First version: June 18, 2004
Revised: January 24, 2008

PATRICIA LEDESMA LIÉBANA

SAS programming skills

This document is not a self-paced SAS tutorial; rather, it focuses on data management skills and features of SAS that are helpful in a research context. However, it does provide the basic elements of the SAS Language. If you have some previous programming experience, the skills listed here should be easy to understand. For a comprehensive SAS tutorial, please refer to "The Little SAS Book" by Delwiche and Slaughter, available in the Research Computing library (room 4219).

Contents

1. Syntax rules

- All commands end in a semi-colon
- SAS statements are not case sensitive. You may use upper or lower case.
- Variables names can be upper or lower case.
- When you refer to "external" file name, it is case-sensitive (interaction with UNIX operating system)
- Commands can extend over several lines as long as words are not split
- You may have more than one command per line
- SAS name rules (for datasets and variables): up to 32 characters long; must start with a letter or an underscore ("_"). Avoid special characters.

2. Basic parts of a SAS program

- There are two basic building blocks in a SAS program: DATA 'steps' and PROC (procedures). SAS procedures do not require the execution of a data step before them.
- In addition, OPTIONS to control appearance of output and log files.
- The DATA step is where you manipulate the data (creating variables, recoding, subsetting, etc). The data step puts the data in a format SAS can understand.
- A SAS data file can have up to three parts: (a) the headers – descriptive information of the dataset's contents; (b) the matrix of data values; and (c) if created, indexes. Indexes are saved to separate "physical" files, but SAS considers it part of the data file. Thus, if an index exists, it should not be removed from the directory where the data file resides. Version 8 data files have the extension ".sas7bdat", while index files have extension ".sas7bndx". Most data files in WRDS have indexes.
- SAS reads and executes a data step statement by statement, observation by observation. All the variables in the portion of memory that processes the current each observation (input buffer or program data vector, depending on whether you are reading "raw" data or a SAS data file) are reset to missing in each iteration of the data step. The RETAIN statement prevents this from happening.
- Missing values: Generally, missing values are denoted by a period ("."). Most operators propagate missing values. For example, if you have three variables (v1, v2, v3) and for observation 10, v2 is missing, creating total=v1+v2+v3 will result in a missing value for observation 10 [if you would like the sum of the non-missing values, use the SUM function: total=sum(v1,v2,v3)]. However, for comparison operators (<, >, <= or >=), SAS treats missing values as infinitely negative numbers (- ∞) [unlike Stata, which treats them as infinitely positive]

- PROCs are used to run statistics on existing datasets and, in turn, can generate datasets as output. Output data sets are extremely useful and can simplify a great deal of the data manipulation.

- DATA step and PROC boundary: Under Windows, all procedures must end with a "RUN;" statement. In UNIX, this is not necessary since SAS reads the entire program before execution. Does, it determines DATA step boundaries when it encounters a PROC and knows the boundary of a PROC when it is followed by another PROC or a DATA step. The one exception to this rule is in the context of SAS macros, where you may need to add the "RUN;" statement.

3. Types of data files

- SAS distinguishes two types of files: "internal" (created by SAS) and "external". Referring to files in SAS maybe the most complicated point in learning SAS. Here, the conventions date to mainframe systems. In brief, files external to SAS are ASCII (or text) files and files generated by other applications (such as Excel, SPSS, etc). Files internal to SAS are divided into "temporary" or "scratch" files, which exist only for the duration of the SAS session, and "permanent" files, which are saved to disk.

$$\text{Files in SAS} \rightarrow \begin{cases} \text{External} \rightarrow \begin{cases} \text{ASCII} \\ \text{Files generated by other programs} \end{cases} \\ \text{Internal} \rightarrow \begin{cases} \text{Temporary, "scratch" - one level names} \\ \text{Permanent - two level names (libref.name)} \end{cases} \end{cases}$$

- SAS scratch files are created and held until the end of the session in a special directory created by SAS in a scratch area of the workstation. When SAS finishes normally, all the files in this directory are deleted.

