PhUSE 2006

Paper CS03

Lessons Learned: Valuable (But Hidden) SAS Details

Dante diTommaso, F. Hoffmann-La Roche, Basel, Switzerland
Benjamin Szilagyi, Novartis Pharma, Basel, Switzerland

ABSTRACT

Sometimes the simplest SAS® task becomes frustrating and unnecessarily complex. Often the complexity of SAS, itself, is the cause. Crucial details are easily hidden among the thousands of pages of SAS documentation, technical notes, release updates, etc. The authors demonstrate a collection of SAS features that are not well known, somewhat counterintuitive or poorly documented, but all are surprisingly useful once learned and mastered. Mysterious SAS behavior that might have inspired "work-arounds" should now become transparent. Topics include environment settings; system options; and operators, functions and statements from Base SAS®, SAS Macro Language® and SAS/Graph®. Gain efficiency and programming confidence by bringing light to a few dark corners of SAS. This review is intended for programmers of all levels.

INTRODUCTION

This paper stems from several bad programming days in recent years. Such days can have several root causes, but this paper deals with only one: incomplete or unintelligible documentation. SAS is a vast system. People had to design, implement & test each feature. Most likely someone else had to explain each to us, the users. For programmers, inaccessible documentation leads to a familiar result: lost time staring at code, holding our heads, rubbing our eyes, thinking "Why doesn't that work?"

Too often, the most expeditious approach is to simply code around unexpected behavior. But this leaves behind a trail of troubling SAS mysteries, and clutters a programmer's confidence with fears of techniques that just don't seem to work properly. In recent years, the authors have had several of these experiences. The situations were so common that we simply had to understand what SAS was doing and why. This paper contains the conclusions of these investigations, along with interesting features or techniques we learned along the way.

Rather than remembering which mysterious techniques to avoid, a confident and efficient programmer understands subtleties of SAS as well as broadly familiar capabilities. Only then can one harness each with equal confidence in the appropriate situations. We hope our investigations take you further towards such expertise, as they have us.

SAS ONLINEDOC, V8 & V9.1.3

Whenever useful, we will reference the appropriate pages from SAS' OnlineDoc. Of course, these references are only useful if you can access this outstanding resource. Luckily, SAS provides a convenient portal to SAS OnlineDoc and other resources (see **References**, below). Throughout this paper, we will reference v8.2 OnlineDoc, but version 9 users should easily find the same information. These experiences all stem from SAS 8.2 efforts.

SAS SYSTEM OPTIONS

Before examining specific examples, let's first consider the programming environment. Several SAS System options are so useful that all SAS programmers should be aware of them on nearly a daily basis. But most likely, as with the authors, these have remained unknown for too many years.

-RTRACE AND -RTRACELOC

Next time you want to know exactly what files SAS accesses during a particular session, activate this pair of invocation options. SAS will then create an additional log of each resource it accesses. You may expect SAS OnlineDoc (v8 or 9.x) to describe these options under:

►Base SAS ►**SAS Language Reference: Dictionary**
►Dictionary of Language Elements ►**SAS System Options**

However, these options are not available under all operating systems. Therefore, SAS only mentions them in the relevant "Host Specific Information" sections. For example:

►Base SAS Software ► **Host Specific Information**
►(Windows | UNIX) Environment ► Host-Specific Features ►**SAS System Options**

Unfortunately, creating a complete resource log is not this simple, since SAS only logs path & filename and not other crucial information such as file date or size. A bit more programming is necessary to fill in the missing information. Even then, keep in mind that between SAS accessing a file and you processing the -RTRACELOG log file to fill in the missing info, the files on disk may have changed. Perhaps this seems unlikely, but it is nonetheless a real risk.

MPRINT AND MFILE ...

Most SAS programmers learn about the MPRINT option the same day they begin using SAS Macro Language. However, few learn about its partner, MFILE. Unlike partners -RTRACE & -RTRACELOC, above, programmers can turn MPRINT & MFILE on and off throughout a program. The greatest benefit is while debugging complex macros.

The MFILE option instructs SAS to begin writing resolved macro code to the file specified in the special MPRINT fileref. The MPRINT system option must be active. Don't let the MPRINT system option and MPRINT fileref confuse use; they have the same name but one is the familiar SAS System option, while the fileref lets SAS know where to write resolved macro code.

