THE CASE FOR GUIDELINES: A SAS® SYSTEM STYLE PRIMER

Frank C. Dilorio, Computer Sciences Corporation

INTRODUCTION

Whether subconsciously or by slavish devotion to published documents, most people adhere to a set of rules when writing even trivial applications. This paper presents a set of programming guidelines suitable for a variety of SAS applications. It emphasizes that their successful use requires consideration of user, program, and system contexts.

The successful use of guidelines also requires that variations in individual style be respected. This suggests that the traditional call for standards may not be appropriate. The positive impact of guideline use can be significant, especially in SAS-dominated environments.

An Example

Let's begin by posing a seemingly simple problem to some users: add two numeric variables across all observations in a SAS dataset and display the result.

Even though it sounds straightforward there can be a surprising amount of variation in the programs. A casual user might use PROC MEANS or UNIVARIATE. A more advanced user might use SUMMARY and PRINT. The adventurous user may use TABULATE. The user who has seen simple requests mushroom into complex ones may write a DATA step, anticipating additions such as control breaks for subtotals and pagination.

Each user produced the requested figures and all proceeded differently. Why? At least part of the reason lies in differing environmental contexts: people, program requirements (both real and perceived), and programming environments differ. A prescribed method suitable for one environment may be inapplicable, difficult to implement, or even counterproductive in another.

Another reason for the different programs lies in the nature of SAS itself. Did the user see it as a low-level (PL/I-like) language, a high-level "black box", or a combination of the two? SAS offers an unusually high number of creative and effective paths to a "correct" solution. Its schizophrenic nature is at once a blessing and a source of confusion.

Another reason why people attacking the same task may produce different programs is style: the same task's requirements can be satisfied with different programs. The gap between guidelines and disparate, effective programs is programming style.
batch mode, and were often for "mission critical" applications.
• Assembly and Third Generation languages such as FORTRAN and COBOL were the principal tools. Timesharing environments were primitive, expensive, or nonexistent.
• The computing resource was commonly used for fairly primitive, mathematically based calculations.

Given this environment it makes sense that a rules-oriented approach to machine use would emerge. The system was, after all, a scarce, costly resource whose access and use had to be regulated. Compare this to the current computing environment:
• Vast amounts of computing power are within grasp of even low-level managers' budgets. Personnel costs now dominate the decision-making calculus.
• Both "end-users" and professionals have access to the computing resource. User populations are often dominated by people who are less interested in programming than they are pursuing inquiry in their field of substantive interest.
• Use of the machine is often unstructured, ad hoc, and exploratory, usually taking place in real time. Applications range from trivial to sublime to "mission critical".
• Tools for communicating requests are more sophisticated. "User friendly" menus, 4GL's, artificial intelligence, and other interactive tools abound.
• The range of feasible tasks is bounded only by the user's imagination. Numeric, text, graphic and image manipulation are all commonplace.

The latter-day definers of standards and allocators of computing power are justifiably overwhelmed by this decentralization and explosion of resources. They are faced with a wide range of users, applications, data types and hardware all within the same organizational environment.

The control problem is further compounded in SAS-dominated environments. These tend to be dominated by end-users using SAS's diverse product offerings in a wide variety of applications. Standards must accommodate this breadth of use. They must also respect the variation attendant in the applications' heuristics.

Guidelines

The perceived benefits of traditional standards are difficult to refute. A possible alternative to a standards-dominated environment is one utilizing programming aids, or guidelines.

Guidelines answer the question "what, given our environment, constitutes good programming practice?" Rather than the standards' focus on the "what" and "how" of a task, they suggest general techniques and why their adoption would be beneficial. Pedagogy, rather than control, is the objective. Recommendations, rather than strictures, predominate.

A set of guidelines is not an exhaustive collection of rules covering all possible contingencies. Rather, it is a subset of these, selected by features of the computing environment. The environment filters out inappropriate rules and constraints. A reasonable number of appropriate guidelines ensures that they will not be ignored, their purpose circumvented or subverted.

The features of the site's computing environment are central to the guidelines' success. They are discussed below.

