Paper TT05

Job Security:
Using the SAS® Macro Language to Full Advantage

Arthur L. Carpenter
California Occidental Consultants, Oceanside, California

ABSTRACT
As has been discussed in other papers on the topic of Job Security, those working in the field as Job Security Specialists must always be ready to take advantage of all types of programming techniques. Fortunately the SAS® Macro Language contains many hidden gems that can provide many hours of programming pleasure.

By examining how these macro language gems relate to the code that is generated by the macro language, the specialist can gain further insight and understanding into why things work the way that they do. This paper will present a few of these gems, how they relate to the DATA/PROC step programmer, and how the specialist can take full advantage of them.

KEYWORDS
Macro variables, %SCAN, macro comments, statement style macros, macro variable collisions

INTRODUCTION
As a code generator the macro language has a number of advantages to the Job Security Specialist. Not the least of these, can be the subtle implementation of security techniques such as those described in other papers on this topic (Carpenter,1993 as well as Carpenter and Payne, 1998 and 2001). However due to the depth of the macro language it is not limited to the application of these 'tried and true' techniques. The complexities of the macro language and the way that it interfaces with the DATA and PROC steps gives the specialist additional and sometimes extremely interesting tools.

SYSTEM OPTIONS
Turning off Debugging Options
The system options MPRINT, MLOGIC, SYMBOLGEN, and MFILE are especially useful when debugging macros. Turning these options off removes virtually all debugging notes. Conversely if your macro is running very well you might consider turning them on as they can create very extensive LOGs.

Autocall Without SASAUTOS
The macro language includes a number of functions such as %LEFT, %TRIM, %LOWCASE, and %VERIFY that are actually stored in the SAS autocall library. These functions become unavailable if the autocall library is not available. You can turn off the library all together by specifying the NOMAUTOSOURCE. Better yet leave the library turned on (MAUTOSOURCE), however do not include the automatic *fileref* SASAUTOS in the library designation. The following OPTIONS statement makes sure that the autocall facility is turned on without making the SAS autocall library available.

```
options mautosource sasautos=(myprojmac grpmaclib ~~sasautos~~);
```

Compiled Macro Libraries without Source Code
Prior to SAS9, when storing macros in a compiled stored macro library, it was not possible to store the source code as part of the library. Since SAS9 includes an option to store the code, and since the availability of this code makes the job of the Security Specialist more difficult, you should double check the %MACRO statement to make sure that the SOURCE option is not used.

```
%macro aerpt(dsn, stdate, aelist) / store ~~source~~;
```

Implied or Statement Style Macros
Implied or statement style macros were available before the current macro language was implemented. Although still available these macros are rarely used as they are resource intensive and their programming syntax can cause confusion. Of course it is the latter advantage that is appealing to the Job Security Specialist. Calls to implied macros do not use the percent (%) sign and consequently can be easily confused with Base language statements.

DATA STEP EXPECTATIONS

Assigning a Data Step Value Using %LET

Since macro language statements are executed before the DATA step is even compiled it is not possible to assign a the value of a DATA step variable value to a macro variable using the %LET statement. This of course should not discourage the Job Security Specialist, because many programmers new to the use of the macro language are unfamiliar with these timing issues. The data set SASHELP.CLASS has values for the variable NAME for each observation.

```
data new;
   set sashelp.class;
   %let fname = name;
   if "&fname" < 'C' then put "Name starts with A or B " name=;
   run;
```

Of course the %LET is executed before there are any values assigned to the PDV. This means that the text in the %LET is not seen as a variable name but merely as the letters n-a-m-e. In fact, since the macro variable &FNAME is assigned the text value n-a-m-e, and since letter lowercase n sorts after all uppercase letters, the IF is never true.

Conditional Execution of a %LET and %PUT

Another technique is to attempt to conditionally assign a value to a macro variable using the %LET. In the following code we want to change the value of the macro variable &TEENS to `teenager` only if there are one or more teenagers in the incoming data table.

```
data new;
   set sashelp.class;
   *** Determine if this student is a teenager;
   if age > 13 then %let teens = teenager;
   ***;
   run;
```

As it turns out the %LET is executed before the DATA step is compiled and will always have the value of `teenager` regardless of the values taken on by AGE. In this case the IF statement becomes:

```
   if age > 13 then ***;
```

A simple extension of this concept and one that can add a great deal to the specialist's list of subtle techniques might use a %PUT instead of the %LET. In the following example the IF conditionally executes the assignment statement.

```
data new;
   set sashelp.class;
   if age > 15 then %put 'Driving Age';
   /* Adjust weight for all students*/
   weight = weight + 5;
   run;
```