The MPRINT option prints resolved code to the SAS log. But each line in the log begins with the macro name, so this is not immediately usable. The MFILE approach, however, provides code that you can submit directly to SAS.

... AND THE SAS DATA STEP DEBUGGER

Now imagine having to debug a system of programs that is even moderately complex. MFILE prints out immediately useable code; saving the intermediate data sets that this code uses is trivial; and the SAS Data Step Debugger helps you step through the MFILE code using the intermediate data sets. Using this combination of underappreciated features can greatly simplify code review & debugging.

The simple-to-use Data Step Debugger is a vastly useful and underused SAS resource. While a tutorial is beyond the scope of this paper, programmers can greatly improve their trouble-shooting skills by spending the necessary five minutes learning about it (see also **References**, below).

►Base SAS ►**SAS Language Reference: Dictionary**
►Appendices ►**Data Step Debugger**

Recommendation:	Combine MPRINT, MFILE, intermediate data sets and the Data Step Debugger to reduce debugging efforts. How SAS produces an unexpected result quickly becomes transparent.

SERROR

The function %SYMEXIST() is available to version 9 users. But for the rest of us, it is difficult to confirm the existence of a SAS Macro symbol without generating a `WARNING`. Note that the question of *existence* is different from the question of what *value* a symbol holds. Programmers can easily check *value* with a statement like: "`%if &macsym = YES ...;`"; or the DATA STEP version, "`if resolve('&macsym') = ...;`". However, if the symbol has not been declared, then SAS generates a familiar `WARNING` and possibly an `ERROR`:

```
WARNING: Apparent symbolic reference MACSYM not resolved.
ERROR: A character operand was found in the %EVAL function or %IF condition
where a numeric operand is required. The condition was: &macsym = YES
```

Enter the SERROR system option. This turns off the `WARNING`, above, which is very useful for developing code that simply needs to check whether a macro symbol already exists before taking the appropriate action.

Unless used carefully, this option can actually make debugging code rather difficult, since it can hide the original source of code failure. Therefore, programmers should centralize the checking of symbol existence, and turn SERROR back on as soon as these checks are complete.

MERGENOBY=

SAS initializes the MERGENOBY system option to NOWARN by default. We cannot imagine why, since its sole task is to warn a programmer that they have forgotten the BY statement for a MERGE step. We doubt that we have ever *intentionally* written a MERGE step without a BY statement. SAS calls this "One-to-One Merging", as opposed to the "Match-Merging" that we use almost exclusively in clinical research:

►Base SAS ►**SAS Language Reference: Concepts**
►Data Step Concepts ►Reading, Combining, Modifying ►**Combining SAS Data Sets: Methods**

If you have never read through the Match-Merge section, referenced above, then we highly recommend you do so soon. We recommend WARN as the default value for the MERGENOBY option. If you even need a no-BY merge, it is simple to eliminate the warning for that rare situation.

Recommendation:	Review the details of common clinical merges such as Match-Merging, above; and Make MERGENOBY=WARN your default option setting.

MSGLEVEL=

PROC SQL automatically warns programmers about ambiguous references to variables from multiple data sets:

```
WARNING: Variable <VAR-NAME> already exists on file <LIB-NAME>.<DSET-NAME>.
```

However, the DATA STEP MERGE statement is not so careful, by default. Unknown system option MSGLEVEL controls this behavior. By default, MSGLEVEL=N, which means SAS does not warn programmers that values from a secondary data set are overwriting values from a prior data set. System option MSGLEVEL=I changes the default. We believe the default setting promotes careless programming. Still, the DATA STEP MERGE will not produce a warning like that of SQL. Instead, it produces a non-standard INFO message:

```
INFO: The variable <VAR-NAME> on data set <LIB-NAME>.<DSET-NAME-1> will be
overwritten by data set <LIB-NAME>.<DSET-NAME-2>.
```

Recommendation:	Make MSGLEVEL=I your default option setting; and Automate log scanning, making sure the scanner searches for the non-standard SAS message "INFO: The variable ..." (see **Log Scanning**, below).

TAKE ADVANTAGE OF THE SAS IDE

So far these SAS system options apply to both batch & interactive modes. For interactive SAS sessions, several features are available that may solve long-standing complaints you have with SAS' integrated development environment (IDE). The IDE includes the familiar Program, Log and Output windows, along with the Explorer and Results windows. Most SAS programmers quickly recognize the value of task-specific configuration of the Keys and Toolbar for each of these windows.

