MACROS: TIPS, TECHNIQUES, AND EXAMPLES

Andrew M. Traldi, Advanced Quantitative Solutions, Inc., Alpharetta, GA

ABSTRACT

Most SAS® users are aware that SAS has a macro facility, but might be unsure of how they can use it or are fearful that macros are too difficult. Although macros can be complex, they can be very helpful in writing general-purpose SAS programs; in some instances, they are absolutely critical to an application.

WHAT IS THE SAS MACRO FACILITY?

The purpose of the SAS macro language is to generate text which is used in SAS programs; this text can be any valid SAS code: statements, variables, text strings, PROC steps, etc. In its simplest form, a macro variable can be used for text substitution in SAS code. Consider the following example:

```
%let state=GA;
%let month=Jul2001;

…
proc print data=permlib.sales
(where=(state_code="&state" and
month_year=input("&month",monyy7.
));
title "Sales report for &state /
&month";
run;
```

These statements could be useful if you provide reporting by month and region and you want to be able to generate reports for different states and months easily. This example assumes that there is a dataset PERMLIB.SALES that contains sales data and has variables `state_code` and `month_year` that we can use to select the desired observations. Note that we haven't even used a macro here, just macro variables for simple text substitution.

One important difference between macro code and SAS code is that the macro code is compiled **prior to** regular SAS code, and the code generated by the SAS macro compiler is then processed by the SAS compiler. Here is a silly example that clearly illustrates this difference:

```
data dumb;
if 1 eq 2 then do;
   * this will never be
executed!;
   xxxyyyzzz;
end;
else do;
   put 'Hello'; …
end;
run;

203  data dumb;
204  if 1 eq 2 then do;
205       * this will never be
executed!;
NOTE: SCL source line.
206       xxxyyyzzz;
          ---------
          180
ERROR 180-322: Statement is not
valid or it is used out of proper
order.

207  end;
208  else do;
209     put 'Hello';
210  end;
211  run;

%macro dumb;
data dumb;
%if 1 eq 2 %then %do;
   %* This will never be
compiled!;
    xxxxxyyyyy;
%end;
%else %do;
    put 'Hello';
%end;
run;
%mend dumb;
%dumb

MPRINT(DUMB):   data dumb;
MPRINT(DUMB):   put 'Hello';
MPRINT(DUMB):   run;

Hello
```

In the first part of this example, even though the statement in the `if 1 eq 2 then do` group will never be executed, it is still compiled and causes a syntax error. In the second case, the

statement within the `%if 1 eq 2 %then %do` group is successfully compiled by the macro compiler, but because it is never executed, it is never passed to the SAS compiler. This is an example of conditional execution vs. conditional compilation.

ENVIRONMENT

A short discussion of the macro variable environment is probably in order. The environment can be explicitly specified with either the `%global` or the `%local` statement. The value of a global macro variable is available throughout the program – open code as well as within macros. A local macro variable's value is available only in the macro where it is defined (therefore, a `%local` statement is not valid in open code).

Consider the following example:

```
%global var1;
%let var1=hello;
%let var2=world;
%put ** in open code var1=&var1
var2=&var2 **;

%macro test;
%put ** in test: var1=&var1
var2=&var2 var3=&var3 **;
%mend test;
%test

%macro test2;
%local var2;
%let var1=hi;
%let var2=universe;
%let var3=hello, world;
%put ** in test2: var1=&var1
var2=&var2 var3=&var3 **;
%test;
%mend test2;
%test2;
%test

%put ** in open code var1=&var1
var2=&var2 var3=&var3 **;
```

