

# Some SAS Tricks and Tips

Presentation to the Vancouver SAS User Group  
2007-10-31

Howard Cherniack  
Centennial Systems Consultants  
[cherns@compuserve.com](mailto:cherns@compuserve.com)

This is an adaptation of the PowerPoint presentation I made to the Vancouver SAS User Group. Putting it in a document instead of on slides allows me to use complete sentences and include a bit more detail.

These are some random and miscellaneous techniques that are working tools for experienced SAS programmers, but which may not be obvious to those who have not yet had occasion to discover them. Many of these will be familiar to members of the audience, but with luck, some of them might be new to some. Almost all apply to Base SAS DATA step programming. I am assuming a basic familiarity with the elements of the DATA step. I've tried to make the code fragments as simple and illustrative as possible—clearly real-life programs will be more complex.

Note: in some examples, line numbers are shown—these are just for identifying lines for the Notes following. I've tried to use SAS keyword colouring, but since I don't have SAS at home, I may be a bit off in some places.

## Resources

There are a number of resources available to the SAS programmer with a question or problem or curiosity:

- SAS documentation is sometimes a little difficult to navigate through, but it is very complete, with overviews and examples in addition to descriptions and syntax. SAS Online Help is typically installed along with SAS; you can also find the online version at <http://support.sas.com/onlinedoc/913/docMainpage.jsp> and pdf versions of some parts at [http://support.sas.com/documentation/onlinedoc/91pdf/index\\_913.html](http://support.sas.com/documentation/onlinedoc/91pdf/index_913.html).
- Papers delivered at SUGI (SAS Users Group International, now the Global SAS Forum). These are collected by year, but easily found through ordinary web search engines.
- SAS-L, the SAS Mailing List maintained at the University of Georgia. This is a forum inhabited by SAS experts, statistical experts, experienced

SAS users, and beginners. Questions and problems posted to the list will be promptly debated and answered by those who know or have an opinion. If you subscribe, you can look forward to receiving digests of maybe 20-40 messages a day. There may not be a better way to understand the scope and richness of SAS than eyeballing these digests. In addition, there's a search facility for the list's archives. Subscribe or search at <http://listserv.uga.edu/archives/sas-l.html>

- SAS now maintains its own discussion forums, starting at <http://support.sas.com/forums/index.jspa>
- SAS has opened [sasCommunity.org](http://sascommunity.org) as a community resource, containing both blogs and forums. [http://sascommunity.org/wiki/Main\\_Page](http://sascommunity.org/wiki/Main_Page)
- SAS at home: The SAS Learning Edition 4.1 is SAS for \$US 199—an even better value with the dollar at its new value. It can be run on one computer for non-production use only, will die at the start of 2012, is limited to 1500 observations, doesn't have full product support, and has a few other limitations. Check it out at <http://www.sas.com/apps/pubscat/bookdetails.jsp?catid=1&pc=61043>

## SAS as a data manipulation tool

SAS started out life primarily as a statistical tool, but it is also a magnificent data manipulation tool:

- It can read and write just about every input and output format, on just about every platform, known to mankind.
- Its data input is very flexible—since the input pointer can be moved around and the programmer has full control over output, SAS can read or write hierarchical files, with multiple observations per data record or vice-versa. The same applies to data output.
- The handling of invalid data is also flexible—various kinds of invalid data can be translated to various types of "missing," invalid data can be written to another file, or one can ignore the entire record.
- Although SAS was originally designed for handling sequential data sets, PROC SQL allows set-oriented handling of the data, which is sometimes advantageous.
- SAS has a number of features, seldom found in other languages, that make life easier and more efficient for programmers.

### Faced with unknown data?

Sometimes one gets a data file and doesn't really know what's on it. Use SAS to look at it: The `LIST` statement gets SAS to put up a ruler in the log and print some lines (as optionally limited by the `OBS=` option of the `INFILE` statement); where a line contains an unprintable character, a convenient hexadecimal display shows what's "really" there.

```
1) data _null_ ;
2)   infile "X:\data\mystery.foo" obs=2K ;
3)   input ;
4)   list ;
5)   run ;
```

Notes:

- 1) `_null_` is a special SAS dataset that is not a dataset at all—it tells SAS that one wants a `DATA` step, but not an output SAS dataset.
- 2) the `obs=` option tells `infile` that only the first two kilobytes (K) of the file are to be read. This option may also be specified in numbers of records (as defined by the operating system) or megabytes (M).
- 3) the `input` statement is required for the `list` statement to work. It may specify variables to input, but need not.
- 4) the `list` statement tells SAS to display the input data.
- 5) the `run` statement tells SAS that this is the end of the `DATA` step (or procedure). It is not always required for batch jobs (SAS will take a `data` or a `proc` as implying the end of whatever comes before, but this can sometimes lead to confusions, and it's always better to be explicit. When one is running interactively on a PC, it's required for the execution.

