This paper is derived from a tutorial, in which I presented tools for reading and processing hierarchical and relational data using the SAS System. As such, the figures used are the transparencies I used in that tutorial, and this text is a brief explanation of each of those figures.

This paper presumes some elementary familiarity with the SAS System, especially the language used in the DATA step. Specifically, it presumes familiarity with the use of the INFILE and INPUT statements to read data external to the SAS System, and to create SAS datasets. It also presumes some elementary familiarity with basic principles of data processing. Anyone with a few months experience using the SAS System to process data should meet these requirements.

The first portion of this paper discusses the logical structure of a SAS job, and the activities that occur when a DATA step is compiled, and when it is executed. The second portion of the paper covers the SAS System tools needed to read data with multiple lines per observation, and multiple observations per line. The third portion covers processing of hierarchical data with the SAS System, and the last portion covers processing relational data with the SAS System.

All SAS jobs consist of a series of one or more steps. There are two kinds of steps: DATA and PROC steps. DATA steps start with the DATA statement, and PROC steps start with the PROC statement. Each step is compiled and then executed separately. Any one step is compiled and executed by the SAS System before the system begins processing the next step. Figure 1 shows a SAS job consisting of six steps, three DATA steps and three PROCedure steps.

The major task of a DATA step is to create one or more SAS datasets. All SAS PROCedure steps expect to encounter their data in the form of SAS datasets. Most PROC steps read SAS datasets and usually perform some analysis and produce some form of printed report. Thus, one may consider the major task of DATA steps is to create SAS datasets for later PROC steps.

At compile time (Figure 2), the SAS System performs three major tasks. First, the system uses INFILE names and SAS library names to link to the actual system files. Second, the Program Data Vector (PDV) is built. Third, machine code is generated.

In the example in Figure 2, the SAS dataset name used in the DATA statement, INVOICE, is a single-level SAS dataset name, indicating that it is to be created in the WORK library (exactly where that is is system dependent). There is no INFILE statement as the data follow in-stream as indicated by the CARDS statement.

The PDV is temporary storage for one observation in the SAS dataset being created (it also includes some variables which the SAS System places there, which will not be discussed at the moment). SAS builds the PDV by recognizing, during compilation, variable names used in the SAS code, and inferring their attributes from their usage. It also uses information contained in the descriptor portion of existing SAS datasets, used in any SET, MERGE, or UPDATE statements to add variables to the PDV. In this example (Figure 2), the DATA step compiler processes the INPUT statement and determines that the PDV needs to contain four variables, NUMBER, NAME, DATE, and AMOUNT. The use if the dollar sign in the INPUT statement indicates to the compiler that the variable name immediately proceeding each dollar sign is a character variable. Because the INPUT statement uses list input (column and formats are not described), and this input statement is the first time the compiler encounters these variable names, the three character variables are eight byte character variables (capable of holding eight characters).

At execution time, the machine code generated during the compile phase is executed repeatedly, until an end-of-file condition is reached during an INPUT statement. Here, in this example, each execution of the DATA step program reads one line of input data and creates one observation during the SAS dataset. The input statement reads data from the input buffer and places it in the PDV, and, at the bottom of the step the PDV is copied as it exists at that moment, to the output SAS dataset, becoming the next observation.

Figure 3 contains one way to read data with multiple records per observation to be created. Actually, the same code as used in the previous example will work, because list input will simple refill the input buffer as needed when it encounters the end of the record during the input statement. Figure 4 contains two more ways to read the same data. The first uses two INPUT statements. Each INPUT statement causes SAS to refill the input buffer and then add values to the PDV based upon what is currently in the input buffer. The second uses one INPUT statement with the "/*" pointer control item. The "/*" pointer control item is an explicit instruction to refill the input buffer, i.e., to read the next line of data. Both produce exactly the same SAS dataset.

Figure 5 contains several more examples which read the same data. They use the "#" pointer control item to tell the SAS System to create a multi-line input buffer, and then to read from specific lines in that buffer. These examples all produce the same SAS dataset, except that in the last example in Figure 5, because the variables DATE and AMOUNT occur first in the INPUT statement, the compiler encounters them first, and places them in the PDV first, and hence they occur first in the SAS dataset. The dataset contains the same observations on the same variables; it is just that the physical order of the variables in the SAS dataset has changed. Since we always refer to variables by name, their physical order in the SAS dataset us almost always irrelevant.

