I. Abstract

Submitting SAS code to SAS for execution is only one use for the code we write. This paper explores the idea of SAS code as data about the work we are doing. If we explore these data—even with the simplest tools—we'll learn useful things about our work. This idea is illustrated with examples of what it is easy to do today, with proposals for what is certainly possible with a little more effort and with a discussion of what would be desirable in the future. Thinking and talking about how we use SAS code as data today could help guide the development of more powerful tools for SAS users tomorrow.

II. Introduction

Submitting SAS code to SAS for execution is only one use for the code we write. We also read it and quote it to each other to describe what we have done, are now doing or will do. Using the machine to scan SAS code can help locate a needed file, debug the code we're writing or give us a better understanding of code we've already written. Treating SAS code as data about the work being done with SAS is unusual for a SUGI paper, which by tradition should focus on generating SAS code that does marvelous new things.

Muller et al. (1981) give many simple but practical ideas about managing a batch SAS environment. Their recommendations include techniques for expressing the relationships between jobs, OS files, SAS datasets, DDNAMES and printouts by following naming rules and conventions. The problems they address have certainly become more complex with the general availability of interactive SAS under 3 different operating systems. Easy access to Interactive SAS makes it easier to write more SAS code than we can easily manage. All of those marvelous new things we can do are less useful if we can't manage the whole process effectively.

Nugent (1986) discusses many of the same problems, but from the perspective of a software designer and with more theory to guide him. He notes that, with the availability today of computing power only dreamt of 15 years ago, we have discovered the meta problem of maintaining coherency in our analysis paths. We can do so much so quickly we now must pay attention to organizing and knowing our voluminous output.

To solve these problems, Nugent proposes a set of programs that represent a stream of existing source code in such a way that the code can be examined in fairly sophisticated ways. This paper deals with these problems on a level that is intermediate between Muller et al. (1981) and Nugent (1986).

We need to be able to manage the cumulated SAS code and outputs from many sessions, even though this paper focuses on tools to examine an individual session (a single file). Conceptualizing the data analysis path across files and sessions is essential, as Huber (1986) comments: "in text editing (or programming) the final document (or program, respectively) is everything; you need not preserve its antecedents. Data analysis is different: just as with financial transactions, you cannot check the correctness of an analysis without inspecting the path leading to the conclusions."

However, audit trails can be very important in large, complex programming efforts. And they can be helpful in quite small, but complex efforts. SAS code is an essential part of the evidence we have available about our data analysis paths, whether we are fitting stochastic models or generating descriptive tables or combining information from several different sources. Where analysis stops and programming begins is sometimes hard to tell with SAS. Experience with the techniques illustrated in this paper and a little reflection should remind us that there is a lot more to SAS code than whether it works or not; that's why we want to test it and filter it with computer tools of various sorts (aside from submitting it to SAS).

III. Finding key SAS statements

The first examples use VMS command procedures to extract highlights from existing SAS code. Some general discussions of post-processing statistical code are cited.

Think of examining the 5 files of SAS code related to a project that you haven't been able to work on during the last 2 weeks. Imagine that the context has now changed, the goal has been re-stated, and you need to rewrite some of the jobs. Most of the files are on one directory on one machine but some are written for batch and others are unedited SAS statements collected by the Display Manager in fairly long interactive sessions. Where do you start? 1) You can stare...
at the file names till you remember what each file was for, 2) scroll through the code a screen-full at a time, 3) browse the slush pile of old printouts, or 4) loose patience and print out all of the SAS code you think might have anything to do with the project.

A. Useful filters

The filters discussed in this section won't replace any of these time-honored methods, but they can supplement all of them. A filter, in the sense used by Kernighan and Plauger (1976), is one of those surprising number of programs that have one input, one output, and perform a useful transformation on data as they pass through. We want filters to be convenient for doing simple tasks like listing all lines with comments, titles, and references to external files or permanent SAS datasets.

