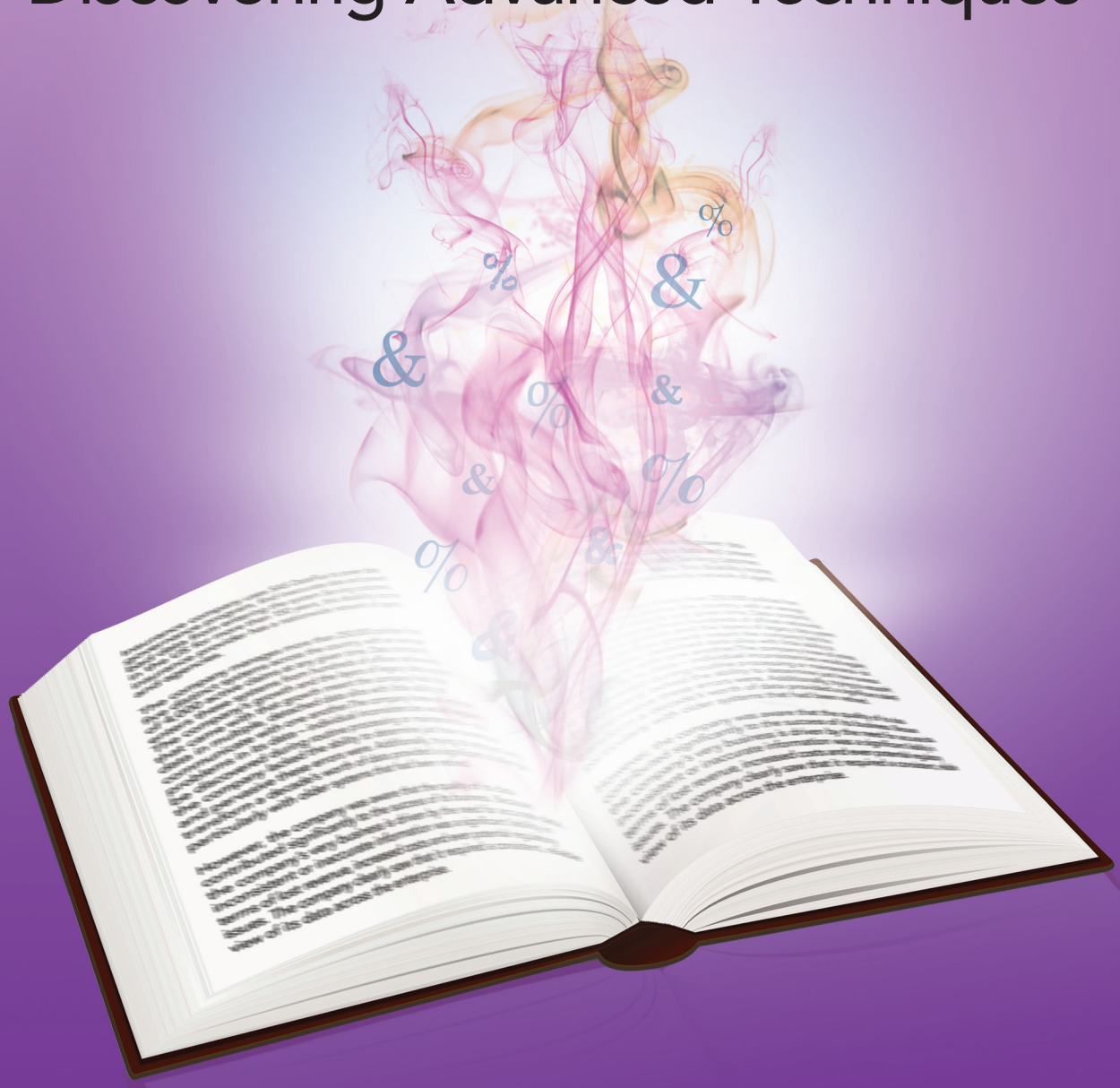
sas

SAS® Macro Language Magic

Discovering Advanced Techniques



Robert Virgile

Part 1: Preparation

What makes a macro look like magic? How can a short macro enable an end user to generate sophisticated, accurate results? To write such macros, programming wizards apply their knowledge, boosted by creativity and years of experience. Of course, preparation begins with:

- Reading
- Taking courses
- Programming

But magical results require more preparation than that. Behind the scenes, these same programming wizards also:

- Strive to write simple programs.
- Keep up with advances in the software.
- Experiment to determine how the software really works.

