Abstract

User friendly programming should be the aim of all software development. The idea should apply not only to application programs for the end user but to the development of programming languages for use by application programmers. However, developers have given relatively little attention to the ergonomics of programming languages—how do the thought processes of the human mind affect user friendly programming? I start by describing how a minor change in the use of the semicolon provides ergonomic benefits to SAS programmers. I then follow with a potpourri of procedures I use, some of which are in direct conflict with recommendations made by authors at previous SUGI's and relate them to my own general philosophy of programming. We all have constraints imposed upon us and cannot always follow every procedure in textbook fashion as we might desire. Part of programming is knowing the trade-offs involved.

The Lowly Semicolon

Ergonomics and computer programming. Webster defines ergonomics as the "science concerned with the characteristics of people that need to be considered in designing and arranging things that they use in order that people and things will interact most effectively and safely." Computer programming is certainly one of those "things" alluded to in this definition. However, much remains to be done in applying this concept to programming languages. For example, although Fortran 0 was first designed around 1954, it was a full 10 years later in 1964 that Bohm wrote the first paper that might be regarded as an attempt to look at the human element in programming. Dijkstra followed with "Programming Considered as a Human Activity" in 1965 and his well-known 1968 paper, "GOTO Statement Considered Harmful." However, these papers are only a first step. Today we can and should go much further in implementing ergonomics in the broadest sense of the word. For example, examine the use of the semicolon in SAS programming and other languages which use a free format.

The Problem. What is the most common mistake made by new programmers? It is one that each of us will likely continue to make occasionally throughout our careers. We all know it is forgetting to place a semicolon at the end of each statement. In fact, a study of 589 Pascal programs by Ripley and Druseikis reported that the omission of a single token accounted for 41 percent of the errors found, and in almost half of these instances the missing token was a semicolon. The omission of a semicolon accounted for one out of every five errors! Why is this error so common? Let me speculate briefly.

We all do many things by habit. In fact, without habit we probably could not live very effectively—we would spend all our time concentrating on trivia. However, our habits are not etched in stone. There seems to be a pattern or sequence involved in most habits, and if something happens to interrupt that sequence, there is a very good chance it will not be completed. Regardless of the theoretical basis for the behavior, I believe empirical analysis supports this contention. Consider some examples:

How many times have you left the keys in your car's ignition? Compare this with the number of times you have forgotten to put the key in the ignition. I'll bet you have left your keys in the car much more frequently than you have forgotten to put them in the ignition. The reason, I believe, is that putting the keys in the ignition is at the start of a sequence of events in operating your car—and if the keys are in your hands, you've got to do something with them! However, removing your keys is another story. It occurs at the end of the sequence of events; at a time when you are more likely to be distracted by other people or by thoughts about what you will be doing next (the sequence to follow).

Now, contrast the number of times you've seen a sentence started without capitalizing the first word against the number of times you've seen a sentence without an ending period. More mistakes occur at the end of this sequence of events than at the beginning—especially among those just developing skills in sentence writing. Again, the mind is preoccupied—in this case, evaluating alternative means of communicating a particular thought—and becomes sidetracked from the requirement of ending the sentence with a period.

I believe computer programmers have a similar problem. Their thought process is much clearer when they start to write (or type) a statement than when completing it. For whatever reason the sequence of thoughts required to enter the complete statement is interrupted before the statement is completed, and the semicolon is omitted.

The remedy and its advantages. What can we do about the problem? Like many other programming languages, SAS software requires each statement to end with a semicolon. I believe this is a mistake in design from an ergonomic point of view. However, just as a three dimensional
cylinder may look like either a rectangle or a circle if viewed from the proper perspective, let us, as programmers, view the use of the semicolon from another perspective. Except for the last program statement, the semicolon may be viewed as a token used to separate statements. Therefore, the software-SAS software in particular-doesn't care whether the semicolon is used at the beginning or end of a statement—except for the special requirements of the first and last statements. In addition, the SAS System provides a null statement and is very consistent in the way it uses the semicolon. These features allow us to overcome this design discrepancy. The semicolon may, with minor changes in programming habits, be placed at the beginning of each statement with several advantages.

Make fewer errors. As previously stated, actions completed at the beginning of a sequence are less likely to be interrupted than those required at the end of a sequence. The result is less programming errors. In addition, if you use cards statements to input data, the first program line following the data must always have a semicolon. Using this new technique, you'll always meet that requirement, regardless of the length of the first programming statement following the data.