What specific character strings are searched for is less important than providing yourself with some method for scanning SAS code by machine. Nugent (1986) gives the following reasons for getting the machine to do it:

1. Searching through a script to find a variable or command reference is a tedious process.
2. An analyst makes mistakes when searching manually through the script.
3. We have proven A.I. technology which can be applied to this problem.

B. SSAS

The SSAS command (for See SAS) is a 1-line VMS command procedure that searches for certain kinds of SAS statements in a file of SAS code. This version searches for:

1. Title statements
2. %Let statements
3. Comments
4. References to permanent datasets (the string "db", as in jdsDB.tuit87).

I find SSAS to be useful. Its utility to me may depend on the SAS code being searched, the amount of discipline that has gone into writing comments, the nature of my final product (so often just a SAS dataset), or the problems characteristic of using administrative records for research purposes. The discipline of making the names of all SAS data libraries end in "db" makes it easy to find references to permanent data sets.

Source code for SSAS is in Appendix A. The VMS search command has many options that make it a very useful tool. Coding a similar search with SAS itself would be quite simple, although I found I wanted to quickly scan a series of files from outside SAS.

C. SASHEDR

SASHEDR searches the code for strings similar to the ones SSAS searches for but writes the selected lines to a header inserted at the top of the file, with the name of the file, the owner, the date, and a marker so it won't run twice on the same file. It provides a start toward documenting a file of SAS code that can give enough cues to combat writer's block. The same caveats about utility apply to SASHEDR that apply to SSAS.

Appendix B has the source code. Figure 1 shows sample output from SASHEDR.

D. Other possibilities

Other possibilities include searching for other combinations of strings. Setting on which strings requires that we think about likely scenarios in a session, as argued in several papers in both Gale (1986) and Boardman and Stefanski (1986). Lewis (1983) offers an example of further analysis and manipulation following the search for keywords in FORTRAN code.

The tools described or proposed in this paper arose in a less "statistical" context than what is suggested in Huber (1986), Nugent (1986), or Oldford and Peters (1986a and 1986b). These references provide background on the idea of SAS code as data, although examples given in this paper were developed before I came across these references.

IV. A Microscope for the Program Data Vector

For most of us, analysing a set of data begins with getting it into a SAS data set. Given a reasonable SAS dataset we can go far in describing and analyzing the problem at hand. Getting it in a SAS dataset is also a reasonable first step in examining SAS source code by machine. The first step of Nugent's (1986) program is to generate a data base that describes the logs (termed scripts) which other programs can then process to answer specific questions. The approach in Cowell et al. (1986) is to modify the S environment (Becker and Chambers, 1984) so that comments and date stamps are automatically added to the S script. An analysis session is then represented graphically by a different program (under a different operating system!) and the user can add comments and interact with a description of the analysis as a whole. This section describes what these ideas might suggest in a SAS environment.

SAS datasets are the backbone of the SAS System, and the following discussion proposes three ways that SAS might create a SAS dataset (like the one now generated by PROC CONTENTS, hereafter termed the SAS Log Dataset or SLD) to describe all the SAS datasets in a job stream. The first is a SAS PROC, which would create a SAS dataset by reading a file of SAS code. The second is a global option, where SAS would append observations to the SLD for each DATA or PROC step in which a data set is created. The third would

* James W. Barbour contributed a good deal of the coding for SASHEDR.
append to the SLD only when the user requests it by coding a SAS dataset option. Each one of these alternatives could be advantageous under certain situations or assumptions.

A. The output SAS dataset

All three proposals would create an SLD which would be like the one now produced by PROC CONTENTS, except that it would have one observation for each variable in each SAS step (for the entire Program Data Vector, including variables brought in by a SET statement, variables that are dropped, etc.). The SLD could also have additional analytic variables indicating the step number or line number (relative to the beginning of the job stream), whether the step was a DATA step (or if a PROC, then which one), and whether the variable was dropped, kept or renamed.