Since the %PUT is executed first and the comment is ignored, the IF effectively becomes:

```
   if age > 15 then weight = weight + 5;
```

%SCAN with Quotes

Since macro variables are preceded with an ampersand, text does not need to be quoted like it does in the DATA step. In the following example &NAMES contains a list of last names of the people of interest in a study. The names are always separated with a blank. We want to use the %SCAN function to isolate the second name in the list.

```
%let names = Smith Smithers Smothers;
%let lname = %scan(&name,2,' ');
```

As anticipated this code performs as anticipated almost all the time. Since 'almost all the time' is a code phrase for 'Job Security Opportunity', we need to better understand what is happening here. The 'almost' will be observed if the last name contains an apostrophe, such as O'Mally.

```
%let names = Ogden O'Hare O'Mally;
```

```
%let lname = %scan(&name,2,' ');
```

&LNAME contains only O since the quote marks are also seen as word delimiters. The %SCAN function would be correctly specified as:

```
%let lname = %scan(&name,2,%str( ));
```

Using * to Comment a Macro Statement

It is not unusual to comment out a macro statement by using an asterisk style comment. Let's say that the user would like &DSN to contain the value of `mydata`, the following %LET statements will do the trick since the first and last %LET statements have been commented out.

```
*%let dsn = clinics;
%let dsn = mydata;
*%let dsn = testdat;
```

This works fine in open code, but something odd (also a code word for 'Job Security Opportunity') happens when these same three statements are included inside of a macro definition.

```
%macro testit;
   *%let dsn = clinics;
   %let dsn = mydata;
   *%let dsn = testdat;
   %put dsn is &dsn;
%mend testit;
%testit
```

The %PUT statement shows that the value of &DSN is `testdat`. Even though the Enhanced Editor marks the lines to be commented in green, all three %LET statements are executed (the final definition is the one that sticks). The * are then applied (after the %LET and %PUT have executed). As an aside, this could have other Job Security ramifications as the call to %TESTIT will resolve to a pair of asterisks (**).

LANGUAGE COMPLEXITY

Using Triple Ampersands, &&&var

Macro variable references with three ampersands are indirect references to another macro variable. In other words a macro variable of the form of &&&VAR holds the name of a macro variable that in turn holds the value of interest. By adding another layer of indirect references we have of course increased the complexity of our code. It turns out that there can be some minimal efficiency gains by using triple ampersands, however by far the greatest gain is in the area of Job Security.

The macro %NAMEONLY returns the second word in the parameter that is passed into the macro. In this case we want to extract the name portion of a two level table name. The macro variable &DATASET is created and then its value is passed into the macro where the data set name is written on the local symbol table with the macro variable name &DSN.

```
%macro nameonly(dsn);
   %scan(&dsn,2,.)
%mend nameonly;

%let dataset=sasclass.clinics;
%put name is %nameonly(&dataset);
```

From the Job Security standpoint, it is a bit too clear what is going on, and who really cares that the data set name is now stored in two locations (&DATASET and &DSN)? We can use a triple ampersand macro variable to save a storage location and, more importantly, make the code more complex.

```
%macro nameonly(dsn);
   %scan(&&&dsn,2,.)
%mend nameonly;

%let dataset=sasclass.clinics;
```

```
%put name is %nameonly(dataset);
```

Now we are no longer passing in the data set name but rather the name of the macro variable that holds the data set name. &&&DSN resolves to &DATASET which resolves to the data set name. We no longer are storing the name in two locations.

Using Quadruple Ampersands, &&&&var

There are numerous examples in the SUGI literature of papers with topics like "A Six Ampersand Solution, and Why It is Cool". Of course you already know that two adjacent ampersands resolve to one ampersand. In fact if there are an even number of ampersands in front of the macro variable name, there might as well only be one. Perhaps the programmer that had the six ampersand solution was a Job Security specialist and did not know it. When you see a macro variable of the form &VAR, consider changing it to &&&&&&VAR.

Macro Arrays, &&var&i

While the macro language does not really support the concept of a macro array, we often use the &&VAR&I construct to mimic a macro array. Effectively there are three ampersands, so we have a macro variable that holds the name of another macro variable. A series of macro variables with the form &var1, &var2, etc. form a vector of values which have been indexed with the macro variable &i.

The use of this type of macro array is common, but it is generally not intuitive to most users. Even many macro programmers that are fairly advanced, often have to stop and think about how the arrays work. This, of course, means that a Job Security Specialist should become familiar with their usage.