- SAS permanent files are written to a "SAS library", which is simply a directory in the system with a collection of SAS data files. In the program, the directory is assigned a nickname, called a "libref" in the manuals, with a LIBNAME statement. Subsequently, to refer to a permanent file, you type the libref, followed by a period, followed by a name.

- The examples below build a data file. Until a complete version of the file is created, all intermediate data files are "scratch" files. We ignore the LIBNAME statement until the sample code in section 7.

4. Providing data in the program

Use a text editor to create the following program (call the file "ipos.sas"):

```
data ipo;
  input ticker $ offer_prc offer_amount open_prc
```

```
        date_public mmddyy8.;
    format date_public yymmddn8.;
     cards;
     ACAD 7 37.5 7.44 05/27/04
     CYTK 13 86.3 14.5 04/29/04
     CRTX 7 48 7.26 05/27/04
     EYET 21 136.5 30 01/30/04
;
proc sort data=ipo; by descending offer_amount;
proc print data=ipo;
  title 'Recent IPOs';
```

Note that the name of the date variable is followed by "`mmddyyn8.`". This is a date "informat", an instruction that tells SAS how to read the values. SAS will read the dates into SAS dates (number of days elapsed since January 1, 1960), enabling you to do calculations with the date. The format statement includes a SAS format, an instruction to SAS on how to display the date (internally, it keep the number of days since 01/01/1960). The "n" in the format stands indicates that we do not want any separators between the year, month and day (for a slash, the format would be `yymmdds10.`).

To run the program, exit the text editor, type the following command at the UNIX prompt and hit enter to execute:

```
sas ipos
```

SAS will create two files:

- ipos.log: The LOG file has all the messages, errors and warnings that the SAS compiler has issued during execution of your program. Always check it.
- ipos.lst: The listing file contains any "printed" output generated by a procedure.

Use the UNIX "`more`" command to browse these files.

5. Creating variables

New variables can be defined in the data step. For example, in the previous program, type the following statement after the INPUT statement (between INPUT and CARDS):

```
  open_offer_ret=open_prc/offer_prc;
```

Run the program again to view the results. SAS has a wide range of functions that can also be used to modify your data.

6. SAS operators and functions

For a complete listing of functions and their syntax, refer to the SAS/BASE "SAS Language Reference: Dictionary" manual. For detailed explanations of the precedence each operator takes and how SAS expressions are built, refer to the SAS/BASE "SAS Language Reference: Concepts" manual. The list below only includes some of the functions I use most.

Arithmetic	Logical	Comparison
As in Excel: +, -, *, /	And: & or AND	Equal: =
Exponentiation: **	Or: | or OR	Less than: < or LT
	Not: ^ or NOT	Less than or equal: <= or LE
		Greater than: > or GT
		Greater than or equal: >= or GE
		Not equal to: ^= or NE

An additional important operator not included in the categories above, is the concatenation operator, which creates one alphanumeric by concatenating two character strings: ||

Functions work across variables (in rows). Some important functions:

- ABS(x): absolute value
- DIFn(X): difference between argument and its n-th lag
- EXP(x): raises *e* to x power.
- FLOOR(x): integer that is less than or equal to x. Same results as INT(x)
- INDEX(argument, excerpt) searches an argument and returns the position of the first occurrence.
- LAGn(x): lagged values from queue (LAG(.) is first lag)).
- LOG(x): natural log of x (LOG2(x) logarithm to the base 2, LOG10(x) logarithm to the base 10)
- MAX(x), MIN(x): Maximum and minimum
- MEAN(X), SUM(X)
- N(x) number of non missing values
- NMISS(x) number of missing values
- SUBSTR(argument, position, length) retrieves a portion of a given string

7. Reading "raw" data – external files

To read data from an ASCII file, you must specify the location of the file using either the FILENAME statement or the INFILE statement.