The SLD could be filtered in various useful ways—a lot easier than searching for character strings, since an SLD dataset would be fairly rich. A subsetting if could easily follow all numeric variables (or just one particular variable) through the entire job stream. A dataset like the SLD could help us examine the history of a particular variable—more analytically than the present SAS history manager permits. Joining several SLDs from multiple SAS sessions would be a fairly good first step toward a data dictionary for an entire project. The output dataset would also be useful for mundane tasks like removing steps that were repeated as the SAS user repeatedly submitted a snippet of SAS code—so we could keep only the version of the code that finally works.

The proposals described here are a special case of the more general proposal in Nugent (1986).

B. PROC PDV

PROC PDV would create an SLD by reading a file of SAS statements. It might be called as follows:

```sas
PROC PDV INDD-RECALL OUT=WORK.SASDSET;
```

The INDD keyword could point to a sequential file, to a member of a PDS, or to the Display Manager's RECALL buffer (processing its contents directly instead of having to perform a PRINT RECALL Display Manager command).

A primitive version of PROC PDV might be implemented using a technique similar to the one in SSAS, by searching for drop, keep, var., attr., length. As Nugent (1986) points out, however, the farther a tool gets from searching for simple character strings, the more it has to know about the internal language syntax. I have tried to write DATA steps to construct an LDS dataset and it's not easy.

C. The PDV = WORK.MICROSCP Global Option

The following arguments suggest that it would be better to link the process that constructs the SLD more closely to the SAS supervisor as it constructs the Program Data Vector for execution.

1. A program smart enough to construct the Program Data Vector would have to change with SAS.

2. Constructing the Program Data Vector depends on the peculiarities of each PROC and DATA step; PROC TRANSPOSE, for example, creates a program data vector that depends entirely on the Interaction between the SAS code and the SAS dataset being processed.

3. The presence of %MACROs can immensely complicate the interpretation of SAS code.

Therefore it may be more effective to have a global SAS option that, once turned on, adds observations to the LDS every time it constructs a Program Data Vector.

D. The (LDS=DATASET) Data Set Option

Another approach (which might offer the user a little better control) would be to add observations to the LDS only when the user specifically requests it by coding a SAS data set option. This would allow specification of the LDS as follows:

```sas
PROC SUMMARY DATA=JOSSB.TUIT (LDS=PDVOUT); CLASS YEAR; VAR COST;
OUTPUT OUT=TOTALS (LDS=PDVOUT) SUMO;
```

V. My wish list for the SAS Display Manager

The SAS Display Manager has come a long way from its first implementation on the VAX in Version 4 to its most advanced form in Version 6 on the PC. Its features for manipulating SAS code include a text editor that is as good as the ISPF editor it was modeled after. (I even like it better than my favorite text editor when I consider that it allows me to submit the code directly to SAS and immediately examine the log and the output.) However, it could go much further in providing cognitive tools to manipulate existing SAS code more skillfully. For example, it should have artificial intelligence features, along the lines of Huber's (1986) Laboratory Assistant. Here are some suggestions:

1. The Display Manager might automatically construct AF screens that would elicit comments to be written to the SAS log explaining the relationships between the datasets in the current session, or recording information about external files (where the data came from, why collected, etc.). Like Huber's (1986) Laboratory Assistant, AF screens could ask you "Why did you do that" or, after generating a plot, "What do you see?"

2. The Display Manager could provide a window that is a schematic map of data set dependencies in the current session; it should allow this window to be saved either with the output or the session log.

3. Insert comments anywhere in the recall buffer or log directly from either the program window or from the "comments window" in Version 6.

Generally, the Display Manager should help SAS users do work—beyond our preconceptions of "editing". Here are some other thoughts on serv-
ices that the Display Manager could provide in the Program Window that would help it more intelligently support us as we work on SAS code.

1. "Find", "replace" and other Display Manager commands should be smart about SAS code: for example, it would be nice to have a meta character in the find command that would anchor the search to the beginning of a SAS statement in the log or the recall buffer; thus you could search for the last DATA statement.

2. We should be able to include SAS code directly from the records kept by the history manager to the program window.

3. It would be useful to toggle between recall-by-SAS-stepboundaries and recall-by-Display-Manager-submit-boundaries.

4. Since we bring in big pieces of existing SAS code that we may want to execute by steps, have the ability to submit only the beginning n lines from the Program Window, or submit down as far as a "stop" line command.

