PhUSE 2017

Paper PP15

# System Macros are Useful - You Can %Bquote Me on That

Melanie O'Neill, PPD, Bellshill, Scotland

## ABSTRACT

In a world where technological advances allow tasks to be carried out at a rapid pace, are there any hidden gems within SAS that could save us more time if we only knew they were there? The word "macro" can make some programmers uneasy, but SAS holds a wealth of automatic system-defined macros that are simple to use as functions and can help reduce both the development time and run time of programs.

This paper gives details on how the macro processor works when executing code and provides examples of SAS quoting macro functions and macro functions that can evaluate expressions, highlighting the situations they can be used in, the options needed and possible restrictions.

## INTRODUCTION

### MACRO FACILITY

The macro language within SAS is a string based language. Together with the macro processor they make up the macro facility.

The macro language is the text used to interact with the processor, which carries out the tasks. There are two delimiters used with the processor - & is used for a macro variable and % is used for a macro statement.

When using macros in SAS, statements and commands are created as needed by the macro facility which are then interpreted and used in the same way as standard code.

### PROGRAM AND MACRO PROCESSING

When a section of code is run in SAS it is sent to the 'input stack' which is an area of memory. From here the program is then changed from characters to tokens. Different tokens are then sent to different areas of SAS to be processed, e.g. the macro processor and the data step compiler. These different areas of SAS work together to execute a program and the interactions between them determine how a macro is processed and also the point in the program that the macro should be called.

### TOKENS

There are four general types of tokens:

- Literal is a string of characters in quotation marks.
- Number are digits and date/time values.
- Name is a string of characters starting with an underscore or letter.
- Special tokens are characters which have a special meaning or function within SAS e.g. = / +

### PROGRAMS WITHOUT MACRO PROCESSOR ACTIVITY

The following code is submitted to the input stack:

```
data ds1;
 set ds;
  trtn=2;
run;
```

The first part of the code, the word data, triggers the data step compiler which continues to read in the tokens from the input stack until it reaches the end of the data step, which is the run statement. The data step instructions are then carried out.

#### PROGRAMS WITH MACRO PROCESSOR ACTIVITY

At the start of each SAS session a symbol table is created that stores the value of automatic and global macro variables.

The following code is submitted to the input stack during the session:

```
%let ds=test;
data trt;
  set &ds.;
   treat="Y";
run;
```

The first part of the code, the %let statement, triggers the macro processor. When the macro processor is triggered by % followed by a character it removes the %let and creates a record in the symbol table where ds=test. No other processing occurs while the macro processor is triggered. The semi-colon after the %let statement ends the activity in the macro processor with the next token, the word data, triggering the data step compiler. The tokens are read in to the data step compiler until the & triggers the macro processor. At this point the macro processor replaces the macro variable with the corresponding text from the symbol table. The remaining tokens are then read into the data step compiler until the end of the data step.

Any macro variables in the symbol table that are created within the program will stay in the symbol table until the SAS session is closed.

#### MACRO QUOTING

Special characters are often required as text which can confuse the macro processor as it does not know whether the special character should be seen as text or as a symbol from the macro language.

There are various system macros within SAS that mask the meaning the special character has within the macro language and allows the macro processor to see the character as text. A selection of these system macros will be examined in further detail within this paper.

### %BQUOTE

%BQUOTE is an execution function which instructs the macro processor to view special characters created by resolving macro variables as text instead of viewing them as part of the macro language. Unlike compilation functions such as %STR where the macro processor views special characters as text whilst constructing the macro, execution functions like %BQUOTE resolve when the macro is executed.

When the macro processor encounters the %BQUOTE function it resolves the macro expression as far as it can and masks the result. If there are references to macro variables that cannot be resolved then a message is issued to the log.

%BQUOTE can be used to mask a variety of special characters such as parentheses, quotation marks, mathematical operators and semi-colons. The advantage %BQUOTE has over other functions such as %STR is that unmatched quotation marks, parentheses and percent signs do not have to be used in conjunction with %.

Any quotation marks or parentheses are viewed by %BQUOTE as special characters which are then masked when the macro is executed; this allows each quotation mark or parenthesis to be masked independently of others.

The following code uses %BQUOTE to deal with the single quotation mark found in the text string “Investigator’s Decision”.

```
data reason;
 text="Investigator's Decision";
  call symput('reason', text);
run;

%macro valid_reason;
  %if %bquote(&reason) ne %then %put reason is valid;
  %else %put reason is not valid;
%mend valid_reason;
%valid_reason;

1      data reason;
2       text="Investigator's Decision";
3        call symput('reason', text);
4      run;

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

```
5      %macro valid_reason;
6       %if %bquote(&reason) ne %then %put reason is valid;
7       %else %put reason is not valid;
8      %mend valid_reason;
9
10     %valid_reason;
reason is valid
```

The log shows the reason is seen as valid when %BQUOTE is used. If %BQUOTE is omitted from the macro using the code below then a different result can be seen.

