COMBINATION PROGRAMMING BLOCKS AND SYMBOLIC VARIABLE POOLS

Robert H. Levy

ABSTRACT

This article describes some of the design issues raised in programming batch report writers of 5,000 lines using the SAS* macro facility. The issues include logical program structuring, the relationship of macro code to SAS code, and coordination of referencing environments or symbolic pools with the program structure. Some "rules-of-the-thumb" to simplify coding macros are suggested.

COMBINATION PROGRAMMING BLOCKS

Report writing applications in SAS without macros -- consisting only of data steps and proc steps -- require simpler design considerations than the same applications do in procedural languages. In Cobol or Fortran modules are often designed to perform a single function. A module will edit, compute, or write a report but will rarely perform all three functions. Since SAS performs many of the tasks required of the programmer in procedural languages, SAS modules usually have mixed functions. A regression proc step, for example, combines data selection, computation and reporting. A data step will precede it to read in and edit data. The basic design unit in SAS without macros, however, is end user output rather than programming function. That is, SAS report writing programs without macros are built around the procs that produce the output desired by the end user.

In SAS applications with macros, however, the power and flexibility of the macro facility require design considerations resembling those for procedural languages. An important consideration is dividing the program into manageable sections or design units. Manageable sections can be understood by system designers and programmers; such sections can be tested and maintained relatively independently of the rest of the program. Without these units a batch program would be a mass of inline code. In SAS report writing programs without macros these sections are the proc and data steps that combine to produce the end user output. These sections are easy to identify since each proc statement contains a hard-coded data set name. In batch report writing programs using the macro facility this decision is more complicated. For example, the statement "$A &B;" could mean "run any proc with any data set". That is $A can have the value of any proc and $B the value of any data set. If the values for these symbols were entered in a batch environment, numerous lines of macro and data step code would be required to ensure that the program would never abend. That is, the names of the procs and data sets would have to be validated; and macro statements would be required to implement this logic. The macro statements needed to determine which data sets and which procs would be valid pairs, along with the data steps and proc steps generated, form a combination programming block. This is the largest manageable section for batch report writing programs using SAS macros.

The macro statements within a programming block create a relationship between data and proc steps that is frequently dynamic. That is, it is based on parameters or the results of data steps. Such relationships might include:

- Conditional execution of a proc step based on the results of a previously executed data step (such as suppressing the proc step due to insufficient data).
- Conditional specification of the proc step by parameters (such as choosing which data step variables to process in the proc).
- Conditional specification of data step variables by parameters (a data step variable is formed according to rules specified by input parameters and the proc processes this variable).
- Multiple executions of the proc step according to conditions specified as parameters (such as the creation of data subsets or a particular number of executions).

The evaluation of macro code provides the branching and sequencing capabilities that SAS without macros lacks, generating data step and proc step code for the paths taken within the programming block. Thus, a major design consideration must be specifying the macro based structure of these blocks and their relations to each other.

519
In an orderly, structured program these combination blocks will be arranged hierarchically. That is, the main driver routine will activate blocks subordinate to it. These blocks will activate other blocks subordinate to themselves. All levels will have the option of calling macro subroutine modules -- which are the smallest manageable section of a SAS program. Branching will be controlled by %IF %THEN %ELSE statements. Transfer of control to a routine is thus restricted by the logical structure of the program. A diagram of this structure resembles a pyramid: one routine at the apex expanding to many routines at the base.

![Hierarchical Program Structure](image)

**Figure 1. Hierarchical Program Structure**

**SYMBOLIC CODE WITHIN BLOCKS**

The programming blocks incorporate symbolic information -- macro statements and macro variables -- within themselves. This symbolic information has its own logical structure which is potentially independent of the overall program structure. In the simplest case the macro statements are part of the block hierarchy. If the macro code surrounds a data or a proc step, meaning the code is completely exterior to it, then the macro code may be said to be integrated with the overall block structure. This condition also exists if the macro code is bounded before and after by data or proc steps. Figure 2 illustrates exterior code:

```sas
%IF &A = 1 %THEN %DO;
   DATA ONE;
   X = Y + 2;
   RUN;
%END;
%ELSE %DO;
   DATA ONE;
   RUN;
%END;
```

**Figure 2. Exterior Code.**

If, however, the macro code is interior to the proc or data step then the macro code is independent of the block structure and less comprehensible to the programmer. Figure 3 illustrates this case:

```sas
DATA ONE;
%IF &A = 1 %THEN %DO;
   X = Y + 2;
%END;
RUN;
```

**Figure 3. Interior Code.**

In this example the interior code is evaluated out of the order in which it appears to be written. That is, the %IF is evaluated before the data step is compiled and produces either the data step line " X = Y + 2" or nothing. The code appears to indicate that an assignment statement is being executed, on condition that &A = 1. In fact, however, a branch is taking place; but it is a compiler branch, not an execution time or data step branch. The data step code produced is exactly the same as the code in figure 2. However the branch in figure 3 is invisible. It is never seen in its proper sequence before or after the data step is compiled. It therefore cannot be understood as part of the programming block structure. In fact, it is more accurately understood as a complex, symbolic (or macro variable) reference.

**SYMBOLIC POOLS**

Macro variable references, as opposed to macro code, conform to a logical structure separate from the hierarchy of the blocks. This structure resembles a series of pools rather than a pyramid. First, a macro variable reference -- by definition -- is attached to neither a data step nor a proc step. Instead, it is attached to a referencing environ-
This referencing environment may be global (program-wide) or local (an individual routine and, potentially, all routines subordinate to it). The pool of macro variables available to a given macro routine, therefore, includes all globals, the local variables created in higher level routines, and its own local variables. Figure 4 illustrates the symbolic pool available to each routine in the sequence "A" calls "B" calls "C". The symbolic pool available to subroutine "C" includes the globals, the locals defined in "A", the locals defined in "B", and the locals defined by subroutine "C" has defined.

