Demystifying the SAS® Macro Facility - by Example

Harry Droogendyk, Stratia Consulting Inc.
Marje Fecht, Prowerk Consulting Ltd

ABSTRACT

The SAS macro facility enables you to apply a wealth of useful, uncomplicated, real-world solutions to enhance your coding pleasure, reduce coding effort, and minimize error. As your business applications inevitably become more complex, the SAS macro facility is indispensable to:

- reduce code repetition
- increase control over program execution
- minimize manual intervention
- create modular code.

Unfortunately, the SAS macro facility is often perceived as confusing, difficult to use, and only really comprehensible by a few long-time macro masochists. This presentation removes the mystery of macros and provides coding tips, macro solutions, and methodologies you can take away and implement immediately.

INTRODUCTION

Since the SAS Macro facility can be perceived as *mysterious* until you understand the inner workings, we will begin by exploring "common gotchas" to help you avoid some traps. Then, we will develop macro coding examples to explore

- macro variables
- macro functions
- macro definition
- macro programming statements.

The majority of this paper is equally applicable to both SAS 8 and SAS 9. If something is only available in SAS 9, it is noted.

COMMON *GOTCHAS*

GOTCHA # 1

Do you repeatedly copy and paste steps that are quite repetitious? Consider this program:

```
title "Sales 200511";
proc print data = sales_200511 noobs;
run;
title "Sales 200512";
proc print data = sales_200512 noobs;
run;
title "Sales 200601";
proc print data = sales_200601 noobs;
run;
title "Sales 200602";
proc print data = sales_200602 noobs;
run;
```

While this example may not seem dreadfully repetitious or high-maintenance, it could become tedious to work with as it "grows". Later examples in this paper will demonstrate easy and effective ways to generate this code with macro.

GOTCHA # 2

Timing is the key to using macro functionality. If you misunderstand the timing, you will get unexpected results! In addition to the timing issues, another typical macro processing misunderstanding revolves around "Who does what?"

When you submit the following program, why isn't the title reflective of the TxnSize values in the output data set?

```
data gotcha;
    set DailyTxns;
    if txnSize>50000 then do;
    %let alert = NOTE: BIG Transactions;
  end;
run;
proc means data=gotcha mean min max;
  var TxnSize;
  title "Average Txns  &alert";
run;
```

```
Average Txns  NOTE: BIG Transactions
Mean          Minimum         Maximum
-------------------------------------
23941.62      20000.00        49685.48
-------------------------------------
```

Since this program ran without error, but the results were *not* as we expected, there's obviously a problem, and the problem is both **TIMING** and **"Who Does What?"!**

When you first began to use SAS, you likely learned that steps are compiled, prior to executing. For the DATA step in this example, the ***SAS word scanner*** works through the entire DATA step (up to and including the run;) prior to executing the DATA step. When macro triggers are encountered, such as % and & followed immediately by a "word", the ***macro processor*** looks at the token and the adjoining "word", to resolve or carry out the specified instruction.

Consider the processing of the example above. When the DATA step compilation begins, the first three statements are scanned and placed in the ***execution stack***, awaiting the end of the step. Next, the word scanner encounters

```
%let alert = NOTE: BIG Transactions;
```

When the word scanner passes the tokens **%** followed by the adjoining word **let** to the macro processor, the macro processor requests all text through to the semicolon. The macro processor evaluates the **%let** statement, creates a macro variable named `alert`, and stores `NOTE: BIG Transactions` as its value in the ***macro symbol table***. NOTE that the %let statement is NOT placed back in the execution stack. Instead, the macro facility carried out this instruction prior to the end of DATA step compilation.

Continuing, the word scanner passes the next two statements to the execution stack. At the completion of DATA step compilation, the **execution stack contains**:

```
data gotcha;
  set DailyTxns;
  if txnSize>50000 then do;
  end;
run;
```

The data values have NO IMPACT on the value of the macro variable ALERT. Now we know why the title is wrong!