Multi-subscripted Macro Arrays

The macro array described above uses a single macro variable as an index to point to the requested array element, and since there is a single index this type of array contains a vector of values. If we add additional indexes we can create multi-dimensional arrays. Within the coding for these multi-subscripted macro arrays can be found a number of Job Security opportunities. Some of the complexities of the topic are discussed in Carpenter (2004, Section 13.2).

Depending on the situation, a two dimensional macro array might be addressed as: &&var&i&j, &&var_r&i._c&j, or even &&&var&i. As you can easily see keeping track of two indexes adds programming complexity and hence Job Security.

Symbol Tables and Macro Variable Collisions

Macro symbol tables have the wonderful property of allowing the same macro variable name to appear in any number of symbol tables. Since a given macro variable name can, at any given time, hold different values in different symbol tables, it is very possible to cause a macro to write the value of a macro variable to the 'wrong' symbol table.

The rules used by the macro facility to decide which symbol table is to be used can be fairly arcane (Carpenter, 2004, Section 13.3.5), and thus an opportunity for the Job Security Specialist. In the following example the macro variable &i is used in both of the macros, however since there is no %LOCAL statement in %CHKSURVEY, &i will only appear once - in the symbol table for %PRIMARY. As a result the %DO loop in %PRIMARY will **NEVER** be executed for &i = 2, 3, 4, 5, or 6.

```
%macro chksurvey(dset);
   %do i = 1 %to 5;
      .... code not shown....
%mend chksurvey;

%macro primary;
   .... code not shown....
   %do i = 1 %to &dsncnt;
      %chksurvey(&&dsn&i)
   %end;
   .... code not shown....
%mend primary;
```

Macro variable collisions are discussed in more detail in Carpenter (2005).

INTERESTING MACRO LANGUAGE ATTRIBUTES

Using Characters with Attitude

There are a number of characters and combinations of characters that have special meaning to the macro parser even though they are not strictly part of the macro language. Much like a teenager that is starting to rebel, these characters seem to have a bit of attitude. We know for instance that quote marks must still come in pairs so some of

the techniques that are commonly used in the Base language can also be applied in the macro language. These include blurring techniques such as:

```
%let dsn1 = 'clinics;
%let dsn2 = olddata';
```

Of course, because of the way it colors the code, the Enhanced Editor tends to make these techniques a bit difficult, so we do need to be a bit more subtle. The following %IF works just fine - usually.

```
%if &state = CA %then %do;
```

If &STATE happens to take on the two digit postal code for Oregon (OR), Nebraska (NE) or in SAS9 Indiana (IN) the %IF results in an error.

```
%if OR = CA %then %do;
```

The OR will be seen, not as an abbreviation for Oregon, but as a comparison operator. We could prevent the OR from being seen as a comparison operator by using a quoting function..

```
%if %bquote(&state) = CA %then %do;
```

Quoting functions are used to mask the attitude of these characters, however not all quoting functions work the same and often the differences are subtle (do I hear a Job Security code word?). The %STR and %QUOTE functions mark some of the special characters with a special masking character which just happens to be a percent sign. Since the macro language already uses a % sign this provides an opportunity for confusion. The macro %CALC is not called in the following %LET:

```
%let result = %str(%%calc);
```

Quoting Functions and Invisible Quotes

Quoting functions mark text with invisible quote marks. Since these marks are hard to see, it is possible to quote a macro in such a way that the user is unaware that it has been quoted. Suppose that somewhere in your program you create the following macro variable &PLACE as shown here:

```
%let city = Phoenix;
%let place = %nrstr(&city);
```

Whenever the &PLACE is used it will resolve to &CITY, however it will resolve no further. This can be 'special' for someone who is expecting the macro variable to resolve to Phoenix, and can see no reason why it should not resolve.

Non-integer Comparisons

When the macro language makes a comparison, it first checks to determine if it should be a numeric or alphabetic comparison. This is not necessary in the Base language since SAS knows if it is dealing with numeric or character variables. The macro language checks both sides of the comparison operator for integers. Remember that an integer has no decimal point (1.0 and 1. would both be non-integer). If both sides of the comparison are integers a numeric comparison is made.

```
%if &wt1 lt &wt2 %then %do;
```

If either &WT1 or &WT2 is not an integer the values will be compared alphabetically. This can be handy since

```
%if 10.1 lt 2.1 %then %do;
```

would result in a TRUE determination. When you know both values will be numeric and one or both might be a non-integer, you should avoid using the %SYSEVALF function which would force a numeric comparison.

```
%if %sysevalf(&wt1 lt &wt2) %then %do;
```

