, modify, and work with text strings. Initially this may seem like a minor difference, but because macro functions deal with text, they can be used to build SAS code. This becomes the powerful advantage of the macro language. Macro functions can be divided into three categories:

- character functions;
- evaluation functions;
- quoting functions.

### MACRO CHARACTER FUNCTIONS

This type of macro function provides information about the string it takes as argument (see following example: the %length function). For a more extensive list and more detailed information, see the ‘SAS Guide to Macro Processing’ (on the shelf in my cubicle).

Some of the macro character functions that have analogous DATA step functions include:
An important distinction between the use of macro functions and data step equivalents is:

- **DATA step functions work on character strings, numeric values, and DATA step variable values.**

- **Macro functions are applied to text strings that NEVER contain the values of DATA step variables.**

Several of these functions have two forms, with and without a Q at the start of the function name. Functions with names that start with Q (quoting) remove the meaning from special characters including the ampersand (&), percent sign (%), and mnemonic operators in returned values.

**%LENGTH**

The %length function returns the length of the string it takes as an argument. In the following example the objective is to determine if the length of a variable name is smaller or longer than 8 characters (some softwares don’t accept variable names longer than 8, so it can be necessary to change the names before exporting the data set) and replace the variable by the same one but with a new name.

**EXAMPLE 10:**

```sas
%macro export (name,newname);
  %if %length(&name)>8 %then %do;
    data year;
    set year;
    &newname=&name;
    drop &name;
  %end;
%mend limit;
```

<table>
<thead>
<tr>
<th>Macro Function(s)</th>
<th>Analogous DATA Step Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>%INDEX</td>
<td>index</td>
<td>First occurrence of a text string is located</td>
</tr>
<tr>
<td>%LENGTH</td>
<td>length</td>
<td>Character count</td>
</tr>
<tr>
<td>%SCAN %QSCAN</td>
<td>scan</td>
<td>Search for the n\textsuperscript{th} word in a text string</td>
</tr>
<tr>
<td>%SUBSTR %QSUBSTR</td>
<td>substr</td>
<td>Select text based on position</td>
</tr>
<tr>
<td>%UPCASE</td>
<td>upcase</td>
<td>Convert to upper case</td>
</tr>
</tbody>
</table>
%INDEX

SYNTAX
%INDEX(argument1,argument2)

The %INDEX function searches the first argument (ARGUMENT1) for the first occurrence of the text string which is contained in the second argument (ARGUMENT2). If the target string is found, the position of its first character is returned as the function’s response (0 if not found).

EXAMPLE 11

This example stores three words in the macro variable &X. The %INDEX function is then used to search in &X for the string TALL and the result is then displayed using the %PUT statement.

%LET X=LONG TALL SALLY;
%LET Y=%INDEX(&X,TALL);
%PUT TALL CAN BE FOUND AT POSITION &Y;

Notice that the TALL as the second argument is not in quotes. The %PUT results in the following text being written to the LOG:

TALL CAN BE FOUND AT POSITION 6

%LENGTH

The %LENGTH function determines the length (number of characters) of it’s argument. The number of detected characters is then returned. When the argument is a null string the value of 0 is returned.

SYNTAX
%LENGTH (argument)

EXAMPLE 12

In the macro %LOOK the name of the incoming data set is checked to see if it exceeds 8 characters.

%Macro LOOK(dsn,obs); %put &wscan &wqscan;
%if %length(&dsn) gt 8 %then
%put Name is too long - &dsn;
%else %do;

PROC CONTENTS DATA=&dsn;
TITLE "DATA SET &dsn";
RUN;
PROC PRINT DATA=&dsn (OBS=&obs);
TITLE2 "FIRST &obs OBSERVATIONS";
RUN;
%MEND LOOK;

The LOG shows that the following data set name exceeds 8 characters:
53 %look(demographics, 5)

Name is too long - demographics

%SCAN and %QSCAN

SYNTAX
%SCAN(argument1,argument2[,delimiters])
%QSCAN(argument1,argument2[,delimiters])

The %SCAN and %QSCAN functions both search a text string (ARGUMENT1) for the n\textsuperscript{th} word (ARGUMENT2) and returns its value. If ARGUMENT3 is not otherwise specified the same word delimiters are used as in the DATA step SCAN function. For an ASCII system these include the following (for EBCDIC the \textasciitilde{} is substituted for the \textasciicircum{}):

\texttt{blank . < ( + | \& $ * ) ; ^ - / , \% > \textasciitilde{}}

%QSCAN removes the significance of all special characters in the returned value.