Following the execution of the DATA step, the word scanner continues parsing the PROC MEANS step. When the word scanner encounters the next macro trigger in the TITLE statement, it passes `&alert` to the macro processor. An ampersand followed by a name is a request for a macro variable resolution, so the macro processor grabs the current value of the macro variable in the symbol table, and passes it back to the word scanner. This value is now placed in the execution stack followed by the remainder of the PROC MEANS step.

How should we have solved this problem? The **SYMPUT** routine is useful to create macro variables when the values are data dependent.

MACRO VARIABLES

As demonstrated in our first two Gotchas, macro variables provide a way to substitute **text** into a SAS program at compile time. That **text** can encompass anything from a single character to an entire program. Macro variables enable you to easily maintain your production programs since you can change a value once where it is assigned and profit from substitution throughout your program, wherever a macro variable reference appears. Often, you will populate macro variables from data values, which lets you create *table driven* programs that adapt or react to your data.

There are two types of macro variables.

- Automatic Macro Variables are created by SAS and typically exist for the duration of your session (job).
  - Sysdate9 contains the date that your SAS session (program) began. Ex: 27Mar2006
  - SYSSCP identifies the Processor (operating system). Eg: WIN
  - For a complete list of automatic macro variables, see SAS Online Documentation.
- User defined macro variables are created within your program.
  - `%let mode = PROD;`
  - `Call symputx('Nobs' , N);  ** SYMPUTX is a SAS 9 function;`

To view the contents of a macro variable, you can use a %put statement:

```
%put Mode is &mode;

%put _user_;       ** to view the contents of all user defined macro variables;

%put _automatic_;  ** to view the contents of all automatic macro variables;
```

NOT JUST A "WORD"

Macro variables can contain any text. The maximum length of a macro variable is SAS version dependent.

```
%let code = %str(data a; i = 7; run;);
options symbolgen;
&code;
```

```
SYMBOLGEN:  Macro variable CODE resolves to
 data a; i = 7; run;
NOTE: The data set WORK.A has 1 observations and 1
variables.
```

What would the value of the macro variable CODE be if we excluded the quoting function, %str, and used the following?

```
%let code = data a; i = 7; run;
```

&Code would result in the value

```
data a
```

Since a semicolon is used to end a %let statement, a %str function is used to *hide the literal meaning of special characters* such as the semicolon.

USING MACRO VARIABLES

CUSTOM DATE VALUES

Typically, the first ***real use*** of macro variables for most people revolves around date values. Most programs require subsetting or custom text that is date dependent. Macro variables are extremely useful for including date values in file names, titles, footnotes, and subset criteria.

The DATA step with the symput routine is the classic approach (prior to SAS 6.12) to create and store custom date values in a macro variable. The following step creates a macro variable named `date` containing today's date.

```
data _null_;
    call symput('date',put(today(),yymmddN8.));
run;
title "Report As of &date";
```

```
====== title ========
Report As of 20060327
```

You can avoid the unnecessary DATA step by using the macro function %sysfunc.

```
%let date = %sysfunc(today(),yymmddN8.);
title "Report As of &date";

%let log = CpmgnA_&date..log;
```

If you do not need an intermediate macro variable, use %sysfunc directly in your code.

```
title "As of %sysfunc(today(),yymmddN8.)";
%let num = %sysfunc(today(),8.);    ** Beware of spaces ! **;
%put **&num**;    **   16887**
```

Each of the examples above shows the macro variable reference within a double-quoted string. That's **very** important!! If the macro variable (or %sysfunc function) were enclosed in single-quotes it would be *hidden* from the macro processor and it would not resolve. `title 'Report As of &Date';` would display the contents of the single quotes, **`Report As of &Date`**, rather than substituting the macro variable value.

The setting of **&log** brings up an important point. As we've already discovered, the macro processor is invoked when a macro trigger immediately followed by a "word" is encountered. Where there is any ambiguity as to where the end of the "word" lies, the macro variable reference **must** be followed by a period to end the "word". If the macro variable **&date** was being used to define a report name, a specification of &date_report.xls wouldn't achieve the desired result. The macro processor would search the symbol table for a macro variable named **date_report** rather than **date**. To force the end of the word, a period must be specified: **&date._report**.xls.

