A Tutorial on the SAS® Macro Language
John J. Cohen, Advanced Data Concepts LLC, Newark, DE

ABSTRACT
The SAS Macro language is another language that rests on top of regular SAS code. If used properly, it can make programming easier and more fun. However, not every program is improved by using macros. Furthermore, it is another language syntax to learn, and can create problems in debugging programs that are even more entertaining than those offered by regular SAS.

We will discuss using macros as code generators, saving repetitive and tedious effort, for passing parameters through a program to avoid hard coding values, and to pass code fragments, thereby making certain tasks easier than using regular SAS alone. Macros facilitate conditional execution and can be used to create program modules that can be standardized and re-used throughout your organization. Finally, macros can help us create interactive systems in the absence of SAS/AF.

When we are done, you will know the difference between a macro, a macro variable, a macro statement, and a macro function. We will introduce interaction between macros and regular SAS language, offer tips on debugging macros, and discuss SAS macro options.

MACROS
The Macros consist of collections of regular SAS program statements, macro variables, macro language statements, and macro functions contained within a %MACRO and a %MEND. The %MACRO statement includes a name and the macro is called using the macro’s name preceded by a “%”.

MACRO VARIABLES
Macro variables are indicated by preceding a variable name with an ampersand (&). Following regular SAS variable naming conventions, we will create a macro variable called &WHEN. We will assign a value to &WHEN when we call the macro. This value (“9409” in our example) will be substituted in the space taken up by the macro variable &WHEN at execution time. Previewing our discussion of macro options, with SYMBOLGEN on, your SAS log will note that:

SYMBOLGEN: Macro variable WHEN resolves to 9409
In Figure 2 when macro CLAIMREP executes, observations from dataset CLAIMS will be checked for a value of DATE equal to the value we assigned to &WHEN ("9409") or the prior month (&WHEN-1, which would resolve to "9408"). These will be output into dataset REPORT and printed. Note that we also use &WHEN in the TITLE statement, thereby automatically documenting which month is contained in the report. We can submit next month’s program by simply changing the macro variable value in the call to:

```sas
%CLAIMREP(WHEN=9810);
```

We can verify last year’s report with:

```sas
%CLAIMREP(WHEN=9709);
```

This construct allows us to enter a value (using &WHEN) into the program just once, even though we use the value three times. In a complicated program with many referenced to a particular parameter value, date range, contract, company, or the like, the chances of forgetting or mistyping one are remarkably high.

**MACRO STATEMENTS**

Our first macro statements were the %MACRO and %MEND. Other macro statements, often similar to their regular SAS equivalents, also start with the % symbol, such as %DO, %END, %IF%THEN, %ELSE, %GLOBAL, %LOCAL, and %LET. In the following example, we will run an annualized version of %CLAIMREP, once for each of the last ten years. We will use the macro language to generate ten copies of the program.

```sas
%MACRO CLAIMREP;
%LET CURYEAR = 98;
%DO YEAR = 89 %TO 98;
  %IF "&YEAR" = "&CURYEAR" %THEN %LET NOTE = FOOTNOTE "THRU &SYSDATE";
  PROC PRINT DATA = CLAIMS;
    WHERE YEAR = &YEAR;
    TITLE1 "ANNUAL REPORT FOR 19&YEAR";
    &NOTE;
  %END;
%MEND CLAIMREP;
%CLAIMREP;
```

**Figure 2 - Passing Values With Macro Variables**

```sas
%MACRO CLAIMREP(WHEN=);
DATA REPORT;
  SET CLAIMS;
  IF DATE = &WHEN THEN CURRENT = AMOUNT;
  ELSE IF DATE = (&WHEN - 1) THEN PRIOR = AMOUNT;
  ELSE DELETE;

  PROC PRINT DATA=REPORT;
    VAR CURRENT PRIOR;
    TITLE1 "MONTHLY CLAIMS REPORT FOR &WHEN";
    TITLE2 "AVOIDING "HARD-CODING"";
  %MEND CLAIMREP;

/* the macro call */
%CLAIMREP(WHEN=9809);
```
In Figure 3, the macro CLAIMREP will generate ten copies of the regular SAS program, one for each iteration of the %DO loop. Each copy will print a report for the year assigned to it by the %DO loop. The current year’s version (here set to “98”) will include a footnote describing the completeness of the data.