### Flexible input—multiple observations from single input-data rows

Usually, the `input` statement reads a record at a time, then goes on to the next record. However, SAS allows the programmer to move the input pointer programmatically, with the "trailing `@`", which tells SAS to keep the pointer from going on to the next record.

A simple example:

We have input data with a student's name in the first 10 columns, then five 5-column fields, with scores or something. We want to get these into a SAS dataset with each score as a separate observation.

Input data:

```
John          50.  60.  30.  20.  10.
Chris         15.  72.  95.  30.   5.
```

Here's some code:

```

1) data scores ;
2)   infile "X:\myfile.dat" ;
3)   input StudentName $10. @ ;
4)   do TestNo = 1 to 5 ;
5)     input score 5. @ ;
6)     output ;
7)     end ;
8)   input ;
9)   run ;

```

Notes:

- 3) The "@" at the end leaves the input pointer on the current record, at column 11.
- 4) Here we will execute the code between `do` and `end` five times, with the variable `TestNo` taking on the values 1 through 5.
- 5) We read the score with a five-column numeric format, and the trailing "@" leaves the input pointer immediately after that field.
- 6) The `output` statement puts a row into the SAS dataset `scores`.
- 8) Since, after the completion of the loop, the pointer is still on the input record, we add another `input` statement to tell SAS to go on to the next one.

#### Flexible input—different kinds of records

Suppose we have an input data set that is made up of two records: records with a "A" in column 1 have a value X in columns 10-14, and other records, with something else in column 1 and the value of X in columns 4-7.

```

A           300.
B  400.
B   25.
A           150.

```

Here's how we can read it:

```

1) data bar ;
2)   infile "X:\data\myfile.foo" ;
3)   input rectype $1 @ ;
4)   if ( rectype = 'A' )
5)     then input x 10-14 ;
6)     else input x 4-7 ;
7)   run ;

```

Notes:

- 3) the "@" at the end leaves the input pointer on the current record, at column 2.
- 5) Here we are using column input. To read columns 10-14, we could also have specified `input @10 x 5.`—that is, move the pointer to column 10, then read X in the five columns starting there.

### Some examples of SAS's programmer friendliness

SAS has some language features that other languages should have:

#### Truncated character comparisons

The relational expression in

```
if ( 'AB' = 'ABC' )
```

evaluates to False, since the shorter element is padded to the length of the longer. On the other hand,

```
if ( 'AB' =: 'ABC' )
```

is True—the colon after the equals sign tells SAS to truncate the longer element to the length of the shorter. This also works for other relational operators like > or <.

#### Three-way comparisons

When we want to see if some number (say, for example, b) has a value between two other numbers (say, a and c), in other languages we would usually have to write the equivalent of

```
if ( a < b ) and ( b < c )
```

This is a common enough form, and SAS gives us a short cut:

```
if ( a < b < c )
```

(This form can of course use other relational operators, such as <=, =, and so on.)

#### Short-cut accumulation

Here's another common form that we would have to write this way in most other languages:

```
a = a + b
```

Again, SAS gives us a short cut:

```
a + b
```

(This form also implies `retain a 0`, but it's usually better to specify `RETAIN` explicitly. More on this later.)

#### Some useful functions

SAS has a large number of useful functions; here is just a bare sampling of some of them:

- `catx( sep, a, b, c ... )`—removes leading and trailing blanks from the character expressions a, b, c, ..., then concatenates them, separating them with the character expression sep.
- `comp1( str )`—compresses multiple blanks in the character expression str to single blanks.
- `scan( str, num, delims)`—returns the numth word (as defined by delimiters in the character expression delims) of the character expression str.

- `substr( str, posn, len )` on the right side of an assignment returns the substring, starting at position `posn` for `len` characters, of the character expression `str`.
- Here's an unusual one: `substr( str, posn, len )` on the **left** side of an assignment takes the character value of the right side of the expression and inserts it into the string `str`, at position `posn` for `len` characters.
- `strip( str )`—returns the contents of the character expression `str` with leading and trailing blanks removed.