That's helpful, but isn't this still a different situation than the **&log** example above? Why were two periods specified after **&date**? The first period ends the **&date** variable, the second is the period within the filename, separating the name from the **log** file type suffix.

CONTROLLING EXECUTION

Macro variables are useful for simple execution control within your program. For example, suppose that when you are in development mode, you only use a 1% sample of your data and you read / write to a development location. When you move to production, you read all of the data and you work with production locations.

```
** To enable sampling in TEST, set sample to null or %str();
** To avoid  sampling in PROD, set sample to an asterisk to comment out line;

%let mode   = PROD;
%let sample = *;
%let yyyymm = 200510;

libname cc "H:\sugi31\&mode\&yyyymm\Cmpgn";

data cc;
  set cc.campaigns;
  &sample if ranuni(1) > .99;
  where LOB =  '610';
run;
```

CREATING AN "IN" LIST USING MACRO VARIABLES

Dynamic data-driven IN lists result in less program maintenance, since the program will easily adapt to new information. The DATA step can be used to construct the list, and store it in a macro variable, but utilizing SQL requires less effort. The SQL **INTO:** stores values into a macro variable, with separators as specified. In this example, the values will be quoted, separated by commas and stored in **reglist**.

```
proc sql noprint;
select distinct quote(trim(region)) into: reglist separated by ","
  from sashelp.shoes
 where returns > 10000;
```

```
   quit;
   %put &reglist;
```

====== LOG ========
"Africa","Canada","Central America/Caribbean","Middle East","Pacific","United States","Western Europe"

To use the list, specify

```
   where region in ( &reglist )
```

If you work with a database system that requires an IN list with single quotes instead of double quotes, you can use a translate function outside of the quote function to make the change.

Notice in this example we assumed that the variable feeding the IN list is character. Wouldn't it be nice to setup the SQL to *adapt* to the data type? And, wouldn't it really be nice if you could just call up this code segment whenever you need an IN list, without having to copy it to all of your programs?

The next section will demonstrate that this example lends itself nicely to a generalized solution that enables you to supply the table name, column name, data type, and macro variable name as part of a ***macro invocation***.

MACRO LANGUAGE

We've already discovered that SAS macro variables are really not that mysterious. They simply provide a means of substituting **text** into our SAS program during compilation. In fact, SAS macro variables can contain entire SAS statements or even an entire program.

The function of the SAS macro language isn't that much different than SAS macro variables - SAS macros simply *generate* text. In addition to allowing simple text substitution, SAS macros provide looping and decisioning functionality to control text generation. Before *generalizing* the IN list example using macros, let's try a few simple macro examples to make this clearer.

DEFINITION OF A MACRO

Each macro must begin with a **%macro** *macro_name;* statement and end with a **%mend;** statement. Virtually anything else may be coded between the two statements. The macro is invoked by specifying the macro name prefaced with a percent sign.

```
%macro sug_macro;
    %put SAS rocks!!!;
%mend sug_macro;

%sug_macro;
```

This macro is very simple. The **%put** statement results in the text following the **%put** to be written to the log:

```
SAS rocks!!!
```

WHY USE A MACRO?

Macros are most often used to generate SAS code (which is really just text). In real world situations which call for a macro solution, the generation of the SAS code may demand conditional processing or loops to generate the required code. While macro **variables** may be used anywhere, **%if** /**%then**/**%else** and **%do** macro **statements** may *only* be invoked within a macro. Other macro statements such as **%global**, **%let**, **%put**, etc.. may be specified in open code. (Note that not all commands beginning with a percent sign are macro statements, eg. **%include**).

In addition, macros allow for the writing of modular code that can be used like functions. Sections of utility code, even short snippets, may be written in macro form and made available in a common environment and invoked in multiple programs by different users. Utility code of this nature is most useful when the macro is invoked with parameters to define/customize the macro results, eg. specification of an input data set.