Improved readability makes for easier error checking. Compare the following two code segments. Is there any question as to which is easier to check for the missing semicolon? That "ragged right edge" is the key, especially in a really long list of statements complete with combinations of right parentheses, single and double quotes, and commas as well as our lowly semicolon. As the statement lists show, the placement of a semicolon before each statement in conjunction with proper indentation of statements furthers the goal of every programmer—improved readability, and results in easier checking for missing semicolons.

```plaintext
SELECT (A); WHEN ('FR') X='MISSING DOCUMENTS: - TOTAL'; WHEN ('FU') X='FINANCING: NO. UNITS'; WHEN ('SU') X='FINANCING: DOLLAR AMT.' WHEN ('TV') X='FINANCING: - TOTAL NO.' OTHERW ise X='************MISSING***********'; END;

SELECT(A); WHEN ('FR') X='MISSING DOCUMENTS: - TOTAL'; WHEN ('FU') X='FINANCING: NO. UNITS'; WHEN ('SU') X='FINANCING: DOLLAR AMT.' WHEN ('TV') X='FINANCING: - TOTAL NO.' OTHERW ise X='************MISSING***********'; END;
```

Easier to alter. The SAS System provides a high level language which allows many options when executing various procedures. If you add an additional line to specify additional requirements in the program statement, you must then go back and remove the semicolon from what had previously been the end of the statement. This in itself is extra work, and, if you get into too big a hurry, you'll forget to remove it and end up executing your program with the statement terminated before it gets all the newly added information. The SAS software then tries to read the added information as a new, separate programming statement, and the program terminates with an error. The conventional method either requires an additional step to remove the semicolon or results in an error upon execution of the program. However, using the new method,

```plaintext
;DATA YEAR80(KEEP=YEAR X Y)
```

requires only the insertion of the new information on an additional line to become

```plaintext
;DATA YEAR80(KEEP=YEAR X Y)
YEARS(KEEP=YEAR A B)
```

Easier to comment out statements with editing software. Many times I have, knowing better, placed several statements on one line. When this happens there is no easy way to have the editing software comment out the additional statements. This occurs because I (correctly) consider the comment symbol as occurring at the beginning of a statement, but consider the semicolon as occurring at the end of the statement. I sometimes get in a hurry and end up commenting out the first statement on a line but not those "extra" statements. This happens because my editing software only changes the first occurrence on each line. However, all editing software allows the changing of one series of characters into another. Thus ";" becomes ";*" without any problem and,

```plaintext
;W=X ;X=Y ;Y=Z
```

becomes

```plaintext
;W=X ;X=Y ;Y=Z
```

with one simple stroke.

For these reasons I find that using a semicolon at the beginning of a statement is much more "natural" and improves my productivity. In addition, SAS software is extremely consistent in its use of the semicolon. This makes the transition relatively painless. However, there are still several adjustments to be made to your programs since SAS software, like all other programming software, has not been designed to require a semicolon before each statement.

Converting current programs. For consistency, I always place a semicolon before the first statement. Since my first statement is always the name of my operating system source file placed in a comment, this is no problem for me to remember. Likewise, a semicolon must be placed before the first data line when using a CARDS statement. In using

579
One exception in using the technique is when using name style macros calls. In this case the "prelimiter" to separate it from the other statements. And finally, a semicolon must be placed at the end of the program.

One exception in using the technique is when using name style macros calls. In this case the "prelimiter" to separate it from the other statements. And finally, a semicolon must be placed at the end of the program. Following the macro call may cause the macro to malfunction (see SAS User's Guide: Basics, Version 5 Edition, p. 658). Should this occur, the "prelimiter" of the statement following the macro call should be omitted. (How many of us use name style macro calls?)

Programming Techniques

Programming Philosophy. Now that I've (hopefully) convinced you there are valid reasons to "put the horse before the cart" in an ergonomic sense, I'd like to share with you some other programming techniques I use, emphasizing their practicality. To start, I believe programming is both an art and a science. However, to no other group is the old adage "Give a person enough rope, and he'll hang himself" more appropriate than to computer programmers. In fact, I believe the success of the Macintosh is directly related to the fact that programmers and users are limited in the way they are allowed to accomplish tasks. There is a price to pay for this required structure, but for those without the discipline to develop and apply their own structure, the benefits are well worth the price. Fortunately, we as SAS programmers are not so narrowly constrained. We have some latitude to develop our own structure, but we must maintain the discipline to develop and apply it.