## The "comb"

A common problem is to categorize a variable based on upper and lower limits, leading to a lot of code like this:

⊘

```
if ( 0.01 < measure ) and ( measure <= 0.02 ) then grade = 0 ;
if ( 0.02 < measure ) and ( measure <= 0.05 ) then grade = 1 ;
if ( 0.05 < measure ) and ( measure <= 0.10 ) then grade = 2 ;
...
```

SAS's three-way comparisons make things a little more straightforward:

⊘

```
if ( 0.01 < measure <= 0.02 ) then grade = 0 ;
if ( 0.02 < measure <= 0.05 ) then grade = 1 ;
if ( 0.05 < measure <= 0.10 ) then grade = 2 ;
...
```

Neither of these approaches check `measure` for validity, and both perform redundant tests. A better approach would use the `SELECT` group, which goes through a series of `WHILE` tests, executing the first one that is True and then leaving the group: Here's a code fragment:

```
1) ....
2) select ;
3)   when( missing(measure) ) error / "*** Missing!" measure= ;
4)   when( measure <= 0.01 ) error / "*** Too low!" measure= ;
5)   when( measure <= 0.02 ) grade = 0 ;
6)   when( measure <= 0.05 ) grade = 1 ;
7)   when( measure <= 0.10 ) grade = 2 ;
8)   ...
9)   otherwise                error / "*** Too high!" measure= ;
10)  end ;
11) ....
```

Notes:

- 2) if the `SELECT` statement has a value in parentheses, then the value in each `WHEN` clause is compared to that value, and the first match is executed. If there is no value in the `SELECT` statement, then the first `WHEN` clause with a `True` value is executed.
- 3) `missing(measure)` is `True` if and only if the variable `measure` is some SAS missing value.
- 3), 4), and 10) The `error` statement puts a note in the SAS log, using the syntax of the `put` statement—this particular construction skips a line (/), followed by a piece of text (what's in quotation marks), and displays the value of the variable `measure`. It will also display the values of all variables in the dataset.
- 5) Note that there is no need to test whether `measure` is greater than 0.01—we could not have come to this place if that were not the case.
- 9) The `otherwise` clause of the `select` statement is `True` if none of the preceding `while` clauses are `True`. If this clause is omitted and none of the `while` conditions is `True`, then SAS will report an error.

## Data breaks and RETAIN

If a dataset is sorted by some variables, the `BY` statement recognizes "breaks," that is, rows in which the `by`-variables change value. Break recognition is hierarchical—a break in a higher-level variable automatically implies a break in all lower levels. This is useful when, for instance, various kinds of `PROC` processing is desired for each break. However, the `BY` statement is equally useful in the `DATA` step.

When a variable is created within a `data` step (i.e., not in an input dataset), that variable is initially set to Missing for each new input row (to be assigned a new value within the `DATA` step), **unless**: the `RETAIN` statement is used to tell SAS to retain the value of a variable from one input row to the next.

`RETAIN` is useful for summing over rows, and also for inter-row comparisons. (The `LAGx` functions also return previous values of a variable, but aren't quite as flexible, and their usage can be tricky.)

Here's an example that uses both data breaks and `RETAIN`:

We have an input SAS dataset, `foo`, with variables `a`, `b`, and `val`, sorted by `a b`:

<u>a</u>	<u>b</u>	<u>val</u>
1	1	10
1	2	20
1	3	50
2	1	10
2	1	20
2	3	30
2	3	50

We want to create three output datasets:

Dataset `Allobs`, containing the sum of the values of `val` in the input dataset;

Dataset ObsByA, containing the sum of the values of val for each value of a in the input dataset; and

Dataset ObsByAB, containing the sum of the values of val for each combination of a and b in the input dataset:

```
AllObs:  SumAllObs
         190
ObsByA:  a  SumAObs
         1  80
         2 110

ObsByAB: a  b  SumABObs
         1  1   10
         1  2   20
         1  3   50
         2  1   30
         2  3   80
```

(If this is all we wanted to do, we'd probably be better off using something like PROC SUMMARY. Typically, though, there's more complicated stuff going on.)

Here's the code:

```
1) data AllObs (keep=SumAllObs)
2)     ObsByA (keep=a SumAObs)
3)     ObsByAB(keep=a b SumABObs) ;
4) set foo end=last_obs ;
5) by a b ; /* (previously sorted by a b) */
6) retain SumAllObs SumAObs SumABObs 0 ;
7) if (_N_=1) then SumAllObs = 0 ;
8) if first.a then SumAObs = 0 ;
9) if first.b then SumABObs = 0 ;
10) SumAllObs + val ;
11) SumAObs + val ;
12) SumABObs + val ;
13) if last.b then output ObsByAB ;
14) if last.a then output ObsByA ;
15) if last_obs then output AllObs ;
16) run ;
```

Notes:

- 1), 2), and 3) This data statement specifies **three** separate output datasets. The `keep` dataset option specifies which variables are to be kept in the dataset.
- 4) The `end=` option of the `set` (or `merge`, for that matter) statement creates a new variable, `last_obs` that will be True at the last observation of the input dataset. Any legal SAS variable name would do, and the variable is not saved in the output dataset(s).

