

But, It Runs

Grace J. Lossman, ASG, Inc., Cary, NC

Abstract

Time after time in both industry and academia, SAS programmers claim exemption from the rules of computer science. They claim justification for this because "...I'm just a SAS programmer!", or "...because I've had no formal training." These fallacies and others will be examined in this paper. This overview will be a discussion of style, efficiency, coding, testing and debugging, couched in a semi-formal presentation of program analysis and design. Attention will be given to the importance of knowing elementary details of the operating system being used to run SAS.

Introduction

This tutorial presents an overview of the facets to consider when writing programs. More than that it is meant to be a philosophy. The intent will be to point out areas that are often neglected by beginning programmers either from lack of knowledge or from a desire to hurry. Specific attention with examples will be given to programming strategies and it will be shown that this is directly related back to quality software definitions espoused by the industry.

These are meant only as an opening for further study. The goal is to use the SAS system efficiently. While the strategies and methodologies presented should be especially helpful to the beginning programmer, this information can be of equal interest to project and data managers.

Background

SAS, unlike many other programming languages, (notable and divergent examples

being C and Fortran, the former highly structured while the latter is less so), can be particularly tricky. SAS by nature of its seemingly facile and unassuming form, deserves from us more attention to detail and rigor than the more fundamentally classical languages. The time saving PROC's encourage the non-programmer, or perhaps the less formally trained programmer to often disjointedly piece together a running program. No consideration seems to be given to the numerous linkages, to style, or to planning. These linkages between data step programming and the PROC's are neglected and the program becomes the infamous bowl of spaghetti.

Quality programming involves developing and testing software applications. Software excellence begins with dedication to the goals of software engineering as recognized by the industry as fundamental requirements. Quality software is defined by four interrelated issues:

- Modifiability
- Understandability
- Reliability
- Efficiency

In considering modifiability remember that most of us are working under industrial requirements which dictates we must produce, produce and produce in the shortest time frame. Having a global view of the project means that your code is written with generic considerations which will allow it to be taken and easily modified for use on another project.

Understandability dictates that it is written in a logical and analytical manner which is easy to quickly comprehend and easily modified for use on another project.

Reliability insists that the code is robust and without exception withstands all conditionals applied to that particular venue. The code written is global and not local.

Efficiency is multifaceted and covers both the avenues of human endeavor and machine performance. It leads to both a reduction in runtime and a reduction in effort.

To fulfill these criteria, much advance planning is needed. The majority of time should always be spent in the design phase where understanding of the requirements is solidified.

This paper will be offering a particular perspective based on these four points.

Overall Quality

The SAS program runs and a return code is received indicating an all clear. The SAS log is viewed and no information is seen which might cause concern. What is known at this point? That it is error free? That it is correct? What does this mean?

Your supervisor wants macros. Maybe he has been told that this is the hallmark of a great programmer and he figures he'll be a great boss if he has a great programmer. At this point most SAS programmers are either satisfied or they trot off to do the macro stuff, because a great boss and a great programmer have to work hand in hand no matter what the problem dictates. But, what about the program?

1. Was adequate and appropriate use of the operating system made. Consider the CPU. Consider I/O. As a consultant I once shared an office with someone who said to me, '...it's hard for me to believe I could be a better SAS programmer by knowing something about the operating system....'. So understanding all your tools doesn't make you better at your job?

2. Was the programming well thought out? Was there a lot of repetitive code where a macro might have been more appropriate?

Were sorts over used? What about storage and space?

3. Was a flow chart made before the programming process started? In other words, was the problem fully understood at the beginning? The single most important point in all of programming is planning. The problem must first be understood and then it can be solved. The project needs to drive SAS and not SAS drive the project.

4. Was a good programming style adhered to? Is there a consistency of usage of statements in similar situations? Was documentation included in the program?

Program Strategies

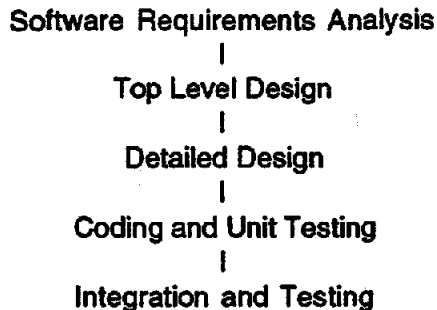
The output of the four original points: modifiability, understandability, reliability, and efficiency are linked by a global kind of quality directly to the last four points: operating system use, programming constructs, planning and programming style. The quality of your output is directly related to these last four point which are directly linked to the definition of quality software.