My programming environment. I work for an audit organization as a statistician. This fact places some constraints on the way I organize and carry out my work. I am in a support position for the organization's primary function. I am part of a team effort in which timeliness of product output is a major consideration. As such, anything I can do to make the auditors' jobs easier will be a great help. I work to help minimize the total team effort required to produce the end product. My environment includes:

1. No SAS software for my Personal Computer. Everything is done on a mainframe through TSO.

2. I have no high speed printers to aid in program development by providing quick job turnaround.

3. Efficiency is important only for very large data sets which I analyze at the beginning of the audit cycle.

4. Data used to make statistical projections is in a special format. It is on computer spreadsheets, designed for workpaper exhibits, not for input to the mainframe. It is also being updated on a continuous basis. Keeping two sets of data to update on a continuous basis is not a good option!

5. Timeliness is very important. However, only a few runs will be made for each program. This allows a trade off. I will use the computer in a less efficient manner if it means an overall savings in manpower and/or time which translates into deadlines met and/or money saved.

The specific programming techniques I use depend on external constraints. Although I have many programming principles that I will not violate, I do use some techniques that previous SUGI papers downplay. Program clarity is extremely important, but also a matter of opinion. Sometimes, trade-offs must be made to accomplish the organization's mission. Often, there are techniques I use in one project which would be inappropriate in another. The important thing is to explicitly recognize the specific techniques that should be used in the current project.

Communicating with the client. One technique we should all be familiar with but probably neglect to use is to have the client provide a diagram of the output desired from the computer. Many times communication between the computer consultant and client is poor. The client talks in generalities instead of specifics and may not know exactly what he wants. Moreover, many times what the client says he wants is not what he actually wants. The computer can provide the desired output only if the consultant really knows what is desired! The act of diagramming the desired output forces the client to be specific and helps him clarify in his own mind exactly what he wants. It also provides a focal point for discussion of what the computer program is supposed to do and provides concrete information for the programmer to use in planning the programming and documentation. Remember, without knowing the output needed by the client, you cannot determine the information required as input. And finally, the diagram helps generate ideas for alternative output formats. I am continually looking for ways to convert program output into a more usable form for the client.

Establishing naming conventions. A very common bit of advice is to use meaningful variable names. I find this suggestion much more appropriate for my Turbo Pascal programs than for my SAS programs. It is impossible to have meaningful variable names for 200 variables which cover possibly 6 main areas and 20 subareas of a subject when names lengths are limited to 8 characters. Consequently, proper programming techniques and naming conventions for the job series are essential. Some of my ideas are:

580
I have had, on occasion, a need to use multidimensional arrays. In this case, a numbered variable list or a range variable list becomes very useful (see SAS User's Guide: Basics, Version 5 Edition, p. 18), especially if a very large number of variables are used or if you wish to use a macro variable to explicitly change the size of an array dimension. In other words, many times in SAS programming, the structure of the data affects the choice of my naming convention.

Slight variations of a name for different purposes can be very confusing and consequently, result in a bombed program. Fortunately, SAS software allows the same name to be used for various purposes, and these purposes are easily recognized by their context. For example, I can use a data set name in my SET statement and the same data set name in the accompanying DATA statement. I can also use a permanent SAS data set name which is the same as the accompanying DD statement name, and the data set may also have a variable of the same name. In addition my OS data set name will likely also have the same SAS data set name embedded in it. Consequently, all of the following statements (including JCL) may be in the same SAS program:

```
/AUDITI DD DSN=FIRSTLVL.AUDITI.DATA,DISP=SHR
;DATA AUDITI
;SET AUDITI.AUDITI
;INFILE AUDITI
;INPUT AUDITI
```

If available and appropriate for the situation, use SAS labels to improve readability of program output.

Commenting. Commenting is essential, but excessive commenting of transparent details detracts from program readability and comprehensibility, actually impedes the debugging effort, and dramatically increases the time required to get a printout of a program listing. Some of my uses of comments include:

1. When using a new, unfamiliar technique or programming statement, I place a comment in the source code giving the publication reference for the idea. In addition, I keep notes in a stenographer's notebook and reference these notes in source code comments.

2. When a section of source code is no longer required, I leave it in the source file as a comment, along with the date the code was last used. For example, I may sort a permanent SAS data set. Because sorting is so expensive and the required order is now secured, I comment out these coding statements and insert the date when I last used the code to make the sort.

3. When using an input statement to input variables from an external file, I use a separate line for each variable and insert a "/* name comment */" after the required program statement information for that variable. In subsequent programs which use the permanent SAS data set created from this external file, I insert this input statement as a comment in the new program. One quick look at this input statement and the meaning becomes clear, even months later. For example,

```
:*INPUT
YEAR /*FISCAL YEAR STARTING OCT. 1 OF
THE CAL. YR*/
AMTO /*DOLLAR AMOUNT ORIGINALLY
CALCULATED*/
AMT1 /*DOLLAR AMT (AUDITED)*/
```

Since a series of programs are required to complete an audit job cycle, I am particularly careful about providing an audit trail of data set files created. Starting with the first data set used in the first program of the series, I give the source, compilation date, and purpose of the data set with comments in the Job Control Language (JCL). As I progress from one computer program in the series to series to another, I copy all these JCL comments to all subsequent programs, whether or not the data sets are required in that particular program.