- 5) The `by` statement is the key to the break processing. If dataset `foo` is not sorted by the variables `a` and `b`, we will get an error message and processing will be abandoned.
- 6) The `retain` statement specifies that these three variables are to retain their values between input observations, and also that they are initialized to 0 (this initialization is not absolutely required, since we will be re-initializing them each time around, but it costs nothing to be explicit. Similarly, the forms of the accumulations in **10**), **11**), and **12**) imply both the `retain` and the initialization, but again the statement makes explicit what is going on.
- 7) SAS maintains an automatic variable `_N_` that gives the number of the input observation. When the value of this variable is 1, this is the first observation of the dataset. As it happens, we have already initialized the variable `SumAllObs` to 0 in the `retain` statement, but again this is more explicit.
- 8) `first.a` is True for the first observation with a new value of `a`—that is, for the first and fourth observations of the input data set—and False otherwise. This is an appropriate time to start the accumulator `SumAObs` to zero.
- 9) `first.b` is True for the first observation with a new value of either `a` or `b`—that is, for the first, second, third, fourth, and sixth row. So we set the value of the accumulator `SumABObs` to zero.
- 10**), **11**), and **12**) roll up the values in the accumulators.
- 13**) `last.b` is the reverse of `first.b`—it is True for the **last** value of a unique combination of `a` and `b`. The output statement will generate a row of the output dataset `ObsByAB`—if we had not specified the dataset in this statement, a row would have been generated in each output dataset.
- 14**) `last.a` is the reverse of `first.a`, and the time we want to generate a row for the dataset `ObsByA`.
- 15**) Here we use the temporary generated variable `last_obs`, created in line **4**). It is True for the last row of the input dataset(s), so it's time to write the one row into the output dataset that summarizes the entire input one.

### The IN= dataset option

Sometimes we have multiple input datasets, and it's important to know which dataset some row comes from. Here we have two input datasets, `a` and `b`, sorted by some common matching variable `x`. We want to know how many observations come from each of the input datasets, and how many from both (as merged by the matching variable).

```

1) data _null_ ;
2)   merge a ( in = in_a )
3)       b ( in = in_b )
4)       end=the_end ;
5)   by x ;
6)   retain num_a num_b num_both 0 ;
7)   num_a + in_a ;
8)   num_b + in_b ;
9)   num_both + ( in_a and in_b ) ;
10)  if the_end
11)  then put / num_a= num_b= num_both= / ;
12)  run ;

```

Notes:

- 1) The DATA step normally creates an output dataset. If we just want to do some processing, but not create a dataset, we specify the special "no-dataset dataset" `_null_`.
- 2) and 3) The `in=` dataset option creates a temporary (non-saved) variable that will be True only if a row comes from that dataset, and False otherwise. For a given input row, `in_a` will be True only if that row comes from dataset a, and `in_b` will be True only if it comes from dataset b. Note that since the rows are being merged with a common `by` variable, an observation could well come from both input datasets.
- 4) Again we create a temporary variable, `the_end` this time, that will be True only at the last observation of the input datasets.
- 6) We want our accumulators to keep their values between the input observations, and to start out at zero.
- 7) and 8) SAS has only numeric and character variables; boolean variables ("True" and "False") are actually numeric: True is 1 and False is 0. So the variables `in_a` and `in_b` will have the values 1 or 0, and these values can be used in ordinary arithmetic, such as adding to an accumulator.
- 9) Boolean variables can also be used in logical expressions, which return the boolean values of 1 or 0. Here, the expression `( in_a and in_b )` is True (=1) if and only if both `in_a` and `in_b` are True (=1). The value of the expression can then be used arithmetically in our count.
- 10) Here we use the temporary variable created in line 4).
- 11) In this case, we just want a note in the log. This `put` statement will skip a line, print the values of our three accumulators, and skip another line.

## Dates, Datetimes, Times

Dates are probably the largest source of problems and questions on the SAS-L mailing list than any other single topic. This seems strange, since deep down they're pretty simple.