Additionally, there is much interplay between some of these points and sometimes an increase in one leads to a decrease in another. Knowing your operating system now becomes of particular importance and understanding how your program will handle these resources. Tradeoffs are always possible in the environment for a given situation.

These strategies all highlight planning and understanding an overview of the project. As we proceed it will be demonstrated by example either directly or indirectly how all these are joined together. Cross referencing the examples we view will add clarification to these issues. While much of the material is simply a quantification of conventional wisdom, a more detailed discussion of these areas can be gained by the references listed at the close of this paper.

Beginning Tutorials

A few words about the design phase of a project/program before we proceed. Viewing a typical project development life cycle may help put this idea in perspective.



Further analysis of this topic is beyond the scope of this paper. The above is offered for orientation.

Program Strategy Examples

How do all these principles of modifiability, understandability, reliability, and efficiency relate to everyday work? How do the four methods for quality programming relate to the previous principles? In an applied sense how do we know our program is more or less efficient?

The 'quality' methods - operating system use, program coding, program design and program style - are directly related to the four principles. Our programs will then be more easily modified (because good style was used to code), understood, (again clear coding and style), reliable, (much planning from original flow chart/design), and efficient (operating system used appropriately, clear code written, good design and style). The SAS system makes it very easy for us to dissociate this rather relative term from the abstract. SAS tools exist for collecting performance and storage statistics and these provide a guideline for testing various run-time techniques.

Your SAS program uses system resources in the form of CPU time, I/O operations and memory. The performance statistics from these resources can be collected when your program compiles and executes by using the SAS system

options STIMER and FULLSTIMER.

Resource testing by different programming techniques is called benchmarking. When benchmarking is done it is important to use consistency in both your code and the environment so that appropriate comparisons can be made.

Always separate code clearly and include a RUN after a PROC or complete DATA step.

This:

```
data test;
    set abc;
    if a = 5 then b = 7;
run;

proc freq data=test;
    tables a/list out=freqtest;
run;

proc print data=freqtest;
    title 'test set abc';
run;
```

Not this:

```
data test;set abc;
if a=5 then b=7;
    proc freq;
    tables a/list;
    proc print;
run;
```

The first set of code will print performance statistics after each RUN which will make sense because the programmer can break down and analyze problem areas while the next set is useless. No performance statistics will be printed until the RUN statement and at that point it will be impossible to locate any real problem area. Which code is easier to understand? Which is easier to modify?

Likewise, do benchmarking under similar system conditions by making sure that roughly the same number of users are on the system and

that there are no resource intensive processes (e.g., system backups) going on at that time. Also, don't forget to keep the same method of operation consistent (i.e., batch vs. interactive).

And finally, never base your conclusions on a single run.

Good style should be present at all times and it is not a separate issue to be considered alone. I would however like to begin by considering some simple stylistic issues which will make writing easily modifiable and reusable code much easier.

The example which I am first going to present is an extreme one, but one which will more than show the importance of good logic and style. I was given this program to audit and it was written by the same person I spoke of earlier who identified herself as an excellent SAS programmer.

A few brief excerpts first and then we will discuss this:

1) both poor and/or incorrect programming practices.

```
proc sort data=&STDY.d.stat out=stat;
by studyno center pt examno;
proc sort data=&STDY.d.stat out=ce;
by study center subj; run;
```

Explanation: first sort has no RUN to release it to the compiler. Both sorts are coded all on one line. This is incorrect. This code can not be either easily modified or understood.

2) incorrect programming practices

```
if dimiscop=2 & totcss < 3 then efftr=1; else
if dimiscop=1 | sum(eryth,prurit)>1 then
efftr =2; else efftr=9;
```

Explanation: besides being hard to read this is very inefficient code (more later on that). Always put the next statement on the next line

when a semi-colon is encountered.

3) incorrect style and very poor programming.

```
if center=0 or pt=0 then delete;
length thstchr thendchr $7;
if 0<wkspther<=4 then wksince=1;
else if.....
else if.....
```

Explanation: LENGTH statement needs to be after SET statement. This code is mentally disorganized and no indentations are used. Again it is not easily modifiable, easily understood or efficient.

4) incorrect style

```
if anymc=1 then do; if dtmcst='M' then do;
dstart=.; ncstchr='MISSING'; end;
else do; d=1*(substr(dtmcst,3,2)); if do=0
then do; d=15; daymiss=1; end;
```

Explanation: figuring out this before you modify it is a real nightmare, I know!