Here is the resulting SASLOG:

```
1    %global var1;
2    %let var1=hello;
3    %let var2=world;
4    %put ** in open code
var1=&var1 var2=&var2 **;
** in open code var1=hello
var2=world **
5
6    %macro test;
7    %put ** in test: var1=&var1
var2=&var2 var3=&var3 **;
8    %mend test;
9    %test
WARNING: Apparent symbolic
reference VAR3 not resolved.
** in test: var1=hello var2=world
var3=&var3 **
10
11   %macro test2;
12   %local var2;
13   %let var1=hi;
14   %let var2=universe;
15   %let var3=hello, world;
16   %put ** in test2: var1=&var1
var2=&var2 var3=&var3 **;
17   %test;
18   %mend test2;
19   %test2;
** in test2: var1=hi
var2=universe var3=hello, world
**
** in test: var1=hi var2=universe
var3=hello, world **
20   %test
WARNING: Apparent symbolic
reference VAR3 not resolved.
** in test: var1=hi var2=world
var3=&var3 **
21
22   %put ** in open code
var1=&var1 var2=&var2 var3=&var3
**;
WARNING: Apparent symbolic
reference VAR3 not resolved.
** in open code var1=hi
var2=world var3=&var3 **
```

The default environment for a macro variable is what I would call *downward* global. That is, the value of the macro variable can be referenced (and changed) in the environment where it first appears (open code or a macro) as well as in any macros which are invoked from that environment. In the first statement, the `%global` isn't really necessary, because the assignments are made in open code. However, note the behavior of the macro variable var3 – which is not given an explicit environment with either a `%global` or `%local` statement when it is defined in macro test2: when macro test is invoked from within the macro, the value of var3 is available, but not when it is invoked from open code. Note also that var2 is declared as a

local variable in macro test2, so that the value that it is assigned only remains while macro test2 is executing – when test is invoked again in open code, the value given to var2 inside of macro test2 is no longer available. This may seem a little cumbersome at first, but it allows for a great deal of flexibility.

TIPS

Define all macro variables as either global or local.
Set aside specific variables for `%do` loop indices and **always** define them as local to avoid inadvertently changing their values in other macros.

ASSIGNING VALUES TO MACRO VARIABLES

We have seen how macro variables can be assigned values with the `%let` statement. However, there are many instances where we need to reference SAS datasets for the values that we want. The CALL SYMPUT statement is used to assign values to macro variables during DATA step execution, while the SYMGET function is used to retrieve values during DATA step execution (macro variables can also be resolved directly, during DATA step compilation). This example illustrates the use of SYMPUT and SYMGET:

```
%global month year;
%let month=7;
%let year=2002;

data _null_;
length test mthname $ 16;
* These statements are
equivalent;
month=mdy(&month,1,&year);
put month= date9.;
month=mdy(SYMGET('month'),1,SYMGE
T('year'));
put month= date9.;
mthname=put(month,monname9.);
put mthname=;
* Now load month name into a
macro variable;
CALL SYMPUT('mthname',mthname);
* Use SYMGET to get value of
macro variable;
test=SYMGET('mthname');
put test=;
test="&mthname";
put test=;
run;

%put ** &mthname **;
```

Note how we integrated the macro variable into a regular SAS statement, first by resolving it directly and then by using the SYMGET function. It is important to remember that macro variables that are set using CALL SYMPUT are not available to be resolved directly by the macro compiler until *after* the DATA step is finished; this distinction can be seen in the resulting SASLOG:

```
1    %global month year;
2    %let month=7;
3    %let year=2002;
4
5
6    data _null_;
7    length test mthname $ 16;
8    * These statements are
equivalent;
9    month=mdy(&month,1,&year);
10   put month= date9.;
11   month=mdy(SYMGET('month'),1,
SYMGET('year'));
12   put month= date9.;
13   mthname=put(month,
monname9.);
14   put mthname=;
15   * Now load month name into a
macro variable;
16   CALL SYMPUT('mthname',
mthname);
17   * Use SYMGET to get value of
macro variable;
18   test=SYMGET('mthname');
19   put test=;
20   test="&mthname";
WARNING: Apparent symbolic
reference MTHNAME not resolved.
21   put test=;
22   run;

NOTE: Character values have been
converted to numeric values at
the places given by:
      (Line):(Column).
      11:11   11:29
month=01JUL2002
month=01JUL2002
mthname=July
test=July
test=&mthname
NOTE: DATA statement used:
```

```
      real time      0.01 seconds
      cpu time       0.01 seconds


23
24   %put ** &mthname **;
**      July        **
```

Note that the macro variable mthname cannot be resolved directly while the DATA step is still executing, but is available in the `%put` statement immediately afterwards. Also, even though the macro variables month and year contained numeric values, SAS has to perform a character to numeric conversion when the SYMGET function is used. This is because the macro compiler treats all macro variables as text strings, even when they are valid numeric values[1].