They appreciate the relationship between the SAS language and macro language. The vast majority of macro applications generate SAS code. As a result, any steps that improve your SAS skills will broaden the range of SAS programs you can envision and increase the value of the macro language. These steps go hand in hand:

- Expand your knowledge of SAS software.
- Select a simpler, more appropriate SAS tool to accomplish a programming task.
- Write simpler, more powerful macros.

The first two chapters of this book illustrate these preparatory steps. Chapter 1, “SAS Language Preparation,” focuses on the SAS language with a little bit of macro language mixed in. Chapter 2, “Shifting Gears: Macro Language,” shifts the focus more heavily into the macro language.

Chapter 1: SAS Language Preparation

Macro language is useless unless you can visualize the SAS program that results. So we begin by examining the three learning steps and how they apply using the SAS language. This chapter emphasizes simplicity, keeping current, and experimentation.

1.1 Keep It Simple

Perhaps you have never seen:

- A one-page macro to perform character-to-numeric conversion (instead of using the INPUT function)
- A multi-page program to present three-dimensional tables in two dimensions (instead of using the LIST option within PROC FREQ).

Unfortunately, such complex programs exist. Make a point of simplifying your programs so that they won't need to be added to this list.

This first simplification uses statements that might appear in a DATA step. The original version is overly complex:

```
data last_one;
   set municipalities;
   by state county town;
   if last.state or last.county or last.town;
run;
```

When LAST.STATE or LAST.COUNTY is true, the condition LAST.TOWN must also be true. Therefore, a somewhat simpler program will generate the exact same result:

```
data last_one;
   set municipalities;
   by state county town;
   if last.town;
run;
```

Without macro language, the difference is trivial. But suppose a macro parameter holds the list of BY variables:

```
by_varlist=state county town,
```

If macro language has to generate one program versus the other, the workload becomes very different. We will revisit this scenario in Chapter 2.

Common programming tasks abound with opportunities for simplification. Compare these two ways to conclude a DATA step:

```
if eof then do;                  if eof;
   average = total / n;          average = total / n;
   output;                       run;
end;
run;
```

The shorter ending generates the same result. A more complete application calculates means within BY groups. A few different styles exist, including this common one:

```
data totals;
   set dairy;
   by cow;
   if first.cow then do;
      tot_milk=0;
      milkings = 0;
   end;
   tot_milk + (2*pints + 4*quarts);
   milkings + 1;
   if last.cow;
   average_milking = tot_milk / milkings;
run;
```

However, a variation in style generates a shorter program:

```
data totals;
   set dairy;
   by cow;
   tot_milk + (2*pints + 4*quarts);
```

```
   milkings + 1;
   if last.cow;
   average_milking = tot_milk / milkings;
   output;
   tot_milk=0;
   milkings=0;
run;
```

The revised version might look a little awkward at first. But besides being a line shorter, it removes a level of indentation and actually runs slightly faster. Notice how the first version checks every observation for both FIRST.COW and LAST.COW, but the revised version checks only for LAST.COW.

Consider this all-too-common case of overly complex code.

1.1.1 Programming Challenge #1

Simplify the program, eliminating any unnecessary steps:

```
proc sort data=midwest;
   by state;
run;

data midwest;
   set midwest;
   counter=1;
run;

proc means data=midwest sum maxdec=0;
   class state;
   var counter;
   title 'Number of Observations for Each Midwest State';
run;
```

1.1.2 Solution

Certainly, the PROC SORT is unnecessary. The CLASS statement works with unsorted data. But there is more to eliminate. If you are having trouble locating the extra complexities, focus on the title before reading any further.

The key is figuring out what the program is trying to achieve. Because it counts the number of observations for each STATE, this replacement might come to mind:

```
proc freq data=midwest;
   tables state;
   title 'Number of Observations for Each Midwest State';
run;
```

A number of themes are emerging here. Choose simple, direct SAS tools. The simpler the SAS code, the simpler the macro version will be. Increase your knowledge of SAS to expand the variety of tools available for tackling a given programming problem. The more tools that you have at your disposal, the better your chances of constructing a simpler looking program. And the simpler the SAS code, the easier it is to write and to maintain the macro version.

Along those lines, here are a few suggested topics for independent self-study. Learn how to:

- Use the CNTLIN= option on PROC FORMAT to create a format from a SAS data set.
- Use the Output Delivery System (ODS) to capture procedure output as a SAS data set.
- Extract information from dictionary tables.

These topics will serve you well, even if you never venture into the world of the macro language.

1.2 Keep Up with the Software

Keep up with advances in the software. Even though SAS is continually advancing the software, many of the advances are not key breakthroughs. Rather, they fall into the category of nice (rather than essential) tools that make programming easier.