Note that we assign values to &CURYEAR and &NOTE using a %LET statement, and to &YEAR with a %DO statement (all without ampersands). &SYSDATE is an automatic macro variable created by SAS. In TITLE1, the phrase “19&YEAR” will resolve to “1989” the first time through the %DO loop, to “1990” for the second iteration, and on to “1998” in the last run through. Similarly, the WHERE statement will be comparing values of DATA step variable YEAR to a macro variable value of “89” the first time, “90” the second, and so on.

MACRO FUNCTIONS

Macro functions operate much as regular SAS functions, except that the arguments are within the context of the macro language. Most operate on character strings (e.g. %SUBSTR) or control the exact interpretation of macro special symbols (e.g., %STR). Macro variables are character variables so the %EVAL function may be required to perform arithmetic within macro statements. Figure 3 above included a simple instance of conditional execution (whether or not to create a value for the FOOTNOTE statement). In the macro system displayed in Figures 4A and 4B we will have the program elect to create a backup of the CLAIMS dataset if we are running the report at year’s end (i.e., month 12).

```
MACRO CLAIMREP(WHEN=); /* main macro */
%GLOBAL COUNT TOTAL;

    /* conditional macro call to auxiliary macro %BACKITUP */
%IF %SUBSTR(&WHEN,3,2) = 12 %THEN %BACKITUP;

DATA REPORT;
    SET CLAIMS END=EOF;
    RETAIN COUNT TOTAL 0;
    IF DATA = &WHEN THEN DO;
        CURRENT = AMOUNT;
        %ADDER; /* macro call to auxiliary macro %ADDER */
    END;
    ELSE IF DATE = (&WHEN - 1) THEN DO;
        PRIOR = AMOUNT;
        %ADDER; /* macro call to auxiliary macro %ADDER */
    END;
    ELSE DELETE;

IF EOF THEN DO;
    CALL SYMPUT(‘COUNT’,LEFT(PUT(COUNT,COMMA5.)));
    CALL SYMPUT(‘TOTAL’,LEFT(PUT(TOTAL’,DOLLAR5.2)));
END;

PROC PRINT DATA=REPORT;
    VAR CURRENT PRIOR;
    TITLE1 “MONTHLY CLAIM REPORT FOR &WHEN”;
    TITLE2 “&COUNT CLAIMS TOTALLING &TOTAL”;
    &TITLE3;
%MEND CLAIMREP;

%CLAIMREP(WHEN=9712); /* macro call */
```
The macro in Figure 4A generates our monthly claim report much as in Figure 2. Using macro %ADDER, we will accumulate a claim count and total amount only for the records we want. At the end of file, we will pass these sums to TITLE2 in the report. As we are running the program for December of 1997/end of the year (i.e. the month is “12”), we will also want to call the backup program (macro %BACKITUP).

Figure 4B - Macro Functions – Auxiliary Macros (%Adder and %Backitup)

```
%MACRO BACKITUP; /* conditional */
%LET YEAR = %SUBSTR(&WHEN,1,2);
%LET BEGYEAR = %EVAL(%YEAR * 100);
%LET ENDYEAR = %EVAL((&YEAR+1) * 100);
LIBNAME OUT “U367294.CLAIMS&YEAR..BACKUP”;
DATA OUT.CLAIMS&YEAR;
  SET CLAIMS;
  IF &BEGYEAR < DATE < &ENDYEAR;
%LET TITLE3 = TITLE3 "19&YEAR BACKED UP ON &SYSDATE”;
%MEND BACKITUP;

**************************************;

%MACRO ADDER; /* for illustration */
COUNT = COUNT + 1;
TOTAL = TOTAL + AMOUNT;
%MEND ADDER;
```

The %SUBSTR function allows us to distinguish between the year and month portions of &WHEN. The month value will cause macro %BACKITUP to execute (conditional upon the month being December). The year value is used in conjunction with the %EVAL function to produce begin and end values for selecting the data in the CLAIMS dataset to output. We use the CALL SYMPUT function (technically a DATA step statement) to take DATA step variables (COUNT and TOTAL) and output their values to macro variables of the same names.