Question: What is the environment for the macro variable mthname**,** since it is not defined with a `%local` or `%global` statement?

Answer: Since it is defined by a CALL SYMPUT in open code, it is a global macro variable. However, if this DATA step were inside a macro, then the value would not be available outside the macro.

Another method of setting macro variables is to use PROC SQL. For example, suppose we want a list of all numeric variables in a dataset:

```
proc contents data=sasuser.admit
noprint out=_cont_;
run;

proc sql noprint;
select name into: numeric_vars
separated by ' '
from _cont_
where type eq 1;
quit;

%put &numeric_vars;

33   proc contents
data=sasuser.admit noprint
out=_cont_;
34   run;

NOTE: The data set WORK._CONT_
has 9 observations and 40
variables.
NOTE: PROCEDURE CONTENTS used:
      real time      0.01 seconds
      cpu time       0.01 seconds

35
36   proc sql noprint;
37   select name into:
numeric_vars separated by ' '
38   from _cont_
39   where type eq 1;
40   quit;
NOTE: PROCEDURE SQL used:
      real time      0.00 seconds
      cpu time       0.00 seconds


41
42   %put &numeric_vars;
Age Date Fee Height Weight
```

PARAMETERS

The macro language allows for passing of parameters in much the same way as other programming languages; those of you who develop SAS/AF applications or have used PASCAL or FORTRAN are probably used to passing parameters to functions and subroutines.

A SAS macro can have two types of parameters: positional and keyword. Positional parameters are defined only by their order in the macro invocation and must always be included in the macro invocation, while keyword parameters are defined by the name of the parameter and do not have to be included. A macro can contain both positional and keyword parameters, but the positional parameters must come first. Here is an example of a macro with keyword parameters:

```
%macro smart_print
(dsn=_LAST_,title=,by=,id=,var=,d
snopt=,options=);
%* print the specified dataset,
using the specified variables
   in the BY, ID, and VAR
statements and included options;

%if &by ne %then %let by=BY &by;
%if &id ne %then %let id=ID &id;
%if &var ne %then %let var=VAR
&var;
%if %quote(dsnopt) ne %quote()
%then %let dsnopt=%str (
(&dsnopt) );

TITLE "&title ";
PROC PRINT &options DATA=&dsn
```

```
&dsnopt;
&by;
&id;
&var;
RUN;
%mend smart_print;
```

Here is the macro invocation and resulting SASLOG:

```
options mprint;
%smart_print(dsn=sales,var=name
customer
amount,dsnopt=%str(where=(state
eq 'GA')),by=state,
  title=GA sales, options =
noobs);

10
11   %macro smart_print
12
(dsn=_LAST_,title=,by=,id=,var=,d
snopt=,options=);
13   %* print the specified
dataset, using the specified
variables
14       in the BY, ID, and VAR
statements and included options;
15
16   %if &by ne %then %let by=BY
&by;
17   %if &id ne %then %let id=ID
&id;
18   %if &var ne %then %let
var=VAR &var;
19   %if %quote(dsnopt) ne
%quote() %then %let dsnopt=%str (
(&dsnopt) );
20
21   TITLE "&title ";
22   PROC PRINT &options
DATA=&dsn &dsnopt;
23   &by;
24   &id;
25   &var;
26   RUN;
27   %mend smart_print;
28   options mprint;
29
%smart_print(dsn=sales,var=name
customer amount,
dsnopt=%str(where=(state eq
29 ! 'GA')),by=state,
30     title=GA sales, options =
noobs);
MPRINT(SMART_PRINT):   TITLE "GA
sales ";
MPRINT(SMART_PRINT):   PROC PRINT
noobs DATA=sales (where=(state eq
'GA')) ;
MPRINT(SMART_PRINT):   BY state;
MPRINT(SMART_PRINT):   ;
MPRINT(SMART_PRINT):   VAR name
customer amount;
MPRINT(SMART_PRINT):   RUN;

NOTE: There were 100 observations
read from the data set
WORK.SALES.
      WHERE state='GA';
NOTE: PROCEDURE PRINT used:
      real time      0.31 seconds
      cpu time       0.03 seconds
```

