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
Timing is the key to using macro functionality. If you misunderstand the timing, you will get unexpected results!

GOTCHA # 1
What do you expect for the values of the variable teenage_flag in the following program?

```
data heehaw;
  set blah;
  if age > 12 then do;
    %let teenager = Y;
  end;
  else do;
    %let teenager = N;
  end;
  teenage_flag = "&teenager";
run;

proc print data = heehaw;
  var name age teenage_flag;
run;
```
Since this program ran without error, but the results were not as we expected, there's obviously a problem, and the problem is called TIMING!

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

```sas
%let teenager = Y;
```

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 `teenager`, and stores Y 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 and then encounters another `%let`. Again, the entire `%let` statement is passed to the macro processor. Since the macro processor sees that a macro variable named `teenager` already exists in the symbol table, it simply replaces the value `Y` with `N`. The word scanner then continues parsing the code and passes the following to the execution stack.

```sas
end;
 teenage_flag = "N"
```

Since the word scanner encountered another macro trigger, it passes `&teenager` 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 data step.

At the completion of data step compilation, the execution stack contains:

```sas
data heehaw;
   set blah;
   if age > 12 then do;
      end;
   else do;
      end;
      teenage_flag = "N";
   run;
```

Now we know why every observation had a value of "N"!

**GOTCHA # 2**

In addition to the timing issues with macro processing, a typical misunderstanding revolves around “Who does what?” When you submit the following program, why isn’t the variable `next_year` in the output data set?

```sas
%macro test;
   %let update = Y;
   data gotcha;
      set sashelp.class;
      %if &update = "Y" and age = 12 %then next_year = 'Y';
      run;
   %mend test;
   %test;
proc print data = gotcha noobs;
run;
```

< Partial output >

<table>
<thead>
<tr>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>James</td>
<td>M</td>
<td>12</td>
<td>57.3</td>
<td>83.0</td>
</tr>
<tr>
<td>Jane</td>
<td>F</td>
<td>12</td>
<td>59.8</td>
<td>84.5</td>
</tr>
</tbody>
</table>
Although this example includes a macro definition and a macro call, the processing flow remains the same as previously described. When the macro call is executed, the macro processor passes all of the text in the macro definition (5 statements) to the execution stack. The %let statement is then passed to the macro processor and the macro variable update is created and assigned a value. Since they contain no macro triggers, the DATA and SET statements are placed in the execution stack. The entire %if statement is passed to the macro processor and requires a few steps to resolve:

- Macro variable update resolves to Y
- Since Y does not equal "Y", the evaluation results in FALSE
- Since the text age does not equal the text 12, the evaluation results in FALSE
- Therefore, the macro processor does not return the text next_year = 'Y' to the input stack.

Notice a few issues with the program:

- When assigning a value to a macro variable, you should avoid using quotes in the macro variable value.
- Quotes are rarely needed in a %IF evaluation, and then only if they ought to be part of the text comparison.
- The macro processor evaluated the expression age = 12. Since this occurred prior to the execution of the data step, no data step variables existed, so no "data check" was performed. If you want to evaluate the value of a data step variable, you must use a DATA step IF, not a macro %IF.

The correct coding of this statement should have been:

```
if &update = "Y" and age = 12 then next_year = 'Y';
```

### GOTCHA # 3
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.

### MACRO VARIABLES

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).
  - Sysdate contains the date that your SAS session (program) began. Ex: 27Mar06
  - Sysdate9 contains the same information in date9. format. Ex: 27Mar2006
  - 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:

```sas
%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.

```sas
%let code = %str(data a; i = 7; run;);
```

```sas
options symbolgen;
```

```
&code;
```

What would the value of the macro variable CODE be if we used the following?

```sas
%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.

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

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

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

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

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

```sas
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? This example lends itself nicely to a macro that enables you to feed in the table name, column name, data type, and macro variable name.
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 prefixed with a percent sign.