Unlike many database products, SAS has only two data types, numeric and character—how can it have dates, datetimes, and times? Actually, they're just numbers, but

numbers that have special meanings to the formats that express them and the functions that manipulate them. Although you don't really have to know this:

- Dates are the number of days before (<0) and after (>0) January 1, 1960 (0). To express a date constant, use the form '31OCT2007'**d**. (Because SAS uses the Gregorian calendar, dates before the adoption of this calendar—1582 for Britain and its colonies—must be approached with caution.)
- Datetimes are the number of seconds before (<0) and after (>0) 00:00 hours (midnight) of January 1, 1960 (0). To express a datetime constant, use the form '31OCT2007 08:30:00'**dt**.
- Times are the number of seconds after 00:00 hours (midnight) of any day. To express a time constant, use the forms '17:00:00'**t** or '5:00pm'**t**.

Note that this makes date arithmetic pretty easy: to count seven days from a date, you just add seven.

There are formats and informats for just about every way of expressing dates, datetimes, and times, including weeks, quarters, and other calendars. Typically, these kinds of variables should have particular formats permanently associated with them. Note that these formats just control the display of the value—the underlying numeric value is what the date, datetime, or time really "is."

There are functions for many kinds of manipulations on dates, datetimes, and times, including:

- conversions, like extracting dates and time from datetimes; and
  - calculating differences and applying increments.
- Some of these can be a bit tricky: for instance, the function `intck('month', Date1, Date2)` looks as if it calculates the number of months (as specified by the first argument—there are many possible values for this) from `Date1` to `Date2`. Almost: it actually calculates the number of **month boundaries** between the two dates—if the day of the month of `Date2` is earlier than that of `Date1`, then you will have to use the `day` function which returns a date's day of the month, to compare the days:
- `( day(Date2) < day(Date1) )` is a logical expression that is True (=1) when the day of the month of `Date2` is less than that of `Date1`, so `intck('month', Date1, Date2) - ( day(Date2) < day(Date1) )` will return the number of full months that have elapsed between the two dates. (Studying this example, with some experimenting, will greatly aid understanding the whole family of functions.)

In general, keeping dates or datetime or time values in any form other than SAS dates, datetimes, or times is just asking for trouble.

## Missing data values

SQL has a concept of a NULL data value, a value that isn't a value, meaning "we don't know what it is." SAS has something similar, but a little fancier. Since the missing value for a character variable is just a blank, the concept is really relevant for numeric variables.

One of the things SAS was created for was projects like surveys, in which there might be values for "Don't Know" and "Won't Answer" and "Not at Home," all of which should be distinguishable, but generally should not participate in any statistical measure. SAS recognizes 28 different kinds of missing variables, all sorting lower than any negative numeric value: `.` (that's a dot; the "ordinary" missing value); `._` (dot-underscore), and `.A` to `.Z` (dot-A to dot-Z).

Missing values arise from:

- invalid input to an `input` statement or function—for instance an alphabetic character in what should be a numeric field. (Optionally, you can specify that a particular single letter can be read as a special missing value, for instance, an A in a field be taken for the missing value `.A`.)
- impossible mathematical operations.
- unmatched observations in a `merge` (for instance, if two datasets are matched on a variable, but one of them has a match-variable value that the other doesn't, all of the variables in the absent one are set to missing.)
- initialized variables (often the result of a missing `retain` statement).
- an assignment (e.g., `x = . ;`)
- a calculation involving a missing value—for instance, if `x` is missing, then so is `3 * x`

If you are testing for a missing value, always use the form

```
if missing(x) then ...
```

instead of

```
if (x=.) then ...
```

because the latter will miss any of the other possible missing values. They're not common, but they're sometimes used.

## The ATTRIB statement

In its early days, SAS allowed the programmer to specify variable attributes with what might be called functional statements—for instance the `FORMAT` statement would specify formats for variables, the `LENGTH` statement the lengths, and so on. These still work, but sometimes it's a bit more convenient to use a comparatively newer statement, `ATTRIB`, which allows specification of multiple attributes (`length`, `format`, `informat`, `label`) for a single variable. It is good practice to specify every variable used in a program, if only for documentation—not only for others trying to read your program, but also for yourself in a month or two.

For numeric variables, you usually need at least:

```
attrib <var> format=<fmt> label='<lbl>' ;
```

that is, the output format, and a label for documentation. Where there are considerations of storage space, `length` may also be important.

For character variables, the important attributes are the length and documentary label:

```
attrib <var> length=$<len> label='<lbl>' ;
```

(Note that it is the dollar sign in front of the length that marks the variable as a character one.)

## Altering or reordering variables

SAS uses the first appearance of a variable name to specify its attributes and sequence in a dataset. You can change a variable's attributes in a `DATA` step by specifying the new attribute (such as `format` or `label`) **before** the first appearance of that variable:

```
data newds ;
  attrib foo label='Changed label' ;
  set OldDS ; /* contains the variable foo */
run ;
```

This will have the side effect of making `foo` the first variable in the dataset.