Note how the order of the parameters in the invocation is not the same as in the macro declaration and that we did not have to specify the id parameter. If we had used positional parameters, we would have had to not only specify the parameters in the same order but also use placeholders for the unneeded parameters:

Here is the definition and invocation of the same macro with positional parameters:

```
%macro smart_print
(dsn,title,by,id,var,dsnopt,
options);
…
%mend smart_print;

%smart_print(sales,GA
sales,state,,name
customer,%str(where=(state eq
'GA')),noobs);
```

Here, we need to include an extra placeholder for the `id` parameter and specify the parameters in the same order as in the definition. For more complex macros, keyword parameters are preferable.

Note that this print macro did not perform any error-checking (ensuring that the data set exists, that the variables are found, that the options given are valid, etc.). Often, a decision has to be made about how much programming time is worth investing in a macro – depending on how often it will be used, whether it will be made available to other users, etc.

MACRO STYLE AND COMMENTS

There are style issues when writing macro code, just as there are for regular SAS code.

Use good, clean style. This is especially important because macro code is usually less readable than base SAS code. Some examples of good macro style include: indenting %do groups, using white space, and – most importantly – using comments liberally.

Use keyword parameters and define macro variables as needed.

Everyone has programming conventions that he or she likes to use. Here are a few that I use to help keep my macro code as readable as possible:

- Use lower case for all macro code – except text strings that must be upper case.
- Avoid use of the `%goto` statement – it makes the program very hard to follow
- Initialize all global macro variables at the beginning of the program
- Use the %* rather than the * comment statements, understanding the difference between them:

  %* is a **macro compiler comment** – the macro compiler will ignore the statement and it will not print in the SASLOG

  * is a **SAS comment** – the macro compiler will not ignore the statement, so it must be an appropriate place in the macro code for a SAS comment. Here is an example – where would the use of a * comment instead of a %* comment cause an error?

```
%macro _missing(var=,type=);
%* this macro will set a variable
var to missing, vartype=C
indicates a character variable,
else numeric - must be called
from a DATA step;
%* if character, use blank ;
%if %upcase(type) eq C %then %do;
   &var = ' ';
%end;
%* if numeric, use .;
%else %do;
   &var = .;
%end;
%mend _missing;

options mprint;

data dumb;
if x=0 then
  %_missing(var=Y,type=N);
run;
```

If the first or third comments were written using a * comment, this would not work. If the first comment had a * comment, the SAS compiler would see this statement: if x=0 then * this macro will … ; Y=**.** This will of course cause an error, because an inline comment must be enclosed within /* and */**.** Why would the third comment cause a problem? The macro compiler is looking for an %else statement immediately after the end of the first %do loop – it ignores the %* comment, but treats the * comment as a statement, and therefore will produce an error when it comes to the %else statement. Incidentally, this can happen in base SAS as well – note that the following will cause an error:

```
data one;
…
if x=1 then do;
 …
end ;;
else do;
 …
end;
run;
```

In most cases, a double semi-colon does not matter to the SAS compiler, but here it expects the else statement to immediately follow the end of the do-loop, and the extra semi-colon causes it to compile an additional statement, producing an error.

It's also important to remember that the de-bugging process for macro code is more difficult than for regular SAS code. Because of that, it is even more critical to document programs that include macros. The `mprint, mtrace`, and `symbolgen` options make it easier to examine what is being generated by the macro compiler.

AN EXAMPLE WITH UTILITY MACROS

This is an example of how macros can be used to save time and run programs more efficiently. For example, suppose that there are large flat files that contain transactional records that come

out of a billing system each month for different markets, and that these become available at different times. Rather than waiting for all the files to be available, we would like to provide reporting on each market as soon as possible.

