SAS programming tends to isolate a programmer from other programming methods. A programmer may know a standard third-generation language (3GL) method for doing something, yet spend much time and effort "figuring out how to do it in SAS." We hope to narrow this gap between technique and implementation.

In many instances the SAS System provides the tools required to implement standard programming methods as well or better than 3GL programming environments. The programmer has to learn to recognize SAS equivalents to program constructs he or she already knows, and practice implementing them.

We use a series of examples to illustrate basic programming constructs, implementation of these constructs in typical 3GL programs and SAS equivalents. We hope to extend the SAS System, using SAS Macro variables and functions, SQL procedures and metaprograms, while preserving the natural advantages of SAS.

**PROGRAMMING CONSTRUCTS AND SAS**

Some basic and useful 3GL programming constructs have no direct analogies in SAS data step programs. Skilled programmers who try to transplant these constructs into a SAS program risk misapplying them. Programmers with more experience working in the SAS System usually discover methods for working around some of SAS's features, but (speaking from experience) they often use limited, unduly complex or inefficient constructs.

**Example 1: Creating scalar variables.**

A common 3GL programming method involves reading data from one source and setting a flag variable to a value if the data contain a specific value or values. Later parts of the program can then refer to the flag variable, as in

```plaintext
until(EOF)
{ read(file1,var1);
  if var1="Site1"
    then flag="N";
}
until(EOF)
{ read(file2,var2);
  if flag="N"
    then ...
}
```

Perhaps the condition, var1="Site1", anywhere in file1 dictates that the program use a more complicated algorithm for updating var2. Status flags containing integer numbers or character strings often control the operations of 3GL programs.
The same construct in a SAS program will not work the same way, as any SAS programmer will tell you:

* This program will not work!.

data lib._out1;
  set lib.in1;
  flag='0';
  if var1="Site1"
    then flag='N';
run;
data lib._out2;
  set lib.in2;
  if flag='N'
    then ....

In the 3GL program, a table connected to a program block, such as the one shown, points from the variable name FLAG to a scalar memory location (one capable of holding a single number or character string). At the end of the first data step of the incorrect SAS program segment (after the first run statement), the SAS library catalog contains rows keyed to the dataset _out1. Each row points from the row containing the dataset name and variable name to a column of the dataset. If, in a later data step, the program does not refer to the dataset containing the variable FLAG, it cannot access the variable. SAS uses the SAS library catalog, not the programming environment, to store the access path to variables. In fact, the same variable name related to different datasets in the catalog identify completely different SAS variables. They may even have different types!

Even if a later data step does refer to the dataset in a set or merge statement, so it has access to its catalog entries, the flag variable may have different values in different observations (rows) of the dataset. Experienced programmers use the retain statement to make a SAS flag variable simulate a scalar variable.

Misunderstandings of the distinctions between 3GL variables and SAS variables have cost a fortune in programmer time and effort (again speaking from experience)! For those tasks requiring a scalar variable in a SAS data step program, a relatively simple technique does the trick:

Define a SAS macro variable by calling the SAS symput function:

```
data lib._out1;
  set lib.in1;
  if _n_=1
    then call symput('flag','0');
  if var1="Site1"
    then call symput('flag','N');
run;
data lib._out2;
  set lib.in2;
  if symget('flag')='N'
    then ....
```

The initialization of the macro variable at _n_=1 of dataset lib.in1 prevents run-time errors when the var1="Site1" condition fails in every observation of lib.in1. The symget function has the same effect in most SAS implementations as a more intuitive alternative: "&flag". The alternative reveals the ampersand (&) that typically denotes a macro variable in SAS programming. As usual, SAS replaces the space the macro variable name occupies in the SAS program with the character string value to which the name points. SAS stores the macro variable value in the programming environment, not the SAS library catalog. As a result, it has the same effect as a scalar variable flag in a 3GL program.