Begin with a simple example. The LENGTH function has a special feature:

```
len_name = length(name);
```

When the incoming string is blank, the function returns a 1 instead of a 0. This code would bypass that special feature:

```
if (name > ' ') then len_name = length(name);
else len_name=0;
```

However, a newer function eliminates the need for this bypass operation:

```
len_name = lengthn(name);
```

Unaware of this new feature, a macro programmer might code:

```
if (&varname > ' ') then len_&varname = length(&varname);
else len_&varname = 0;
```

But the macro version should be simpler:

```
len_&varname = lengthn(&varname);
```

Of course the macro version assumes that incoming variable names are no longer than 28 characters, so that LEN_&VARNAME contains a maximum of 32 characters.

Let's move on to another outdated example. Renaming a list of variables used to be tedious:

```
data new;
   set old (rename=(v1=var1 v2=var2 v3=var3
                    v4=var4 v5=var5 v6=var6
                    v7=var7 v8=var8 v9=var9
                    v10=var10));
   * more processing;
run;
```

If a macro were to perform the renaming, the DATA step might look like this:

```
data new;
   set old (rename=(%RENAMING(prefix_in=v, prefix_out=var,
                              first_num=1, last_num=10)));
   * more processing;
run;
```

The macro would be especially handy if there were large numbers of variables to rename or many sets of variables to rename. Here is one form that such a macro might take:

```
%macro renaming (prefix_in=, prefix_out=,
                 first_num=1, last_num=);
   %local i;
   %do i=&first_num %to &last_num;
       &prefix_in.&i=&prefix_out.&i
   %end;
%mend renaming;
```

In fact, there was a time when this level of complexity was necessary. But advances in the software have made it obsolete. Now the software can easily rename a list of variables:

```
data new;
   set old (rename=(v1-v10=var1-var10));
   * more processing;
run;
```

In the next example, the objective is to get rid of variable labels. This step might be needed in PROC MEANS, where the output prints the variable label as well as the variable name. If the labels are wide, the report might no longer fit each variable's information on a single line. The output gets split into sections instead. A simple way to condense the report is to wipe out the variable labels:

```
label var1=' '
      var2=' '
      var3=' '
```

```
        var4=' '
         ...
        varN=' ';
```

Most shorter approaches generate an error message, including:

```
label _all_=' ';
```

A macro could capture the names of all numeric variables in the data set and generate a LABEL statement for each. But a simple feature makes such a macro unnecessary:

```
attrib _all_ label=' ';
```

Just knowing that the simple tool is available makes writing a macro unnecessary.

The next example selects the last word from a series of words. For example, begin with this variable:

```
list_of_words = 'state county town';
```

The DATA step must select the word TOWN as the last word in this list. Here is the DATA step code, using the old-fashioned approach:

```
i=0;
length word $ 32;
if (list_of_words > ' ') then do;
   do until (word=' ');
      i + 1;
      word = scan(list_of_words, i, ' ');
   end;
   word = scan(list_of_words, i-1, ' ');
end;
```

When the SCAN function reaches past the end of the incoming list, it returns a blank. By reaching past the end of the list, the program can then go back and retrieve the last word.

Occasionally, an objective requires such complex code. But in this case, clever programming can dramatically shorten the code from nine statements to two:

```
length word $ 32;
word = reverse(scan(reverse(list_of_words), 1, ' '));
```

Reverse the original string, select the first word, and then reverse the characters in that first word. But advances in the software can simplify the code even further:

```
length word $ 32;
word = scan(list_of_words, -1, ' ');
```

When the second parameter contains a negative number, the SCAN function reads the list from right to left, instead of from left to right. By keeping up with advances in the software, a tricky problem becomes easy.

Consider one final example that works with the COMPRESS function. The existing variable PHONE_NUM holds phone numbers. But they are not in a standard format. These versions (as well as others) might appear in the data:

```
(123) 456-7890
123 456 7890
123 456-7890
123-456-7890
[123] 4567890
```

In order to transform all phone numbers into an identical format, you might begin by removing non-numeric characters. By guessing well, you might eliminate all non-numeric characters:

```
phone_num = compress(phone_num, '-() []');
```

Of course, there are only 256 possible characters, so the last parameter might specify the full set of 246 non-numerics. However, even that solution might not be transferable across operating systems. A better solution would automate instead of guessing:

```
phone_num = compress(phone_num, compress(phone_num,'1234567890'));
```

The interior COMPRESS function identifies all non-numeric characters, and the exterior COMPRESS function removes them.