By the strict rules of relational database systems, the sequence of variables in a dataset shouldn't make much difference—the programmer should specify variables explicitly by name in all cases. On the other hand, when browsing or "eyeballing" datasets, it's often convenient to have variables in some sort of sequence that makes sense. In addition, SAS allows specification of a list of variables based on sequence. (Since sometimes the sequence can be disrupted inadvertently, this is a potentially dangerous practice.)

Variable order within a dataset again goes by first mention of variable names, so putting `attrib` or `label` statements, for instance, in the desired sequence, before a `set` or `merge` statement, as in the example above, will do the job. The `retain` statement, however, does not require specification of attributes, so it's often the most convenient. (Because there is a new value of that variable for every incoming observation, the "official" meaning of `retain` doesn't apply or affect anything.)

## Macro variables

Macro variables allow the programmer to put a character value into a variable, and then use that value by referring to the variable. Macro variables are just text substitution. There are two main ways to create macro variables:

- Outside a `DATA` step:

```
%let foo = bar ;
```

(There are elaborate work-arounds for cases where the value on the right side of the equals sign contains, for instance, semicolons.)

- Inside a DATA step:
 

```
call symput("foo", "bar") ;
```

 (Note that the name of the macro variable is given as a character expression, and that this call does not take effect until **after** the DATA step completes.)
- From the INTO clause of a PROC SQL SELECT statement. This can be very flexible (for instance, either creating one macro variable per row, or concatenating the values of each row into a single macro variable), and is complex enough just to note and not discuss.

Both of the first two create a macro variable named `foo` and assign the character value `bar` to it. The macro variable is invoked as `<macvarname>`:

```
put "&foo" "X&foo.1" '&foo' ;
```

This will put a line into the log:

```
bar Xbar1 &foo
```

The first expression shows that the value of the macro variable `foo` has been substituted for the macro variable itself. The second expression shows that it's possible to include the macro variable as part of something larger: the `"&"` marks the beginning of the macro variable so the `"X"` can go right in front, but we need the dot after the macro variable name to show that we are referring to macro variable `foo` and not `foo1`. Finally, the last example shows the only big difference between single and double quotation marks: a macro variable within double quotation marks is resolved to its value; within single quotation marks, everything is just text and macro variables are not resolved.

What are macro variables good for? Lots of things, including:

- Global text and numeric constants used in several places:
  - ```
%let ProjName = Clinical Study 1 ;
      title2 "&ProjName" ;
```
  - ```
%let DaysInMonth = 30.4375 ;
      AgeMonths = ( CurrDt - DOB ) / &DaysInMonth ;
```
- Passing values from one DATA step to another, or to a procedure. (A simple example: calculating the total number of observations in a dataset in a DATA step, then using `"N=&NumObs"` in the subtitle of a subsequent procedure.
- Being able to write fairly general code, then using macro variables to apply it to specific instances:
 

```
%let TheseVars = x y z ;
...
proc means data=foo ;
var &TheseVars ;
...
```

## Macros

Macros are extensions to the idea of macro variables. They can be elaborate, can contain logic, can include multiple procedures or DATA steps or just little code fragments, but

ultimately reduce to text that can be thought of as being inserted into a SAS program at the point of invocation.

Macros are defined with a `%macro ... %mend` pair. For instance:

```
%macro foo ( param1=, param2=bar ) ;  
  <body of macro>  
%mend foo ;
```

This defines a macro named `foo`, with two parameters, `param1` and `param2`. The parameters can be considered as macro variables within the scope of the macro body, being referred to as `&param1` and `&param2`. If the second parameter is not specified in the invocation of the macro, it will take the default value `bar`. The definition of the macro has no immediate effect, but allows it to be invoked later.

The definition of the macro may be in executed code in a SAS program, or put in a file, with the same name as the macro and the suffix `sas`, in a library (e.g. Windows folder) that is specified in an `options sasautos=` statement, in which case any invocation of a hitherto-unrecognized macro will cause SAS to search that library for that macro.

Macros are invoked as the name of the macro with the percent sign in front:

```
%foo( param1=xxx )
```

This will be expanded to whatever the body of the macro is, and the resulting text will replace the macro call.

A very simple example:

- Define:

```
%macro DtDif2Yr( FromDt=, ToDt= ) ;  
  ( ( ToDt - FromDt ) / 365.25 )  
%mend DtDif2Yr ;
```

Note that there is no semicolon after the body of the macro, since the content of the macro may be in the middle of a line of code that has its own semicolon. Similarly, in this case we want to enclose the expression in parentheses, to make sure that the constant 365.25 remains in the expression.