Environments

Environmental features may be grouped into three categories: user, program, and system. The person charged with designing a set of programming guidelines should ask "how important or relevant are these features in our organization? Can we ignore them without consequence?"

User Environment Issues These include:
• Job function The end-user-professional programmer mix of the site is relevant. It affects the number of guidelines, their sophistication of phrasing, and their likely acceptance.
• Environment Is it better to "carrot" than "stick", especially when the user population is predominantly end-user.

Program Environment Issues Included here are:
• Consumer If a program is not for self-consumption consider the difference in skill level between the author and user. If not for self use the techniques should be lowest common denominator.
• Insularity If the program or the data are part of a system selection of naming conventions, data types, and the like become important.
• Size and Complexity As either increase, so does the need for order and rules. Large or complex tasks also increase the likelihood that guidelines will be adhered to or even sought out by the developer.
• Criticality If the programs or data are "mission critical" the need for quality assurance increases. The range of "correct" behavior narrows as the importance of the application grows.

System Environment Issues The operational environments include:
• SAS Feature availability affects approach and technique. The variety of IPP's installed at a site and the version of Base SAS have a great influence on program design. Cross-system compatibility must be
An Illustration

Two venues from the author's background illustrate the different impact of environments on guideline selection. The first site was a large bank developing a SAS-based management reporting system. The second site was a university data library handling student and faculty requests for U.S. Census data.

The consumers of the bank programs were middle and upper-level managers. The system required more than 12,000 lines of code in over 50 programs, which varied greatly in size and complexity. The available computing power, the complex data manipulations, and the size of the datasets demanded that programs be highly tuned. The original system design was written for SAS release 82.4, and thus did not have the macro language available. Report formats were highly customized, conforming to the accounting and financial analyst interests of the design team.

By contrast, the consumers at the library were usually end-users who would refine and modify a "skeleton" program produced for them. Usually only one program per request was needed. There was adequate computing power available, so programs could be written without using fast-executing but obscure techniques. Output format was usually not an issue: simple listings or summary datasets were the usual products.

Even this quick comparison highlights the influence of environment on design: the same set of guidelines would not be appropriate at both sites. The bank's environment required emphasis on program efficiency and consistent representation of data elements. The library's orientation required guidelines which emphasized straightforward coding techniques and effective use of PROCs.

II. GENERIC GUIDELINES

This section presents a generalized set of design and coding guidelines. They do not purport to be comprehensive, and reflect the author's biases and limitations. User, program and system contexts may render some points irrelevant and others critical.

The discussion covers tools common to all SAS users: the DATA step and procedures found in the BASE and STAT products. The macro language and IFP's such as SAS/AF, SAS/FSP® and database interfaces are not explicitly covered. General design issues are addressed first, followed by items specific to the DATA step and procedure usage.

DESIGN: Presentation

Use consistent case. All program text except comments and character constants should be consistently entered in upper or lower case. This improves readability and makes most program editors' text search functions perform more reliably.
Use blank space liberally. Indent statements in DO loops. Split long or complex statements into generously spaced multiple lines.

Clearly separate units of work. Use RUN, SKIP, and PAGE statements liberally. This makes macro definitions, DATA steps, significant events in DATA steps, and procedures easy to identify.

DESIGN: Naming

Choose meaningful names. Datasets, variables, formats, labels, macros, and libraries should have names which communicate their content. At minimum, choose a name which suggests where the reader can look for further explanation. For example, the variable name "q12a" may be more suggestive than a tortured mnemonic like "mgivusub": it indicates where on a survey instrument the user could look for more information.

Avoid default names. Don't take default dataset names in the DATA statement. Likewise, use the DATA= option in procedures to make explicit dataset references and avoid confusion.

DESIGN: Documentation

Use labels. Variable and dataset labels extend your ability to convey entity meaning.

Establish an audit trail. Leave identifying information on all but the most formal output. Leave the DATE and NUMBER system options turned on. Use TITLES or FOOTNOTES to indicate the name of the program which generated the procedure output.