MACRO EXAMPLES

The “IN list” example we’ve been considering could be “macroized” and made available to the entire user community. That way everyone could benefit from this bit of utility code without needing to write it from scratch each time it is required.

```
%macro shoes_inlist;
    %global reglist;
    proc sql noprint;
           select distinct quote(trim(region))
             into :reglist separated by ','
             from sashelp.shoes
    quit;
%mend shoes_inlist;

%shoes_inlist;
```

While *some* benefit may be realized by including this piece of code in a macro, it’s really only useful when a user requires a list of regions from the sashelp.shoes dataset. Rather than writing a macro specific to one variable within a specific dataset, a really useful macro solution would allow the specification of parameters to produce a “customized” IN list.

```
%macro inlist(ds,fld,mvar);
    %global &mvar;
    proc sql noprint;
           select distinct quote(trim(&fld))
             into :&mvar separated by ', '
             from &ds;
    quit;
%mend inlist;

%inlist(sashelp.shoes,region,reglist);
```

The macro has been renamed to **inlist** and three **positional** parameters have been defined to allow the specification of the input dataset, the field within the input dataset and the name of the macro variable to be created. Each macro parameter value is available to the macro as a macro variable.

We now have a useful macro! We can build and store an IN list into any macro variable for any character field in any dataset accessible to us.

For a couple of reasons, positional parameters should **not** be used where more than one macro parameter may be specified. Positional parameters must be defined in the correct order when the macro is invoked or strange results will occur. In addition, the specification of positional parameters does not afford any “self-documentation” in the invoking program. When writing macros, get into the habit of using **keyword** parameters as demonstrated in the following example:

```
%macro inlist(ds = ,fld = ,mvar = inlist);
    %global &mvar;
    proc sql noprint;
           select distinct quote(trim(&fld))
             into :&mvar separated by ', '
             from &ds;
    quit;
%mend inlist;

%inlist(ds = sashelp.shoes, mvar = reglist, fld = region);
%put &reglist;
```

In this version of the **inlist**, three **keyword** parameters have been defined to allow the specification of the input dataset, the field within that dataset and the name of the macro variable to be created. Upon careful comparison of the macro definition and invocation, it’s apparent that the keyword parameters have been specified in a *different order*

in each. Since keywords have been used, the order is not important and the SAS macro processor will assign parameter values correctly to each keyword parameter. The **SYMBOLGEN** lines in the log show the resolution of macro variables, the **MPRINT** lines show the SAS code generated by the macro.

```
options mprint symbolgen;
%macro inlist(ds = , fld = , mvar = inlist);
    %global &mvar;
    proc sql noprint;
        select distinct quote(trim(&fld))
          into :&mvar separated by ','
          from &ds;
    quit;
%mend inlist;
%inlist(ds = sashelp.shoes, mvar = reglist, fld = region);

LOG:
SYMBOLGEN:  Macro variable MVAR resolves to reglist
MPRINT(INLIST):   proc sql noprint;
SYMBOLGEN:  Macro variable FLD resolves to region
SYMBOLGEN:  Macro variable MVAR resolves to reglist
SYMBOLGEN:  Macro variable DS resolves to sashelp.shoes
MPRINT(INLIST):   select distinct quote(trim(region)) into :reglist separated by ','
from sashelp.shoes;
MPRINT(INLIST):   quit;
NOTE: PROCEDURE SQL used:
      real time           0.01 seconds
      cpu time            0.01 seconds


SYMBOLGEN:  Macro variable REGLIST resolves to "Africa","Asia","Canada","Central
            America/Caribbean","Eastern Europe","Middle East","Pacific","South
            America","United States","Western Europe"
1310  %put &reglist;
"Africa","Asia","Canada","Central America/Caribbean","Eastern Europe","Middle
East","Pacific","South America","United States","Western Europe"
```

While the **inlist** macro has been improved, it is still limited since the IN list will always create a *quoted* list of values. How can the macro be modified to deal with numeric data as well? If we add another parameter to inform the macro that the data set variable is character or numeric, the value quoting can be conditioned using the macro **%if** statement.

```
%macro inlist(ds = , fld = , type = C, mvar = inlist);
    %global &mvar;
    proc sql noprint;
          %if &type = C %then %do;
                select distinct quote(trim(&fld))
          %end; %else %do;
                select distinct &fld
          %end;
          into :&mvar separated by ',' from &ds;
    quit;
%mend inlist;

%inlist(ds = sashelp.shoes, mvar = reglist, fld = region);
```

This example highlights another advantage of keyword parameters. The **type** parameter is defined in the macro definition with a default value of **C**. If **type =** is omitted from the macro invocation, the default value will be used.