We will use the following sample data file:

```
~ple531/sastraining/sampledata.prn
```

In the next example, we will use the INFILE version, where the location of the external file is referenced in quotes:

```
data t1;
infile '~ple531/sastraining/sampledata.prn' firstobs=2;
input year
      ticker $
      data6
      data12
      fiction;
proc print data=t1;
```

Note: The FILENAME would assign a "fileref" or handle to the external file, which would then be used by the INFILE statement. The first lines of the code would be written as follows:

```
filename raw '~ple531/sastraining/sampledata.prn';
data t1;
infile raw firstobs=2;
etc.
```

Note that "raw" is an arbitrary nickname for the external file, which is then used in the INFILE statement instead of the file location. The FILENAME command is structured like the LIBNAME command in section 7 and, if your program is reading several external files, it might be a more organized way of writing the program: declare all the external file names at the beginning of the program (a sequence of FILENAME statements), making easier replication of results if you must move the data files and programs to a different computer.

8. Combining datasets

There are two common types of dataset operations: appending datasets with the same variables or combining (merging) datasets with different variables.

8.1. Appending datasets

In "~ple531/sastraining/additionaldata.txt" there are three additional years of sample data, with four of the same variables as "sampledata.prn". To append or concatenate these datasets, first add the following statements to read the additional data into a SAS dataset:

```
data t2;
infile '~ple531/sastraining/additionaldata.txt' firstobs=2;
input year ticker $ data6 data12;
```

Next, concatenate both files with a new data step:

```
data t3;
set t1 t2;
```

Note: This can also be done with APPEND procedure, which is more resource efficient. Instead of the additional data step, you can try:

```
proc append base=t1 data=t2;
```

In general, `proc append` will be faster than the data step: the `data` step will read both datasets (t1 and t2), while `proc append` will only read the second (t2, the dataset being appended to the end of t1, the base dataset).

8.2. Merging datasets

File "~ple531/sastraining/moredata.txt" has an additional variable, a fictional dummy for fair trade coffee. It also includes the tickers. To merge these data with the rest of our coffee data:

```
data t4;
  infile '~ple531/sastraining/moredata.txt';
  input ticker $ fair;
proc sort data=t4; by ticker;
proc sort data=t3; by ticker year;
data t5;
  merge t3 t4;
  by ticker;
```

One thing to keep in mind to control the result of the merge, is that SAS keeps track of the dataset that contributes to each observation with some internal dummies. Using the "IN=" dataset option (for example, "`merge t3 (in=a) t4 (in=b);`") it is possible to control what observations are in the final dataset. By default, it would be all observations (from t3 and t4). Using an IF statement such as the one below writes only those observations for which both datasets contributed. Similarly, one could specify "`in=a;`", "`in=b;`", "`if not b;`", etc.

```
data t5;
  merge t3 (in=a) t4 (in=b);
  by ticker;
  if a and b;
```

8.3. PROC SQL

PROC SQL is the SAS implementation of the Standard Query Language (SQL) standard to work with databases and is used in Oracle, Microsoft SQL Server, Microsoft Access and other database systems. PROC SQL is a powerful language that allows sophisticated merges. When

you merge datasets with PROC SQL you do not need to sort them. Given that most files in WRDS are indexed, PROC SQL also works fast when creating queries on CRSP and Compustat, for example. Consider the example below, from the WRDS sample file "ina.sas" to retrieve data from the Compustat Industrial file.

To end PROC SQL you must type use the QUIT statement. The SQL query is between PROC SQL and QUIT (as a very long statement). Unlike the rest of SAS, datasets are called "tables", while variable names now have two levels (e.g., inanames.coname): the first level denotes the table of origin (dataset "inanames"), while the second level is the variable name itself ("coname"). There is no reference to SAS libnames except in the "FROM" part of the SQL query.

