

BUILDING PROGRAMS WITH THE SAS® MACRO FACILITY

Philip M. Pochon, Computer Task Group, Inc.

Abstract

The SAS®* macro facility allows a programmer-analyst to create object-like utility functions that may be used to construct task specific modules within a main program. The macro interpreter resides between the source code and the base SAS® interpreter. Macro language statements and functions can thus be used to control the flow of data values and lines of code to the base interpreter.

This tutorial first reviews the principles of constructing SAS® macro statements and functions and then discusses how macro code is processed by, and can control the SAS® interpreters. Several example utility functions are then examined. A final section shows how utility functions are assembled into a generalized program that will generate CONTRAST statements for statistical testing in the GLM procedure.

SAS® Macro Overview

The macro language is a facility within the base SAS® system that allows the programmer to perform certain tasks outside the base SAS® interpreter. This paper concentrates upon one of these tasks: control of the flow of 'normal' SAS® code statements to the base interpreter. 'Normal' SAS® code statements include lines for base SAS® data and procedure steps as well as lines for procedures in such products as SAS/STAT® and SAS/GRAPH®.

Certain rules govern the flow and execution of SAS® code. A SAS program (source code in a program or the pre-compiler) is loaded into the code stack to be read one line at a time. Every line of code (macro or base) must end with a semicolon. When the SAS® interpreter finds a semicolon it interprets the line and executes the macro portion of it. It does not necessarily execute the base SAS® language portions of the line. In most procedures and all data steps base SAS® lines are not executed until a step boundary is crossed.¹

SAS® maintains two program vectors. The first vector is processed by the macro language interpreter. The second vector is processed by the base SAS® interpreter.

Whenever a semicolon is encountered in the code stack the line is first sent to the macro program vector. Here, the following sequence of operations takes place:

- 1) All macro variables in the line are assigned their value from the symbol table.
- 2) The macro code portions of the line are then interpreted, and immediately executed. This may result in:
 - a) Assigning values to macro variables.
 - b) Adding words to the program vector.
 - c) Adding pointers and lines of code to the code stack if a macro function was called.
- 3) After macro processing, if the code stack still contains a semicolon the line is sent to the base interpreter for processing.

Items 2b and 2c are the key processes for building programs with the macro facility. By allowing the programmer to control the flow of code lines into the code stack and control the formation and flow of code lines to the base SAS® interpreter macro is a powerful tool for building SAS® programs.

Use of Macro Variables

Macro variables are by definition character strings. The most direct way to have a macro variable control the flow of a program is to have its value be executable SAS® code.

When building programs for a production system such a process creates more problems than it solves. Structured design techniques, CASE tools and data dictionaries can not handle such a confusion of categories. Maintenance and documentation considerations generally outweigh any gain in reduced coding costs.

Macro variables can also carry names (both table names and column names). The ability to create general utility functions relies heavily on passing all names as parameters. Figure 1 shows such a utility function whose job is to add an annotation marker to a specific column in a specific row of a data set about to be printed. The data set name and two column names are passed as parameters, which will be evaluated by the macro interpreter before the base SAS® interpreter sees the data step lines.

* SAS, SAS/STAT and SAS/GRAPH are registered trademarks of the SAS Institute.
¹ In version 6 SAS® certain procedures (e.g. SQL and DATASETS) do execute lines as they are interpreted.

Figure 1

```
%macro annotvar(_indsn=work.rptpage,
               _column=P458,
               _selvar=ANIMAL,
               _selval="0rowval",
               _locatn=8,
               _char=@annomark);
data &_indsn;
  set &_indsn;
  if (trim(left(&_selvar))=&_selval) then
    (substr(&_column,&_locatn,1))="&_char";
run;
%mend;
```

A second use for macro variables is to pass fixed values that are used in conditional statements. The data step in Figure 1 uses an IF condition in which both the column name and the fixed value are passed as parameters.

Macro %IF statements (which will be examined in more detail later) by definition must use macro variables. Figure 2 shows the use of the SAS® system macro variables SQLOBS and SQLRC to control program flow after a SQL SELECT statement.

Figure 2

```
%macro updttrmt(_indsn=work.codes,
               _outdsn=work.unitage,
               _seidsn=work.treatment);
proc sql;
  create table &_outdsn as
  select entcd as uncode, codef as unitage
  from &_indsn
  where (tabcd='UN') and entcd in
    (select distinct uncode from &_seidsn);
quit;

%if (&sqlobs=0) or (&sqlrc>0) %then %do;
  %primesg(_msgcode=MISSTATLAB)
  %let lethal=Y;
%end;

%mend;
```

If the read fails a second macro function (PRTMSG) is called to print an error message and then a lethal error flag is set to 'Y'. This flag can be used by the calling function to direct any further processing.