Comparison of a Range

In the Base language we might compare a range of values by coding a composite comparison.

```
if 21 le age le 80 then do;
```

When we translate this same technique into the macro language we can have some unexpected ☺ results. The statement:

```
%if 21 le &age le 80 %then %do;
```

will not, as we would expect based on our DATA step knowledge, be interpreted as:

```
%if 21 le &age and &age le 80 %then %do;
```

instead it is interpreted as:

```
%if (21 le &age) and 80 %then %do;
```

The portion `(21 le &age)` will be evaluated first and will result to a zero or 1. If &AGE was 150 the %IF becomes:

```
%if (21 le 150) and 80 %then %do;
```

which becomes:

```
%if 1 and 80 %then %do;
```

which will be true. Composite range comparisons must always be coded as separate comparisons.

```
%if (21 le &age) and (&age le 80) %then %do;
```

Composite Macro Calls

Sometimes you may want to store all or part of the name of the macro which is to be executed in a macro variable. These technique by itself has some value to the Job Security Specialist, however as an added bonus different versions of SAS handle the resolution of the macro call differently (Carpenter, 2004, Section 13.3.5).

```
%let dsn = cm;
%read&dsn;
```

Versions 5.18 and 8.0 will attempt to execute %READ before resolving &DSN. Versions 6, 7, 8.2 and SAS9 will resolve &DSN first and attempt to execute %READCM. You gotta love stuff like this, especially when you are working concurrently with different versions of SAS.

Using CALL EXECUTE

The CALL EXECUTE DATA step routine allows the programmer to control the timing of the execution of certain macro calls and macro statements during DATA step execution. This can be important because normally the compilation of macro calls and statements takes place before the compilation of the DATA step. Since the topic of timing of the events of the macro language is not well understood by most users, adding another layer of complexity can make for even more interesting programming. The CALL EXECUTE routine is discussed in more detail in Carpenter (2004, Section 6.6).

One way to maximum the effect of these timing issues is obtained when CALL EXECUTE is used to call a macro that contains a mixture of macro and Base language statements.

SUMMARY

The macro language adds power and flexibility to SAS, and with the power and flexibility comes a number of opportunities for the Job Security Specialist. The specialist that is conversant in the macro language and the way that it interfaces with the Base language can capitalize on a great many techniques that the other Job Security Specialists will be unable to utilize.

ABOUT THE AUTHOR

Art Carpenter's publications list includes three books, and numerous papers and posters presented at SUGI and other user group conferences. Art has been using SAS® since 1976 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Professional™ and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
P.O. Box 430

Oceanside, CA 92085-0430

(760) 945-0613
art@caloxy.com
www.caloxy.com

REFERENCES

Carpenter, Art, 2004, *Carpenter's Complete Guide to the SAS® Macro Language 2nd Edition,* Cary, NC: SAS Institute Inc.,2004. Most of the examples in this paper were taken from this text.

Carpenter, Arthur L., 1997, *Resolving and Using &&var&i Macro Variables*, presented at the 22nd SAS User's Group International, SUGI, meetings (March, 1997) and published in the Proceedings of the Twenty-Second Annual SUGI Conference, 1997.

Carpenter, Arthur L., 1993, Programming for Job Security: Tips and Techniques to Maximize Your Indispensability, selected as the best contributed paper, presented at the 18th SAS User's Group International, SUGI, meetings (May 1993) and published in the Proceedings of the Eighteenth Annual SUGI Conference.

Carpenter, Arthur L. and Tony Payne, 1998, Programming For Job Security Revisited: Even More Tips and Techniques to Maximize Your Indispensability, presented at the 23rd SAS User's Group International, SUGI, meetings (March, 1998) and published in the Proceedings of the Twenty-Third Annual SUGI Conference, 1998.

Carpenter, Arthur L. and Tony Payne, 2001, A Bit More on Job Security: Long Names and Other V8 Tips, presented at the 8th Western Users of SAS Software Conference (September, 2000), 26th SAS User's Group International, SUGI, meetings (April, 2001), and at the Pharmaceutical SAS User Group meetings, PharmaSUG, (May 2001). The paper was published in the proceedings for each of these conferences.

Carpenter, Arthur L. , 2005, Make 'em %LOCAL: Avoiding Macro Variable Collisions, presented at the Pharmaceutical SAS User Group meetings, PharmaSUG, (May 2005).

TRADEMARK INFORMATION