**SAS MACRO NOTES (Ou Zhang)**

**Paper 1: Understanding double Ampersand [&&] SAS macro variables**

* A macro variable name is bracketed with ampersand [**&**] before and period [**.**] after it. While the period may be optional, it is a good practice to include it until you are able to explain why you can omit it!

&variable.

* Notice that we do not use quotation marks around the character string. It may contain blanks. The maximum length of a **macro** variable is up to 32K and may be system dependent. The value starts with the first non-blank character after the equal sign and ends with the last non-blank character before the semicolon.
* The value of any macro variable will be a character string.

**-----------------------------------------------------------------------------------**

**Paper 2:** **The Ampersand (&) Challenge, Single, Double or more?**

%let x=a;

%let a=b;

%let b=c;

%put &x; \* [&]x --> a;

%put &&x ; \* [&&]x --> [&]x --> a;

%put &&&x; \* [&&][&]x --> [&][a] --> b;

%put &&&&x; \* [&&][&&]x --> [&][&]x --> [&]x --> a;

%put &&&&&x; \* [&&][&&][&x] --> [&][&][a] --> [&][a] --> b;

%put &&&&&&x; \* [&&][&&][&&]x --> [&][&][&]x --> [&&]a --> b;

%put &&&&&&&x;\*[&&][&&][&&][&]x --> [&][&][&][a] --> [&]b --> c;

**RULES OF THE GAME**

To deal with any number of ampersands associated with a macro variable, the three basic rules that should be understood and these are:

1. The macro processor reads a macro variable from left to right (i.e. forward scanning).
2. For multiple ampersands, the re-scan rule takes effect until no more ampersands are left to resolve i.e the macro variable resolves its final value OR error messages have been issued
3. ‘Two’ ampersands result into ‘One’ ampersand (&&  &) during the macro resolution process

**Example 2:**

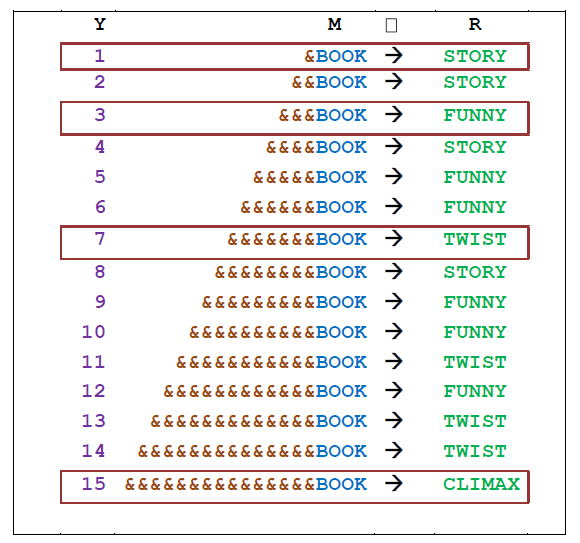
options symbolgen;

%LET BOOK = STORY; \*\*(1)\*\*;

%LET STORY = FUNNY; \*\*(2)\*\*;

%LET FUNNY = TWIST; \*\*(3)\*\*;

%LET TWIST = CLIMAX; \*\*(4)\*\*;



**Y: Number of ampersands pre-fixed to the macro variable**

**M: Macro variable to be resolved**

**R: Final resolved value**

A non-linear equation can be drawn between them as follows (The equation is specific to this example discussed).



In mathematical terms,

- ‘n’ is the number of macro variables that are defined

- ‘y’ is the number of ampersands that are required until the macro variable successfully resolves to its final value

**REVIEW OF MULTIPLE AMPERSANDS &&&**

Multiple ampersands can be used to allow the value of a macro variable to become another macro variable reference. The macro variable reference will be rescanned until the macro variable is resolved.

The following demonstrates how macro variables with multiple ampersands are resolved.

Symbol Table

|  |  |
| --- | --- |
| Macro Variable Name | Macro Variable Value |
| A | FREIGHT |
| B | PASSENGER |
| C | SPECIAL |
| CODE | A |
|  | |

**Paper 3: Resolving and Using &&var&i Macro Variables**

TECHNICAL TIP:

* Professional SAS programmers always refer to the ampersand as “**amper**” when reading code, and only novice programmers pronounce the full word.
* Macro variables are not resolved within single quote ( **‘ ’** ) and they are resolved within double quotes( **“ ”** ).

**MACRO VARIABLE RESOLUTION**

* A macro variable may be appended to SAS code including variables and data set names.
* A period following the macro variable is assumed to be a concatenation operator for that macro variable.
* Period before a macro variable have no special significance.
* Double quotes are use when a macro variable is to be resolved inside of a quoted string.