Use of Macro Functions

The use of macro functions to control the flow of SAS® code rests upon the ability of one macro function to call another macro function. At the point of the call the interpretation of the calling function is halted and a pointer is set at that line in

the code stack. The called function is then inserted into the stack and all of its lines are interpreted and executed. When the called function's %MEND statement is executed the interpretation of the calling function is resumed at the pointer.

Figure 3 shows a typical macro driver function which calls utility functions to do its work.

Figure 3

```
%getptlab(_datadsn=work.rptpage,
          _labdsn=@prtlibdsn,
          _report=SUMMARY,
          _rsitlist=&&assyls1&pagetime,
          _markist=TAMTLAB ASSAY STATISTIC);
%local result1 result2 result3 result4 result5
       marker1 marker2 marker3 marker4 marker5;

%lstlabl(_indsn=&_labdsn,
        _rpttype=&_report,
        _list=&_rsitlist,
        _txttype=COLUMN LABEL,
        _outvar=RESULT)

%ptlablst(_indsn=&_datadsn,
         _list=&_rsitlist,
         _root=RESULT)

%lstlabl(_indsn=&_labdsn,
        _rpttype=&_report,
        _list=&_markist,
        _txttype=COLUMN LABEL,
        _outvar=MARKER)

%ptlablst(_indsn=&_datadsn,
         _list=&_markist,
         _root=MARKER)

%mend
```

Within each pair of calls LSTLABL read up to five column labels from a data set and then PTLABLST sets the values as column print labels. Result columns are processed first, then marker columns. Different lists and roots are passed, and as each function is interpreted its base SAS® code will be different.

Called macro functions always add lines of code to the stack. In this sense the always control the flow of a program. New lines must be processed by the macro interpreter and if any base SAS® code remains it is added to the program vector of the base SAS® interpreter.

Macro functions may directly return values to the code stack of the calling function. Figure 4 shows the use of a macro function (CNTWORDS) shown in figure 5) to set the termination value for a macro %DO loop.

Figure 4

```

%macro calcstat(_indsn=WORK.RESULTS
               _outdsn=WORK.SUMSTAT,
               _byvarst=TIME TRMT ASSAY,
               _clastst=GENDER,
               _inlst=,
               _varlst=RESULT,
               _statlst=MEAN VAR STDERR N,
               _outlst=MEAN VARIANCE STDERROR
                  ANIMALS);
proc means data=@_indsn noprint;
  by @_byvarst;
  var @_varlst;
  %if (@_clastst>) %then %do;
    class @_clastst;
  %end;
  %if (@_idlst>) %then %do;
    id @_idlst;
  %end;
  output out=@_outdsn
    %do s=1 %to %cntwords(_string=@_statlst);
    %scan(@_statlst,@s)=%scan(@_outlst,@s)
  %end;
;
run;

%mend;

```

Figure 5

```

%macro cntwords(_string=NAME1 NAME2 NAME3);
%local count word;
%let count=1;
%let word=%qscan(@_string,@count,%str( ));

%do %while (@word ne );
  %let count=%eval(@count+1);
  %let word=%qscan(@string,@count,%str( ));
%end;

%eval(@count-1)

%mend;

```

The CNTWORDS function has as its next to last line a %EVAL statement which computes the number of words in the passed string. This line is not closed by a semicolon. The word count value remains in the program vector when CNTWORDS is exited and is incorporated as part of the %DO loop line in CALCSTAT. The semicolon that closes this %DO line finally sends the line to the macro interpreter, with the call for CNTWORDS being replaced with an integer number from the unclosed line in CNTWORDS.

In CALCSTAT a similar unclosed line occurs within the %DO loop. The %DO loop is adding code to the OUTPUT statement which eventually is closed by the semicolon that is on its own line following the %END statement. The result is shown by the base SAS® proc MEANS log in Figure 6.

Figure 6

```

proc means data=work.results;
  by time trmt assay;
  var result;
  class gender;
  output out=work.sumstat
    mean=mean
    var=variance
    stderr=stderr
    n=animals;
run;

```

Note that the ID statement does not occur, for its parameter was blank. The %IF...%THEN clause uses a %DO...%END construct to enclose the CLASS and ID statements precisely so that the CLASS and ID statements can be closed with a semicolon.

The basic premise of the CALCSTAT function is list processing. When building production systems using SAS® macro the use of macro variable word lists can be a very powerful aid in generalizing a program. Here, the BY, VAR and CLASS variables are passed as lists and the OUTPUT statement is built from two word lists.