You can customize other aspects of the IDE, although documentation is either hard to find or nearly unintelligible.

RECOVER AN INTERACTIVE SAS SESSION

But first, if you are working in an interactive session, eventually you will submit code that causes an apparently unrecoverable error: unbalanced quotes, bad macro definition, etc. It seems the only solution is to shut down SAS and start a new session. But an even quicker solution is to submit a few meaningless statements that are, in fact, able to save your session: `*))%*))'"*/; %mend; proc save_me; run;`

Note that, by itself, this string is four complete statements: A strange comment (`*))%*))'"*/;`); A macro statement

(`%mend;`); And an invalid PROC block (`proc save_me; run;`). The comment closes any unmatched quote, open parenthesis, or open comment. The macro statement then closes any macro definition fragment.

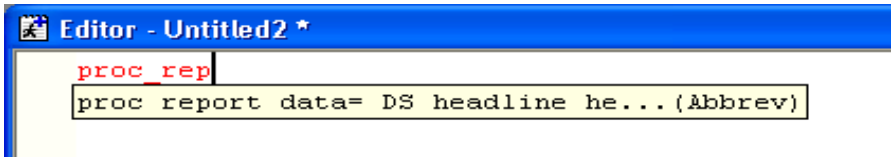
The invalid PROC block simply confirms that SAS has recovered and is once again correctly interpreting code. These statements are able to undo just about any offense you can throw at the SAS interpreter. If your session is in really bad shape, you may have to "submit" the string several times. Once SAS reports `"ERROR: Procedure SAVE_ME not found."`, then you know that SAS has recovered.

Recommendation:	Key definition that handles the embedded, unmatched quotes in these statements: `cle log;gsubmit "*))%*))'";gsubmit '"*/;%mend;proc save_me;run;';`

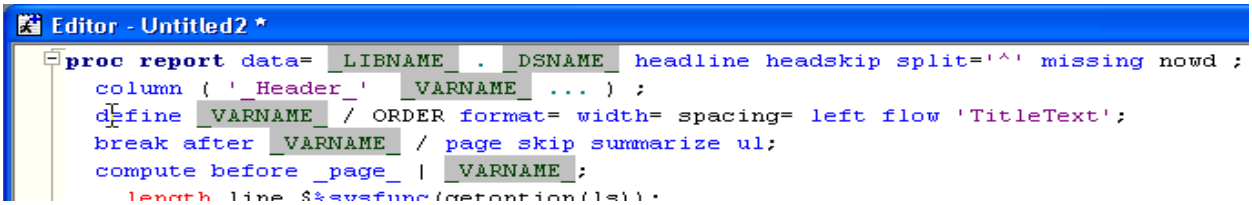
ABBREVIATIONS & KEYBOARD MACROS

Two excellent references already exist to help programmers fully harness the power of Enhanced Editor keyboard macros and abbreviations (see Paul Grant & Arthur Carpenter, in **References**).

Instead of a attempting to replace these brilliant resources, we simply clarify the difference and tempt you with the possibilities. Store both abbreviations and keyboard macros in your customized SAS IDE. **Abbreviations** are most useful for code substitution. For example, instead of trying to remember the main features of complex SAS Procs, simply store generic proc template code in the SAS IDE and give it a unique and simple abbreviation, such as "PROC_REP". Then, the next time you need to write a new proc report block simply type "proc_rep". SAS will recognize the abbreviation and give you the option of "tabbing" to replace the abbreviation with the template code:



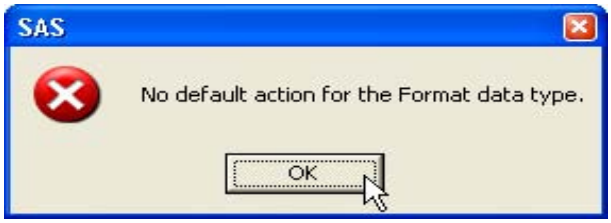
Hitting the "TAB" key gives you instant template code, ready for fine-tuning. Note in the image below, how programmers can also register user-defined keywords, such as _LIBNAME_, which SAS highlights. The appearance of user-defined keywords is also customizable. Paul Grant's paper, above, is exceptionally straightforward; one of the most usable SAS Institute publications you are likely to find.



Keyboard Macros, by comparison, are useful outside the Enhanced Editor. You can, for example, define a simple log scan expression that you can execute in the log window to quickly highlight key phrases.