Notice the absence of quotes in the **%if** statement. Since the macro facility is really just a text processor, everything it finds without a percent sign or ampersand (**%**, **&** macro triggers) is treated as text. When the **&type** symbol is

encountered in the **%if** statement, the macro processor will retrieve the value of **&type** from the symbol table. The retrieved value, **C**, will be compared to whatever is found on the other side of the equal sign. If the two are equal, the text between the **%then** **%do** and the **%end** will be processed.

In this case, because the **type** parameter wasn't specified in the macro invocation, the default value of **C** coded in the macro definition will prevail and the distinct values returned by the query will be quoted.

CREATING A SEQUENCE OR "ARRAY" OF MACRO VARIABLES

In the examples we've seen thus far, the values retrieved / created have been stored in a single macro variable. Sometimes, it's very helpful to create a sequence or "array" of macro variables to process iteratively. For example, a monthly process may need to run for a dynamically determined number of months (see the 1st "gotcha" at the beginning of this paper).

The months could be assigned manually via a series of **%let** statements.

```
%let mth1  = 198001;
%let mth2  = 198002;
%let mth3  = 198003;
%let Mths  = 3;
```

However, if the data is available in a dataset, multiple macro variables can be created in one fell swoop using SQL into:

```
proc sql noprint;
select distinct put(mdy(month,day,year), yymmN6.)
  into :Mth1 - :Mth999
  from sashelp.retail;

  %let NumMths = &sqlobs;

quit;

%put &numMths Months of Dates;
%put Mth 1: &Mth1, Mth &NumMths: &&Mth&NumMths;
```

The query is extracting distinct year/month values and creating a series of **Mth**nnn macro variables. The **999** specified in the upper range of the **into** specification is entirely arbitrary, but it must be a number high enough to contain all of the possible year/month values. If the range specified had been **into :Mth1 - :Mth3**, the query would only return three values. As it is, the number of macro variables created is available from the automatic macro variable **&sqlobs** and saved in the **NumMths** macro variable. After executing the **%put** statements following the SQL statement, the log displays:

```
SYMBOLGEN:  Macro variable NUMMTHS resolves to 58

1320  %put &numMths Months of Dates;
58 Months of Dates
1321  %put Mth 1: &Mth1, Mth &NumMths: &&Mth&NumMths;

SYMBOLGEN:  Macro variable MTH1 resolves to 198001
SYMBOLGEN:  Macro variable NUMMTHS resolves to 58
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable NUMMTHS resolves to 58
SYMBOLGEN:  Macro variable MTH58 resolves to 199404
Mth 1: 198001, Mth 58: 199404
```

This of course begs a question - what in the world does **&&Mth&NumMths** do?!? Earlier in the word scanner / macro processor discussion we discovered that as macro triggers are encountered, the macro processor kicks in and deals with macro tokens. When macro variables are specified, the macro processor will determine their value by looking them up in the symbol table. In the case outlined above, the macro processor will actually pass through the resolution phase several times.

Consider `&&Mth&NumMths`

The word scanner hands the macro processor the first 2 tokens - `&&`

`&&` resolves to `&` which is placed back in the input stack

The word scanner hands the macro processor the next token - `Mth`

`Mth` resolves to `Mth` which is placed back in the input stack

The word scanner hands the macro processor the next 2 tokens – `&NumMths`

`&NumMths` resolves to 58 which is placed back in the input stack

Now the input stack has `&Mth58`

The word scanner hands the macro processor these 2 tokens which resolve to 199404.

Easy?!? Tokenization is actually quite predictable once you get the hang of it.