But SAS 9 makes this task easy to accomplish and easy to read:

```
phone_num = compress(phone_num, '0123456789', 'K');
```

The third parameter indicates that the function should keep, rather than remove, characters found in the second parameter. In fact, that third parameter is quite powerful and supports:

```
phone_num = compress(phone_num,, 'KD');
```

The third parameter uses K for “keep” and D for “digits”. So the program doesn’t even have to spell out the list of digits.

Keeping up with the software will often enable you to write simpler programs.

1.3 Experiment

Whether the subject is cooking, sports, or programming, book learning has limitations. Testing and experimenting enrich the learning process far beyond what reading can do. Expect dismal failures. Expect delightful surprises. Expect to learn by experimenting.

Today's experiment is called "How Low Can You Go?" Can you assign a value to a variable such that either of these comparisons would be true:

```
if numval < . then put 'Found a small numeric!';
if charval < ' ' then put 'Found a small character!';
```

Book learning helps with the numeric value. The software supports 27 special missing values:

```
._ .A .B … .Z
```

Surveys routinely utilize these values to differentiate among various reasons a respondent failed to answer a question. For example, a survey might assign these values for respondents' answers:

1 = strongly agree
2 = agree
3 = neither agree nor disagree
4 = disagree
5 = strongly disagree

It would be inconvenient to assign these values as well:

6 = refused to answer
7 = not applicable
8 = departed the survey early

In that case, subsequent analyses would have to deal with values of 6, 7, and 8 separately for each question. Instead, this scale might be easier:

.A = refused to answer
.B = not applicable
.C = departed the survey early

Now a PROC MEANS would automatically exclude all forms of missing values from its calculations.

Are any of these special missing values lower than a traditional missing value? It turns out that just one of them is lower: ._ is lowest. So these statements would generate a true comparison:

```
numval = ._;
if numval < . then put 'Found a small numeric!';
```

Experimentation would confirm the order of special missing values, but it takes reading to discover that they exist. On the other hand, test programs reveal that there are 32 character values smaller than a blank:

```
data _null_;
   charval =' ';
   put charval $hex2.;
run;
```

In an ASCII environment, the PUT statement writes 20. So the hex codes ranging from 00 through 1F would all be lower than a blank. You won't find these characters on a keyboard, but you certainly can assign them in a program:

```
data test;
   do i=32 to 0 by -1;  /* when i is 32, hex code 20 = blank */
      hexcode = put(i, hex2.);
      charval = input(hexcode, $hex2.);
      output;
   end;
run;

proc sort data=test;
   by charval;
run;

proc print data=test;
run;
```

The results show that the blank falls to the end of the sorted list, whether sorting is performed by CHARVAL or by HEXCODE. All 32 other values for CHARVAL are smaller than a blank. (In an EBCDIC environment there are actually 64 characters smaller than a blank.)

The "How Low Can You Go?" experiment may be over, but do we really understand the software any better than before? These results give some insight into user-defined formats that define the range:

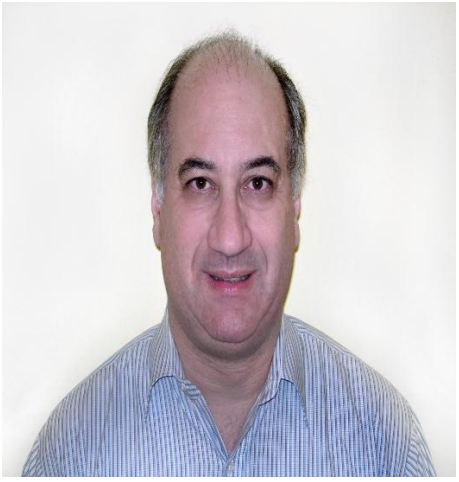
```
low – high
```

For numeric formats, missing values fall outside the range. But for character formats, blanks must be part of this range because there are characters smaller than a blank.

These experiments are just the beginning. As you conduct your own experiments, you will expand your understanding and comfort with the inner workings of the software. That's where the magic begins.

Let's shift into the realm of the macro language.

About The Author:



Robert Virgile is an independent SAS trainer and consultant, with 30 years of experience developing and teaching SAS classes. He has published numerous papers and written problem-solving contests for the Northeast SAS Users Group (NESUG) and SAS Global Forum. Due to his wealth of SAS knowledge, Robert was barred from participating in the NESUG SAS Bowl.

Robert is also the author of *An Array of Challenges – Test Your SAS Skills* and *Efficiency: Improving the Performance of Your SAS Applications.* Learn more about this author and his books by visiting his author page at http://support.sas.com/virgile. You can download free chapters, access example code and data, read reviews, and more.