```
%macro valid_reason;
 %if &reason ne %then %put reason is valid;
 %else %put reason is not valid;
%mend valid_reason;

%valid_reason;
```

```
11     %macro valid_reason;
12      %if &reason ne %then %put reason is valid;
13      %else %put reason is not valid;
14     %mend valid_reason;
15
16     %valid_reason;
ERROR: A character operand was found in the %EVAL function or %IF condition where a
numeric operand is required. The condition
       was: &reason ne
ERROR: The macro VALID_REASON will stop executing.
```

The macro valid_reason fails to run this time because the single quotation mark has caused an error within the %EVAL function.

There are some restrictions to the %BQUOTE function for example a single & or a large number can cause the function to error in certain circumstances.

```
data character;
 text="&";
  call symput('char', text);
run;

%macro valid_text;
 %if %bquote(&char) ne %then %put text is valid;
 %else %put text is not valid;
%mend valid_text;

%valid_text;
```

The code above is similar to the valid reason where %BQUOTE effectively handled the single quotation mark present in “Investigator’s Decision”. However when a single & is used the log output below shows that using %BQUOTE results in the same error message as not using it.

```
1      data character;
2       text="&";
3        call symput('char', text);
4      run;

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


5      %macro valid_text;
6       %if %bquote(&char) ne %then %put text is valid;
7       %else %put text is not valid;
8      %mend valid_text;
9
10     %valid_text;
ERROR: A character operand was found in the %EVAL function or %IF condition where a
numeric operand is required. The condition
       was: %bquote(&char) ne
ERROR: The macro VALID_TEXT will stop executing.
```

However in the case of a single & the %SUPERQ function can be used in place of %BQUOTE. One thing to note from the code below, when invoking the %SUPERQ function the ampersand before the macro variable is not required.

```
     data character;
      text="&";
       call symput('char', text);
      run;

      %macro valid_text;
       %if %superq(char) ne %then %put text is valid;
       %else %put text is not valid;
      %mend valid_text;

      %valid_text;
```

```
36     data character;
37      text="&";
38       call symput('char', text);
39     run;

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


40
41     %macro valid_text;
42      %if %superq(char) ne %then %put text is valid;
43      %else %put text is not valid;
44     %mend valid_text;
45
46     %valid_text;
text is valid
```

The log above shows that the %SUPERQ function has effectively handled the single &.

### %NRBQUOTE

%NRBQUOTE is an execution function similar to %BQUOTE. The same special characters masked by %BQUOTE can also be masked with %NRBQUOTE. %NRBQUOTE differs from other quoting functions due to the fact that macro variables will be resolved where possible but any ampersands or percent signs in the result are not viewed as operators when using other functions such as %EVAL or %PUT.

```
data _null_;
 call symput( 'trt' , 'A&B' ) ;
run ;

%let C = %NRBQUOTE(&trt);
%put C = &C ;

1      data _null_;
2       call symput( 'trt' , 'A&B' ) ;
3      run ;

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


4      %let C = %NRBQUOTE(&trt);
WARNING: Apparent symbolic reference B not resolved.
5      %put C = &C ;
C = A&B
```

The issue with using the %NRBQUOTE is the log message with the warning that the symbolic reference was not resolved. This warning can be eliminated by using the %SUPERQ function.

```
data _null_;
 call symput( 'trt' , 'A&B' ) ;
run ;

%let C = %superq(trt);
%put C = &C ;

6      data _null_;
7       call symput( 'trt' , 'A&B' ) ;
8      run ;

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


9      %let C = %superq(trt);
10     %put C = &C ;
C = A&B
```

### %QSUBSTR

There are various quoting text functions available in SAS that can be used. For example %QUPCASE, %QCMPRES and %QSUBSTR which are useful when trying to manipulate text strings that contain special characters.

%QSUBSTR is used when trying to substring a variable that contains an ampersand or some other special character. The %QSUBSTR function can mask the same special characters as discussed for the %NRBQUOTE function.

```
%let a=AE;
 %let b=MH;
  %let c=%nrstr(&a &b);
   %put C: &c;
    %put With SUBSTR: %substr(&c,1,2);
     %put With QSUBSTR: %qsubstr(&c,1,2);

35     %let a=AE;
36     %let b=MH;
37     %let c=%nrstr(&a &b);
38     %put C: &c;
C: &a &b
39     %put With SUBSTR: %substr(&c,1,2);
With SUBSTR: AE
40     %put With QSUBSTR: %qsubstr(&c,1,2);
With QSUBSTR: &a
```

The use of %NRSTR in the code above causes C not to be resolved. %SUBSTR gives AE as the result because it does not mask the special characters found in C. However %QSUBSTR will mask the special character and so won't resolve &a to AE but will quote the first two characters found in C.

### %QCMPRES

%QCMPRES is another example of a quoting text function. This function is used to remove multiple blank spaces or specific characters from an expression just like the COMPRESS function does in standard code. %QSUBSTR masks the same special characters as %NRBQUOTE so does %QCMPRES.