USING A SEQUENCE OR "ARRAY" OF MACRO VARIABLES

Recall the original sequence of macro variables.

```
%let mth1  = 198001;
%let mth2  = 198002;
%let mth3  = 198003;
%let Mths  = 3;
```

Using these macro variables, consider the following example. The macro enables us to produce as many PRINT steps as required, without repeatedly copying the code. The macro **%do** loop, iteratively processes while incrementing the macro variable **i** from 1 to the value of the **&lmt** parameter. Note that **&lmt** is the macro parameter that is populated with the value of **&Mths** at macro invocation time.

```
%macro print(lmt);
    %do i = 1 %to &lmt;
           title "Sales &&mth&i";
           proc print data = sales_&&mth&i noobs;
           run;
    %end;
%mend print;

%print(&Mths)
```

Partial Log:

The **MLOGIC** lines in the log show the incrementing / iterative activity of the **%do** loop. The text generation nature of SAS macro can be appreciated when the **MPRINT** output is considered.

```
MLOGIC(PRINT):  %DO loop index variable I is now 3; loop will  iterate again.
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable I resolves to 3
SYMBOLGEN:  Macro variable MTH3 resolves to 198003
MPRINT(PRINT):   title "Sales 198003";
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable I resolves to 3
SYMBOLGEN:  Macro variable MTH3 resolves to 198003
MPRINT(PRINT):   proc print data = sales_198003 noobs;
MPRINT(PRINT):   run;

NOTE: There were 12 observations read from the data set WORK.SALES_198003.
NOTE: PROCEDURE PRINT used:
```

```
MLOGIC(PRINT):  %DO loop index variable I is now 4; loop will not iterate again.
MLOGIC(PRINT):  Ending execution.
```

The **%print** macro generated the following code:

```
title "Sales 198001";
proc print data = sales_198001 noobs;
run;

title "Sales 198002";
proc print data = sales_198002 noobs;
run;

title "Sales 198003";
proc print data = sales_198003 noobs;
run;
```

Partial Output:

Sales 198003

month	region	office	sales
198003	Ontario	100	$22,835
198003	Ontario	101	$119,763
198003	Ontario	102	$49,361
198003	Ontario	103	$32,024
198003	Western Canada	200	$113,777
198003	Western Canada	201	$119,663
198003	Western Canada	202	$67,034
198003	Western Canada	203	$65,641
198003	Maritimes	300	$6,147
198003	Maritimes	301	$8,218

MANIPULATING MACRO VARIABLE CONTENTS

In the previous example, the macro was iterating over a sequence or “array” of macro variables. Sometimes, it’s advantageous to store all possible values in a delimited list in a single macro variable and extract the values one by one to process them individually.

This example builds the list of values delimited by the pipe character, and stores them in a single macro variable. Note the use of **`separated by`** to ensure all values are stored in the macro variable. The **`%rip_n_tear`** macro, that follows, will then parse the individual values from the **`&prodlist`** macro variable and use the individual values to build “data-driven” DATA steps.

```
%macro ValueList(ds = , var = , mvar = inlist);
    %global &mvar;
    proc sql noprint;
            select distinct &var
              into :&mvar separated by '|'
              from &ds ;
    quit;
%mend ValueList;

% ValueList (    ds     = sashelp.prdsale,
                 var    = product,
                 mvar   = prodlist);
```

```
%put ProdList contains &ProdList;
```

Partial Log

```
ProdList contains BED|CHAIR|DESK|SOFA|TABLE
```

Now that the **%ValueList** macro has created the macro variable containing the pipe-delimited values, let's have a look at **%rip_n_tear**. The **%rip_n_tear** macro pulls the individual values from the **&prodlist** variable and uses them when building the DATA steps. Note the use of **%scan**, a SAS-supplied macro function that works like its cousin **scan**. None of the parameters of the **%scan** function are quoted since the macro facility is simply processing text; it has no notion of anything but text unless explicitly told so.