Form of **&&VAR&I** where VAR is the root portion of the variable name and the index number is an integer counter.

Example:

SCREEN=GLSCN.descn.p**&&dsn&q...**SCREEN;

**&&dsn&q...**

Notice that **three periods** are required in the SCREEN= option because two passes are needed to resolve the &&DSN&Q.

* The first pass compresses the three periods to two and
* The second pass compress the two periods to one.

**&&dsn&q... 🡪 [&&dsn[&q.].].**

**Paper 4: Advanced Macro Topics**

* It should be noted that global statements such as title, footnote, options, libname etc. are compiled, and then executed immediately.

**A Timing Error**

Suppose we want to retrieve a macro variable value that was created in the same step. This might be because the user is interacting with the step, or perhaps we are using a stored compiled step or an SCL program.

|  |  |
| --- | --- |
| Wrong | Correct |
| data saleexps;  input name $1-10 division $12 ;  if division=’H’ then  **%let mdiv=Hardware;**  if division=’S’ then  **%let mdiv=Software;**  datalines;  Steve H  Bob H  ;  run; | data saleexps;  input name $1-10 division $12 ;  if division=’H’ then  **call symput(‘mdiv’, ‘Hardware’);**  if division=’S’ then  **call symput(‘mdiv’, ‘Software’);**    newvar=”&mdiv”;  datalines;  Steve H  Bob H  ;  run; |

**Another Timing Error (symget)**

Again the reference to &mdiv would be attempted before it was created by SYMPUT, even though it appears later in the program. The run time retrieval routine is typically SYMGET which works fine.

|  |  |
| --- | --- |
| Wrong | Correct |
| data saleexps;  input name $1-10 division $12 ;  if division=’H’ then  call symput(‘mdiv’, ‘Hardware’);  if division=’S’ then  call symput(‘mdiv’, ‘Software’);  **newvar=”&mdiv”;** \*execute m-v before it is been compiled;  datalines;  Steve H  Bob H  ;  run; | data saleexps;  input name $1-10 division $12 ;  if division=’H’ then  call symput(‘mdiv’, ‘Hardware’);  if division=’S’ then  call symput(‘mdiv’, ‘Software’);  \* Use symget to apply m-v;  **newvar=symget(‘mdiv’);**  datalines;  Steve H  Bob H  ;  run; |

**Debugging and Tracing Techniques**

There are essentially five debugging tools in the macro facility.

* %PUT

The %PUT statement is probably the best and simplest method of displaying values at word scan time. %PUT can display text or macro variable references and calls.

A fairly recent addition is parameters to %PUT that not only display a single variable, but can show several.

*\_*ALL\_ shows all variables and there respective symbol table

*\_*AUTOMATIC\_ shows only the system variables

*\_*USER*\_* displays only user variables

*\_*LOCAL\_ displays only local variables

*\_*GLOBAL\_ displays only global variables

* SYMBOLGEN

SYMBOLGEN shows macro variables as they are being resolved.

* MLOGIC

MLOGIC displays decisions and looping that the system makes with %DO, %IF etc.

* MPRINT

MPRINT displays the generated code sent to the SAS compiler.

* MFILE

MFILE routes the MPRINT results to a separate file.

**Paper 5: SAS author's tip: macro functions %EVAL and %SYSEVALF**

* The %EVAL function evaluates expressions using integer arithmetic.
* The %SYSEVALF function evaluates expressions using floating point arithmetic.

With the exception of %sysevalf function, integer arithmetic is the only way macro statements perform arithmetic calculations.

Following are the few examples of macro statements performing integer arithmetic calculations:  
%let one=%eval (3+5);  
%let two=%eval (5\*2);  
%let three=%eval (9/3);  
%let four=%eval (5/2);  
%put The value of one is &one;  
%put The value of two is &two;  
%put The value of three is &three;  
%put The value of four is & four;

Open the Log file and see the results as follows:  
The value of one is 8  
The value of two is 10  
The value of three is 3  
The value of four is 2

The value for macro variable four, should be 2.5, instead it shows only two. That happens because if we perform division on integers, integer arithmetic doesn’t take the fractional part into account.  
When we try to execute the integer arithmetic calculations of values with functional part, :  
  
%let last= %eval (5.0+3.0); /\*INCORRECT\*/  
  
**%EVAL**function only supports integer arithmetic values. The values here in the above statement have a period character to numeric values and because of that the macro processor stops evaluating and produces the following error message: “ ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric operand is required. The condition was: 5.0+3.0 “

The **%SYSEVALF** function can perform arithmetic calculations with operands that have the floating point values.