CUSTOMIZE SAS EXPLORER

The SAS Explorer is excellent for quickly navigating through libraries and data sets. But it too often proves useless for viewing information about other basic SAS entries such as catalogs, formats, and data views. How often have you tried to double-click a format in the SAS Explorer only to see this error:



How unhelpful is that? How is it possible that SAS has no default action for such fundamental elements? Well, end

your frustration by defining your own default action. You can even specify Base SAS code to execute for a particular member or entry type. The best documentation for understanding how to customize the SAS Explorer comes from SAS users, rather than SAS (see Robert DeVenezia & Peter Crawford, in **References**).

With the help of Robert and Peter, I now have instant, flexible access through the SAS Explorer to Catalogs, Formats, Views, etc. Here are a few examples of customized actions:

CATALOGS: *Print formats or create control-out data sets from entire catalogs:*

Action: **&Contents;95**
gsubmit 'proc catalog; contents catalog=%8b.%32b; quit;'; lst;

Action: **&Print formats;95**
gsubmit 'proc format library=%8b.%32b fmtlib;run;'; lst;

Action: **&Control out;133**
gsubmit '%%let li=%8b;%%let na=%32b;proc format library=&li..&na cntlout=&na;run;'; explorer;

FORMATS: *Print or create control-out data sets from formats:*

Action: **&Print format;95**
gsubmit 'proc format library=%8b.%32b fmtlib; select %32b; run;'; lst;

Action: **&Control out;133**
gsubmit '%%let li=%8b;%%let ob=%32b;%%let na=%32b;proc format lib=&li..&ob cntlout=&na;select &na; run;'; explorer;

Note that you must modify the SELECT statements, above, for char formats & informats. Remember that char format names begin with "$", informat names begin with "@" and char informat names begin with "@$".

TABLES & VIEWS: *Display variable NAME instead of LABEL by default when viewing data. Variable names are far more useful than variable labels while programming.*

Action: **&Open;83**
VIEWTABLE %8b.'%s'.DATA colheading=names

Action: **&Query;83**
QUERY DATA=%8b.%32b colheading=names

VIEWS: *Describe either Data Step or Proc SQL views with one click:*

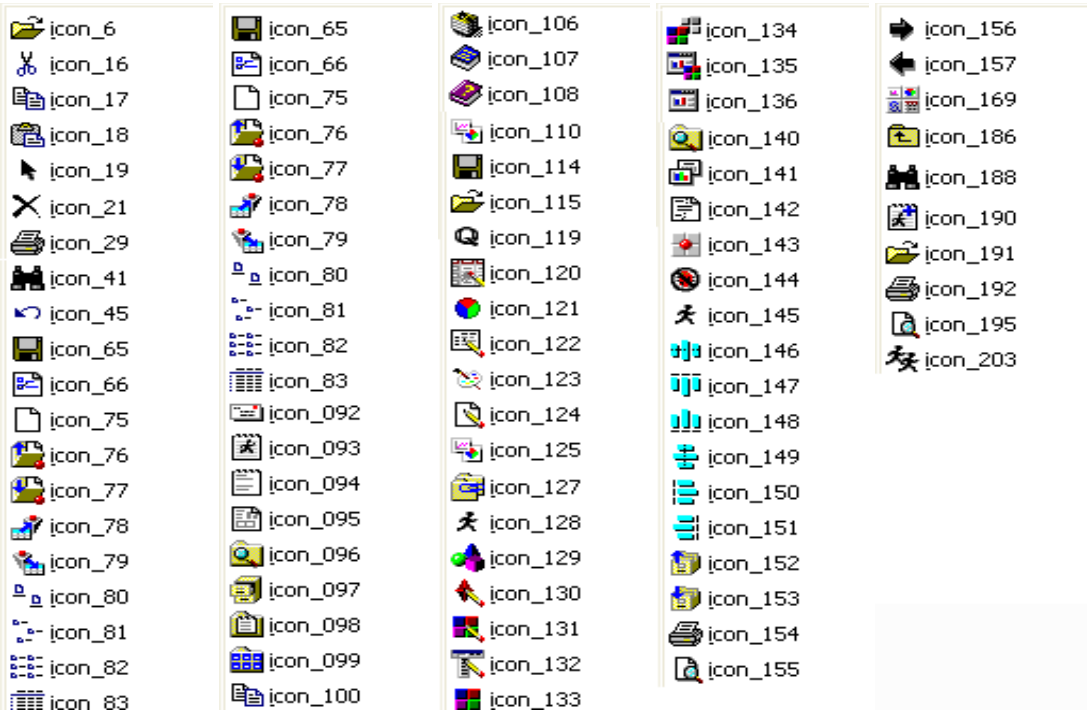
Action: **&Describe;94**
gsubmit 'data view=%8b.%32b; describe; run;proc sql noprint; describe view %8b.%32b;quit;'; log;