The organization and usage of the symbolic pools creates a design consideration for the programmer. He must coordinate the pools with a hierarchical coding structure and the resulting code must be clear to another programmer. Some practical "rules-of-the-thumb" can be formulated. First, the wider the referencing scope of a symbol, the more important the issues of definition and assignment become. A local variable in a subordinate macro has limited scope. A program-wide variable can impact all routines. And a program-wide variable, defined in a lower level subroutine, is likely to become invisible to the programmer. Thus, our program will be well organized if we define all program-wide symbols (or Globals) at a single point in the program -- a macro of their own at the beginning -- and assign a constant value to them where possible. These symbols will be less likely to confuse us because we can easily find their definitions. They have achieved a visibility that would otherwise be missing from the source code.

Second, the more we separate code which manipulates symbolic information from code that does not, the easier such code is to follow. The separation of symbolic code from a preceding data step is particularly important. A RUN statement to mark the end of each data step will prevent the assumption that the symbolic code is part of the data step. The RUN statement, which is not required syntactically by batch jobs, is essential for clarity in SAS batch jobs using macroes. Figures 5A and 5B illustrate the difference.

The SAS macro facility does offer methods for impeding the trickle down effect shown above. The programmer could avoid declaring any globals. He could also mask the passing of local variables by declaring local variables with a LOCAL statement. That is, if variable "x" is declared in routine "A" and again declared in routine "B" the second declaration will make the first "x" unavailable in routine "B". However, the design issue raised by the pool structure is where to define macro variables and how to control their use. Macro variables have widespread usefulness. They can communicate between proc steps, serve as work areas, as program constants, or as actual lines of code. However, if the structure of the symbolic pools in a program is left to chance, hidden dependencies between routines can arise and development and maintenance cycles can lengthen. The structure of the symbolic pools deserves explicit design attention just as the structure of modules does.

**RULES-OF-THE-THUMB**

The organization and usage of the symbolic pools creates a design consideration for the programmer. He must coordinate the pools with a hierarchical coding structure and the resulting code must be clear to another programmer.

Some practical "rules-of-the-thumb" can be formulated. First, the wider the referencing scope of a symbol, the more important the issues of definition and assignment become. A local variable in a subordinate macro has limited scope. A program-wide variable can impact all routines. And a program-wide variable, defined in a lower level subroutine, is likely to become invisible to the programmer. Thus, our program will be well organized if we define all program-wide symbols (or Globals) at a single point in the program -- a macro of their own at the beginning -- and assign a constant value to them where possible. These symbols will be less likely to confuse us because we can easily find their definitions. They have achieved a visibility that would otherwise be missing from the source code.

Second, the more we separate code which manipulates symbolic information from code that does not, the easier such code is to follow. The separation of symbolic code from a preceding data step is particularly important. A RUN statement to mark the end of each data step will prevent the assumption that the symbolic code is part of the data step. The RUN statement, which is not required syntactically by batch jobs, is essential for clarity in SAS batch jobs using macroes. Figures 5A and 5B illustrate the difference.

```
DATA ONE;
  z=4*x;
  %LET p=S;
  w=4*z;
```

Figure 5A. Without RUN.

```
DATA ONE;
  z=4*x;
  %LET p=5;
  w=4*z;
```

Figure 5B. With RUN.
DATA ONE;
  z=4*x;
  w=4*z;
RUN;
%LET p=S;

Figure 5B. With RUN.

Unless there is a contrary requirement, the macro code should normally precede data step code within the same macro. As noted previously, macro code is evaluated before the data step is run. Figure 5C is thus more easily understood than 5A or 5B.

%LET p=5;
DATA ONE;
  z=4*x;
RUN;

Figure 5C. Macro code first.

Third, reducing the amount of interior macro code reduces program complexity. Interior code can be difficult to follow and can produce invisible branches. In many cases a symbolic reference can be substituted for interior code. In the example below "&LINE1" could represent any line of SAS code. The required lines can be created by a line generator macro, placed at the beginning of the programming block in which the lines are used or at the beginning of the program.

DATA ONE;
&LINE1;

Figure 6A. Interior code eliminated.

%IF 6A = 1 %THEN
  %LET LINE1 = %STR(x=y*2);
%ELSE %LET LINE1 = ;

Figure 6B. Line generator.

Fourth, use macro call statements with explicit parameters when values (not variables) are to be transmitted from a higher level routine to a lower level routine. The chances of the lower level routine altering the upper level routine's symbolic pool are thereby reduced, and the macro call statement provides documentation for the symbol usage.

Fifth, the smaller the unit of SAS code the easier it is to isolate and control. Develop prototypes of complex or troublesome macro routines before including them in major programs. This allows extensive testing of a critical or difficult section of code in isolation. Errors are then easier to detect.

CONCLUSION

In conclusion the hierarchical design of a batch report writing program is made more complex by the nature of macro code and of the symbolic variable pools. However, by simplifying the types of macro usage allowed, the difficulty of designing, coding, testing, and maintaining the program can be reduced.

ACKNOWLEDGEMENTS

I would like to thank Marta Harrison, Jerry Levinson, and Jeff Phillips for their helpful suggestions.

REFERENCES


SAS is the registered trademark of SAS Institute Inc., Cary, NC, USA.

AUTHOR CONTACT

Robert Levy
1104 South 46th Street
Philadelphia, Pa. 19143
(215) 386-0476.