Use of Macro Language

The macro language facility is by far the most useful aspect of SAS® macro when we consider the problem of controlling program flow. The basic macro language structures that allow a programmer to control the code sent to the interpreter are shown in figure 7.

Figure 7

```

%IF....%THEN....;
%IF....%THEN %DO;
%IF....%THEN (%DO); %ELSE (%DO);

%DO;
%DO N=1 %to @N;
%DO %UNTIL (@N=@N);
%DO %WHILE (@N<@N);

```

The %IF conditional structures can control which lines of code are executed in the program. The %DO repetitive structures allow code to be created an indeterminate number of times.

The generation of repetitive values within a line of base SAS® code and the generation of repetitive lines of base SAS® code within a procedure are a natural place to apply the %DO macro structures.

This repetition with slight variation is characteristic of CONTRAST statements used with

a number of SAS STAT® procedures. CONTRAST statements take a very formal structure. First the key word CONTRAST must appear. This is followed by a label up to 24 characters in length. The name of the variable (or MODEL statement effect) whose values define the entities to be contrasted follows the label. A set of coefficient values follow the variable name. Options may follow the coefficient vector. Figure 8 shows a sample GLM log with CONTRAST statements specifying a full pairwise comparison of five treatment groups.

Figure 9

```
PROC GLM DATA=WORK.GLMDATA;
CLASS TMT;
MODEL ANGLTRAN=TMT;
CONTRAST "Treatment 0 vs. 1" TMT -1 1 0 0 0;
CONTRAST "Treatment 0 vs. 2" TMT -1 0 1 0 0;
CONTRAST "Treatment 0 vs. 3" TMT -1 0 0 1 0;
CONTRAST "Treatment 0 vs. 4" TMT -1 0 0 0 1;
CONTRAST "Treatment 1 vs. 2" TMT 0 -1 1 0 0;
CONTRAST "Treatment 1 vs. 3" TMT 0 -1 0 1 0;
CONTRAST "Treatment 1 vs. 4" TMT 0 -1 0 0 1;
CONTRAST "Treatment 2 vs. 3" TMT 0 0 -1 1 0;
CONTRAST "Treatment 2 vs. 4" TMT 0 0 -1 0 1;
CONTRAST "Treatment 3 vs. 4" TMT 0 0 0 -1 1;
run;
```

In each CONTRAST statement the base treatment group has a value of '-1' and the comparison treatment group has a value of '1'. The macro function to produce these CONTRAST statements is shown in figure 10. It contains three %DO loops ('i' loop for base treatment groups, 'j' loop for comparison treatment groups and the 'k' loop for coefficient values) and a nested set of %IF...%ELSE conditions.

Figure 10

```
%macro glmpairf_n=5,
    _label=Treatment,
    _effect=TMT;
%local i j k;
%do i=0 %to %eval(&n-2);
    %do j=1 %to %eval(&n-1);
        CONTRAST "&_label &i vs. &j" &_effect
            %do k=0 %to %eval(&n-1);
                %if (&k=&i) %then -1;
                %else if (&k=&j) %then 1;
                %else 0;
            %end;
        ;
    %end;
%end;
%mend;
```

The control of the semicolon is crucial. Within the 'j' loop the function issues the CONTRAST keyword, sets the label and then specifies the MODEL statement variable to be tested. There is no semicolon to close this line. Within the 'k'

loop the %IF and %ELSE statements have only one semicolon, so they close only as macro statements. They send a single value ('-1', '1' or '0') to the program vector to join the CONTRAST key word, label and variable name. When the 'k' loop is completed the semicolon that stands alone closes the CONTRAST statement. Since no step boundary has been encountered it is not executed, but held along with the GLM PROC, CLASS and MODEL statements. The program vector is then cleared and the 'j' loop starts another CONTRAST statement.

This function is useful, but is limited in that the values of the variable have a fixed integer step and must run from 0 to &n-1. We can generalize the function by passing a list of the variable values as a fourth parameter, as shown in figure 11.

Figure 11

```
%macro glmpairf_n=5,
    _list=0 2 4 6 8,
    _label=Treatment,
    _effect=TMT;
%local i j k;
%do i=1 %to %eval(&n-1);
    %do j=2 %to &n;
        CONTRAST
            "&_label %scan(&_list,&i) vs. %scan(&_list,&j)"
            &_effect
            %do k=1 %to &n;
                %if (&k=&i) %then -1;
                %else if (&k=&j) %then 1;
                %else 0;
            %end;
        ;
    %end;
%end;
%mend;
```

This only difference in this function from that shown in Figure 10 is that the label processing has now been changed to read words (with the %SCAN function) from the list passed as a parameter. As long as this word list contains all of the variable's values in ascending order the function can now handle any set of values of any data type- integer, floating point or character.