It is easy to read data in which each line of input
data corresponds to multiple observations. The trailing @ and trailing @@ signs enable us to read such data and create observations by holding the input buffer for the next INPUT statement. Figure 6 contains the rules for use of the trailing @ and @@ signs. Figure 7 contains an example of the use of the trailing @@ sign to read data with an unknown number of observations per line. The trailing @@ sign sets a flag which tells the next INPUT statement not to refill the buffer, but to continue reading the current buffer where it last stopped. The input buffer is filled only when the pointer reaches the end of the buffer and the INPUT statement requests more data. Notice that the effect of the @@ sign crosses step boundaries, i.e., the flag remains set even through subsequent executions of the DATA step.

The single trailing @ sign does not remain in effect across step boundaries, i.e., subsequent executions of the data step cause subsequent INPUT statements to refill the buffer. Figure 8 contains an example of a data step in which we read only part of a buffer, and based on the value of that part, decide whether to read the rest of buffer or delete the entire observation.

We can use the trailing @ sign to read keywords from the data lines, and based on the value of the keyword, decide which variable to read. The next example (Figure 9) contains data in which there are four variables per observation, NUMBER, AMOUNT, NAME, and DATE. The variables occur in the raw data one per line, and they can occur in any order. Thus, in the first group of lines, the order of the variables is NUMBER, NAME, DATE, AMOUNT, while in the second group of lines, the order of the variables is NUMBER, DATE, AMOUNT, NAME. Each group of lines ends with a line containing the word, "END," which the program uses to trigger the output of the PDV to the SAS dataset and the initializing of the variables to zero or blank. Figure 10 contains the output from this example.

We have now developed sufficient tools to read and process hierarchical data, or tree-structured data. A hierarchical data file has header records and detail records, and the information on a header record applies to the information on all detail records (Figure 11). For example (Figure 12), suppose we have raw data with header records containing names, account numbers, and current balance, and following each header record, multiple detail records containing transactions to that account. Header records contain the string "HEADER" in the first six columns, and detail records contain the string "DETAIL" in the first five columns. The program fills the variables NAME, NUMBER, and BALANCE from a header record, but does not output the PDV as an observation. The program uses the detail records to update the balance, and for each detail record, outputs an observation. Thus, the SAS dataset created contains an observation for each detail record, containing (Figure 13) NAME, NUMBER, and BALANCE, which reflects the result of each transaction.

In the previous example, the program created an observation for each detail record. In the next example, the program will create an observation for each header record, but reflecting the information in all the detail records that follow it (Figure 14). The computer program is given in Figure 15 and the data in Figure 16. The data for any one company consists of a header record (identified by a record type of "C") containing the company ID number (COMPNUM) and the company name (COMPANY), and a series of detail records. Identified by a record type of "D," each detail record contains a item number (ITEMNUM), the number of that item ordered (NUMBER), and the name of the item (PARTNAME). When a header record is encountered, the program outputs the PDV, which at that point contains information pertaining to the number of each item bought by the previous company, whose name is still in the PDV. The program then initializes the counters to zero and continues. When detail records are encountered, the program reads the item number (ITEMNUM) and the number ordered (NUMBER) and increments the appropriate counter. (Although the prices of the items were hard coded in the program, they could have been read in from another external file.)

Our final hierarchical example is another complex, hierarchical example, in which the purpose is to create a flat file in SAS dataset form from a set of hierarchical external data. The SAS dataset has one observation for each detail record in the raw data file, and this observation contains information from both the detail record and the parent header record. Figure 17 contains the program listing for the first solution and the sample data. Each group of records comprises a family. The record type of the header record is FAMILY, while the record type of detail records may be FATHER, MOTHER, SON, or DAUGHTER. The header record contains the family name and the number of persons in the family. The first solution (Figures 17-20) does not read the number of persons in the family; it depends on the record type to decide when it has reached all the detail records for a particular family. The second solution (Figure 21) is a slightly more elegant solution which depends on knowing the number of persons in each family to read the detail records with a DO loop.

