The Dirty Dozen: Twelve Common Programming Mistakes

Bob Virgile
Robert Virgile Associates, Inc.

Overview

As programmers begin using the SAS® software, their programs often contain many of the same types of mistakes. A seemingly innocuous error, such as omitting a semicolon, may generate results ranging from an obvious error message to an impossible to decipher error message to destroying an existing dataset. This paper covers typical errors made at an introductory level, including those which produce no error message but still generate the wrong result.

#1: Step-by-Step Execution

SAS programs execute one DATA or PROC step at a time. A program cannot mix and match the steps. Here is a typical attempt:

```
DATA PURCHASE;
SET SALES;
TOTAL = PRICE * QUANTITY;

PROC MEANS DATA=PURCHASE;
VAR TOTAL;

WITHTAX = TOTAL * (1 + TAXRATE);

PROC MEANS DATA=PURCHASE;
VAR WITHTAX;
```

The WITHTAX= assignment statement generates an error message. The syntax would have been correct if the statement had appeared in a DATA step. However, the statement is part of PROC MEANS. So the software generates an error message saying that the assignment statement is invalid or used out of proper order. The message is only partially clear, since the statement is valid. The problem is that it is used out of proper order.

#2: Informat with Columns

In an INPUT statement, informats must immediately follow a variable name. A legal example:

```
INPUT  @21 D_BIRTH MMDDYY8.;
```

An illegal version:

```
INPUT D_BIRTH 21-28 MMDDYY8.;
```

This type of error, while common, is not fatal. The software generates an error message, and the error message is somewhat clear. One hybrid exists, to indicate the number of implied positions after a decimal point. These two statements are both legal, and are equivalent to one another:

```
INPUT AMOUNT 11-18 .2;
INPUT @11 AMOUNT 8.2;
```

Let's move on to situations where the error message is difficult to decipher.

#3: Missing Semicolons

In this program, the software complains that the ELSE statement has no matching IF / THEN statement. What could be the problem?

```
DATA NEW;
INFILE RAWDATA;
INPUT NAME $16.  /* First + Last */
      RACE 17    /* 1-digit code */
      AGE  18-20 /* Some oldies! */

IF AGE >= 65 THEN STATUS='SENIOR';
ELSE              STATUS='JUNIOR';
```

You could stare at this one for a long time if you didn't know that the topic is "missing semicolons." When the INPUT statement is missing a semicolon, the software "thinks" that the IF / THEN statement is part of the INPUT statement. The words within the IF / THEN statement are taken to be additional variables which the INPUT statement should read. The INPUT statement ends with the semicolon before the ELSE statement. With no IF / THEN statement, the ELSE statement is illegal.

The lesson here: when a statement appears to be perfectly valid but still generates an error message, check for a missing semicolon on the PREVIOUS statement.

Consider a DATA step which creates three datasets at the same time:

```
DATA MALES FEMALES BADDATA;
SET IN.EVERYONE;
IF GENDER='M' THEN OUTPUT MALES;
ELSE IF GENDER='F' THEN OUTPUT FEMALES;
ELSE OUTPUT BADDATA;
```

A perfectly legitimate program. But what about this one:

```
DATA MILK
SET ALL.COWS;
CUPS = 4 * QUARTS;
```

Because of the missing semicolon on the DATA statement, this program is asking that three datasets be created in one DATA step: MILK, SET, and ALL.COWS. The program contains no SET statement, and does not read from any source of data. Technically, however, these are legal statements. So all three datasets on the DATA statement get created with one observation and two variables.

The dataset ALL.COWS just got destroyed! This is a truly bad result! In addition, there would be no error or warning messages. (The LOG would contain a note about the number of observations in each of the datasets.) In my opinion, the names SET, MERGE, and UPDATE should be illegal as dataset names to guard against this possibility. Version 7 of the SAS software will provide such an option. A global option will allow the words SET, MERGE, UPDATE, and RETAIN (or other DATA step key words as well) to trigger an error message.

#4: Comments

Commenting errors fall into two categories: using comments improperly and failing to use comments where they should be used.