Macro %ADDER will bring in a program fragment as needed. This saves a little repetitive typing, although a regular DATA step “LINK” or “GO TO” statement would also have served. The “LEFT” DATA step function in the CALL SYMPUT statement offers a measure of formatting control of the right-justified character value that will otherwise be displayed in TITLE2.

**DELIMITERS**

Along with the special symbols & and %, on occasion we need to consider delimiters, characters that indicate the end of a macro variable. We have blanks and equal signs (=) above. Ampersands and percent signs (& and %) also operate as delimiters, as does the period (.). In Figure 4 above, note that LIBNAME refers to data set “CLAIMS&YEAR..BACKUP”. The two periods assure us that at macro resolution time, the first period will be the delimiter for macro variable YEAR (and disappear!), while the second will remain as an operating system delimiter in a long data set name.

We will use macro %CENTER to center text strings for a report produced in a DATA _NULL_ step. Several delimiters are illustrated.
Figure 5 offers a good example of the use of a program fragment, a section of code that could be written using just regular SAS statements, but it would be much more messy. It is also a likely candidate to be saved as a program module in standard library at your site. Note the use of the macro function %STR to allow a macro token (the ampersand) to be passed and to serve as a delimiter. We also have two macro parameters in this macro (&TEXT and &WIDTH). We are allowing the latter to default to a value of “132” --- the width of a mainframe printer page --- but can override this for, let’s say, an LRECL of 80 as in this example:

```sas
PUT %CENTER(REPORT FOR &YEAR,WIDTH=80);
```

**DEBUGGING**

Identifying and fixing macro language errors can be quite a challenge. Good programming practice will help to limit the extent of your task. This includes designing your macro program carefully. Test regular programming steps to confirm that they work (and are not a source of apparent macro errors). Then build and test the macros incrementally, commenting out macro statements irrelevant to your current testing.

As macros get more complicated, be sure to pay special attention to your referencing environment, i.e., the arena in which macros, macro statements, variables, and values will be resolved. In a simple example in Figure 6 below, we attempt to evaluate the value of macro variable &COUNT before it exists. The resulting message in the log will soon become an old friend.

```sas
%MACRO CLAIMSREP;
DATA NULL;
SET CLAIMS END=EOF;
COUNT + 1;
IF EOF THEN CALL SYMPUT(‘COUNT’,LEFT(PUT(COUNT,5.)));
%IF &COUNT > 0 %THEN %DO;
/* additional SAS statements */
%END;
%MEND CLAIMSREP;
%CLAIMSREP;

WARNING: Apparent symbolic reference COUNT not resolved.
```
SAS will try to resolve the macro statement referencing the value of &COUNT before the DATA step—which creates &COUNT and assigns a value—executes. A RUN; statement immediately following the DATA step will correct this timing problem.

Liberal use of %PUT statements are a helpful way to keep track of the current value of a macro variable. These operate much as a DATA step “PUT” statement and write the indicated information in the SAS log. The following %PUT statement:

```
%PUT COUNT IS &COUNT;
```

results in:

```
COUNT IS 24
```

One useful feature of the %PUT is that it is valid anywhere in your program. Many other macro statements are acceptable only inside a macro, and misuse will leave you with a log message such as:

```
ERROR: The %IF statement is not valid in open code.
```

**MACRO OPTIONS**

Options most frequently used in debugging programs are MPRINT, MLOGIC, and SYMBOLGEN. All three result in enhanced log messages. MPRINT displays the resolved regular SAS code which will execute. MLOGIC traces macro execution. SYMBOLGEN will cause the value of macro variables to be automatically listed in the log. The first iteration of macro CLAIMREP from Figure 3 above (i.e., &YEAR=89) will result in the logs for each option, respectively, as shown in Figure 7.