The best SAS documentation for Explorer customization is in the SAS Help file "base.chm", typically located in the SAS installation directory, !SASROOT\core\help\. Search this help file for the unquoted string "pop-up action" to find the page titled **"Changing the pop-up actions and icon for a registered type"**.

If you are willing to risk editing the SAS registry directly, this is the simplest but most dangerous approach. Please note that incorrectly editing the SAS registry can disable your SAS installation. Proceed with caution. You can edit the SAS registry by entering the command "regedit". You will find registered types and their actions under: CORE ► EXPLORER ► MENU ► (ENTRIES | MEMBERS).

Use the "Export" & "Import" functions of the SAS Registry Editor to share these settings within a team. "Export" from the SAS Profile that has standard definitions; then "Import" to the target profile.

In the actions, above, the numbers indicate the icon that SAS uses for right-click menu items. SAS has many icons available for these actions, but it takes a clever DATA STEP and registry hack to trick SAS into revealing the available icons and their indices. Here they are:



BASE SAS

Unfortunately it took us too long to learn very clever features of fundamental SAS functions. Several of these are so useful that they deserve at least brief mention here. We invite the reader to peruse the details at their leisure:

►Base SAS ►**SAS Language Reference: Dictionary**
►Dictionary of Language Elements ►**Functions and CALL Routines**

PUT()

Control the alignment of numbers directly in the put function call, rather than using left(put()). For example: "`put(age, best. -L)`". Alignment options are left, center, and right: -L, -C, and -R, respectively.

INPUT()

Programmers can control what SAS writes to the log when converting character strings to numeric values. This situation arises often while processing clinical data. Times, dates, lab values, etc., are often delivered as text fields. Converting these values for numeric processing often fills SAS logs with alerts about invalid numeric data. While often unfortunate, these invalid data are no surprise and therefore the log noise is really unnecessary and not helpful.

The input() function accepts a format modifier, either "?" or "??", to help control this log noise. The single question mark stops SAS from logging "`NOTE: Invalid argument to function INPUT at line ...`", although the automatic variable _ERROR_ is still set to 1 in the DATA STEP. The two question marks stop SAS from setting _ERROR_=1, in addition to stopping the NOTE. For example: "`labnum = input(labchar,?? best.);`"

SCAN()

The scan function, and the macro analog %scan, both accept negative indices. So pulling off the first or last element of a string are equally easy. Consider this simple, but common situation:

```
%if %index(&dataset, .) %then %do;
  %let libname = %scan(&dataset, 1)
  %let memname = %scan(&dataset, -1);
%end;
```

SAS/GRAPH

According to SAS/Graph documentation, you can specify only one line type per plot statement for reference lines:

►SAS/Graph ►SAS/Graph Software: Reference ►**The GPLOT Procedure ... PLOT Statement**

Such inflexibility does not seem reasonable even to SAS, who have added flexibility in SAS version 9. You can now specify multiple line types quite simply: "LVREF=*reference-line-type* | (*reference-line-type*) | *reference-line-type-list* ".

The question, then, is whether this useful feature is documented for version 8 (and therefore usable in production programs). The answer is unclear since SAS is reluctant to update their SAS OnlineDoc pages with the most current information. But they did release a support note confirming 9.2-style line type specification for 8.2:

►SN-V8+-006672: Reference line options work differently beginning in Release 8.2 of SAS/GRAPH

So, those using SAS version 8.2 can already benefit from improvements published in SAS/Graph v 9.2

SAS MACRO LANGUAGE

It is unlikely you only program in Base SAS. More likely you develop generic or macro code to handle situations like those described above. So it is important to approach macro programming with confidence. Perhaps one of these topics will explain some bad experience you have, but never fully understood.

DICTIONARY.MACROS = SASHELP.VMACRO SHORTCOMING

This SQL data set/view includes variable <SCOPE> which should contain the name of the symbol table (macro level or scope) to which each declared symbol belongs. The problem is that the variable <SCOPE> has length $9, while macro names can be up to 32 chars. So, if you hope to discover the scope of a particular symbol, be careful. Unless you always use short macro names, you are likely to discover an incorrect, truncated macro name.