One caution, clearly stated in SAS documentation but worth repeating for emphasis, warns that the SAS compiler does not execute the symput function until it comes to the end of the data step containing it. Since at that point it has resolved all of the macro variable and data step variable references in that data step, it cannot recognize a macro variable name defined by the symput function in that same data step. Note that in the example given the program defines the macro variable name and value in one data step and refers to it again in another.
Example 2: Table Look-ups

Rather than write out in program statements a long list of conditional assignments of the form, IF VAR1='X' THEN VAR2='Y', skilled programmers often use look-up tables to represent relations between two or more data elements. Many 3GL programs allow the programmer to construct local or global data arrays of multiple dimensions. A table represented by a simple two-dimensional array might show the relation between a variable name and the value of the preceding variable that indicates a skip pattern:

<table>
<thead>
<tr>
<th>variable name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_12</td>
<td>2</td>
</tr>
<tr>
<td>Q_22</td>
<td>1</td>
</tr>
<tr>
<td>Q_23</td>
<td>2</td>
</tr>
</tbody>
</table>

A 3GL program could define the array using direct assignments, or populate the array by reading values from a file. Matching on the variable NAME supplies the index of the array, and the corresponding VALUE has the same row index.

SAS, despite some clever though contrived attempts to provide direct access table look-ups, does not offer a direct analogy to standard 3GL table look-ups. The whole idea of suspending the processing of observations in one input dataset and looking up values in another, then resuming processing of the first dataset, seems in conflict with the basic tenants of SAS datasets processing.

In those situations requiring a large number of look-ups, relative to the number of observations being processed, the SAS MERGE statement offers a better alternative. Joining the look-up table to the primary dataset places the desired value in every observation. The expanded view of the data includes a new variable with the appropriate value. Proc SQL makes the programming even easier, and possibly more efficient.

Situations requiring infrequent references to a small look-up table have equally attractive alternatives in the SAS Macro language. These equivalents to basic 3GL table look-up constructs do not require the sorting or indexing overhead of the method of joining look-up and primary tables.

SAS macro variables will hold a list of tokens (character strings) along with their delimiters. The programmer can define a macrovariable directly in the program or use the symput function to assign values from a dataset to it. Defining the list in a program usually proves simpler during the first stages of program development.

* Example of SAS table look-up program using macro variables;

* Define macro;
%macro inlist(token,lst,limit=200);
  _n=1;
  _t=scan(&lst, _n);
  do while(_t ne "" and _n<&limit);
    if _t=&token then
      _mnum=_n;
      _n=_n+1;
      _t=scan(&lst, _n);
  end;
%mend inlist;

* Define parallel lists in macro variables:
%let vskp1=Q11B Q12B Q23B Q30B Q31B Q32B Q33B Q35B Q36B;
%let vskp2=Q37B Q38B Q39B Q40B Q42B Q43B Q44B Q47B Q48B;
%let vallst=1 1 1 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0;
%let vskp= &vskp1 &vskp2;

* Table look-up segment of data step;
DATA out;
  length _mval $ 8;
  retain vskp vallst _mval;
  SET in;
* Alternative to macro quoting;
if _n_=1
  then do;
    vskp="&vskp";
The SAS macro %inlist looks up a token on a list and returns the position of the token in the list (its index value). A second list, parallel to the first in the sense that token 1 on the second list contains the value(s) related to token 1 on the first list, uses the index value to locate the value in the second dimension of the look-up table. Note that the program defines both the macro %inlist and the macro variables outside a data step. But no matter where they appear, SAS processes the Macro statements before it begins executing any data steps. SAS will actually replace the name of the macro and the names of the macro variables with character strings that become part of the data step program that SAS compiles. Unless the program includes a symput function as shown in Example 1, SAS replaces macros and macro variables before it performs any of the file access operations defined in data step programs.

At the end of the partial data step program in Example 2, the data step variable _mval either contains a blank string (if in a observation the variable name does not match any of the tokens on the first list) or the value on the second list related to token on the first list. The remainder of the data step program can use _MVAL as the desired value from the look-up table.