EXAMPLE 13

The macro variable &X below can be broken up using the %SCAN function.

%LET X=XYZ.ABC/XYY;
%LET WORD=%SCAN(&X,3);
%LET PART=%SCAN(&X,1,Z);
%PUT WORD IS &WORD AND PART IS &PART;

The %PUT returns the following:

\texttt{WORD IS XYY AND PART IS XY}

Notice that the word delimiter (third argument) is not enclosed in quotes as it would be in the DATA step SCAN function.
The `%QSCAN` function is needed when you want to return a value that contains an ampersand or percent sign. This is demonstrated below:

```
%let dsn = clinics;
%let string =
%nrstr(*&stuff*&dsn*&morestuff);  

%put &wscan &wqscan;
```

The `%PUT` writes:
clinics &dsn

Both functions return the value &DSN, but since the meaning of the & is not masked by `%SCAN`, the &DSN in &WSCAN is resolved to clinics.

EXAMPLE 14

```
%macro a;
aaaaa
%mend a;

%macro b;
bbbbbb
%mend b;

%macro c;
cccccc
%mend c;

%let x=%nrstr(%a*%b*%c);
%put X: &x;
%put The third word in X, with SCAN: %scan(&x,3,*);
%put The third word in X, with QSCAN: %qscan(&x,3,*);
```

The `%PUT` statement writes this line:

X: %a*%b*%c
The third word in X, with SCAN: ccccccc
The third word in X, with QSCAN: %c

 `%SUBSTR` and `%QSUBSTR` 

Like the DATA step SUBSTR function these macro functions return a portion of the string in the first ARGUMENT. The substring starts at the POSITION in the second argument and optionally has a LENGTH of the third argument.
SYNTAX

%SUBSTR (argument,position[,length])
%QSUBSTR (argument,position[,length])

As is the case with most other macro functions, each of the three arguments can be a text string, macro variable, expression, or a macro call. If a value for LENGTH is not specified, a string containing the characters from POSITION to the end of the argument is produced.

EXAMPLE 14

%LET CLINIC=BETHESDA;
%IF %SUBSTR(&CLINIC,5,4) = ESDA %THEN
%PUT *** MATCH ***;
%ELSE %PUT *** NOMATCH ***;

The LOG would contain *** MATCH *** since &CLINIC has the value ESDA in characters 5 through 8.

As is shown in the following example, the %QSUBSTR function allows you to return unresolved references to macros and macro variables.

EXAMPLE 15

%let dsn = clinics;
%let string =
%nrstr(*&stuff*&dsn*&morestuff);
%let sub = %substr(&string,9,5);
%let qsub = %qsubstr(&string,9,5);
%put &sub &qsub;

The %PUT will write clinics* &dsn* in the LOG.

%UPCASE

The %UPCASE macro function converts all characters in the ARGUMENT to upper case. This function is especially useful when comparing text strings that may have inconsistent case.

Syntax

%UPCASE(argument)

EXAMPLE 16

The following code allows the user to differentially include a KEEP= option in the PROC PRINT statement.

The %UPCASE function is used to control for variations in the text that is supplied by the user in the macro call.
%macro printit(dsn);
* use a KEEP for CLINICS;
%if %upcase(&dsn)=CLINICS %then
  %let keep=(keep=lname fname ssn);
%else %let keep=;
proc print data=&dsn &keep;
title "Listing of %upcase(&dsn)"
run;
%mend printit;
%printit(cLinICs)

The macro call to %PRINTIT produces the following code.

proc print data=cLinICs (keep=lname fname ssn);
title "Listing of CLINICS"
run;

MACRO EVALUATION FUNCTIONS

The macro evaluation functions evaluate arithmetic and logical expressions in the macro language. They only perform integer arithmetic.

The evaluation functions are the
  - %eval and
  - %sysevalf

%EVAL

In the following example, we assume that a certain procedure produces an integer stored in the macro variable base (for example, a prediction of the number of years required for a specific analysis). If 3 is added to this integer, we could obtain the year up to which proc means have to be run so as to remove the possible estimation errors in the parameter base.

EXAMPLE 17:
%macro add(base);
  %let result=%eval(&base+3);
  %do i=75 %to &result;
    proc means data=year&i;
    var rainfall;
  %end;
%mend add;
We see that once the macro expression is evaluated, the result can be stored in another macro and you can use it immediately.