In the example below, the query itself has been broken into lines for convenience. The first line defines the dataset that will be created (temp2) by selecting variables from dataset inanames (coname and iname) and dataset "temp" (all variables). The second line of the query defines the sources of the datasets: temp is a temporay dataset (defined in earlier SAS statements) and inames is in the "comp" library (the libref assigned to the location of Compustat files in WRDS). Finally, the WHERE portion of the query defines how the merge is done. In this example, observations from temp are matched to observations of inanames on the basis of CNUM (CUSIP issuer number) and CIC (the issue number and check digit). Note that the variables in temp are named left of the equal sign; this implicitly defines a "left join" in which all the records of temp will be preserved (whether there is matching Compustat data or not) and only matching records of the Compustat inanames file.

```
PROC SQL;
  CREATE TABLE temp2 AS SELECT
   inanames.coname, inanames.iname, temp.*
  FROM temp, comp.inanames
  WHERE temp.cnum=inanames.cnum AND temp.cic=inanames.cic;
QUIT;
```

To achieve the same with a MERGE statement, assuming both datasets are sorted by CNUM and CIC, the DATA step would read:

```
data temp2;
merge temp (in=a) comp.inanames;
if a;
```

Given that "inanames" is a large file and that the DATA step will copy each merged observation before evaluating the IF statement, this option is less efficient than the PROC SQL version.

The next example shows a more complex SQL statement in which observations from TAQ consolidated trades (test) and consolidated quotes (test2) are matched based on the date and symbol, as well as time stamp for each record. In this case, for each trade, we match all the quotes in a 10 second window. Note that variable "secs" is a calculation based on variables from different datasets. Also, notice that the time stamp variable was renamed on the fly.

```
proc sql;
  create table combined as
  select test.symbol, test.date, test.price, test.size,
         test2.bid, test2.ofr, test2.bidsiz, test2.ofrsiz,
```

```
          test.ct_time, test2.cq_time,
         (test2.cq_time-test.ct_time) as secs
    from test(rename=(time=ct_time)) left join
       test2(rename=(time=cq_time))
       on -10<=(ct_time-cq_time)<=10
   where test.symbol=test2.symbol and test.date=test2.date;
quit;
```

Some limitations of PROC SQL are: (i) it can merge up to 16 tables, versus SAS normal limit of 100; (ii) if you are merging many tables, PROC SQL can be slower relative to using a MERGE statement.

9. Using procedures to generate datasets

Most (if not all) SAS procedures can generate output datasets that you can use for further processing (merge with another datasets, etc), with either an OUT= option or an OUTEST= option. Note that you can use DATA step options when you create these datasets.

```
proc sort data=t5;  by fair;
proc means data=t5 noprint;
 var data6 data12;
 by fair;
 output out=t6 (drop=_freq_ _type_) mean= / autoname;
proc print data=t6;
```

The resulting dataset can be merged with the original data to "propagate" the mean values, allowing us to use it for further computations:

```
proc sort data=t5; by fair;
data spiff.t7;
  merge t5 t6;
  by fair;
```

For more options to generate output datasets from SAS procedures, see section 16 below, on the "Output Delivery System" (ODS).

10. LIBNAMES and SAS datasets

SAS creates two types of SAS "system" data files: temporary ("scratch" or "work" files) o "permanent". Both types of files are SAS data files. Physical SAS files have the "sas7bdat" extension for SAS versions 7 and up.

Temporary files are files created during a SAS session (either in interactive mode or during the execution of a SAS program. Temporary files are named in the program with a one-level name. For example in "DATA oranges", the dataset "oranges" is temporary. If the program ends successfully, the "oranges" file will be deleted by SAS at the end. If the execution fails, temporary files will not be deleted.

In each SAS installation, there is a default location for the work directory, where temporary files are created during a session. In Kellogg's UNIX server, temporary files are directed to "/scratch/sastemp". This location can be changed by specifying the "-work" option before execution:

```
sas –work /someotherpath progname
```

All default options for SAS are stored in a configuration file (sasv8.cfg) in the SAS program directory.

"Permanent" files are SAS files named with a two-level name during the program. The first level of this name is a handle or "libref" that associates the dataset with a specific directory where it will be created and stored. The libref is created in a LIBNAME statement. The following example associates the "~/training" directory to the libref "spiff". The libref is then used to write dataset "t7" to the directory named in the LIBNAME statement.

```
libname spiff '~/training';
data spiff.t7;
merge t5 t6 (drop=_freq_ _type_);
by fraud;
```

11. BY processing

There are many operations that require handling the data in groups of observations, as defined by common values of one or more variables. Most procedures, if not all, allow BY and/or CLASS statements. BY was also used above for merging data. The following example illustrates another use of BY-processing in a data step.