**Example 3: Searching Arrays**

An algorithm found in Gries and adapted here will search three ordered arrays for the first instance of the same value in each.3 A 3GL program based on the algorithm has f, g, and h as array names, i, j, and k as indexes and ni, nj and nk as array dimensions:

```sas
i=0; j=0; k=0;
do;
case
  of f(i)<g(j) and i<=ni and j<=nj:
    i=i+1; return;
  of g(j)<h(k) and j<=nj and k<=nk:
    j=j+1; return;
  of f(i)<h(k) and i<=ni and j<=nk:
    k=k+1; return;
  otherwise: exit;
end;
if f(i)=g(j) and g(j)=h(k) then match='T'; else match='F';
```

A SAS programmer could use SAS arrays and this type of algorithm to search different sets of column values (denoted by variable names) in an observation (row) of a dataset. The program would then produce the SAS data step variable named MATCH. That variable would contain a 'T' in each observation in which one variable in each of the sets f, g and h has the same value, and a 'F' if not. SAS arrays index the columns of arrays that we call SAS variables, not the actual data elements. A SAS construct directly equivalent to Gries' algorithm would search one variable in each of three SAS datasets for matching values.
The equivalent SAS data step merges three sorting datasets by v1 and subsets the output data set:

```sas
data _out;
  merge dl (in=inl)
         d2 (in=in2)
         d3 (in3);
  by v1;
  if in1 and in2 and in3;
```

A roughly equivalent SAS SQL construct bears little resemblance to Gries' algorithm. It has a declarative rather than a procedural form:

```sql
* proc SQL version;
proc sql;
create table _out as
select *
from lib.d1 as f1,lib.d2 as f2,lib.d3 as f3
where f1.vl=f2.vl=f3.vl
```

The SAS SQL program declares a desired result as a relation represented by a SAS dataset (table). The asterisk (*) in the select statement asks for all variables in the three datasets; wise programmers specify the variables they want as f1.vx,f2.vy,..., to avoid conflicts in variable names.

Both of the SAS programs produce more than specified in Gries' algorithm. The output contains observations of all matches of the related key values, but the first observation, if any, in the data step's output dataset contains the first value matching in each array, or in this case, specified column of each dataset. An empty output dataset indicates no matches.

**Example 4: Looping of Function Calls**

The C PROGRAMMING LANGUAGE presents a classic example of 3GL construct: looping through an array and within each loop using a different element of an array as a parameter of a function. The authors use the same basic construct for reading, sorting and writing a text file. A critical part of the program, in abstract C, has a deceptively simple form:

```c
for (i=1 to max) write object(i);
```

If we take variable names as the objects in an array, a loop through a SAS array, writing the value of a different variable within each loop, uses the same program construct. But, as in the prior example, the program operates on variable names, each pointing to a column of dataset. If we take lines of text as the objects, as in the classic C example, SAS makes the details of page-long C program virtually transparent:

```sas
data out;
  file in;
  input textline $char80.;
run;
proc SQL;
select *
from out
order by textline;
```

For the main part SAS controls the pointers to data array elements and looping through files and arrays. The SAS program declares the results needed and names the data objects required. Using sample declarations both simplifies programming and eliminates many of the myriad possibilities for typographical and logical errors in pointer assignments and loop constraints.

Yet at a higher level, programming constructs such as looping of function calls have a role in SAS programs. SAS programmers often become expert program editors. A good SAS programmer may copy and edit almost all of the code he or she uses.