```
%let dose=5;
 %let freq=3;
  %let total=%nrstr(%eval(&dose   *   &freq));
   %put QCMPRES: %qcmpres(&total);
    %put CMPRES: %cmpres(&total);

17     %let dose=5;
18     %let freq=3;
19     %let total=%nrstr(%eval(&dose   *   &freq));
20     %put QCMPRES: %qcmpres(&total);
QCMPRES: %eval(&dose * &freq)
21     %put CMPRES: %cmpres(&total);
CMPRES: 15
```

The example above is similar to the %QSUBSTR example. In this case %CMPRES resolves the %EVAL to multiply the dose by the frequency to give the total, however %QCMPRES masks the special characters and simply removes the additional spaces from the %EVAL statement.

**%SYSEVALF**

The %EVAL function is used to evaluate both arithmetic and logical expressions. To do this the arguments of the function are converted to numeric or logical expressions from the character values provided. Once the expression has been evaluated the result is then converted back to a character value.

%EVAL can be only used for integer arithmetic expressions. If a value containing '.' is entered into the function an error is issued saying a character operand has been found where a numeric operand is required. To solve this problem the %SYSEVALF function can be used. This works in the same way as the %EVAL function but uses floating point arithmetic and will return a value that has been formatted using BEST32. format.

The value returned by the %SYSEVALF function can be specified using the conversion type option within the function. There are four different conversion types that can be specified:

- BOOLEAN – will return 0 if the result is 0 or missing and will return 1 for any other result.
- CEIL – will return the character version of the smallest integer that is greater than or equal to the result.
- FLOOR – will return the character version of the largest integer that is less than or equal to the result.
- INTEGER – will return the character version of the integer portion of the result, so decimals are not presented.

```
%macro calculate(a,b);
   %let y=%sysevalf(&a+&b);
    %put The result with SYSEVALF is: &y;
     %put  The BOOLEAN value is: %sysevalf(&a +&b, boolean);
      %put  The CEIL value is: %sysevalf(&a +&b, ceil);
       %put  The FLOOR value is: %sysevalf(&a +&b, floor);
        %put  The INTEGER value is: %sysevalf(&a +&b, int);
%mend calculate;

%calculate(20,2.534);

1      %macro calculate(a,b);
2         %let y=%sysevalf(&a+&b);
3         %put The result with SYSEVALF is: &y;
4         %put  The BOOLEAN value is: %sysevalf(&a +&b, boolean);
5         %put  The CEIL value is: %sysevalf(&a +&b, ceil);
6         %put  The FLOOR value is: %sysevalf(&a +&b, floor);
7         %put  The INTEGER value is: %sysevalf(&a +&b, int);
8      %mend calculate;
9
10     %calculate(20,2.534);
The result with SYSEVALF is: 22.534
The BOOLEAN value is: 1
The CEIL value is: 23
The FLOOR value is: 22
The INTEGER value is: 22
```

## CONCLUSION

Using the system defined macros discussed in this paper can help to resolve issues found within text that contains special characters such as unmatched quotation marks or percent signs. Therefore it is possible to create user defined macros to automate and easily replicate analysis.

When implementing macros there are various different errors that can occur at each stage of the macro process. Some common examples are syntax errors during macro compilation or referring to a macro that has not yet been defined.

Since a program using macros could quickly become complex it is a good idea to develop and test the program in sections and once these work as expected and are error-free then combine the individual parts to create the complete program. Before testing macros it is also a good idea to thoroughly proofread the code as a missed semi-colon or typo can cause problems when submitting the macro.

The list below gives some things to check when creating a user defined macro:

• Each %macro statement has a %mend statement to inform SAS that the macro is complete. If the %mend statement is missed further data steps or open code will not be processed but will be interpreted as part of the macro and will appear in the log as text. It is good practice to use the name of the macro in the %mend statement as this allows an easier and quicker way to check that each macro has a corresponding %mend statement and can make the log clearer and easier to debug.

• If %DO statements are used then there should be the same number of %DO and %END statements. If the %DO statements are iterative then there should be %TO values specified. Clear indentation and alignment of nested %DO and %END results in easier identification of issues.

• Macro variables should start with & and macro statements should begin with %. If call symput is used to create macro variables within a data step then those macro variables cannot be referenced within the data step that creates them.

• If any special characters are present in the data being used for macro variables, such as unmatched quotation marks or mathematical operators, then quoting macros, as discussed in this paper, should be used.

## CONTACT INFORMATION

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

Melanie O'Neill
PPD
Fleming House
Strathclyde Business Park
Bellshill, ML4 3NJ
Work Phone: +44 (0) 1698 575097
Email: Melanie.o'neill@ppdi.com
Web: www.ppdi.com