MACRO LOGICAL OPERATOR "OR"

Ever wonder why your compound macro statements don't quite work as expected? There is a big difference between Base SAS & SAS Macro logical operators. In Base SAS, the exclamation point "!" is a mnemonic for "or". In SAS Macro, the exclamation point is just another text char; the symbol "|" is the mnemonic for "or". The invalid mnemonic "!" does not create an error or warning; you simply & silently get the *wrong* results you want. The rule for SCL matches that for SAS Macro: only "|" is a valid mnemonic for "or".

MACRO EXPRESSIONS

Similarly, SAS Macro evaluates compound expressions such as "`%if &low < &myval < &high %then do;`" without generating warning or error messages. But, like the invalid mnemonic for "or", SAS gives you no indication that it has almost certainly misunderstood the instruction. SAS provides no clear explanation, but simply recommends you *not* use compound expressions in SAS Macro:

►Base SAS ►**SAS Macro Language Reference**
►Macro Expressions ►**Arithmetic and Logical Expressions**

… and look for the tiny "**Note:**" under "**Operands and Operators**". Depending on how an expression resolves (moving left to right, as Macro likes to do) SAS can return very strange and unappreciated results.

SUPPRESSING SQL WARNINGS

As mentioned above, PROC SQL generates log warnings if your SELECT statement (eg., "`select * ...`") includes an ambiguous variable reference. This can be helpful, of course, but it also presents us with a problem. How do we specify a list of unique variable names in the SELECT statement? It is not very nice to write or read or maintain long lists of hard-coded variable names. Especially when what we want, as programmers, is often quite simple: all the variables from data-set-1, and all the *other* variables from data-set-2.

Last year I came across a clever, albeit cumbersome, SQL solution to this problem on the SAS Listserver (see **References**). While still no fun to read in its naked form, you could easily wrap it in a brief macro call. The idea is to

have SQL determine all the variables in data-set-2 that are not already in data-set-1. The following code is based on 2 data sets, ONE and TWO, that have a common variable <ID>. Have a look:

```
proc sql noprint;
  * CREATE list of vars in TWO that are not already in ONE *;
  select vlist2.name into :varlist2 separated by ','
  from ( select name from dictionary.columns
         where libname = 'WORK' & memname = 'ONE' ) as vlist1
       RIGHT JOIN
       ( select name from dictionary.columns
         where libname = 'WORK' & memname = 'TWO' ) as vlist2
  on vlist1.name = vlist2.name where vlist1.name is missing;

  * HANDLE the situation of NO additional vars in TWO *;
  %if &varlist2 ne %then %let varlist2=,&varlist2;
  %else %let varlist2=;

  * JOIN the two tables *;
  create table three as
  select one.* &varlist2
  from one JOIN two
  on one.id = two.id order by id ;
quit;
```

POISONOUS COMMENTS

Know your comment types. The wrong comment type can break macro logic in very subtle ways. Then again, the right comment type at the right time can make code much easier to use and maintain. Comment dexterity is a must for the expert programmer.

Consider this trivial example of **Interrupting Macro Logic**:

```
%macro greet(nice);
  * Friendly greeting *;
  %if &nice %then %put HELLO, WORLD! ;

  * Unfriendly greeting *;
  %else %put GET LOST!;
%mend;
%greet(1)
```

which produces: `ERROR: There is no matching %IF statement for the %ELSE.` The problem is that Macro tokenizer correctly interprets the `*comments;` as valid Base SAS statements. Therefore, the second `*comment;` separates the %IF ... %THEN ... statement from the remaining %ELSE statement. When the Macro tokenizer next tries to process the %ELSE statement, it throws out the complaint.

Far more difficult to detect, the Macro tokenizer treats Base SAS `*comments;` as complete statements, *and therefore processes them like any complete Base SAS statement*. As a result, text within a `*comment;` can cause serious problems when that text looks like certain complete Macro statements.

Consider a second trivial example of **Embedded Macro Statements**:

```
%macro empty;
  * %let idx = 1 ;
  * %put IDX = [&idx] ;
```

```
    proc print data=sashelp.class;
      var name age sex;
    run;
  %mend;
  %empty;
```

This appears to be a simple PROC PRINT. But the code fails. The SAS compiler never sees the PROC PRINT statement. Why not?