And, it is *possible* to explicitly tell SAS to process numeric data arithmetically. In the macro below we're manually incrementing **&i** (used as the word counter in the **%scan** function) via:

```
%let i = %eval( &i + 1 );
```

The **%eval** (used for integer data, **%sysevalf** is the floating point counterpart) function causes the macro processor to treat the stuff between the parenthesis as an arithmetic expression. Without the **%eval**, the value of **&i + 1** would be 5 characters long, ie:. the current value of **&i** and the characters **+ 1**, a literal of **1 + 1**.

```
%macro rip_n_tear(ds = , ripvar = , var = );
    %let i = 1;
    %do %until(%scan(&&&ripvar,&i,|) = );
           %let value = %scan(&&&ripvar,&i,|);

           data data_&value;
                  set &ds;
                     where &var = "&value";
           run;

           %let i = %eval( &i + 1 );
    %end;

%mend rip_n_tear;

%rip_n_tear(     ds     = sashelp.prdsale,
                 var    = product,
                 ripvar = prodlist);
```

Partial Log:

```
MLOGIC(RIP_N_TEAR):  %LET (variable name is I)
SYMBOLGEN:  Macro variable I resolves to 1
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable RIPVAR resolves to prodlist
SYMBOLGEN:  Macro variable PRODLIST resolves to BED|CHAIR|DESK|SOFA|TABLE
SYMBOLGEN:  Macro variable I resolves to 2
MLOGIC(RIP_N_TEAR):  %DO %UNTIL(%scan(&&&ripvar,&i,|) = ) condition is FALSE; loop
will iterate again.
MLOGIC(RIP_N_TEAR):  %LET (variable name is VALUE)
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable RIPVAR resolves to prodlist
SYMBOLGEN:  Macro variable PRODLIST resolves to BED|CHAIR|DESK|SOFA|TABLE
SYMBOLGEN:  Macro variable I resolves to 2
SYMBOLGEN:  Macro variable VALUE resolves to CHAIR
MPRINT(RIP_N_TEAR):   data data_CHAIR;
SYMBOLGEN:  Macro variable DS resolves to sashelp.prdsale
MPRINT(RIP_N_TEAR):   set sashelp.prdsale;
```

```
SYMBOLGEN:  Macro variable VAR resolves to product
SYMBOLGEN:  Macro variable VALUE resolves to CHAIR
MPRINT(RIP_N_TEAR):   where product = "CHAIR";
MPRINT(RIP_N_TEAR):   run;

NOTE: There were 288 observations read from the data set SASHELP.PRDSALE.
      WHERE product='CHAIR';
NOTE: The data set WORK.DATA_CHAIR has 288 observations and 10 variables.
```

The **%until** condition looks a little strange doesn't it? There's nothing on the right side of the equal sign. When the **%scan** function attempts to find the sixth item in the **&prodlist** value, it'll find nothing, the same thing that's on the right side of the equal sign. ;-) It works.

FUNCTION STYLE MACROS

Macros we've seen to this point generated complete, stand-alone SAS code. However, macros can generate any amount of text, even code snippets that may not be a complete SAS step or even a complete statement. Often these types of macros are known as "function-style" macros. Function-style macros can be used to generate two different types of results:

1) a snippet of code to be included in another SAS or macro statement
2) an answer deduced with exclusively macro-code to be used in or outside of step boundaries

Consider the following examples of function-style macros.

You may have access to a data mart of SAS data sets where the *many* date values have all been stored in a numeric field in yyyymmdd format. As we know, if dates are *properly* stored in internal SAS format, the many SAS date formats and functions can be brought to bear upon the date value to provide a wealth of programming and execution efficiencies. Rather than writing the required nested input/put statements each time a date field is encountered, a function-style macro does the trick.

The macro call is coded in the *middle* of the SAS **IF** statement. Notice that a statement ending semi-colon is **not** specified on the macro call, nor is a semi-colon found *inside* the macro. If either semi-colon had been specified the **IF** statement would have prematurely, and erroneously, terminated.

```
%macro date(fld);
    input(put(&fld,8.),yymmdd8.)
%mend date;

data a;
    set b;
    if %date(open_date) > '01Jan2005'd;
run;
```

Log:

```
1784  data a;
1785      set b;
MLOGIC(DATE):  Beginning execution.
1786      if %date(open_date) > '01Jan2005'd;
MLOGIC(DATE):  Parameter FLD has value open_date
SYMBOLGEN:  Macro variable FLD resolves to open_date
MPRINT(DATE):   input(put(open_date,8.),yymmdd8.)
MLOGIC(DATE):  Ending execution.
1787  run;
```