Using macros. Macros are essential, but should be used with caution. In some cases, they are the only way to efficiently accomplish an operation which must be repeated a number of times. Clarity is a major consideration. I highly recommend the use of macro variables, especially when a long group of variables will be needed in several places in the program. I group all macro variables together at the beginning of the program.

For testing and debugging programs, I have a macro which prints a specific sequence of observations from a SAS data set, along with a title (comment) which indicates my specific reasons or concern for printing them out. The macro is attached to all programs by concatenating the program to it. I do this by placing the macro source in a separate file and referring to it with a SYSIN DD statement. The macro includes the statements:

```
/*MACRO FR PART(DSN,MSG,FOBS,LOBS)
;PROC PRINT DATA=DSN (FIRSTOBS=FOBS
OBS=LOBS)
;TITLE="DATA SET: &DSN., OBS. &FOBS -
&LOBS •••
&MSG"
;ZMEND FR_PART
```

Data input from a spreadsheet. This can be a real problem when writing an input statement, especially if there are many variables and columns' widths are non-uniform. However, I use the spreadsheet itself as an aid. First, insert a row of digits to represent the columns, numbering from 1 to however many columns are needed for all the variables. This line should be entered as a label, not a number, and could include up to 240 characters...
in a single cell in Symphony. Next, insert a line above the digits and place an "S" (for starting column) in each cell, left justified. Finally, insert another line below the digits and place an "E" (for ending column) in each cell, right justified. You can then simply read the columns occupied by each variable (cell). The left and right justification markers line up directly above and below the starting and ending column. In the following example, we easily read the variables as occurring in columns 1-3, 5-7, 9-19, 21-28, and 30-33. As an added bonus, the descriptive headers supplied by the auditor are plainly visible.

<table>
<thead>
<tr>
<th>S</th>
<th>T</th>
<th>SAMP</th>
<th>CLAIM</th>
<th>AUDITED</th>
<th>ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>1234567810123456783012345678401234567</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20000</td>
<td>15000</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10000</td>
<td>10000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>25000</td>
<td>20000</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Saving program output. It may be useful to save important program output on a computer file. In one case the mainframe somehow changed the accessibility of one of my source files so that it was not available to me. Naturally, this happened at a critical time. I downloaded the output from a SAS program to a Symphony file, used several short Symphony macros to clean up the source code from the SAS log, and uploaded the "new" source code to the mainframe. In well under an hour new program output was forthcoming. In other instances, the output was needed for instructions or reports for agency personnel. Downloading and reformatting program output has, at times, saved much time in retyping and proofreading retyped data, particularly in preliminary draft reports involving hundreds of variables.

Using the Symphony Communications Environment. I use Symphony rather than another communications program because of its CAPTURE mode. It captures all transmissions to and from the mainframe and allows me to instantly look at the interaction between myself and the mainframe. This is great when I've made a mistake. I quickly review and correct it. If necessary, I can upload specific source lines back to the mainframe instead of correcting those same lines on the mainframe source file. As previously explained, I can capture specific lines from output files and use Symphony macros to quickly make changes and then upload them to be used in source files.

Summary

The use of a semicolon preceding, rather than following, each programming statement is an ergonomically sound programming technique which will aid productivity by making allowance for common human tendencies and by increasing efficiency. In addition, various other techniques will improve your productivity when using SAS software. Establishing naming conventions and proper commenting are particularly important. The key to using most techniques is flexibility. Rules of thumb are just that. The programmer must assume responsibility for analyzing the project and choosing the techniques most applicable after considering the various trade-offs involved. The key requirement is that the techniques not be unduly "cute." Readability, clarity, and standardization must take precedence above all other factors. However, the standardization used may vary from project to project. The important thing is to explicitly recognize the standards in effect for the project.

Acknowledgments

Macintosh is a registered trademark of Apple Computer, Inc.
SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA.
Symphony is a registered trademark of Lotus Development Corporation.
Turbo Pascal is a registered trademark of Borland International, Inc.

References


Contact Author

Henry Foster, US Department of Agriculture, Office of Inspector General - Audit, 114 Commerce Street, Room 812, Dallas, TX 75242
Commercial No. (214) 767-2080