This is probably enough for you to get the idea of what was going on here. After I got my blood pressure down, I went to my supervisor and explained that I'd need a day or so to redo this 1300 line program so that I could read it before I had any hope of auditing it. When I pointed out to the original programmer that this was the sloppiest piece of work I had ever seen, she commented, '...Yes, but it runs..' and it was at that moment that my paper was born.

This discussion will begin with coding. Always code clearly as this makes the program easier to understand and easier to modify and as a result increases human efficiency. Use a program header containing the program name, what the program requires, who the author is, etc. Use comments within the body of the program. This will save the original programmer time and anyone making modifications time. Use meaningful variable names. Clearly separate steps in the program by a blank line. Avoid 'cutsey' complex expressions that are both

Beginning Tutorials

difficult to easily understand and not terribly machine efficient. Put all code in your program rather than bunches of %INCLUDEs. Set up neat, logical conditionals.

The end result should be something of this form:

```
*****;  
* Program: Where To Begin ;  
* ;  
* Date: July 28, 1995 ;  
* ;  
* Author: G. Lossman ;  
* ;  
* Requires: Files x, y and z ;  
* ;  
* Produces: SAS data sets i, j & k ;  
* ;  
* Program Description: ;  
* ;  
* Updated: ;  
*****;
```

Always use test data which is realistic and complete. During the development process it is more efficient to test incrementally, than to run the entire program. While this may make perfect sense to all of you, it is surprising to see the number of people day after day who insist that they are debugging as they run the entire program. This is simply ludicrous.

As you do your incremental testing between data steps remember to use PUT statements to output intermediate results within the data step. Never make the assumption that all your raw data is perfect and be in the lookout from less than perfect data from your input medium. Always examine the raw data and use system utilities to make appropriate corrections.

As show earlier with the poor and incorrect placement of the LENGTH statement, it is important to write the statements you need in the order that you need them. It is a CPU efficiency issue for both the order of your statements and their exclusivity.

For a given observation, if only one condition

can be true use a series of IF-THEN/ELSE statements as opposed to IF-THEN statements without ELSE's.

This:

```
if organ = 'heart' then  
    bodygrp = 1;  
else if organ = 'liver' then  
    bodygrp = 2;  
else if .....  
    ;  
else if .....  
    ;  
run;
```

Not this:

```
if organ = 'heart' then  
    bodygrp = 1;  
if organ = 'liver' then  
    bodygrp = 2;  
if organ = .....  
    ;  
if organ = .....  
    ;
```

This is, of course, a CPU issue. In the former example the SAS system stops checking when a condition is true. However, when ELSE's are not used, there is checking for each condition for every single observation.

Missing values are likewise a big problem with CPU efficiency. Always try to make certain when writing expressions that involve missing values to evaluate these last. The fewer the number of operations that involve missing values, the less 'bookkeeping' SAS needs to do.

So how is this problem avoided? Check for missing values prior to a loop.

This:

```
if state ne . then  
do;  
    total = person + state;  
end;
```

Not this:

```
total = persont + state;
```

Advantages of using SAS PROC's are numerous. Many programmers prefer to write a DATA step or develop a macro instead of using what SAS has so thoughtfully provided for them. As opposed to a DATA step with an array statement and a retain, use PROC TRANSPOSE.

Data step program:

```
data x;
  array y{4} x1-x4;
  retain z 1 y;
  set a;
  by num;
  .....
  - more statements -
run;
```

SAS PROC:

```
proc transpose data=x out=transp;
  by num;
run;
```

Take advantage of output DATA sets from PROCs in your programming. PROC CONTENTS has a myriad of nifty options.

And finally in this brief introduction, do not overuse sorts. Remember that sorted observations remain in that order. This little trick will save you loads of CPU, I/O and memory. Plan your sorts by sorting more than one variable at once and use a planned order.

Use KEEP and DROP statements to insure that you work with only the data you need.

I have only touched on introductory topics. I hope these examples encourage you to become a more formal SAS programmer.

References

Dilorio, F. (1989), "The Case for Guidelines: A SAS System Primer", *SUGI 14*, Cary, NC, SAS Institute Inc. pp 496-501.

SAS Institute Inc. (1990), *SAS Programming Tips: A Guide to Efficient SAS Processing*, Cary, NC, SAS Institute Inc.