<table>
<thead>
<tr>
<th>OPTIONS</th>
<th>MPRINT;</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPRINT (CLAIMREP): PROC PRINT DATA=CLAIMS;</td>
<td></td>
</tr>
<tr>
<td>MPRINT (CLAIMREP): WHERE YEAR = 89;</td>
<td></td>
</tr>
<tr>
<td>MPRINT (CLAIMREP): TITLE1 “ANNUAL REPORT FOR 1989”;</td>
<td></td>
</tr>
<tr>
<td>MPRINT (CLAIMREP): FOOTNOTE “THRU 06JUL98”;</td>
<td></td>
</tr>
</tbody>
</table>

```
************
**************;
```

<table>
<thead>
<tr>
<th>OPTIONS</th>
<th>MLOGIC;</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLOGIC (CLAIMREP): Beginning execution.</td>
<td></td>
</tr>
<tr>
<td>MLOGIC (CLAIMREP): %GLOBAL NOTE</td>
<td></td>
</tr>
<tr>
<td>MLOGIC (CLAIMREP): %LET (variable name is CURYEAR)</td>
<td></td>
</tr>
<tr>
<td>MLOGIC (CLAIMREP): %DO loop beginning: index variable YEAR; start value is 89; stop value 98; by value is 1.</td>
<td></td>
</tr>
<tr>
<td>MLOGIC (CLAIMREP): %IF condition “&amp;YEAR” = “&amp;CURYEAR” is FALSE</td>
<td></td>
</tr>
<tr>
<td>MLOGIC (CLAIMREP): %DO loop index variable YEAR is now 90; loop will iterate again.</td>
<td></td>
</tr>
<tr>
<td>MLOGIC (CLAIMREP): %IF condition “&amp;YEAR” = “&amp;CURYEAR” is TRUE</td>
<td></td>
</tr>
<tr>
<td>MLOGIC (CLAIMREP): %LET (variable name is NOTE)</td>
<td></td>
</tr>
</tbody>
</table>

```
**************************;
```

<table>
<thead>
<tr>
<th>OPTIONS</th>
<th>SYMBOLGEN;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYMBOLGEN: Macro variable YEAR resolves to 89</td>
<td></td>
</tr>
<tr>
<td>SYMBOLGEN: Macro variable CURYEAR resolves to 98</td>
<td></td>
</tr>
<tr>
<td>SYMBOLGEN: Macro variable NOTE resolves to FOOTNOTE “THRU 06JUL98”</td>
<td></td>
</tr>
</tbody>
</table>

Frequently a combination of these options is used in debugging programs. A result, though, can be very lengthy and busy logs. Once a program is fully-tested, you may want to turn these options off with:

```
OPTIONS NOMPRINT NOMLOGIC NOSYMBOLGEN;
```
WHEN MACROS REALLY HELP
The preceding examples have been fairly simple. To help illustrate the power of the macro language, we will first examine a simple utility macro in Figure 8. Based on the number of incoming non-blank drug codes, we will calculate the number of lines to reserve for footnote decodes at the bottom of each page of the report. The maximum number of drug codes, a function of the protocol, would normally be established earlier. In this example we will use a %LET statement to assign a value to macro variable &MAX.

Figure 8 - A Simple Utility Macro

```sas
/* concatenate drug code values */
%MACRO CONCAT(VAR_NAME,DIMENSN);
  %DO I = 1 %TO &DIMENSN;
  &VAR_NAME&I ||
  %END;
%MEND CONCAT;

/* calculate lines to reserve */
%GLOBAL RESERVE;
%LET MAX = 20;
DATA _NULL_;  
  SET DRUGS(KEEP=DRUG1-DRUG&MAX) END=EOF;
  ALL = COMpress(%CONCAT(DRUG,&MAX)'');
  IF ALL NE '' THEN RESERVE = CEIL(LENGTH(ALL) / 3);
  ELSE RESERVE = 0;
  IF EOF THEN CALL(SYMPUT('RESERVE',RESERVE));
RUN;
```

Here a macro call within the DATA step will concatenate the values of all the drugs in this protocol/study. The resulting value is then compressed with a DATA step function, evaluated for length of the entire new character string, reserve amount calculated (based on a formula derived from prior work), and the result is then passed to subsequent steps by placing this value in (GLOBAL) macro variable &RESERVE. This macro – in terms of the programming, looks like another SAS function and vastly simplifies our tasks.