Lastly, we will briefly discuss relational data, which are simply a collection of flat, two-way tables, with some restrictions given in Figure 22. Thus, a series of SAS datasets in a SAS data library, may qualify as a relational database. Observations in the various SAS datasets are related by keys. Manipulations we may wish to do to these SAS datasets may be concatenation, interleaving, subsetting observations and/or variables, and a relational join (a SAS MERGE). One way to do this is simply to use the tools SAS gives us to do these: the SET, MERGE, and UPDATE statements, with appropriate
DROP/KEEP options and use of DELETE/subsetting IF statements. Figure 22 contains some relational data (the two datasets relate via the key, NUM), and the program (Figure 24) reads both raw datasets and performs a relational MERGE.

SAS is the registered trademark of SAS Institute, Inc., Cary, NC, USA.

All SAS Jobs consist of DATA and PROC steps

DATA ONE;
...
PROC MEANS;
...
DATA TWO;
...
DATA THREE;
...
PROC PRINT;
...
PROC PLOT;
...

Figure 1

Multiple Records per Observation

DATA INVOICE;
  INPUT NUMBER NAME $ DATE $ AMOUNT;
CARDS;
  12345 SMITH 10/16/86 42.08
  83635 JONES 11/13/86 123.20
  91116 BROWN 12/22/86 4.98
PROC PRINT;

PDV NUMBER NAME DATE AMOUNT

OBS NUMBER NAME DATE AMOUNT
  1 12345 SMITH 10/16/86 42.08
  2 83635 JONES 11/13/86 123.20
  3 91116 BROWN 12/22/86 4.98

Figure 2

DATA INVOICE;
  INPUT NUMBER NAME $ / DATE $ AMOUNT;
CARDS;
  12345 SMITH 10/16/86 42.08
  83635 JONES 11/13/86 123.20
  91116 BROWN 12/22/86 4.98

DATA INVOICE;
  INPUT NUMBER NAME $ / DATE $ AMOUNT;
CARDS;

Figure 4

COMPILING AND EXECUTING THE DATA STEP

Compile time
- File and dataset links established
- The PDV is created
- Machine code is generated

Execution time
- Machine code is executed
- Files and datasets are read and written
DATA INVOICE;
    INPUT NUMBER NAME $ #2 DATE $ AMOUNT;
    CARDS;
   12345 SMITH 
      10/16/86  42.08
   83635 JONES 
      11/13/86 123.20
   91116 BROWN 
      12/22/86  4.98
PROC PRINT;

DATA INVOICE;
    INPUT #1 NUMBER NAME $ 
            #2 DATE $ AMOUNT;
    CARDS;
PROC PRINT;

DATA INVOICE;
    INPUT #2 DATE $ AMOUNT 
            #1 NUMBER NAME $ ;
    CARDS;
PROC PRINT;

PDV DATE AMOUNT NUMBER NAME

Figure 5

Multiple Observations per Record
• Holding onto a record [tightly]
• Use of trailing @ and @@

The trailing @ sign respects step boundaries
The trailing @@ sign does not respect step boundaries

When a trailing @@ is used, a new record is
copied to the input buffer ONLY when an end-of-
record condition occurs

When a trailing @ is used, a record is copied to
the input buffer when
• An end-of-record occurs
• When control cycles to the beginning of the data
  step because
  • The bottom of the data step has been reached
  • A RETURN statement is executed
  • A DELETE statement is executed

Figure 6

Example: Double trailing @ sign

DATA TRANS;
    INPUT NUMBER AMOUNT @@;
    CARDS;
   12345 42.08 83635 123.20
   91116  4.98
   91117  5.65 91118 1024.00
;
PROC PRINT;

PDV NUMBER AMOUNT

OBS  NUMBER  AMOUNT

1   12345  24.08
2   83635 123.20
3   91116  4.98
4   91117  5.65
5   91118 1024.00

Figure 7

Example: Single trailing @ sign
(Conditional execution of the INPUT statement)

DATA TRANS;
    INPUT NUMBER @;
    IF NUMBER<90000 THEN INPUT AMOUNT;
    ELSE DELETE;
    CARDS;
   12345 42.08
   83635 123.20
   91116  4.98
   91117  5.65
   91118 1024.00
;
PROC PRINT;

PDV NUMBER AMOUNT

OBS  NUMBER  AMOUNT

1   12345  24.08
2   83635 123.20

Figure 8
### Complex Example: Keyword Input

```sas
DATA INVOICE;
LENGTH NUMBER AMOUNT 8 NAME DATE $ 8;
RETAIN NUMBER NAME DATE AMOUNT;
INPUT TYPE $ 8;
IF TYPE='END' THEN DO;
  OUTPUT;
  NUMBER=0; AMOUNT=0; NAME=''; DATE='';
END;
ELSE IF TYPE='NUMBER' THEN INPUT NUMBER;
ELSE IF TYPE='AMOUNT' THEN INPUT AMOUNT;
ELSE IF TYPE='NAME' THEN INPUT NAME $;
ELSE IF TYPE='DATE' THEN INPUT DATE $;
CARDS;
```

```sas
NUMBER 12345
NAME SMITH
DATE 10/16/86
AMOUNT 42.08
END
NUMBER 83636
NAME JONES
DATE 11/13/86
AMOUNT 123.20
```

### PROC PRINT:

```sas
PROC PRINT;
Figure 9
```

<table>
<thead>
<tr>
<th>OBS</th>
<th>NUMBER</th>
<th>AMOUNT</th>
<th>NAME</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12345</td>
<td>42.08</td>
<td>SMITH</td>
<td>10/16/86</td>
</tr>
<tr>
<td>2</td>
<td>83636</td>
<td>123.20</td>
<td>JONES</td>
<td>11/13/86</td>
</tr>
</tbody>
</table>

**Figure 10**

### Hierarchical Data
- Header and Detail Records
- (Parent and Child Records)
- Equal and Unequal Numbers of Detail Records
- Known and Unknown Numbers of Detail Records
- Detail Records can be Header Records, too

**Figure 11**

### Complex Hierarchical Example

```sas
PROC PRINT;
Figure 12
```

```sas
PDV TYPE NAME NUMBER BALANCE AMOUNT
```

<table>
<thead>
<tr>
<th>OBS</th>
<th>NAME</th>
<th>NUMBER</th>
<th>BALANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SMITH</td>
<td>12345</td>
<td>1010</td>
</tr>
<tr>
<td>2</td>
<td>SMITH</td>
<td>12345</td>
<td>960</td>
</tr>
<tr>
<td>3</td>
<td>SMITH</td>
<td>12345</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>JONES</td>
<td>83635</td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td>JONES</td>
<td>83635</td>
<td>800</td>
</tr>
</tbody>
</table>

**Figure 13**

### Complex Hierarchical Example

Accumulating multiple totals

**Figure 14**

### OPTIONS LS=78 NOOVP;

```sas
DATA FILE (KEEP=NAME NUMBER BALANCE);
RETAIN NAME NUMBER BALANCE;
INFILE CARDS END=ENDFLAG;
INPUT TYPE $ 6;
IF TYPE='HEADER' THEN
  INPUT NAME $ NUMBER BALANCE;
ELSE IF TYPE='DETAIL' THEN DO;
  INPUT AMOUNT;
  BALANCE+AMOUNT;
  OUTPUT;
  END;
ELSE ABORT;
CARDS;
```

```sas
HEADER SMITH 12345 1000.00
DETAIL 10.00
DETAIL -50.00
DETAIL 40.00
HEADER JONES 83635 500.00
DETAIL -300.00
DETAIL 600.00
PROC PRINT;
```

**Figure 12**
OPTIONS LS=78 NOOVP;
DATA REPORTS(DROP=ITEMNUM PARTNUM
PARTNAME TYPE NUMBER);
LENGTH COMPANY $ 33;
RETAIN COMPNUM COMPANY LASTREAD;
INFILE CARDS END=ENDFLAG;
IF ENDFLAG=0 THEN INPUT TYPE $ 1 0;
IF TYPE='C' OR LASTREAD THEN DO;
WTOTAL=3.25*WIDGETS;
TTOTAL=4.50*TOP;
BTOTAL=1.25*BOTTOM;
STOTAL=6.50*SIDE;
IF _N_ ^= 1 THEN OUTPUT;
IF LASTREAD THEN STOP;
END;
ELSE IF TYPE='I', THEN DO;
INPUT ITEMNUM NUMBER PARTNAME $;
IF ITEMNUM=301 THEN WIDGETS=0;
ELSE IF ITEMNUM=302 THEN TOP=0;
ELSE IF ITEMNUM=303 THEN BOTTOM=0;
ELSE IF ITEMNUM=401 THEN SIDE=0;
IF ENDFLAG THEN LASTREAD=1;
ELSE IF TYPE='I' THEN DO;
ELSE IF ITEMNUM=301 THEN WIDGETS=0;
ELSE IF ITEMNUM=302 THEN TOP=0;
ELSE IF ITEMNUM=303 THEN BOTTOM=0;
ELSE IF ITEMNUM=401 THEN SIDE=0;
IF ENDFLAG THEN LASTREAD=1;
END;
FORMAT BTOTAL STOTAL TTOTAL WTOTAL 6.2;