To understand this, you must first pretend, just for a few minutes, that you are the Macro Language tokenizer. First, you see the "*", so you know you will pass a `*comment;` to the SAS compiler. You just need to know where the comment ends. So you start looking for the concluding ";". But along the way you see a macro %let statement, which also ends with a ";". And then a macro %put statement, still within the same comment, which uses up the second semi-colon. Then finally you see the `*comment;` string "`proc print data=sashelp.class;`", which you justifiably pass to the SAS compiler as a comment. Then you pass the meaningless statement "`var name age sex;`" to the compiler.

And that makes the compiler mad.

Recommendation:	Simply learning the difference between comment types is not always sufficient. SAS programmers should also understand how SAS processes various types of code.

BONUS: LOG SCANNING

ERRORs & WARNINGs are quite easy to find in a SAS log. But what about those devilish NOTEs? SAS provides no comprehensive list of what NOTES it might generate. Most users & SAS teams have their list of "prohibited" log NOTEs. But beyond these *inclusive* lists, what lurks undetected in a vast log file, or a set of 100 logs?

Inspired by a novel technique presented in a user paper (see **References**), and having access to a decade of SAS log files, we undertook a different approach: the *exclusive* NOTEs list. Instead of looking for "uninitialized", "could not be found" and similar troubling SAS messages, we identified unique patterns of SAS NOTEs. Many are purely informational and pose no risk to the validity of results. In the **Appendix**, we present a simple "grep" text-search script (HP_UNIX, but simple enough to port to other operating systems) that ignores these safe, informational NOTEs and reports anything left over. As it always should be, the programming team can then conduct their own risk assessment based on this complete report of what SAS encountered while performing analyses.

This log scanner, of course, also reports any MERGE step INFO messages described above.

Some disturbing NOTEs are unavoidable, even if they cause no harm. The best a team can do, for now, is to be aware, and adopt simple programming standards to minimize confusion or concern. For example, SAS offers advice for eliminating or evaluating two such NOTEs:

```
Note: BY-line has been truncated at least once.
WARNING 32-169: The quoted string currently being processed has
become more than 262 characters long.  You may have unbalanced quotation marks.
```

Recommendation:	Do not risk missing SAS NOTEs that effect the interpretation or validity of results; use an *exclusive* log scanner to minimize the risk of overlooking critical NOTEs in SAS logs.

CONCLUSION

Too often, time constraints force programmers to side-step mysterious problems, rather than trace them through SAS documentation to find an explanation. From the start, SAS programmers rarely spend much time reading the details in SAS documentation. We mostly learn from peers. One unfortunate consequence is that we can easily miss useful and time-saving techniques based on fundamental concepts which we never properly learned.

Recommendation:	Even experienced programmers should return to the basics and regularly browse SAS documentation. The time you same may be your own.

REFERENCES & RECOMMENDED READING

SAS DOCUMENTATION

SAS Documentation Portal: http://support.sas.com/documentation/onlinedoc/index.html
including SAS OnlineDoc v8 and v9.1.3: http://v8doc.sas.com/sashtml/, http://support.sas.com/onlinedoc/913/

SAS SYSTEM OPTIONS

Data Step Debugger: http://v8doc.sas.com/sashtml/lgref/z0208245.htm

SAS ENHANCED EDITOR & THE SAS EXPLORER

Abbreviations and Macros: http://support.sas.com/sassamples/papers/0303_saseditor.pdf (Paul Grant)
http://www.pharmasug.org/2003/BestPapers/cc025.pdf (Arthur Carpenter)

Customize SAS Explorer: http://support.sas.com/sassamples/quicktips/05mar/viewtable.html (intro)
http://www.devenezia.com/downloads/sas/actions/index.php (R. DeVenezia)
http://www2.sas.com/proceedings/sugi31/237-31.pdf (Peter Crawford)
http://support.sas.com/sassamples/quicktips/05mar/viewtable.html (SAS tech)
http://www.sas.com/offices/europe/uk/newsletter/feature/14apr_may05/default_action.html (SAS UK newsletter)

SAS MACRO LANGUAGE

Suppressing SQL warning: http://listserv.uga.edu/cgi-bin/wa?A2=ind0310e&L=sas-l&D=0&P=19548
(thanks to Sigurd Hermansen and Paul Dorfman).