To use BY-processing in a data step, first you data must be sorted according to the variables needed in the BY statement. When the BY statement is invoked in the DATA step, SAS automatically creates a set of internal dummy variables that identify the boundaries of each group. In the following example, it will create four dummy variables: first.sic, last.sic, first.permno, and last.permno. First.sic takes a value of one for the first observation in every SIC code, and is zero otherwise. Last.sic will be one in the last observation of every SIC code and zero otherwise.

These internal variables can be used for a variety of purposes. In the example below, we are keeping the last observation of every "permno", and in the example that follows in section 12, we use the internal variables to create a counter.

```
PROC SORT DATA=t1; BY sic permno date;
DATA test;
SET t1;
BY sic permno;
IF LAST.permno THEN OUTPUT;
```

12. Creating counters

The following example creates a counter (called "qobs") that counts that starts over for each group. In this case, it is used to subset the first two observations in each group.

```
PROC SORT DATA=subset1;
   BY qdate DESCENDING idtdtd;
DATA subset2;
  SET subset1;
  BY qdate DESCENDING idtdtd;
  RETAIN qobs;
  IF first.qdate THEN qobs=0;
  qobs=qobs+1;
  if qobs in(1,2);
```

The RETAIN statement is extremely powerful. SAS normally reads and processes observation by observation in a data step. In each iteration, all the variables in the "program data vector" (the area of memory where the observation processed is being read) are reset to missing before reading the next observation. The RETAIN statement changes this behavior and keeps the value of the previous iteration. This is what enables the counter to increase from iteration to iteration.

13. Lags in panel datasets

The LAGn(argument) function can be used to create different lags of a variable. In the context of a panel dataset, the problem is the SAS DATA step does not distinguish between different cross-section units: if dealing with a monthly panel of securities, the lagged value of variable for the first observation of every cross-section unit (starting with the second one) will be set to the value of the variable for the last observation of the previous cross-section unit. A quick solution is to use BY processing and "clean" the lags:

```
PROC SORT DATA=test; BY permno date;
DATA test; set test;
  BY PERMNO;
  lagprc=LAG(prc);
  IF FIRST.PERMNO then lagprc=.;
```

Higher order lags can be created by specifying the order in the function: LAG2(prc), LAG3(prc), etc.

There is no "lead" function, but the LAG function can be used to create leads by using it after sorting the dataset in descending order (`BY permno DESCENDING date;`)

14. Arrays

SAS arrays are named groups of variables within a DATA step that allow you to repeat the same tasks for related variables. Arrays may have one or more dimensions. In the following example, we use only one dimension. Suppose you have received a data file where each observation is a firm (identified by permno) and 552 columns, each of them for a daily return, where missing values were coded as -99. Instead of writing 552 IF statements ("`if ret1=-99 then ret1=.; if ret2=-99 then ret2=.; etc`"), you could define an array that groups all 552 columns and the cycle through each variable replacing the values:

```
data test;
 set retdata;
   array returns(552) ret1-ret552;
   do i=1 to 552;
     if returns(i)=-99 then returns(i)=.;
   end;
   drop i;
```

There is more flexibility in SAS arrays than this example shows. For example, if the firm identifier was a character variable (and all variables that you want in the array were numeric), instead of declaring the list of variables in the array as "`array returns(552) ret1-ret552`", you could declare it as "`array returns(*) _numeric_`". In this case, SAS will group all the numeric variables in the array and count how many there are (the array dimension). The DO statement would then be rewritten as "`do i=1 to dim(returns);`"

15. Writing ASCII output datasets

15.1. Plain ASCII files: FILE and PUT statements

Within a data step, the combination of the FILE (naming the destination of the data) and PUT (which writes lines to the destination named in FILE) statements provides great flexibility. You may use it to generate anything from a plain ASCII data file to a LaTex table by including the proper formatting. The code below provides a very simple output file:

```
data temp;
   set inststat;
   file 'spectruminst.dat';
   put cusip $ year totsh noinst;
```

The PUT statement can be very refined. For example, if the data being written contains missing values, then you may want to write a formatted PUT statement, in which each variable will take a specific column location in the output file (e.g. cusip in columns 1-8 of the file, year in columns 10-13, etc).

15.2. Creating or reading CSV files: EXPORT and IMPORT

PROC EXPORT and PROC IMPORT are procedures that allow SAS to create and read comma-separated-values (CSV) files, which include headers. These files are easy to read in Matlab (csvread function – start reading in row 2 to avoid the header) and Stata (insheet command).

```
proc export data=temp
     outfile="spectruminst.csv"
     dbms=csv
     replace;
```

When you run a SAS program that includes a PROC EXPORT or a PROC IMPORT, be sure to specify the "-noterminal" option, as these procedures normally require X Windows:

```
sas –noterminal program_name
```

16. Useful procedures

16.1. "Flipping" your data: PROC TRANSPOSE

PROC TRANSPORT, part of the SAS/BASE procedures, allows to restructure a dataset converting variables into observations and vice versa.

```
FILENAME spiff 'externaldata/multi.csv';
PROC IMPORT DATAFILE=spiff OUT=test DBMS=CSV;
 GETNAMES=YES;
*PROC CONTENTS DATA=test; *PROC PRINT DATA=test (OBS=10);
 PROC TRANSPOSE DATA=test OUT=test1 NAME=ticker PREFIX=date;
   VAR msft ibm ba xrx;
   ID date;
   PROC PRINT DATA=test1;
```

16.2. Dealing with rolling windows: PROC EXPAND

The following program uses PROC EXPAND (part of SAS/ETS) to compute a moving sum over a 3-month window. The example in question is for cumulative returns, calculated using natural logs. The data must be sorted by the group identifier (permno) and time identifier (date); the variable specified in the ID statement must be a SAS date or time. By default, PROC EXPAND uses the spline method to interpolate missing values; by specifying METHOD=NONE, this is avoided. Finally, in the CONVERT statement, the options NOMISS and TRIMLEFT (which will set to missing the first two observations of every BY group) will ensure that all the computed cumulative returns include three dates. Otherwise, the first return for each permno would be in the second date, computed with only two observations.

```
DATA test; INFILE "rets2.txt";
```

```
INPUT permno date YYMMN6. ret;
FORMAT date YYMMN6.;
logret=LOG(ret+1);
PROC PRINT DATA=test (OBS=12);
PROC EXPAND DATA=test OUT=out METHOD=NONE;
 BY permno;
 ID date;
 CONVERT logret=movsum / TRANSFORMOUT=(NOMISS MOVSUM 3 TRIMLEFT 2);
DATA out;
 SET out;
 cumret=EXP(movsum)-1;
PROC PRINT DATA=OUT;
```

This procedure can also compute statistics for centered windows, forward looking windows, lags, leads within a period, etc, so it can be a welcome alternative to a more complex data step.

17. ODS to generate customized datasets from PROCs

Starting with version 7, the "Output Delivery System" (ODS) allows the user to control each portion of a procedure's output. In particular (among many features), ODS allows the user to create datasets from specific portions of the output that could not be sent to a dataset with an OUT= or OUTEST= option. The complete ODS documentation is available in the SAS/BASE manual "The Complete Guide to the SAS Output Delivery System".

To use ODS to create a dataset from a PROC statement:

1. Find out how SAS calls the piece of output you need. For this, type "ODS TRACE ON" before the PROC you need to examine, and "ODS TRACE OFF" after the PROC. Example:

   ```
   ods trace on;
   proc model data=uspop;
         pop = a / ( 1 + exp( b - c * (year-1790) ) );
         fit pop start=(a 1000  b 5.5  c .02);
   run;
   ods trace off;
   ```

2. Run the program and examine the log file – it will include a listing of each object of output sent to the LST file. For example, the table with parameter estimates in PROC MODEL is called "ParameterEstimates".