A somewhat more complex use of macros in Figure 9 is designed to achieve results which would be considerably harder without the use of the macro language. We seek to capture information regarding the distribution of a test statistic for a particular sample where determining the appropriate function and then calculating the actual distribution of the statistic would be quite difficult. By accumulating PROC MEANS output for each of 99 iterations of a sample, we may instead simply measure the nature of our final dataset.

Figure 9A - A Not So Simple Set of Macros – The Analysis Data Set

```sas
DATA TRPK;
  /* the analysis dataset */
  SET TRPEAK;
  RENAME CTR = Y0
            CPK = Y1;
RUN;
```

We start by reading in the analysis data set, performing certain rename steps to get the variables lined up for the next step in our process.
Macro TTEST in Figure 9B creates T-scores and accumulates each successive run into DATA ALLT (if the results for that run pass statistical muster, i.e., the PVALUE is within the desired range). Using conditional execution, the very first run is used to create the initial dataset. All subsequent runs are appended (again, if the observations pass statistical muster).

Figure 9C - A Not So Simple Set of Macros – Macro REPORT

/* select title */

/* report results */

/* create PROC MEANS output */

/* if first iteration of sample */

/* all subsequent iterations */
The REPORT macro in Figure 9C uses conditional execution to assign the value for macro variable &TITLE, which is used subsequently in the Title of the report. The entire process is supervised by the CONTROL macro in Figure 9D.

### Figure 9D - A Not So Simple Set of Macros – Macro CONTROL

```sas
%MACRO CONTROL;
%DO T = 1 %TO 99;
    %IF &FIRST = %THEN %LET FIRST = YES;
    %ELSE %LET FIRST = NO;
    %TTEST;  /* call TTEST macro */
%END;

DATA _NULL_;  /* test continuity, capture min & max */
    SET ALLT END=EOF;
    RETAIN MIN 100 MAX 0 LAST 0;
    IF _N_ = 1 THEN LAST = ITERAT;
    ELSE DO;
        IF ITERAT NE (LAST + 1) THEN DO;
            CALL SYMPUT('CONTIN','NO');
            STOP;
        END;
        ELSE LAST = ITERAT;
    END;
    IF ITERAT < MIN THEN MIN = ITERAT;  /* test each observation for new minimum */
    IF ITERAT > MAX THEN MAX = ITERAT;  /* test each observation for new maximum */
    IF EOF THEN DO;
        CALL SYMPUT('MIN',MIN);  /* output final min, max to macro vars */
        CALL SYMPUT('MAX',MAX);
    END;
%REPORT;  /* call REPORT macro */
%MEND CONTROL;
%
```}

The %CONTROL macro in Figure 9D manages (controls) the entire process work flow, once the initial test data are read in. Macro %TTEST is called for 99 iterations of the statistic-generating step. The resulting output is combined in data set ALLT and after the last iteration, passed back to %CONTROL. The nature of the resulting distribution is measured (continuous or discontinuous, minimum and maximum) and placed in macro variables &CONTIN, &MIN, and &MAX, respectively. These macro variable values are processed further when macro %REPORT is called to determine the appropriate TITLE statement. This data-driven title is then printed along with the data.

### CONCLUSIONS

We have seen that the SAS macro language can add considerable power and sophistication to our work. We also have seen that there are enough differences in this syntax from our standard DATA step world that these new tools should be integrated into our tool sets in a measured fashion. Identify areas where your current approaches are tedious, repetitive, or otherwise fraught with a level of complexity that cries for a better solution. Where available, find good examples of working macro code from colleagues, SAS Institute resources and the many SAS user group conferences. Start slowly and incrementally, test and document thoroughly, make sure that you understand each successive step, and accumulate a library of useful programming tools.
REFERENCES
The macro language has proven exceedingly popular and any of the SAS user conferences' proceedings will most likely contain several papers on this topic. The SAS manual is another good resource. A very short list of references might include the following:


ACKNOWLEDGMENTS
SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

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

John Cohen
Advanced Data Concepts LLC
Newark, DE
(302) 559-2060
jcohen1265@aol.com

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *