SAS Interactive

There are 2 approaches to using SAS. The SAS GUI can be used to run interactive SAS commands. The following UNIX command will start the GUI:

```
kingkong:~:% sas &
```

"`kingkong:~:%`" is the prompt; you don't type that. The first string, "`kingkong`", is the name of the machine. The second string, "`~`", (which is the tilde character we discussed last time) represents the Present Working Directory (environment variable `$PWD`) which, in this case, is your home directory (environment variable `$HOME`) for which the abbreviation "`~`" is used for typing and display. The third string, "`%`", just represents the end of the prompt; what follows is what you type. The command is "`sas`" and followed by "`&`" runs the command in its own shell so that the prompt shell is free to be used for other commands. Now you can create/edit SAS code and submit it by pressing F3 or via point-and-click, but you will not be able to continue until the SAS code execution is complete. We don't recommend this approach; it is only here for completeness.

SAS batch with XEmacs/ESS

The recommended approach is to run batch SAS programs from the powerful XEmacs editor with the ESS package (you don't need to know the details of how the XEmacs/ESS/SAS interface is accomplished; it's the job of the system administrator to alleviate the implementation burden from the users). The PCOR UNIX FAQ http://www.mcw.edu/PCOR/Education/FAQs.htm has a nice overview of XEmacs/ESS at http://www.mcw.edu/PCOR/Education/SAS/XEmacs.htm

SAS Initialization

Before your SAS program runs, SAS processes your "`autoexec.sas`" file(s). First it processes, the "`autoexec.sas`" file in your home directory, if any, and then the PWD, which is abbreviated ".", if any. After that, it runs your SAS program. This is convenient since the "`autoexec.sas`" file(s) are convenient places to put SAS statements that are tedious to repeat over and over again. There are 2 statements that often fall into this category: `LIBNAME` and `OPTIONS` (we will use upper case to represent SAS statements, but lower case statements are allowed by SAS and even encouraged since they are easier to type; we will use lower case for situations where the user can choose the naming such as variables and datasets). The `OPTIONS` statement tells SAS how you want it to behave. And you will be surprised to find that the most reasonable options are not necessarily the defaults for some reason (possibly for backwards compatibility concerns). So, a nice first step is to open your "`~/autoexec.sas`" file, if any, and see if the following commands are present:

```
options ls=80 ps=60 errors=max nofmterr mprint noovp probsig=2 spool
dkrocond=error;
libname library "/library/lib";
title;
```

If not, then include them and submit the "`~/autoexec.sas`" file as a batch job with F3. It should complete nearly instantaneously. Then press F5 to go the "`~/autoexec.log`" file. F5 will take you to the first error message, if any. If it just takes you to the beginning of the file, then you know the job completed successfully and there were no syntax errors. You may also have noticed that the status of

the job was echoed at the bottom of the XEmacs editor in the minibuffer region. If the message was something like "`[1]   + done          nohup nice sas autoexec -rsasuser`", this is also an indication that the SAS job was a success (the return code to the shell was zero). If you had seen something else like "`[1]   + exit 2        nohup nice sas autoexec -rsasuser`", that is an indication that the job failed (the return code was two). Pressing F5 will take you to the error message in "`~/autoexec.log`". A middle ground is also possible. You made a syntax error, but it was not substantial enough to cause the SAS job to fail; then you would have seen something like "`[1] + exit 1        nohup nice sas autoexec -rsasuser`" (the return code to the shell was one). If you press F5, then you will see that there are no error messages, but you will see a warning some where. F5 will not take you to warnings since there are many cases where SAS will issue warnings, but the priority has been placed on error messages which you will find of the most concern. In any case, you should have a habit of rigorously inspecting your "`.log`" file after each run. This goes for the new user all the way to the most seasoned SAS programmers.

The second command above requires some explanation. The `LIBNAME` statement associates a directory with a SAS name called a libref. The libref "`library`" is associated with the UNIX directory "`/library/lib`". Permanent SAS datasets can be stored in this directory by naming them "`library.something`" where "`something`" can be whatever you want as long as it starts with an alphabetic character or the underscore (after the first character, numerals can be used as well). Once again, note that "`something`" may be mixed case, but the resulting SAS dataset will always be lower case: "`/library/lib/something.sas7bdat`". This is for explanatory purposes only since the libref "`library`" is read-only and you cannot create permanent SAS datasets there. The purpose of libref "`library`" is to store the formats catalog "`library.formats`". This is where all of our user-defined formats will go (as opposed to the SAS-supplied formats which came with our installation of SAS software). These formats are created by the SAS program "`/library/sas/formats.sas`".

If you open the SAS program "`/library/sas/formats.sas`" you will see a fairly complicated program that mixes DATAstep code with some PROCs (`SORT`, `FORMAT` and `PRINT`). You probably won't get much out of looking at the program itself. However, if you press F6, this will take you to "`/library/sas/formats.lst`" where you will see a listing of the formats that have been created for your use. This is a work-in-progress since formats are often created and/or revised as the needs arise.

Data Types and Formats/Informats

SAS has 2 types of data: character and numeric. Character variables can be up to 32767 characters in length. However, character variables longer than, say, 40 are frowned upon since they are difficult to display. You use the `LENGTH` statement to set this (BEWARE: if not explicitly stated, SAS will set the length of your character variable based on the first observation which may be too short and cause you problems with later observations).

```
DATA new;

LENGTH char $ 10;

INPUT char;

...
```

```
RUN;
```

Numeric variables can contain both integers and real numbers. By default, positive and negative integers with an absolute value of 9,007,199,254,740,992 or less can be represented exactly (that's 2^{53} or about 9 quadrillion which is almost 10^{16} or 16 digits; Stata only supports integers up to 9 digits). Since integers and real numbers are of the same type, occasionally integer operations will result in a near-integer value rather than an integer (what are known as "bruised" integers) due to roundoff error. SAS supplies the FUZZ() function for these eventualities; it returns the nearby integer if it is within 10^{-12}. And, SAS will convert hexadecimal numbers to integers on-the-fly.

```
x=15; * All 3 lines store 15 as the value of the numeric variable x;
x=FUZZ(14.999999999999);
x=0Fx; * which can also be written as '0F'x or "0F"x;
```

Numeric missing data is represented by "." in SAS (which is also the convention adopted by Stata). Most operations on missing values result in missing values (and notes in the .log). However, less-than and greater-than comparisons need to be coded carefully since missing data is considered to be the smallest negative value (basically, negative infinity). The right way of dealing with this is faciliated by the N() function which counts the number of non-missing items in a list. For example, N(1, 2, ., 3) results in 3.

```
* if x can be missing, this is WRONG;
*IF x<=10 THEN …;
*ELSE IF x<20 THEN …;
* if x can be missing, this is RIGHT;
IF N(x) THEN DO;
     IF x<=10 THEN …;
     ELSE IF x<20 THEN …;
END;
```

Similar concerns surround boolean indicator variables: 0=False and 1=True. SAS interprets non-zero, non-missing numbers as True and zero or missing numbers as False. Therefore, unless care is taken, missing results will automatically be interpreted as False.

```
* if x can be missing, this is WRONG;
*IF x THEN …;
*ELSE …;
* if x can be missing, this is RIGHT;
IF N(x) THEN DO;
     IF x THEN …;
     ELSE …;
END;
```

It should be noted that there are other representations of missing: $._{-}$, $.$, $.A$, …, $.Z$. And these missing values have an ordering with $._{-}$ being the smallest and, $.Z$, the largest. Some of the datasets that we get from others have used "alphabetic" missing values so beware.

SAS also stores dates as integers (SAS is Y2K compliant, but you can easily perform non-compliant operations so beware). For SAS, day 0 is 01/01/1960 (which is the same for Stata so dates can be exchanged as integers). For comparison purposes, Microsoft Excel uses 12/30/1899 (this is not a typo).

```
format x date9.; * this format corresponds to the on-the-fly
         entry method below, but there are others;
x=15;
x='16JAN1960'd; * which can also be written as "16JAN1960"d;
x='16JAN1960'D; * which can also be written as "16JAN1960"D;
put x;
```

All store 15 as the value of the numeric variable x. SAS also has on-the-fly entry for times and datetimes. However, times are rarely encountered in the data we see so we only mention it in passing. Furthermore, on-the-fly entry is convenient for dates, but more often you are reading in the data from a text file than assigning each variable a value manually. In this case, we need an INFORMAT; the term SAS uses to convert character text to a numeric variable.

```
DATA new;
     INFILE "birth.txt";
     INPUT birth MMDDYY10.; * reads MM/DD/YYYY where / can be
                                  a special character or blank;
          * use DDMMYY10. for DD/MM/YYYY and
            and YYMMDD10. for YYYY/MM/DD;
     age=%AGEYR(birth, '01JAN2009'd); * calculates age in years;
RUN;
```

Notice that we have called the function %AGEYR() which calculates the number of years between a starting and a stopping date. The % signifies it is a SAS macro function call. We will discuss SAS macros in greater detail later. But, what this macro relies upon is the SAS date function

`INTCK('MONTH', starting, stopping)`. This function counts the number of intervals between two dates (here intervals are months which we will limit ourselves to other than to say that weeks, quarters and years are also possible). The naming of this function is mysterious, however, we can think of the `C` as to count. Another useful function is `INTNX('MONTH', starting, n)` which returns a date advanced from `starting`, `n` intervals. Again, the naming is suspect, but we can think of the `N` as the `n` to advance (please let me know if you have a better way to remember these function names since I still look them up often). There is an optional 4th argument to this function which has to do with the alignment of the date returned: the default is `'B'` for the beginning of the month; `'M'` for the middle and `'E'` for the end are also possibilities.

SAS macro functions like `%AGEYR()` serve an important purpose. They allow us to concisely code an operation that can't be concisely coded with the SAS language itself. SAS macros generate the longer set of necessary instructions which you will see in your `.log` file. Furthermore, they often perform something that is not so easily done with the SAS language. This allows you to perform those actions with a more general understanding of the goal rather than an intimate knowledge of the steps necessary to get there. These PCOR SAS macros are available on kingkong as part of the SAS installation. But, they can be installed on any system running SAS. For the SAS macros themselves, see http://www.mcw.edu/PCOR/Education/SASMacros.htm And to install them, see http://www.amstat.org/chapters/milwaukee/workshop/install.pdf

Sorting, Merging and Subsetting

It is often necessary to sort and merge SAS datasets. For example, suppose you have a dataset of Medicare inpatient claims and you would like to merge with hospital characteristics:

```
PROC SORT DATA=cmsm.inp OUT=inp; * creating temporary dataset inp;
     BY provider;     * sorted by provider with is the hospital id;
RUN;
DATA inp; * creating new inp dataset with variables from oscar.pos;
     MERGE oscar.pos inp;
     BY provider;
RUN;
```

Merging is rather flexible. You can merge by multiple SAS variables: the collection of which is called a key. And, you can merge multiple datasets at a time; suppose there are N datasets. Then, N-1 datasets must have unique keys. That means, for example, `oscar.pos`, must have only one record for each value of `provider`. The last dataset in the list, in this case `inp`, only has to be sorted by the same key, but does not have to be unique.

Now suppose that we want to subset this dataset. There are several possibilities. Suppose that we want to subset based on the hospital's location. In Medicare, the first 2 digits of a hospital's `provider` number represent the state that it is located in. In `formats.lst`, we see that WI is 52 in the format `$SSAST`.

```
DATA wi; * creating new inp dataset with variables for WI only;
     MERGE oscar.pos inp;
     BY provider;
```

```
          WHERE provider=:'52';
RUN;
```

The WHERE clause is used when all N datasets have a common variable: this is very fast since SAS just ignores all records that do not satisy this clause. However, if you want to subset more generally, you can use a subsetting IF or a DELETE/OUTPUT statement. For example, let's suppose that we only wanted to know how many admissions there were to all medical school-affiliated hospitals in WI. Then we would need both strategies plus the RETAIN statement:

```
DATA wi;
     MERGE oscar.pos inp(IN=inp END=last);
     * the inp variable is 1 when the record comes from the inp
  dataset and 0 otherwise;
     * similarly, the last variable is 1 for the last record of
  the inp dataset and 0 otherwise;
     BY provider;
     WHERE provider=:'52';
     RETAIN total 0; * create total variable which maintains its
                       value across observations;
     IF inp & medsch>0 THEN total=total+1;
  * increment total: when incrementing by 1 can be just total+1;
     IF last THEN OUTPUT;
     *IF last; * previous statement could also be a subsetting IF;
     *IF ^last THEN DELETE; * or it could have been this as well;
RUN;
```

The RETAIN statement is helpful, but with a BY statement, it is usually not what you really want. For example, to tally all admissions at each hospital we will use the %_RETAIN() macro which generalizes RETAIN.

```
DATA wi;
     MERGE oscar.pos inp(IN=inp);
     BY provider;
     WHERE provider=:'52';
     %_RETAIN(VAR=total=0, BY=provider);
  * create total variable which maintains its value across
   observations within a BY-group only;
     IF inp & medsch>0 THEN total=total+1;
     IF last.provider THEN OUTPUT;
     * the last.provider variable is created whenever a BY provider
```

```
      statement is present:  it is 1 for the last record of the
          merged wi dataset and 0 otherwise;
    * if you had more BY variables, then you would have more
      last. variables.  However, you usually want the last
          last. variable.  Similarly, first. variables are created
          and you usually want the last first. variable;
RUN;
```

Arrays and Looping

Most programming languages provide arrays and looping capabilities and SAS is no different. However, arrays are handled slightly differently. Arrays are not a permanent feature of a SAS dataset. Rather, they are a temporary short-hand that is present only in the DATAstep where they are defined. They often rely on variable lists. For example, a list of variables may be written as follows in either a PROC or a DATAstep:

```
X1-X5 /* expands to X1 X2 X3 X4 X5 */
```

Another type of list relies on the order that the SAS variables appear in the SAS dataset (which is defined by their order of creation which can be displayed by `PROC CONTENTS`):

```
X1—-Z5 /* expands to X1-X5 Y1-Y5 Z1-Z5 assuming that order */
```

So, two DATAstep definitions of arrays might be:

```
ARRAY X(5) X1-X5;
```

```
ARRAY XYZ(3, 5) X1—-Z5;
```

Let's look at a warehouse example. A warehouse is a dataset that summarizes a much larger dataset so that you may use the warehouse rather than the larger dataset which may save a lot of time and save you the extra SAS coding that may be very complex. Usually, a warehouse has a unique key where the larger dataset does not; another bonus. For example, the `cmsmf.charlson12` dataset summarizes the diagnoses and their corresponding dates appearing on inpatient (`cmsmf.inp`) and outpatient (`cmsmf.outp` and `cmsmf.carrier`) claims according to the NCI Combined definitions with the 12-month time window.

```
DATA nci;
      SET cmsf.charlson12(KEEP=hic c20030101-—c20061217);
      * for datasets with a large number of variables,
    just bring in those that you really need with KEEP=;
   * conversely, you can DROP= those that you don't need
    when that is easier;
      BY hic;
   ARRAY diag(2003:2006, 12, 17) c20030101-—c20061217;
```

```
   * for each year, and each month, there are 17 diagnoses;
     ARRAY wgt(17) _TEMPORARY_ (0.000 0.845 0.224 0.713 1.192 0.471
0.751 0.228 0.000 0.450 0.023 0.210 1.188 0.000 0.000 0.201 0.396);
  * these are the NCI Combined weights for breast cancer patients
   estimated by Klabunde et al. 2007;
    * a _TEMPORARY_ array has no associated variables;
  * any array can be initialized with a list of numbers in
   parentheses, but it only makes sense for a _TEMPORARY_ array
      because the initialization automatically RETAINs those values;
    DROP c20030101-—c20061217 i;
 * again, you should DROP variables that you no longer need
  especially important with a large number of variables;
    * KEEP should be used when that is easier.  For example,
  KEEP hic year month nci;
    DO year=2003 TO 2006;
        DO month=1 TO 12;
            nci=0;


            DO i=1 TO 17;
                nci=nci+diag(year, month, i)*wgt(i);
                OUTPUT;
            END;
        END;
    END;
RUN;
```

Spreadsheets and Stata

Technically, the SAS Transport format, `.xpt`, is the format for transporting datasets from one SAS user to another. However, you can't always assume that the person receiving the data will have SAS due to the expense. If they do, then with recent versions of SAS (7, 8 and 9), the SAS dataset files themselves, `.sas7bdat`, are portable. Therefore, the SAS Transport format is not used much any more.

Comma Separated Values, `.csv`, files are a de facto interchange standard between spreadsheet applications like Microsoft Excel and statistical applications like SAS and Stata. As has been noted, SAS and Stata are fairly consistent: they share the ".” for missing values and the same day 0 for numeric dates, 01/01/1960, but Stata only supports integers up to 9 digits. Therefore, `.csv` files are often used to tranport data between SAS and Stata as well. The trickiest thing to handle is the various

choices for day 0: Microsoft Excel uses 12/30/1899.

There are 2 easy ways to import a `.csv` file into SAS. The first few lines of a `.csv` file with dates are as follows:

```
"CSID","Wave","Code","Death_dt"
10086,3,2310,39364
10107,1,2112,39702
10156,,,39304
10204,3,2310,39345
...
```

If the file has only a few variables as this one does, then the following may be the easiest:

```
DATA deaths;
  INFILE "deaths.csv" DSD FIRSTOBS=2;
* the DSD option is for .csv files;
* FIRSTOBS=2 skips the first line which has variable names;
  INPUT csid wave code death_dt;
  FORMAT death_dt DATE9.;
  death_dt=death_dt+'30DEC1899'd;
* switch day 0 from the Microsoft Excel setting to SAS equivalent;
RUN;
```

If the file has more than a few variables, then consider the `%_CIMPORT()` macro:

```
%_CIMPORT(INFILE=deaths.csv, OUT=deaths, DAY0='30DEC1899'd,
NUMDATES=death_dt, /* NUMDATES= is a list of numeric dates */
ATTRIB=death_dt FORMAT=DATE7.
/* the ATTRIB statement allows you to set the FORMAT/LABEL/LENGTH
  of a list of variables: when it used outside of a SAS macro,
  then you do NOT use the first equal sign in ATTRIB=, but
  the second equal sign is still necessary                  */ );
```

There is more documentation within the macro itself which you can find at

`/usr/local/sasmacro/_cimport.sas`

Similarly, when you need to create a file for use with Stata or a spreadsheet program, consider the `%_CEXPORT()` macro; you can find the documentation within the macro itself (which is the case for all of the macros that start with an underscore): `/usr/local/sasmacro/_cexport.sas`