As a general rule, programs should contain a comment at the beginning, showing (at a minimum) the name and purpose of the program. Other information, such as creation date, name of the programmer, and relationship of the program to other programs, can be added if appropriate. Different styles exist, ranging from a single comment statement to a formal block that looks more like this:

```
*****************************
**                         **
** Program:                **
**                         **
** Date:                   **
**                         **
** Purpose:                **
**                         **
** Programmer:             **
**                         **
*****************************;
```

Despite the number of lines, this is still just a single SAS comment statement beginning with an asterisk and ending with a semicolon.

My personal preference is to use the first title line to indicate the name of the program which created the output. Many times, the output gets separated from the program. It's nice to be able to look at some output months down the road and to know immediately which program created that output. As long as the client finds the output acceptable in this form, it makes good sense to reserve the first title line for the program name.

Beginning errors of comission fall into a few categories. First of all, it is perfectly okay to leave a blank line in a SAS program. It's even a good idea to use spacing (as well as indentation) to make your programs easy to read. However, there is no need to comment out that blank line.

Secondly, when using a comment statement, make sure it ends with a semicolon. While this might be overkill, it is acceptable:

```
* PROGRAM:  PROJECT/LIB/PROG1.SAS;
*                                ;
* PURPOSE:  ILLUSTRATE COMMENTS  ;

DATA SENIORS;
INFILE RAWDATA;
INPUT NAME $ 1-20;
```

Of course, it would work equally well if the final two asterisks and the first two semicolons were removed. Omitting a final semicolon is a common error:

```
* PROGRAM:  PROJECT/LIB/PROG1.SAS

  PURPOSE:  ILLUSTRATE COMMENTS

DATA SENIORS;
INFILE RAWDATA;
INPUT NAME $ 1-20;
```

Now the comment statement ends with the first semicolon, after the word SENIORS. The software complains that the INFILE statement is invalid or used out of proper order, because the program does not contain a DATA statement.

Finally, when using embedded comments, avoid placing /* in columns 1 and 2 of the program.

```
DATA JUNIORS;
INFILE RAWDATA
/* SOMETIMES USE RAWDATA2 */;
INPUT NAME $ 1-20;
```

Under the MVS operating system, /* in columns 1 and 2 is considered to be job control language, not part of SAS language. This program would immediately end with an incomplete INFILE statement. Even if you're not working under MVS, it is usually a good idea to design your programs to be portable from one machine to the next. So avoid /* in columns 1 and 2.

#5: Lengths of Variables

The DATA step assigns a length to variables based on the first mention of the variable. For example, this code assigns GENDER a length of 4:

```
IF TYPE=1 THEN GENDER='MALE';
ELSE GENDER='FEMALE';
```

The order of the data has no effect on the length of GENDER. The first value of TYPE could be 1 or 2. For that matter, all values of TYPE could be 2. Still, the variable GENDER has a length of 4.

This situation produces no error or warning message. GENDER takes on values of MALE and FEMA.

A straightforward solution exists. Assign a length to GENDER BEFORE the IF / THEN statements:

```
LENGTH GENDER $ 6;
```

#6: Missing versus 99

Often, data contain a special code to indicate a missing value. For example, a survey might ask, "On a scale of 1 to 10, please rate the importance of these items." The survey might record answers of 1 to 10, and use the following codes for missing answers:

97 = Don't know
98 = Refused to answer
99 = Want to check with spouse

These values cause havoc when processing the data later. For example, to avoid averaging in the nonresponsive answers with legitimate answers, PROC MEANS would have to process just one variable at a time:

```
PROC MEANS DATA=SURVEY;
VAR Q1;
WHERE Q1 NOT IN (97, 98, 99);
```

A better solution would be to recode these values to missing values in the permanent dataset. The SAS software supports 28 different missing values (., .A through .Z, and ._), so you won't lose the distinction between 97, 98, and 99:

```
IF      Q1=97 THEN Q1=.A;
ELSE IF Q1=98 THEN Q1=.B;
ELSE IF Q1=99 THEN Q1=.C;
```

#7: $8 vs. $8.

These INPUT statements produce quite different results:

```
INPUT @21 LASTNAME $8.;
INPUT @21 LASTNAME $8;
```

With a dot, 8. means "read 8 characters beginning at the current location on the line (column 21)." LASTNAME will contain the contents of columns 21 through 28. Without a dot, 8 means "read the contents of column 8." Even when the INPUT statement first moves to column 21 (@21), LASTNAME will be whatever is in column 8. The second INPUT statement really says, "First, move to column 21. Then take the value of LASTNAME from column 8." That's exactly the result, with no error or warning message.

#8: Using ELSE

ELSE can cause trouble in a few ways. Failing to use ELSE makes a program take slightly longer to run. For example, the second set of statements runs faster than the first:

```
IF GENDER='F' THEN TYPE='FEMALE';
IF GENDER='M' THEN TYPE='MALE';

IF GENDER='F' THEN TYPE='FEMALE';
ELSE IF GENDER='M' THEN TYPE='MALE';
```

Failing to use ELSE above is a minor error. The only bad thing that happens is that the program takes longer to run. When beginning to use ELSE, programmers frequently forget to consider the possibility of bad data:

```
IF GENDER='F' THEN TYPE='FEMALE';
ELSE TYPE='MALE';
```

GENDER may have contained values other than M or F. Finally, in allowing for bad data, programmers may use poor logic:

```
IF GENDER='F' THEN TYPE='FEMALE';
IF GENDER='M' THEN TYPE='MALE';
ELSE TYPE='??';
```

Since the program contains only one ELSE statement, TYPE will never be FEMALE. Any values of FEMALE get changed to ?? by the final statement. Again, the solution is straightforward:

```
IF GENDER='F' THEN TYPE='FEMALE';
ELSE IF GENDER='M' THEN TYPE='MALE';
ELSE TYPE='??';
```

One final note. While the SAS software provides extreme flexibility with respect to spacing and indentation, there must be a blank before the word THEN. Many letters (T, D, X, B) trigger a different interpretation of your expression if they immediately follow a closing quote.

#9: Testing Programs and Subsetting

In the testing stage, programmers often run on a subset of the data until the bugs are worked out. Beginning programmers run through a few variations on subsetting technique, such as (from worst to best):

```
IF _N_ <= 100;
IF _N_ > 100 THEN STOP;
OPTIONS OBS=100;
```

Still, all of these techniques can run into trouble when trying to produce such a sample while also using a subsetting IF. For example, the objective here is to test on a sample of 10 observations:

```
OPTIONS OBS=100;

DATA SAMPLE;
INFILE RAWDATA;
INPUT NAME $ RANK $ SERIAL $;
IF RANK='GENERAL';
```

It turns out that the first 100 observations contain zero observations having RANK='GENERAL'. So the programmer tries again using:

```
OPTIONS OBS=1000;
```

Again, there are no generals in the first 1000 observations. So the programmer tries:

```
OPTIONS OBS=5000;
```

All of a sudden the program runs for a while, because there were 2000 generals in the first 5000 observations. To get around this trial and error approach, count the number of generals in the DATA step:

```
DATA SAMPLE;
INFILE RAWDATA;
INPUT NAME $ RANK $ SERIAL $;
IF RANK='GENERAL';
N + 1;
OUTPUT;
IF N=100 THEN STOP;
```

Without an OUTPUT statement, SAMPLE would contain only 99 observations. The STOP statement means "right here, right now, without doing any futher work." The DATA step would have ended before outputting the final observation.

#10: Counting with PROC MEANS

Use each procedure for its intended purpose. PROC FREQ counts, while PROC MEANS computes statistics. The most common error in this category is trying to count with PROC MEANS:

```
PROC MEANS DATA=SALES N;
VAR AMOUNT;
BY STATE;
```

versus

```
PROC FREQ DATA=SALES;
TABLES STATE;
```

Given that SALES contains the numeric variable AMOUNT, and given that AMOUNT never has a missing value, and given that SALES is in sorted order by STATE, the N statistic in PROC MEANS counts how many times each STATE appears in the dataset.

Use PROC FREQ to count. Look at the complexity of the programs as well as all the conditions that have to be right for PROC MEANS to do the job correctly. In addition, what happens if AMOUNT takes on a missing value? PROC MEANS still works. However, the N statistic no longer counts the number of times each STATE appears. It actually counts the number of nonmissing AMOUNTs for each STATE. While there is no error message, the output is incorrect.

#11: DISP=(NEW,CATLG,DELETE)

This is JCL, not SAS language. Although this error applies to one operating system only, I have included it here because its effects are quite difficult to trace.