- Invoke:

```
AgeInYrs = %DtDif2Yr( FromDt=DOB, ToDt=CurrDt ) ;
```

This will resolve to:

```
AgeInYrs = ( ( CurrDt - DOB ) / 365.25 ) ;
```

## Arrays

Arrays are a way of treating variables (and constants) as aggregates in DATA steps. Unlike arrays in other programming languages, SAS arrays are usually made up of "ordinary" variables and constants—they're usually just a convenient way to handle them. Arrays exist only within the context of a single DATA step—they do not carry over to the dataset or subsequent DATA steps. A simple example:

We have a dataset that includes some blood measures in "conventional" units; we want to replace these with values in SI units:

```
1) data converted ;
2)   set raw ;
3)   array rawvals [3] Glucose HDL   Bilirubin ;
4)   array conv    [3] _temporary_ ( 0.0555 0.0259 17.1 ) ;
5)   array newvals [3] GlucSI   HDLSI BiliSI ;
6)   do j = 1 to 3 ;
7)     newvals[j] = rawvals[j] * conv[j] ;
8)   end;
9)   drop j Glucose HDL Bilirubin ;
10)  run ;
```

Notes:

- 3) Glucose, HDL, and Bilirubin are variables in dataset `raw`, but we will want to refer to them more conveniently. This `array` statement creates an array named `rawvals`, says that there will be three elements in the array, and names them.
- 4) These are three constant conversion factors. Since they will not enter the output dataset, it is not necessary to give them individual names; the `_temporary_` keyword says that they have no independent existence, and the three numbers in the parentheses are the constants assigned to the three elements of the array.
- 5) These are three variables we wish to create in the new dataset. (Properly, they should have `attrib` statements specifying their attributes.)
- 6) This `do` statement tells SAS to loop around the range of the `do` (that is, up to the `end` statement) three times, with the index variable `j` taking on the values 1 through 3.
- 7) This is the conversion: element `j` of the `newvals` array assigned the value of element `j` of the `rawvals` array times element `j` of the `conv` array.
- 9) We don't want the old variables in the new dataset; neither do we want the variable `j`, which we're just using as a temporary index variable. The `drop` statement tells SAS not to include them in the new dataset. (`drop`, and its mirror image `keep`, can also be useful as dataset options.)

### Using SAS to control MSWord (Windows only)

Dynamic Data Exchange (DDE) is an old Windows technology, now considered to be superseded by Object Linking and Encoding (OLE). SAS can act as a DDE "client," controlling a Microsoft Word (or Excel) "server" through commands in the WordBasic language. The basic technique is found in Viergever & Vyverman, "Fancy MS Word Reports Made Easy: Harnessing the Power of Dynamic Data Exchange: Against All ODS, Part II," in SUGI 28 paper 16-28 <http://www2.sas.com/proceedings/sugi28/016-28.pdf>. The WordBasic Help file that lists and explains the WordBasic commands is at <http://www.microsoft.com/downloads/details.aspx?FamilyID=1a24b2a7-31ae-4b7c-a377-45a8e2c70ab2&DisplayLang=en>. Briefly, the technique works like this:

```

1) data _null_ ;
2)   call sleep(1,1) ; /* delay 1 second */
3)   rc = system('start winword') ;
4)   call sleep(1,1) ; /* delay some more */
5)   run ;

6) filename word dde 'winword|system' notab ;

7) data _null_ ;
8)   file word ;
9)   put '[Insert "Hello world"]' ;
10)  put '[FileSaveAs.Name="foo",.Format=0]' ;
11)  put '[FileExit 1]' ;
12)  run ;

```

Notes:

- 1) the purpose of this DATA step is to fire up a copy of Microsoft Word.
- 2) and 4) These calls to the SLEEP function just pause everything for a second each. It appears that Windows needs this time to fire up Word and set up the communications channels to it.
- 3) The SYSTEM function sends some text to the operating system; this particular text fires up Word. The function returns a code indicating success, but we can ignore it in this simple example.
- 6) This creates a filename "word" and designates it as a DDE channel; the 'winword|system' is the "DDE triplet" (don't ask) that tells Windows that we want a channel to Word. Finally, we tell SAS to ignore the tab characters that are typically inserted between DDE commands..
- 8) Here we tell the DATA step that the contents of PUT statements are to be sent through the DDE connection.
- 9) A WordBasic command saying "Insert this text at the current location"
- 10) A WordBasic command to save the current document as a Word .doc file named foo.doc.
- 11) A WordBasic command to exit this instance of Word, saving changes.