Simple pairwise comparison CONTRAST statements require a fixed pattern made up of only three possible values. Other CONTRAST statements have more elaborate sets of coefficients. Figure 12 shows a GLM log with trend test CONTRAST statements.

Figure 12

```
PROC GLM DATA=WORK.GLMDATA;
CLASS TRMT;
MODEL ANGLTRAN=TRMT;
CONTRAST "Treatment 0 vs. 1" TRMT -1 1 0 0 0;
CONTRAST "Treatment 0 vs. 2" TRMT -1 0 1 0 0;
CONTRAST "Treatment 0 vs. 3" TRMT -3 -1 1 3 0;
CONTRAST "Treatment 0 vs. 4" TRMT -2 -1 0 1 2;
run;
```

As trend test CONTRAST statements the basic structure of each line remains unchanged. However, the comparisons are limited only to the control (first) group versus all other groups, and the coefficients take different values.

Because odd numbered and even numbered comparisons do have similar coefficient structures (their step sizes are constant) we can use three basic rules to compute the coefficients. If 'i' is the comparison group location (2 to n) and 'j' is the coefficient location (1 to n) then we can compute each coefficient value as:

- 1) If the coefficient location is higher than the comparison group set the value to zero.
- 2) Else if the comparison group is odd numbered (step=2) compute the value as $(1-i)+(j-1)*step$
- 3) Else the comparison is even numbered (step=1) so compute the value as $(-i/2)+(j-1)$

Note that the SAS® macro %EVAL function performs only integer arithmetic so the divide in option 3 will round. The implementation of these formula in a macro function is shown in figure 13.

Figure 13

```
%macro glmtrnd(_n=5,
               _list=0 2 4 6 8,
               _label=Treatment,
               _effect=TRMT);

%local i j step;
%let step=2;
%do i=2 %to &_n;
  CONTRAST
    "&_label %scan(&_list,i) vs. %scan(&_list,&i)"
    &_effect
    %do j=1 %to &_n;
      %if (&j>&i) %then 0;
      %else %if (&step=2) %then
        %eval(%eval(1-&i)+%eval(%eval(&j-1)*&step));
      %else
        %eval(%eval(-&i/2)+%eval(&j-1));
      %end;
    ;
      %if (&step=1) %then %let step=2;
      %else %let step=1;
    %end;
  %mend;
%mend;
```

The primary difference between the pairwise and trend test CONTRAST statement generators is that the former can use totally fixed values, whereas the latter must compute each value other than the zeros that hold positions beyond the comparison group.

The functions shown in the logs are subsumed within single effect models but work equally well testing a single effect within a two effect model (e.g. treatment over time). Proc GLM does not require the programmer to provide coefficients for the second effect or the interaction term.

Other statistical procedures that use CONTRAST statements are not so simple to code as GLM CONTRAST statements. Proc CATMOD does not accept a full coefficient row. For comparisons of high to control the final value must be expressed by subtracting it from the other coefficients in the statement.

This can be accomplished by holding the coefficients in a macro variable word string. Once all coefficients values have been computed as for a GLM CONTRAST statement a final loop subtracts the last value from all other coefficient values. The coefficient values 1 to n-1 are then selected using a %SCAN function.

Summary

The complexity of the GLM trend test function shown in Figure 13 and its elaboration as discussed for proc CATMOD indicate that SAS® macro can handle relatively difficult code control problems. As a general rule it is best to keep macro functions relatively brief, structured and performing only one task.

The keys to writing good macro functions are good structured analysis and design, and a clear understanding of what is going on in the code stack and program vectors. For practical purposes SAS® macro has taken the main structuring features of the 'C' programming language and made them available to the SAS® programmer. If you can write good 'C' code you can write good SAS® macro code.

When should you consider using the macro facility? Macro should be of use if you:

- 1) find yourself repeatedly using the same small pieces of SAS® code
- 2) have a situation where the same analyses and/or reports must be generated for a number of variables or data sets.

- 3) need to control program sequence: repetition and conditional execution are basic in the macro language
- 4) ever write a SAS® program longer than 50 lines.

Point 4 is included to stress the importance of modular program structuring. Testing and maintaining large programs can consume a large proportion of a DP shop's budget. Macro can simplify these tasks. For purely monetary reasons SAS® macro should have a place in every SAS® programmer's toolbox.