SAS/GRAPH

Defining multiple linetypes: http://support.sas.com/techsup/unotes/SN/006/006672.html

SAS LOG SCANNER

Inspired by a clever user paper: http://www.nesug.org/html/Proceedings/nesug01/cc/cc4008.pdf

Upon request, the authors can provide the full list of SAS NOTE patterns.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Dante diTommaso
Hoffmann-La Roche
CH-4002, Basel, Switzerland
Work Phone: +41 61 687-4314
Email: dante.di_tommaso@roche.com

Benjamin Szilagyi
Novartis Pharma
CH-4002, Basel, Switzerland
Work Phone: +41 61 324-2369
Email: benjamin.szilagyi@novartis.com

APPENDIX: THE ULTIMATE LOG SCANNER

Upon request, the authors can provide the full list of SAS NOTE patterns.

```
#!/usr/bin/sh
#
# USAGE: SASLogScan filename-or-filemask
#
grep -En "^ERROR|^WARNING|^NOTE|^INFO: The variable" ${1} \
| grep -Ev \
-e "NOTE:  [0-9]+ RECORDS WRITTEN TO " \
-e "NOTE: %INCLUDE \(level [0-9]+\) (ending|file|resuming)" \
-e "NOTE: (DATA statement|The SAS System) used:" \
-e "NOTE: (Format|Informat|Libref|SQL view|Table|The data set) .+ has been (output|written
to|deassigned|defined|dropped|updated)" \
-e "NOTE: .+ values have been converted" \
-e "NOTE: .+ was successfully created\." \
-e "NOTE: A total of [0-9]+ records " \
-e "NOTE: AUTOEXEC processing (beginning|completed)" \
-e "NOTE: Appending .+ to .+" \
-e "NOTE: Building list of graphs from" \
-e "NOTE: Changing the name " \
-e "NOTE: CALL EXECUTE generated line\." \
-e "NOTE: Compressing data set " \
-e "NOTE: Copyright " \
-e "NOTE: DATA STEP view saved on file " \
-e "NOTE: Deleting .+ \(memtype=[A-Z]+\)" \
-e "NOTE: Deleting entry .+ in catalog " \
-e "NOTE: Directory for library .+ contains" \
-e "NOTE: Enter greplay commands or statements\." \
-e "NOTE: Entry .+ has been (imported|transported)\." \
-e "NOTE: Fileref .+ has been deassigned." \
-e "NOTE: Format .+ is already on the library\." \
-e "NOTE: Graphs on .+" \
-e "NOTE: Index .+ deleted\." \
-e "NOTE: Input data set is already sorted" \
-e "NOTE: Input data set is empty\." \
-e "NOTE: Lib(name|ref) .+ (refers to the same|has been deassigned|was successfully)" \
-e "NOTE: Line generated by CALL EXECUTE" \
-e "NOTE: Line generated by the (invoked|macro)" \
-e "NOTE: Missing values were generated" \
-e "NOTE: No (observations|rows) (in data set|were selected|were updated)" \
-e "NOTE: PROC SQL statements are executed immediately" \
-e "NOTE: PROCEDURE [A-Z]+ (printed|used)" \
-e "NOTE: Processing on disk occurred" \
-e "NOTE: Renaming variable .+ to " \
-e "NOTE: Running on " \
-e "NOTE: SAS (initialization|Institute|System|\(r\))" \
-e "NOTE: Table .+ created, with [0-9]+ rows " \
-e "NOTE: The (data set|file|infile|file/infile) .+ (has [0-9]+ obs|has been updated|but
appears|is:)" \
-e "NOTE: The .+ printed page" \
-e "NOTE: The BASE Product product " \
-e "NOTE: The above message was for " \
-e "NOTE: The execution of this query involves performing" \
-e "NOTE: The query requires remerging summary " \
-e "NOTE: There were [0-9]+ observations read from " \
-e "NOTE: This (installation|session) is (running|executing) " \
-e "NOTE: Where clause has been (augmented|replaced)." \
-e "NOTE: Writing (HTML|ODS PDF|RTF) (Body|Contents|Frames|output)" \
-e "NOTE: Your system is scheduled " \
-e "NOTE: [0-9]+ (line|observation|record|row)[s]{0,1}( was| were){0,1}
(added|deleted|inserted|read|updated|with dup|written)" \
-e "NOTE: [0-9]+ duplicate (observations|key values) " \
-e "NOTE: view .+ used:"
```