Typically, of course, there would also be a great deal of formatting and such in the Word document. And much of the code would be in a macro. The V&V article uses SAS to create a mail-merge in Word, inserting boilerplate, text, and SAS data appropriately. The technique can also be used for other applications, such as creating a Word document from a SAS report. (More below).

## ODS, PROC REPORT, MSWord

SAS's comparatively new Output Delivery system does two things:

- It allows the capture as data (in a SAS dataset) of the output of some procedures that normally produce only LIST (printed) output. This can

be a bit tricky, but it also can be very useful. Check out the `ODS OUTPUT` statement.

- It formats SAS output as HTML, RTF, PDF, or Postscript, instead of the traditional monospaced lineprinter output that mainframe users are familiar with. This can produce some very pretty output, but is more suited for "low-density" tables and such than ones that are full of data.

(RTF, by the way, is the expression of a formatted document as a markup-language file, completely in a restricted set of alphanumeric characters, with formatting expressed as commands. It can easily be imported into Word. To see the formatted expression of the file, open it in Word or WordPad; to see what it looks like "underneath," open it in Notebook.)

A common task is putting a table or listing generated by `PROC REPORT` into a Microsoft Word document. Typically, this will involve using ODS to put the output into RTF form, then importing that file into Word. Unfortunately, this doesn't always work very well. For example:

- Statements like `compute before/after _page_` are not recognized by RTF, which regards them not as page headers or footers, but as document headers and footers. `TITLEn` and `FOOTNOTEn` statements will show as proper headers and footers, but these are difficult to format, and are limited to ten lines each.
- There seem to be no ODS templates for closely-packed tables or listings, comparable to the 132-characters-by-66-lines lineprinter output.
- Although there are ways to insert fields such as "Page x of y" into an RTF document, many versions of Microsoft Word don't always interpret these correctly. This is a fault of Word, not RTF—for further information, try doing a web search on "page x of y" (with the quotation marks).

My solution was to use `PROC PRINTTO` to send SAS output to a text file, and then use DDE to get Word to insert the output files (print or graph—I found that the CGM format seems to work best) into a Word document, adjusting the document formatting to match the procedure output (e.g., landscape orientation, margins, font (which governs characters per line), lines per page, etc., adding the "Page x of y" at the bottom of each page, and so on. This can require quite a bit of setup (best done in macros, of course), but can work pretty well.

## Using SAS to generate SAS code

Finally: SAS is flexible enough that it can be used to generate and then execute its own code. Here's a very simple example: we have a dataset with a bunch of variables in it, and, for some reason, we want to rename each variable by adding "X" to the beginning of each name. Renaming variables in SAS is done with the RENAME statement within a DATA step, but there are too many variables to type all of the RENAMES, and besides, we want to make this a general procedure.

```
1) proc contents data=oldDS noprint out=coldS(keep=name) ;
2)   run ;

3) data _null_ ;
4)   set coldS ;
5)   file temp "C:\temp\stmts.sas" ;
6)   attrib stmt length=$60 label='a RENAME statement' ;
7)   stmt = catx( " ", "rename", name, "= X"||name, ";" ;
8)   put @3 stmt ;
9)   run ;

10) data newDS ;
11)  set oldDS ;
12)  %include "C:\temp\stmts.sas" ;
13)  run ;
```

Notes:

- 1) PROC CONTENTS is a SAS procedure that displays information about a SAS dataset, in this case oldDS. The noprint option says that we don't want a printed report; the out= option says that we want the procedure's output is to be sent to a SAS dataset, in this case coldS; the keep= dataset option says that the only variable we want saved in this generated dataset is name, which our PROC CONTENTS documentation tells us contains the name for each column in the dataset.
- 4) The input to this step will be the dataset generated just above; each row contains the name of a column.
- 5) This statement says that step output (generated by the PUT statement) is to go to the external file named. (The folder had better exist, or SAS will report an error.)
- 6) We define a variable stmt as 60 characters long.
- 7) We construct the contents of the variable stmt. The first argument to the catx function is the separator that will be inserted between the other elements; in this case it is a single blank. For example, for a name with value foo, the resulting value of stmt will be:  
rename foo = Xfoo ;
- 8) This statement writes a line of the variable stmt to the output file, with a couple of blanks before it. The result will be a file with a rename statement for each row of the input dataset, each row corresponding to one column name in the original dataset.
- 10) Here we will create a new dataset newDS from the old dataset oldDS.

*12)* the `%include` statement, which looks like a macro but isn't really, sucks the contents of the just-generated file into the input stream. The result will be that the step will effectively contain all of the `rename` statements generated in the previous step, accomplishing the renaming of all columns.