Logical expression can also be evaluated:

**EXAMPLE 18:**

```
%macro biggest(a,b);
%let logic=%eval(&a>&b);
%mend biggest;
```

%SYSEVALF

You can use this function to perform non-integer arithmetic and the function will even return a non-integer result from an arithmetic operation.

**SYNTAX**

%SYSEVALF(expression[,conversion-type])

The EXPRESSION is any arithmetic or logical expression which is to be evaluated and it may contain macro references.

The second argument, CONVERSION-TYPE, is an optional conversion to apply to the value returned by %SYSEVALF. Since this function can return non-integer values, problems could occur in other macro statements that use this function but expect integers.

When you need the result of this function to be an integer, use one of the CONVERSION-TYPEs. A specification of the CONVERSION-TYPE converts a value returned by %SYSEVALF to an integer or Boolean value so it can be used in other expressions that require a value of that type. CONVERSION-TYPE can be:

- BOOLEAN 0 if the result of the expression is 0 or missing, 1 if the result is any other value.
- CEIL round to next largest whole integer
- FLOOR round to next smallest whole integer
- INTEGER truncate decimal fraction

The CEIL, FLOOR, and INTEGER conversion types act on the expression in the same way as the DATA step functions of the same (or similar) names i.e. CEIL, FLOOR, and INT.

**EXAMPLE 19**

The following table shows a few calls to %SYSEVALF and the resulting values:
MACRO QUOTING FUNCTIONS

Quoting functions allow the user to pass macro arguments while selectively removing the special meaning from characters such as &, %, ;, ‘, and “. Most of these functions are not commonly used and are even less commonly understood. Although they are powerful and can even be necessary, programming solutions are usually available that do not require the use of the quoting functions.

All quoting functions are not alike. Consult the documentation to get the gory details, however the following three functions should solve most of your quoting problems.

%STR

The most commonly used macro quoting function is %STR. Often it is used along with the %LET statement to mask semicolons that would otherwise terminate the %LET.

In the following example we want to create a macro variable &P that contains two SAS statements;

%LET P=PROC PRINT DATA=DSN; RUN;

Because the semicolon following DSN terminates the %LET statement, the macro variable &P contains PROC PRINT DATA=DSN which will almost certainly result in a syntax error due to the missing semicolon.

The %STR function masks the semicolon by quoting it.

%LET P=%STR(PROC PRINT DATA=DSN; RUN;);

This results in the macro variable &P being correctly assigned the two statements.

PROC PRINT DATA=DSN; RUN;
%BQUOTE

The `%BQUOTE` function is probably the best choice as an overall quoting function. It eliminates many of the limitations of the `%STR` function, and it will also remove the meaning from unmatched symbols that are normally found in pairs such as quotes and parentheses. The following `%LET` will cause all sorts of problems because the apostrophe will be interpreted as an unmatched quote.

**Example 20:**

```sas
%let a = Sue's new truck;
The %STR function will not help because %STR does not mask quote marks, however %BQUOTE does.

%let a = %bquote(Sue's new truck);
%put &a;
```

This will correctly resolve to assigning “Sue’s new truck” to the macro variable a.

%UNQUOTE

Once a quoting function has been used, the text remains quoted. Since these "quotes" are hard to see, even in the LOG, this can cause problems for the programmer that does not anticipate that quoting functions may have been used. If you need to remove or change the effects of any of the other quoting functions, the `%UNQUOTE` is used.

Three macro variables are defined below, but the second, &OTH, is defined using the `%NRSTR` function.

This means that &CITY can not be resolved when &OTH is resolved. When the `%UNQUOTE` function is applied to &OTH its value (&CITY) is seen as a macro variable which is also resolved.

**Example 21:**

```sas
%let city = miami;
%let oth = %nrstr(&city);
%let unq = %unquote(&oth);
%put &city &oth &unq;
The LOG shows:
miami &city miami
```

Although &OTH looks like any other macro variable in the %PUT statement, it will not be fully resolved because it is quoted, thus preventing &CITY from being resolved.
VI. USING DATA STEP FUNCTIONS IN THE MACRO LANGUAGE

Two macro tools allow the user to execute virtually all of the functions and routines available in the DATA step as part of the macro language. The %SYSCALL macro statement calls DATA step routines and the %SYSFUNC macro function executes DATA step functions.