3. Use the object name to create the output file. Furthermore, you can "turn off" the listing with the "ODS LISTING OFF" statement. This is extremely useful if you need to run the same procedure repeatedly for a large number of units – if you add the NOPRINT option to the procedure, no output will be written at all. Example:

   ```
   ods listing close;
   ods output ParameterEstimates=d2 (drop=Esttype);
   proc model data=uspop;
   ```

```
        pop = a / ( 1 + exp( b - c * (year-1790) ) );
        fit pop start=(a 1000  b 5.5  c .02);
quit;
ods listing; /* Allow output of PROC PRINT to be listed */
proc print data=d2;
 title "ODS output dataset";
```

18. IML – Interactive Matrix Language

SAS has a matrix language in which expressions are close to Matlab. You can read a SAS dataset into a matrix or produce a dataset from a matrix – you may go back and forth between using SAS data steps and procedures and using IML. Like in PROC SQL, you end the procedure with a QUIT statement.

The following example shows some functions in IML: (i) first, we read the "fake" dataset into a matrix beta1; (ii) compute betax the inverse of beta1^T*beta1 and print it (the matrix will be printed in the LST file); (iii) define a column vector and a 2 by 2 matrix providing the data in IML; (iv) define a row vector with elements: x1, x2, x3; (v) create a new dataset, fake2, with the contents of betax. The CREATE statement defines the dataset with the characteristics of the betax matrix, using newvars as the source of the column names. The APPEND statement "populates" the dataset with the data from the matrix. Note that the transpose symbol is the backquote character, generally on the top left of a keyboard; alternatively, you may write t(beta1).

```
data fake;
input cusip $ v1 v2 v3;
cards;
a 1 34 65
b 1 78 32
c 2 12 01
d 4 54 29
;
proc print data=fake;
proc iml;
 use fake;
 read all var _num_ into beta1 [rowname=cusip colname=names];
 print beta1;
 betax=inv(beta1`*beta1);
 print betax;
 columnvector={1,2,3,4};
 twobytwomatrix={1 2, 3 4};
 print columnvector, twobytwomatrix;
 newvars='x1' : 'x3';
 create fake2 from betax [colname=newvars];
 append from betax;
 close fake2;
quit;
proc print data=fake2;
```

19. Macros – an example

The next step in learning SAS is to learn macros. The SAS macro language allows you to run a program for different parameters, such as tickers, dates, etc.

- Build your macros step by step based on a program that works. At each stage of the program, use PROC PRINT to take a look at intermediate output.
- Add one macro variable at a time.
- Test extensively. For example, test for two tickers, then for 10. Make sure that the macro produces the desired results before trying it on the entire universe of tickers! At each stage of the program, use PROC PRINT to take a look at intermediate output.
- Complex macros are extremely hard to debug, which is why you should follow the rules above.

Suppose you have some data in which you would like to replace the missing values for each variable with the median for the variable. In this example, there are two variables, but the macro was written for any number of variables. First, we compute the median with PROC MEANS, and transpose the output dataset so that there will be only one variable (col1) with the medians for each variable.

```
data fake;
input id $ v2 v3;
cards;
1 37483 .
2 . 83478
3 758 7848
4 584 4785
5 849 49549
6 .   73
7 . .
8 347 .
;
proc print data=fake;

proc means data=fake noprint;
  var _numeric_;
  output out=fake2 (drop=_type_ _freq_) median= / autoname;

proc transpose data=fake2 out=fake2;
proc print data=fake2;
```

Next, we use the SYMPUT function, one of the DATA step interfaces to the macro facility. The SYMPUT function assigns a value produced in the data step to a macro variable. Since the macro could be used for many variables (and we don't want to count), the macro variable names are automatically generated by appending the observation number to the "med" prefix: med1, med2, ... , med_n_. The LEFT function trims leading blank spaces that would be added if there were more than 9 observations – numbers are "right aligned": e.g., . With the help of the "END="

option in the SET statement (which defines a dummy variable that will be set to 1 when it reaches the end of the file, in the last observation). We also define an additional macro variable "maxi", which will have the total number of observations (number of medians, in this case).

```
data _null_;
   set fake2 end=last;
call symput('med'||left(_n_),col1);
if last then call symput('maxi',_n_);
run;
```

Now that we have the necessary ingredients, we can write the macro that will replace missing values with the corresponding median. Macro statements are identified by the leading percentage (%) sign, while macro variables are identified by the leading ampersand (&). You start a macro by defining it: `%macro macroname;` (where `macroname` is a name assigned by the user) and end the macro with a `%mend;` statement. To run the macro, you use assigned the macro name: `%macroname;`.