The construct for looping of function calls provides an alternative to copying and editing code. Say we need a program to check for duplicate ID's in different datasets. All of the datasets have multiple keys, such as center+person+specimen, center+person, center+specimen or center+person+year as key
lists. Ten different datasets need checking weekly, but the list may change on short notice. The macro %chkdup, will identify duplicates and print them. It serves as our function. It requires two arguments: the name of the dataset and the key list for that dataset. To define an argument list, we define two SAS macro variables:

```
%* Using abbreviated dsn and variable names;
%let dsn=f1 f2 f3 f4 f5 f6 f7 f8 f9 f10;
%let keylist=c s p, c p, c s p, c p y, c p y, c s p, c p, c s, c p;
```

The first macro variable serves as both an index for the loop and the first dimension of a look-up table. Recall that the look-up table construct uses the data step SCAN function. In contrast, we must use the SAS Macro Language %SCAN function to select a dataset name for each loop. Because the %chkdup function has to scan more than one dataset for duplicates, repeated application of the function requires more than one data step. The loop must operate outside the data steps. Data step functions will not work outside data steps, but the SAS Macro functions will.

Macros in effect do the program file copying and editing for the programmer. They produce much the same benefits as functions that call functions.

By SAS convention the Macro language programs appear within a macro first, followed by the macro call:

```
%macro loop(argl);
%let __n=1;
%let __t=%scan(&argl,%eval(&__n),');
%do %until(__t=);
  %*<look-up key list, using __t to find index>;
  %chkdup(__t,key list);
  %let __n=%eval(&__n + 1);
  %let __t=%scan(&argl,%eval(&__n),');
%end
%loop
```

The program loops through the list of dataset names, using the %chkdup formation to write data step programs and SAS procedure cells. The key list in the %chkdup parameter list came from a macro look-up (not shown).

This program construct can yield benefits beyond its initial saving, if any, of programming time. The technique moves critical specifics of the program out of the main program and into simple parameter lists. Because the program rewrites itself when the parameter lists change, a data editor who understands very little about SAS or any other programming language can respecify the program by changing the parameter lists.

**Example 5: Metaprogramming**

The SAS System keeps track of dataset and variable names, and it uses this information to locate data elements related to the names. Programmers can view the library catalogs. In most cases SAS programmers keep stacks of printouts listing the variable names and types that describe SAS datasets, and they refer to them continually while writing SAS programs.

Metaprograms operate on the data that describe actual data elements, not on the data elements directly. The SAS procedure, PROC CONTENTS, has the option for writing a table of the variable names and their types to a SAS dataset. A program that captures data from the PROC CONTENTS output dataset and uses these data in the program would fit the definition of a simple metaprogram.

Suppose a transaction file contains an ID, a variable name, a prior value and an update value. To update a record, the program merges the update file with the primary dataset by ID, checks the variable to verify that it has the prior value and, if so, replaces the variable value with the update value. This more secure process, as opposed to a standard overwrite, requires a much greater programming effort. In this example, we focus on a simpler task: verifying that the variable names in the transaction file match the names on the primary dataset. This
type of an automated edit performed prior to a major dataset access operation can save much time and expense.

The 3GL program to handle this task would fit into the table look-up construct (see Example 2). For the variable name column on each row of the transaction file, the 3GL program scans an array of valid variable names for a match. If it does not find a match for the name on the transaction file, the program reports the error.

Using SAS we can subtract the table produced by PROC CONTENTS from the transaction file, matching on the variable names. For example,

```sql
proc sql;
select ID, date, varname, val
from Tran
where varname not in (select distinct name from Contents);
```

Working with the SAS contents of the file instead of an external list of legal variable names has certain advantages. Changes to the dataset catalog will not produce a cascade of false errors. Also, the programmer does not have to construct and test a new legal list of variable names to apply the same procedure to a new pair of primary and transaction datasets.

**SUMMARY**

The SAS Macro language's macrovariables and functions, combined with the newer Proc SQL and metaprograms, extend the range of the SAS system. SAS programmers who recognize the equivalence between standard programming constructs and SAS declarations will increase their productivity and gain a greater appreciation of the SAS System.

---

1SAS has a collection or error messages set aside for these occasions. But it can make matters worse if you access the wrong SAS variable and no run-time errors occur!


4The SQL version does not necessarily guarantee that the first observation in the output dataset will have the first instance of matching values in the three datasets. Relational operations do not guarantee any order of observations unless specified.