Here are some thoughts on relationships between windows and the rest of SAS.

1. Announce how the display manager will be an open architecture (just as users can add PROCS to the SAS System and functions to the DATA step or IML) so that users can enhance it.

2. When working with large amounts of output, it would be nice to be able to point to a particular step in the LOG and go immediately to the corresponding point in the OUTPUT.

3. It would be helpful to read a logical file named "RECALL," just as we can now write to logicals LOG, PUNCH or PRINT. In this way PROC PDV or a DATA step could process all the code that has been written interactively up to the present moment in the current session.

VI. Conclusion

The idea of processing SAS code by machine has been presented in 3 different contexts. Some of the literature currently appearing on data analysis environments was reviewed. Suggestions for how some of these ideas could be applied to SAS internals were proposed.

I would urge each other and SAS Institute to deal with challenges of Interactive SAS by developing more tools to process our existing SAS code with intelligent computer programs. This topic is so important that there should be a whole session about it at the next SUGI.

VII. Contact Information

The author can be contacted through BITNET (SMITH_JD@COLORADO) or by writing:
Office of University Management Systems
University of Colorado
Campus Box 50
Boulder, CO 80309

SAS is a registered trademark of SAS Institute, Inc., Cary, NC, USA.
Figure 1

**** SYS$USERDEVICE: [ACSRRES.SMITHJD.SASWRK] PSFBASE2.SAS3 ****
filename psf db [admpsf.studentdb.database] pdbn dx.dat
output out=admpsf.pscnt (rename=(_freq=count) drop_type)
proc print data=admpsf.pscnt (obs=48)
** [SMITH_JD] ** 11-OCT-1985 09:11:37.97 **
**********************************************************************;
IX. Appendices

A. Source code for SSAS
$ search 'Pl' 'data ','format ',',"db.","proc ","title","$let ","set","value"

B. Source code for SASHEDR
$ SASHEDR.com / R-T: John Smith & Jim Barbour Mon 8-26-1985
$ insert a header of information in a file of SAS code: owner,
$ time, file name, and all lines with the following strings
$ in them: title, file, x ' db -- all of them within a
$ regular SAS comment.
$ if pl .eqs. ** then p1 = f$logical('code')
$ getp1: if p1 .eqs. ** then read/promp=">$File: * sys$command p1
$ if p1 .eqs. ** then goto getp1
$ expand name of file, with .SAS if necessary
$ full = "$parse(pl,'sys$disk:[]].sas')
$ full = f$search(full)
$ if full .eqs. ** then goto filenotfound
$ assign/nolog sys$logln: scratch.delete_me scratchfile
$ owner = f$file_attributes(full,"UIC")
$ now ~ fstime()
$ flag_str = "<< SAS Header Information Created by SASHEDR >>"
$ ver_str = "V 2.0 ", now"
$ len = f$length(flag_str)
$ open/read input 'full'
$ read input line
$ close input
$ if f$extract(0,len,line) .eqs. flag_str then goto file_headed
$ open/write tinout scratchfile
$ write tinout ""flag_str" "ver_str"
$ write tinout ""flag_str" "full" "now"
$ close tinout
$ assign /user nia0: sys$output
$ Just put the EDT commands in the file. The system detects a line beginning
$ with a dollar sign ($) as an end of file.
$ edit :=
$ edit/edit 'full'
$ include scratchfile =doc
$ copy =main all 'title' to =doc e-1
$ copy =main all 'file' to =doc e-1
$ copy =main all 'x' ' to =doc e-1
$ copy =main all 'db.' to =doc e-1
$ // = doc e-2 all ";"
$ // = doc e-2 all ""
$ =doc e ;************************************************************************;
$ copy =doc w to =main b
$ edit/doc('scratchfile')
$ delete scratchfile
$ exit
$ filenotfound: write sys$error "SASHEDR ERROR: 'pi' not found."
$ exit
$ file_headed: write sys$error "*** File already contains header information."
$ exit

1252