The advantage of the second type of "function-style" macro can be illustrated with the following example. Consider an example where a Master data set is merged with a daily Transaction data set unless the Transaction data set doesn't exist or contains zero observations. SAS provides a number of functions and/or coding solutions to ascertain the existence of a data set and the number of observations in the data set:

- exist() function

- number of observations is available via:
  - nobs = on SET statement
  - sashelp.vtable nobs value
  - attrn() function

The existence and number of observations in a data set **can be determined** using the DATA step or SQL:

```
%macro exist_obs;
    data _null_;
          call symput('exist',exist('yourlib.transaction'));
    run;

    /*  Data step results in error if data set does NOT exist-must be conditioned */
    %if &exist = 1 %then %do;
          data _null_;
                if 0 then set yourlib.transaction nobs = nobs;
                call symputx('nobs',nobs);   * v9 only ;
                stop;
          run;
    %end;
%mend exist_obs;

%exist_obs;
```

Or, a simpler method using SQL:

```
%let nobs = 0;
proc sql;
    select nobs into :nobs
      from sashelp.vtable
     where libname = 'YOURLIB'
       and memname = 'TRANSACTION';
    %let exist = &sqlobs;  /* No rows returned if dataset doesn't exist  */
quit;

%put EXIST = &exist NOBS = &nobs;
```

A macro-only solution is the most versatile. As written, **%attrn** is a generic utility macro that may be used to retrieve any numeric data set attribute (eg. CRDTE – data set creation date, MODTE – last modified date, NVARS – number of variables) for any data set. Note that the **%attrn** macro returns the number of observations via the **bolded partial *macro* statement**.

```
%macro attrn(ds,attrib);
    %let dsid = %sysfunc(open(&ds,is));
    %if &dsid >0 %then %do;
         %sysfunc(attrn(&dsid,&attrib));
    %end;
    %let rc = %sysfunc(close(&dsid));
%mend attrn;
```

The decision to proceed with the Master / Transaction merge is made by another macro which verifies both the existence of the Transaction dataset and, using %attrn, the number of observations in it. If both conditions are satisfied, the **%do_merge** macro is invoked to merge the Master / Transaction data sets.

```
%macro check_transaction(ds);
    %if %sysfunc(exist(&ds)) %then %do;
        %if %attrn(&ds,nobs) > 0 %then %do;
                %do_merge
        %end; %else %put No observations in &ds;
```

```
    %end; %else %put &ds does NOT exist;
%mend check_transaction;

%check_transaction(yourlib.transaction);
```

MISCELLANIES

Helpful tidbits with no other home.

Description	v8	v9
Check if a macro variable exists	search SAS-L archives for a "macro variable exist whitlock"	%symexist(mvar) or symexist(mvar)
Delete a macro variable	To assign a NULL value, use %let mvar = ;	%symdel(mvar) or symdel(mvar)
Define macro variable scope	%global %local	
Determine macro variable existence within scope		%symglobal(mvar), %symlocal(mvar), symglobal(mvar), symlocal(mvar)
Copy macro variables from local to remote SAS/Connect environment	%syslput	
Copy macro variables from remote to local SAS/Connect environment	%sysrput	
Alter case of macro variable value, making %if comparisons case-insensitive	%lowcase %upcase	
Definitive guide to macro "quoting" or *hiding* meaning in macro	Ian Whitlock's "A Serious Look at Macro Quoting" Paper	

CONCLUSION

Is the SAS macro facility mysterious? Not really, especially when it's viewed as simple text replacement and text generation. A number of examples have shown where macro's text replacement functionality helps reduce coding effort, program maintenance, and potential for errors. Macro code reduces repetition, allows the writing of modular code and the ability to control program flow.

ACKNOWLEDGEMENTS

We thank Laura House for lending us her eyes and expertise and giving this paper a thorough read. Any errors that remain are entirely the work of Marje and Harry.

CONTACT INFORMATION

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

Marje Fecht
Prowerk Consulting
Marje.Fecht@prowerk.com

Harry Droogendyk
Stratia Consulting
sesug@stratia.ca