Here are some of the examples where **%SYSEVALF** function becomes handy:

%let test= %sysevalf(1.0\*3.0);  
%let final= %sysevalf(1.5+2.8);  
%let last= %sysevalf(5/3);  
%put The value of test is &test;  
%put The value of final is &final;  
%put The value of last is &last;

The %PUT statements display the following messages in the log:

The value of test is 3  
The value of final is 4.3  
The value of last is 1.66666666666666

**%SYSEVALF** function perform arithmetic calculations and the result of the evaluation can be a floating point value like in the final and last macro variable case, but as in integer arithmetic calculations, the result is always a text.

The **%SYSEVALF** function be used in conjugation with other functions like, **INTEGER**, **CEIL**, and **FLOOR**.  
For example, the following %PUT statements return 3, 4 and 3 respectively:  
%let val=3.8;  
%put %sysevalf(&val,integer); \*Value returns in the log is 3;  
%put %sysevalf(&val, ceil); \*Value returns in the log is 4;  
%put %sysevalf(&val,floor); \*Value returns in the log is 3;

Difference between **%eval** and **%sysevalf functions** can be understand better with the following example.

%let value=9;  
%let value2=5;  
%let newval=%sysevalf(&value/&value2);  
%let newval1=%eval(&value/&value2);

%put &newval;

%put &newval1;

\*Ans: newval=1.8;  
\*Ans: newval1=1;

**Paper 6: Take an In-Depth Look at the %EVAL Function**

* When %EVAL encounters an arithmetic operation containing non-integer values or characters, it displays an error message about finding a character operand where a numeric operand is required.
* In macro expressions, a period (.) can be used as a separator for macro variables. Operands that contain a period character cause an error when they are part of an integer arithmetic expression. However, if a period appears in a resolved logical expression, it will be treated as a regular character.
* Unlike quotes in the DATA step where encompassing quotes (“ ”) are not part of the value, quotes in macro expressions are part of the value and are treated just like any other regular characters. For example:

%put %eval("0" = '0');

%put %eval("0" = 0);

%put %eval(" " = );

* Specify a conversion type for the %SYSEVALF function. If you use the %SYSEVALF function in macro expressions or assign its results to macro variables that are used in other macro expressions, then errors or unexpected results may occur if the %SYSEVALF function returns missing or floating point values.

To prevent errors, specify a conversion type that returns a value compatible with other macro expressions.

**Paper 7: Check variable existence in a dataset**

1. Macro-based approach via SYSFUNC VARNUM:

%*\* Open the dataset \*;*

%let dsid = %sysfunc(open(work.dataset));

%if %sysfunc(varnum(**&dsid**, x)) > **0** %then %put NOTE: Variable x exists!;

%*\* Close the dataset (important!);*

%let rc = %sysfunc(close(**&dsid**));

This is particularly useful inside of a macro that accepts a variable or list of variables in a dataset as a parameter.

1. Dictionary tables in PROC SQL:

**proc sql**;

select name

from dictionary.columns

where libname = 'WORK' and

memtype = 'DATA' and

memname = 'SOMEDATASET' and

upcase(name) = 'X';

**quit**;

If the result set from the query is empty then the variable does not exist in the dataset. The variables of interest in the COLUMNS dictionary table are:

* LIBNAME, the libref name in all caps
* MEMTYPE, the the member type, i.e. data or view, of the object in all caps
* MEMNAME, the name of the dataset
* NAME, the name of the variable

1. SASHELP.VCOLUMN:

**data** \_null\_;

set sashelp.vcolumn;

where libname = 'WORK' and memtype = 'DATA' and memname = 'SOMEDATASET';

if upcase(name) = 'X' then put 'NOTE: Variable x exists!';

**run**;

The VCOLUMN view in the SASHELP library is akin to the COLUMNS SQL dictionary table, however the SQL dictionary table will generally be faster as the SASHELP views are generated using the SQL dictionary tables.

1. USING A DROP STATEMENT IN A DATA STEP:

One method that wasn't suggested is using a drop statement in a data step. It isn't case sensitive and will produce warnings in the log for any variable that isn't in the dataset. In the following example, gender and score are not on the data set SASHELP.CLASS:

**data** test (drop=SEX gender Age height weight score);

if **0** then set sashelp.class;

stop;

**run**;

1. Check if a variable exists in a dataset

%macro VarExist(ds, var);

%local rc dsid result;

%let dsid = %sysfunc(open(**&ds**));

%if %sysfunc(varnum(**&dsid**, &var)) > **0** %then %do;

%let result = **1**;

%put NOTE: Var &var exists in **&ds**;

%end;

%else %do;

%let result = **0**;

%put NOTE: Var &var not exists in **&ds**;

%end;

%let rc = %sysfunc(close(**&dsid**));

**&result**

%mend VarExist;

**Paper 8: Guidelines on Writing SAS® Macros for Public Use**