The following utility macros typically would be included in a macro library or an autoexec file.

```
%macro _mprint;
%global mp;
%* return current mprint setting
in mp;
%let
mp=%sysfunc(getoption(mprint));
%mend _mprint;

%macro exist(dsn);
%* determines if a dataset exists
and returns yes/no and number of
obs;
%global exist nobs dsndate;

%_mprint;
options nomprint; %* turn off
mprint;
%let exist=no;
%if &dsn ne %then %do;
    %* create dummy dataset;
    data;
    run;

    options nodsnferr;
    data _null_;
    set &dsn(in=in1) _last_;
    if in1 then call
symput('exist','yes');
    stop;
    run;
    options dsnferr;
%end;
%if &exist eq yes %then %do;
   %* determine number of
observations and last modify
date;
   proc contents data=&dsn
noprint out=_cont_;
   run;

   data _null_;
   set _cont_;
   call symput('nobs',
compress(put(nobs,12.)));
   date=datepart(modate);
   call symput('dsndate',
put(date,date9.));
   stop;
   run;
   proc datasets lib=work nolist;
   delete _cont_;
   run;
   quit;
%end;
%else %do;
   %let exist=no;
   %let nobs=0;
   %let dsndate=;
%end;
options &mp;  %* return to
previous mprint setting;
%mend exist;

%macro fexist(fname);
%* determines if an external file
exists;
%global fexist;
%_mprint;
options nomprint;

%let fexist=no;
%if "&fname" ne "" %then %str(

    data _null_;
    if 0 then infile "&fname";
    call symput('fexist','yes');
    stop;
    run;

);
options &mp;  %* return to
previous mprint setting;

%mend fexist;

* This program will read
transactional files for each
market, when available, and
create datasets for reporting.
Also, it will update a dataset
indicating which markets are
available.

%let month=jan2002;  * desired
month;

* Load market codes into macro
variables and set value of mkts.
Typically, this would be done
either in an autoexec file or
with a format, etc. - here we are
just using %let statements;

%let mkt1=ATL;
%let mkt2=MIA;
%let mkt3=ORL;
%let mkt4=JAX;
```

```
%let mkt5=LEX;
%let mkts=5;

libname out '/home/sasdata';

%macro _read;
%local i filenme;

%do i=1 %to &mkts;
   %local complete&i;
   %let
filenme=/billingdata/trans_&month
._mkt&&mkt&i...txt;
   %* Check to see if we have
already a dataset for this
market;

%exist(out.mk&&mkt&i.._&month);
   %if &exist eq no %then %do;
       %* No dataset - see if the
transactional file is available;
       %fexist(&filenme);
          %if &fexist eq yes
%then %do;
         %* read transactional
file;
           data
out.mk&&mkt&i.._&month;
         infile "&filenme";
         input …;
                run;
       %end;
       %else %do;
           data _null_;
         file print;
         put "&filenme not
available";
           run;
       %end;
    %end;
    %else %do;
        data _null_;
        file print;
        put "dataset for market
&&mkt&i / &month already exists";
        run;
    %end;
%end;
%* produce a dataset indicating
which markets are available;
%do i=1 %to &mkts;

%exist(out.mk&&mkt&i.._&month);
   %let completed&i =
%upcase(&exist);
%end;
data out.&month._markets;
length market complete $ 3;
%do i=1 %to &mkts;
    market="&&mkt&i";
    complete="&&completed&i";
    output;
%end;
run;

proc print
data=out.&month._markets;
title "Markets that are now
available for &month";
run;
%mend _read;

options mprint;
%_read;
```

Here are some quick notes about this program. Note the use of the && to resolve the macro variables mkt1, mkt2, etc. This is somewhat analogous to the way arrays are used in a DATA step. Also, why is the period needed in the dataset name `out.&month._markets`? We use a period to indicate to stop resolving the macro variable name at that point (without the period, the macro compiler would try to retrieve the value of the macro variable jun2002_markets, which does not exist).

CONCLUSION

We have just scratched the surface of what can be done with macros. For the beginning user, they can be used to make programs more automated. Once we feel more comfortable with them and understand how powerful they are, we can use them in complex production applications.

CONTACT INFORMATION

Your comments and questions are encouraged. Please contact the author:

Andrew M. Traldi
Advanced Quantitative Solutions, Inc.
5825 Culler Court
Alpharetta, GA 30005
770-418-9167
atraldi@bellsouth.net