The tricky part in this macro is how SAS resolves it: SAS will read the entire macro and replace the macro variables with their assigned values. We want it to cycle from the first variable and corresponding median to the last (nth) variable. For the first variable in the array, we want to assign the first median for each missing value. Notice that in the IF statement, the item corresponding to the median has two ampersands: "`&&med&n`". This is to force SAS to resolve the macro in two passes. In the first one, for n=1, the IF statement will resolve to: "`if vars(1)=. then vars(1)=&med1;`"

In the second pass, &med1 will be replaced by the actual value for the median of the first variable in the array. Without the double ampersand, SAS would try to evaluate the IF statement in the first pass ("`if vars(1)=. then vars(1)=med1;`") and produce an error since it expected a numeric value. Finally, note that the n index in the %do statement is also a macro variable.

```
%macro substit;
data fake3;
set fake;
array vars (*) _numeric_;
  %do n=1 %to &maxi;
    if vars(&n) = . then vars(&n)=&&med&n;
  %end;
%mend;
%substit;

proc print data=fake3;
```

20. Using SAS manuals

SAS has extensive documentation for each of its modules. The complete set of manuals is available online:

http://support.sas.com/onlinedoc/913/docMainpage.jsp

All DATA step statements, functions and options are explained in detail in the BASE SAS manual. The SAS macro language is also part of BASE. The following table provides a list of important procedures and their location in the SAS manuals. Most procedures allow you to create output datasets and avoid listing output with the "NOPRINT" option.

Manual	Procedure	Use
Base	append	Append one dataset to another
	contents	List variables in a dataset
	corr	Computes correlations (Pearson, Spearman, …)
	datasets	Lists datasets available in a specified library; allows you to delete datasets within a program
	export	Translate dataset to a different format, such as CSV
	freq	Computes frequencies and crosstabs
	import	Import files in a different format, such as CSV. The Windows version allows you to read Excel or Access files with this procedure.
	means	Computes summary statistics
	print	"Prints" observations in the output (LST) file. Important options are OBS=n to control the number of observation and specifying the variables to be printed with the VAR statement.
	rank	Computes ranks and can divide observations in groups (percentiles, deciles, etc)
	transpose	Restructure dataset
	univariate	Computest detailed univariate statistics, including plots of a variable's distribution
STAT	logistic	Logistic regression (logit, probit, cloglog)
	nlin	Non linear estimation
	probit	Probit
	reg	Regression, OLS
ETS	arima	ARIMA models, including identification tests (ACF, PACF, Q tests, etc)
	autoreg	GARCH models
	expand	Time series data interpolation, data manipulation, etc
	mdc	Multidimensional choice models (nested logit, multinomial probit, conditional logit, etc). Experimental in version 8.
	model	Estimate one or more nonlinear equations: OLS, 2SLS, SUR, 3SLS, GMM, FIML
	qlim	Qualitative and limited dependent variable models: logit, probit, bivariate logit/probit, etc. Experimental in version 8
	syslin	Simultaneous linear equations
	tscsreg	Panel data – disadvantage: does not allow for unbalanced panels
	x11	Seasonal adjustment
IML	IML	Matrix language
OR	nlp	Non Linear Programming

SAS also has a GRAPH module, which produces very good graphics. Useful additional recommendations

An extremely useful reference book (not part of the on-line documentation) is the SAS Institute's "Combining and Modifying SAS Data Sets: Examples." Also useful is the volume by Boehmer, Broussard and Kallunki, "Using SAS in Financial Research."