Explain with comments. Use a comment block to explain the entire program's function, then smaller blocks before DATA steps and procedures. Insert comments at critical points in DATA steps or anywhere else requiring an explanation of calculations or logic. Don't overcomment: too much explanation is as bad as too little, and program changes are not always reflected in the accompanying comments.

DESIGN: Environment

Document use of obscure options. If an option will make casual reading difficult, highlight its use with comments. IMPLMAC, $*, and WORK= are examples of these options.

Restrict %INCLUDE'd code. Don't reset system options within included external code. Don't branch out of an included DATA step segment.

DESIGN: Strategy

Consider cross-system incompatibilities. SAS and system features differ across platforms, causing unforeseen problems. For example, the ASCII and EBCDIC collating sequences differ. This can throw a dataset between a micro and an IBM mainframe out of sort order even though the data were not physically rearranged.

The collating sequence difference can cause a poorly constructed comparison to produce unexpected results. This comparison:

```sas
if char < '0' then ... ;
```

will catch alphabetic characters in the EBCDIC collating sequence, but not on machines which use ASCII, since 0 through 9 are smaller than any alphabetic character. This comparison would execute correctly in both environments if written:

```sas
if A{'Ol<=char<='9') then ... ;
```

Don't overcode. Exploit SAS's mixture of Third and Fourth Generation languages. A long DATA step may be more effectively written as a sequence of DATA steps and/or procedures. Output dataset capabilities of many procedures add to this DATA-procedure mix.

Don't clever code. Consider the "consumer" of the program and ask if he/she can understand obscure or compact coding conventions. A verbose, multiple statement sequence may be more effective in the long run than a single compact statement. Which of the following is more readily understood?

```sas
x = a > b * c > d;
```

```sas
if a > b & c > d then x = 1;
else
 x = 0;
```

Develop incrementally. Don't code the entire problem at once. Develop the most important layers of the application first and deal with display and tuning issues later. Work with a subset of the data to control expense and speed job turnaround.

Assume growth. Even the most innocent-looking applications tend to grow in size and complexity. Avoid restrictive data structures, variable naming conventions, and calculation methods. Put programs, data, and formats in separate libraries even if it is not immediately apparent that such elaborate organization is needed.

Organize programs and data thematically. Use your system's subdirectory, partitioned dataset and other file management features to group items related to a task. Data can be in
one library, programs in another, and formats
in a third.

DATA Step: Design

Group unexecutable statements. Statements
which "declare" variables, set up arrays, and
perform other housekeeping chores should be
grouped: LENGTH, ARRAY, and RETAIN statements
should go near the top of the step. DROP, KEEP,
RENAME, ATTRIB, and LABEL statements should go
near the bottom, beyond any executable code.
This placement scheme allows quick location of
a particular type of statement.

Overwrite cautiously. Avoid overwriting
to variables and datasets. Ask if you can really
afford to lose the original entity’s contents.
Don’t reuse intermediate variables in a DATA
step: their replacement can impede the debug-
ning process.

Declare character variables. Use LENGTH
statements to explicitly set the length of
character variables. This is particularly
important when a variable is created by char-
acter functions.

Identify RETAINed variables. RETAINed vari-
bles behave differently than others, so treat
them accordingly. Specify them in a RETA:IN
statement or begin their names with a readily
identifiable character, such as an underscore
(_).

DATA Step: Flow of control

Restrain use of GOTOs. A WHILE or UNTIL
clause as part of a DO statement or an IF-THEN-
ELSE sequence can usually accomplish the same
purpose as a GOTO, and can do so more elegant-
ly. However, if the WHILE or UNTIL condition
becomes too convoluted consider using a GOTO.
GOTOs should only branch down, and never branch
out of %INCLUDEd code.

Ensure closure of IF-THEN-ELSE sequences.
When assigning variable values or when branch-
ing, test all logical conditions, using an
unconditional ELSE to capture all remaining
comparisons. This is especially critical when
a SET is placed in a DO loop: every variable is
implicitly RETAINed, so unassigned values are
"left over" from previous observations.

Simplify logical expressions. Write logical
conditions and expressions as you would speak
them. Avoid complex, "not"-filled, or excess-
vously long logical conditions. Reword the
condition: use parentheses to highlight and
clarify the order of evaluation, minimize use
of negatives, and break complex conditions into
several statements.