SYNTAX

%SYSFUNC(function-name(functionarguments)[,format])
%QSYSFUNC(function-name(functionarguments)[,format])

EXAMPLE 22

The following example shows three ways to add the current date to a TITLE. The automatic macro variable &SYSDATE is easy to use but cannot be formatted.

Prior to Release 6.12 most users created a DATA _NULL_ step with an assignment statement and a CALL SYMPUT to create a formatted macro variable. The DATA step can now be avoided by using the %SYSFUNC macro function.

data _null_;  
today = put(date(),worddate18.);  
call symput('dtnull',today);  
run;  
title1 "Automatic Macro Variable SYSDATE &sysdate";  
title2 "From a DATA _NULL_ &dtnull";  
title3 "Using SYSFUNC %sysfunc(date(),worddate18.)";  

The following three titles are produced:
Automatic Macro Variable SYSDATE 10APR00
From a DATA _NULL_ April 10, 2000
Using SYSFUNC April 10, 2000

The leading spaces before the date in the second two titles is caused by the date string being right justified. The LEFT and TRIM functions can be used to remove the space, however care must be exercised or a couple of problems can be encountered.

The first is that function calls cannot be nested within a %SYSFUNC. Fortunately this is rather easily handled because %SYSFUNC requests can be nested.
Secondly the resolved values of interior calls to %SYSFUNC are used as arguments to the outer calls. When the resolved value contains special characters (especially commas), they can be misinterpreted. The following revised TITLE3 will not work because the interior %SYSFUNC uses a formatted value which contains a comma.

**EXAMPLE 23:**

```sas
title3 "Using SYSFUNC %sysfunc(left(%sysfunc(date(),worddate18.)))";
```

*After the inner %SYSFUNC is executed the result is:*

```sas
title3 "Using SYSFUNC %sysfunc(left(April 10, 2000))";
```

*Because of the comma, the LEFT function will see two arguments (it is expecting exactly one), and the message 'too many arguments' is generated.*

The %QSYSFUNC function can be used to mask special characters in the text string that is passed to the next function. Rewriting the TITLE statement using %QSYSFUNC as is shown below eliminates the problem with the comma.

```sas
title3 "Using SYSFUNC %sysfunc(left(%qsysfunc(date(),worddate18.)))";
```

*TITLE3 from above becomes:*

*Using SYSFUNC April 10, 2000*

## VII. AUTOCALL MACROS THAT MIMIC FUNCTIONS

The AUTOCALL facility allows the user to call macros that have been defined outside of the execution of the current program. A number of these macros are provided with the base macro language and are described in the Macro Language Elements section of the *SAS® Macro Language: Reference, First Edition* reference manual. Although these are, strictly speaking, macros, they act like functions.

Commonly used Autocall macros include:

- `%CMPRES`
- `%LEFT`
- `%LOWCASE`
- `%TRIM`
- `%VERIFY`
%LEFT
This macro can be used to left justify a macro argument.

In the earlier example for %QSYSFUNC the DATA step LEFT function was used, this title can be further simplified by using %LEFT.

**EXAMPLE 25:**
```
title3 "Using SYSFUNC
%left(%qsysfunc(date(),worddate18.))";
```

%VERIFY
While %INDEX and its variations search for specific strings, %VERIFY determines the position of the first character that is **NOT** in a text string. The following example subsets a string starting at the first character that is not a number.

**EXAMPLE 26:**
```
%let code = 2000SUGI25;
%let part =
%substr(&code,%verify(&code,1234567890));
&PART will contain:
SUGI25
```

%CMPRES
The %CMPRES macro removes **multiple** blanks (as well as leading and trailing blanks) from a text string. This macro is similar to the COMPBL DATA step function. In the following example a numeric value is placed into a macro variable using the SYMPUT routine. In the conversion process a series of leading blanks are added to &FIVE.

**EXAMPLE 27:**
```
data _null_; 
  x=5;
  call symput('five',x);
  run;
  %put *%cmpres(&five)*;
%let five = *%cmpres(&five)*;
%put &five;
```

*The resulting LOG shows:* 
138
VIII. FROM DATA STEPS TO A MACRO VARIABLE: CALL SYMPUT

CALL SYMPUT is a SAS language routine that assigns a value produced in a DATA step to a macro variable. It is one of the DATA step interface tools that provides a dynamic link for communication between the SAS language and the macro facility.

Syntax
CALL SYMPUT(argument-1,argument-2);

argument-1: specifies a character expression that identifies the macro variable that is assigned a value. If the macro variable does not exist, the routine creates it.

argument-2: specifies a character expression that contains the value that is assigned.