```sas
%macro sugi_macro;
  %put SUGI 31 rocks!!!;
%mend sugi_macro;
%sugi_macro;
```

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

```
SUGI 31 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.

```sas
%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.
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:

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.
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 3rd “gotcha” at the beginning of this paper ).

The months could be assigned manually via a series of \%let statements.
\%let mth1 = 200412;
\%let mth2 = 200501;
\%let mth3 = 200502;
\%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:

```sql
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 $Mth\text{nnn}$ 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.

```sas
%let mth1 = 200412;
%let mth2 = 200501;
%let mth3 = 200502;
%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.

```
%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 200502
MPRINT(PRINT): title "Sales 200502";
SYMBOLGEN: && resolves to &.
SYMBOLGEN: Macro variable I resolves to 3
SYMBOLGEN: Macro variable MTH3 resolves to 200502
MPRINT(PRINT): proc print data = sales_200502 noobs;
MPRINT(PRINT): run;

NOTE: There were 12 observations read from the data set WORK.SALES_200502.
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 200412";
proc print data = sales_200412 noobs;
run;

title "Sales 200501";
proc print data = sales_200501 noobs;
run;

title "Sales 200502";
proc print data = sales_200502 noobs;
run;
```
Partial Output:

Sales 200502

<table>
<thead>
<tr>
<th>month</th>
<th>region</th>
<th>office</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>200502</td>
<td>Ontario</td>
<td>100</td>
<td>$22,835</td>
</tr>
<tr>
<td>200502</td>
<td>Ontario</td>
<td>101</td>
<td>$119,763</td>
</tr>
<tr>
<td>200502</td>
<td>Ontario</td>
<td>102</td>
<td>$49,361</td>
</tr>
<tr>
<td>200502</td>
<td>Ontario</td>
<td>103</td>
<td>$32,024</td>
</tr>
<tr>
<td>200502</td>
<td>Western Canada</td>
<td>200</td>
<td>$113,777</td>
</tr>
<tr>
<td>200502</td>
<td>Western Canada</td>
<td>201</td>
<td>$119,663</td>
</tr>
<tr>
<td>200502</td>
<td>Western Canada</td>
<td>202</td>
<td>$67,034</td>
</tr>
<tr>
<td>200502</td>
<td>Western Canada</td>
<td>203</td>
<td>$65,641</td>
</tr>
<tr>
<td>200502</td>
<td>Maritimes</td>
<td>300</td>
<td>$6,147</td>
</tr>
<tr>
<td>200502</td>
<td>Maritimes</td>
<td>301</td>
<td>$8,218</td>
</tr>
</tbody>
</table>

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, found at the bottom of the page, will then pull the individual values from the &prodlist macro variable and use the individual values to build “data-driven” data steps.

```sas
%macro ValueList(ds = , var = , mvar = inlist);
    %global &mvar;
    proc sql;
        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:

```sas
%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, i.e.: the current value of &i and the characters + 1, a result of 1 + 1.

```sas
%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
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.

```sas
%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:

```sas
%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_;
   %end;
```

The examples shown are provided as an introduction to the subject matter. The detailed discussion of these topics will be included in the paper. The intent of these examples is to illustrate the use of function-style macros and their advantages over traditional methods. Further discussion and exploration of these topics will be provided in the paper.
if 0 then set yourlib.transaction nobs = nobs;
call symputx('nobs',nobs);   * v9 only ;
stop;

run;
%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));
   %sysfunc(attrn(&dsid,&attrib))
   %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_update 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_update
         %end; %else %
         %put No observations in &ds;
      %end; %else %
         %put &ds does NOT exist;
   %end;
%mend check_transaction;

%check_transaction(yourlib.transaction);

MISCELLANIES
Helpful tidbits with no other home.

<table>
<thead>
<tr>
<th>Description</th>
<th>v8</th>
<th>v9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check if a macro variable exists</td>
<td>search SAS-L archives for a &quot;macro variable exist whitlock&quot;</td>
<td>%symexist(mvar) or symexist(mvar)</td>
</tr>
<tr>
<td>Delete a macro variable</td>
<td>To assign a NULL value, use %let mvar =;</td>
<td>%symdel(mvar) or symdel(mvar)</td>
</tr>
<tr>
<td>Define macro variable scope</td>
<td></td>
<td>%global %local</td>
</tr>
</tbody>
</table>
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
  sugi@stratia.ca

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 trademarks of their respective companies.