DISP=(NEW,CATLG,DELETE) is part of the standard JCL DD statement to create a new SAS data library. The three parameters mean:

NEW: the current status of the dataset. It doesn't exist, so create one by that name

CATLG: when the job successfully completes, catalog the dataset by name.

DELETE: if the job ends abnormally instead, delete the dataset.

What happens when the program contains a SAS error, such as:

```
DATAIN.LARGE;
```

Despite the SAS error, the MVS operating system determines that the job completes successfully. The third parameter (DELETE) only kicks in if their was a SYSTEM error, not a SAS error. (For example, the job requested 1 minute of CPU time but actually hit that limit without completing.) So the operating system creates the dataset, and catalogs it by name.

Next, the programmer fixes the SAS error, changing the statement to:

```
DATA IN.LARGE;
```

When the new job gets submitted (using the original JCL), the operating system has a problem. The job says DISP=NEW (create a new dataset under that name), but a dataset already exists with that name. The dataset may contain garbage, but it does exist. Two things could happen at this point, depending on how the systems programmers have set up the operating system:

1. The system may bounce out the job with a JCL error. That's the preferable result.

2. The job may run, but the system would create a second, uncataloged version of the dataset. The original version, containing the garbage, remains cataloged by name. So when a program refers to the dataset later, it always finds the dataset containing the garbage.

If the programmer notices the situation, it's easy enough to correct. Either delete the dataset before rerunning the job, or else change the JCL to read DISP=OLD instead of DISP=NEW. However, this error is extremely difficult to notice. The only message is buried in the JCL log (not the SAS log) with a small note next to the dataset name saying NOT CATLGD 2. Even experienced programmers rarely check that portion of the output. In fact, many programmers inspect the output on the terminal screen, discover there is no longer an error message, and delete the job so it isn't even available to inspect later.

On a related note, most MVS systems periodically examine the disk packs for uncataloged datasets, either deleting them or reporting them to the owners. Usually (perhaps not if you are a timesharing customer) you would not rack up a bill forever for storing an uncataloged dataset.

#12: Failing to Check the LOG

Just because you get output, it doesn't mean the program is correct. Check the SAS log to confirm that the program ran as expected. Here are a few situations where you will find output, but the program is incorrect.

When a procedure uses a BY statement, the software works with one value of the BY variable at a time. It does not check the entire dataset to make sure that the data are in sorted order. Instead, output gets generated for values of the BY variable until the software discovers an observation is not in sorted order. At that point, the software generates an error message on the SAS log. Net result: there is output (albeit incomplete) but the SAS log contains an error message.

Some methods of execution involve submitting a program.SAS file and automatically generating a program.LOG and program.LIS (or LST) file. When running jobs in this fashion, the previous execution of the program already created a .LIS file. If a revised job runs, it may contain errors and not generate a .LIS file. However, the .LIS file still exists with reasonable looking output, since it is left over from the previous execution of a working program.

Finally, the SAS log may contain notes about logic errors which make the output incorrect. Consider this program:

```
DATA INCOMES;
MERGE MOM POP;
BY ID;
TOTALINC = MOMINC + POPINC;

PROC MEANS DATA=INCOMES;
VAR TOTALINC;
```

When you inspect the output, it might look very reasonable. But consider what would happen if POPINC were actually a character variable that usually took on numerals as a value, but could occasionally be "N/A" or "> 100000." For most observations, the software would convert the numerals to the proper numeric value, and compute TOTALINC. But when POPINC could not be converted to numeric, TOTALINC would be missing. PROC MEANS would automatically discard those observations and print the average value of the nonmissing TOTALINCs. How could you know this was happening? Check the SAS log!

What should you check for? At a minimum, check for error and warning messages. Verify that each dataset contains approximately the expected number of variables and observations. Check for any notes indicating an unexpected action took place, such as numeric to character conversion, or missing values being generated as a result of performing mathematical operations on missing values. Finally, verify that each procedure generated approximately the expected number of pages of output.

All comments and questions are welcome! Contact information:

Bob Virgile
Robert Virgile Associates, Inc.
3 Rock Street
Woburn, MA 01801
(781) 938-0307
virgile@mediaone.net