Call symput takes a value from a data step and assigns it to a macro variable. You can then use this macro variable in later steps.

EXAMPLE 28:

If AGE >=21 THEN CALL SYMPUT (“status”, “adult”);

Else call symput (“status”, “minor”);

These statements create a macro variable named &status and assign it a value of either Adult or Minor depending on the variable age.

EXAMPLE 29:

When performing logistic regression, we often need to create dummy variables based on all possible values of another variable. For instance, we want to create dummy variables for the variable CON which has over 400 different integer values from 1 to 506. Basically we need to do the following:

\[
\begin{align*}
&\text{IF CON} = 1 \text{ THEN CON1} = 1; \text{ ELSE CON1} = 0; \\
&\text{IF CON} = 2 \text{ THEN CON2} = 1; \text{ ELSE CON2} = 0; \\
&\ldots \\
&\text{IF CON} = 506 \text{ THEN CON506} = 1; \text{ ELSE CON506} = 0;
\end{align*}
\]
It is not practical to write this many statements. Our goal is to use the SYMPUT routine to obtain this code automatically.

In the following program, a sample data set TESTDATA with 12 observations and 1 variable is first created in step (1). Then in step (2), a data set UNIQUE is created containing 8 unique CON values. In step (3), the SYMPUT routine assigns the largest value of CON to the macro variable N. CALL SYMPUT is executed once when the DATA step reaches the end of the data set. In step (4), the macro variable N’s value is retrieved and CALL SYMPUT is executed 506 times to create 506 macro variables M1-M506 with the initial value 0. The PUT function is used to eliminate a note that numeric values have been converted to character values. The LEFT function is used to left-align the value of the index variable, I, to avoid creating macro variable names with blanks. In step (5), CALL SYMPUT is executed 8 times and the values of the 8 macro variables created in step (4) are updated with the values of the corresponding CON. The 498 macro variables without the corresponding CON values will remain the initial value 0. Step (6) is a macro that generates all dummy variables for all possible values of CON. By using the %GOTO statement and statement label, the dummy variables without the corresponding CON values will not be created. Note that the double ampersand is necessary to cause the macro processor to scan the text twice first to generate the reference and then to resolve it. Step (7) invokes the macro GETCON to create the dummy variables for every observation in the data set TESTDATA. The last step prints the output data set with dummy variables shown in Table 1.

/* (1) Create a sample data set TESTDATA. */
DATA TESTDATA;
INPUT CON;
CARDS;
1
7
34
115
7
1
487
34
506
57
7
43
; RUN;

/* (2) Get the unique values of CON. */
PROC SORT DATA=TESTDATA OUT=UNIQUE NODUPKEY;
BY CON;
RUN;

/* (3) Assign the largest value of CON to the macro variable N. */
DATA _NULL_;
SET UNIQUE END=LAST;
IF LAST THEN CALL SYMPUT('N', PUT(CON, 3.));
RUN;

/* (4) Assign the initial value 0 to all macro variables. */
DATA _NULL_;
DO I = 1 TO &N;
CALL SYMPUT('M'||LEFT(PUT(I, 3.)), '0');
END;
RUN;

/* (5) Assign the value of CON to the corresponding macro variable. */
DATA _NULL_;
SET UNIQUE;
CALL SYMPUT('M'||LEFT(PUT(CON, 3.)), PUT(CON, 3.));
RUN;

/* (6) Macro to generate dummy variables. */
%MACRO GETCON;
%DO I = 1 %TO &N;
%IF &&M&I = 0 %THEN %GOTO OUT;
IF CON = &&M&I THEN CON&I = 1;
ELSE CON&I = 0;
%OUT: %END;
%MEND GETCON;

/* (7) Create dummy variables. */
DATA TESTDATA;
SET TESTDATA;
%GETCON RUN;

/* (8) Print the result. */
PROC PRINT DATA=TESTDATA;
TITLE 'Table 1. List of CON with dummy variables';
RUN;

Table 1. List of CON with dummy variables
### IX. DEBUGGING OPTIONS

**SYMBOLGEN | NOSYMBOLGEN**: when this option is on, SAS prints in your log the values of macro variables.

**MPRINT | NOMPRINT**: when this option is on, SAS prints in your log the standard SAS code generated by macros.

**MLOGIC | NOMLOGIC**: when this option is on, SAS prints in your log details about the execution of macros.