DATA Step: Calculations

Simplify complex expressions. Break complex
calculations into several statements. At mini-
num, insert blank space across multiple lines
To improve legibility.

Use functions. There is no reason to hand-
code a square root function or loop through an
array to compute a univariate statistic. Func-
tions simplify your code, reduce the number of
program statements, and make program output
more reliable.

Use arrays. Statements which fall into a
predictable pattern are usually suitable for
array processing. This makes code compact,
clearly identifies the nature of the calcula-
tions, and simplifies maintenance.

Avoid mixed data type calculations. SAS
will do its best to convert a character vari-
able used arithmetically. It sometimes suc-
cedes, and always consumes a surprising amount
of CPU time doing so. Be sure you’re working
only with numbers. Use the INPUT function to
perform conversions prior to the arithmetic
operation.

Use formats and IN to recode. The IN opera-
tor and user-written formats with PUT functions
are effective and efficient means to recode
both numeric and character variables. Unless
you have a fairly small number (say, fewer than
10) of categories to collapse
avoid using IF-
THEN-ELSE sequences for recoding.

PROCEDURES: Strategy

Review capabilities. Use of some PROCes
tends fall into computational ruts. Period-
ically review all the capabilities of commonly
used procs to keep up to date and avoid unne-
cessary work. This is especially true with new
releases of SAS.

Reconcile Log to output. Suppress the "I
finally got some output!" response. Output may
appear to be "reasonable", but there may be Log
messages indicating problems with number of
cases, missing values, inability to invert,
redundant parameters, plot values out of range,
and so on. Not all procedures will print warn-
ing messages on the output file.

Use DATASETS. Use the DATASETS procedure
when the only operation you need to perform on
a disk-resident SAS dataset involves changing
directory information. There is no need to
pass through the data when all you want to do
is rename a variable or change a format.

Use new PROCes incrementally. When using a
new or "long lost" procedure, don’t start with
a complex request unless you are sure of the
impact of the options and parameters. Start with as many defaults as possible and develop the more complex analyses gradually. This is especially appropriate for multivariate statistics and most graphics (who ever got a complicated PROC GCHART right the first time?).

Exploit output datasets. Familiarize yourself with the nature and contents of the datasets created by procedures. The best way to understand their structure is to print them, along with a CONTENTS listing.

PROCEDURES: Calculations

Let the PROCs do the work. Unless computer resources are at an absolute premium there is no reason for computing univariate statistics across observations in a DATA step. Let MEANS and SUMMARY do the work they were designed and optimized for.

Choose among overlapping procedures. Many procedures have similar capabilities. You can often realize great savings in time (yours and the computer's) if you are aware of which procedure is best suited for your task. For example, if you simply want a compact listing of means, MEANS would be preferable to UNIVARIATE. If your work is more exploratory, UNIVARIATE's output would be more appropriate.

Use IML for set-oriented calculations. When calculations simultaneously go "across" and "down" observations consider using SAS/IML. Its compact notation reduces the chance of errors due to improper handling of retained variables, complex arrays, and the like.

III. CONCLUSION

The variety and complexity of contemporary programming environments makes the specification and administration of standards a difficult and usually unnecessary task. A set of context-sensitive guidelines will not only achieve many of the standards' objectives but also instruct the users in good programming practice.

Ultimately, the organization as a whole as well as the users will benefit. Programs will come from a population of users who are able to retain their personal style and learn from the lessons and techniques promulgated by the guidelines.

Acknowledgements

Some of the ideas expressed here are not new either to the SAS world or programming in general. I owe a debt of thanks to those who participated in a spirited 'SAS Style' discus-

Further Information

The author can be reached at:

Computer Sciences Corporation
PO Box 13886
MD B3-03
Research Triangle Park, NC 27709
(919) 541-1552

Bitnet: DIORIO@NIEHS

Some of the ideas expressed here are not new either to the SAS world or programming in general. I owe a debt of thanks to those who participated in a spirited "SAS Style" discus-

501