Figure 15

CARDS;
C 1402 SMITH MANUFACTURING COMPANY
I 301 1 WIDGET
I 302 2 TOP
I 303 1 BOTTOM
I 401 4 SIDE
I 301 2 WIDGET
I 302 4 TOP
I 301 1 WIDGET
C 1592 JONES COMPUTER SERVICES
I 301 3 WIDGET
I 302 1 TOP
I 301 1 WIDGET
I 401 6 SIDE
PROC PRINT;
ID COMPANY;

Figure 16

Complex Hierarchical Example
Creation of a Flat File
From Hierarchical Data

DATA PERSONS;
INPUT TYPE $ 1;
IF TYPE='FAMILY' THEN DO;
INPUT LASTNAME $;
DELETE;
END;
ELSE IF TYPE='I', THEN DO;
INPUT ITEMNUM NUMBER LASTNAME $ AGE HEIGHT;
PROC PRINT NOOBS;
VAR TYPE LASTNAME NAME AGE HEIGHT;
PROC PRINT NOOBS;
VAR WEIGHT SYSTOL DIASTOL PULSE;
RETAIN LASTNAME;
CARDS;
FAMILY SMITH 6
FATHER JOHN 42 68 148 130 82 67
MOTHER MARY 43 66 115 110 75 77
SON JOHN 20 70 135 130 95 77
SON DICK 17 68 190 120 80 82
SON HARRY 14 66 135 105 65 80
FAMILY JONES 3
MOTHER SELMA 66 60 110 110 75 77
DAUGHTER LISA 12 55 75 105 65 60
DAUGHTER JUDY 43 60 180 140 130 90
FAMILY BROWN 2
PROC PRINT NOOBS;
VAR TYPE LASTNAME NAME AGE HEIGHT;
PROC PRINT NOOBS;
VAR WEIGHT SYSTOL DIASTOL PULSE;

Figure 17

FATHER DARYL 45 69 170 120 75 65
MOTHER BRENDA 40 60 130 130 100 85
PROC PRINT NOOBS;
VAR TYPE LASTNAME NAME AGE HEIGHT;
PROC PRINT NOOBS;
VAR WEIGHT SYSTOL DIASTOL PULSE;

Figure 18

TYPE LASTNAME NAME AGE HEIGHT
FATHER SMITH JOHN 42 68
MOTHER SMITH MARY 43 66
SON SMITH JOHN 20 70
SON SMITH DICK 15 66
SON SMITH HARRY 14 66
MOTHER JONES SELMA 62 60
DAUGHTER JONES LISA 12 55
DAUGHTER JONES JUDY 43 60
FATHER BROWN DARYL 45 69
MOTHER BROWN BRENDA 40 60

Figure 19
WEIGHT  SYSTOL  DIASTOL  PULSE  
148     130     82     67  
115     110     75     77  
210     135     95     77  
190     120     80     82  
175     110     85     85  
135     105     65     80  
110     145     110    75  
 75     105     65     80  
180     140     130    90  
170     120     75     65  
130     130     100    85  

Figure 20

DATA PERSONS (DROP=NUMBER I);  
INPUT TYPE $  LASTNAME $  NUMBER;  
DO I=1 TO NUMBER;  
  INPUT TYPE $  NAME $  AGE  HEIGHT  
    WEIGHT  SYSTOL  DIASTOL  PULSE;  
  OUTPUT;  
END;  
CARDS;  
FAMILY SMITH 6  
FATHER JOHN 42  D  148  130  82  67  
MOTHER MARY 43  D  115  110  75  77  
SON JOHN 20  D  70  210  135  95  77  
SON TOM 17  D  68  190  120  80  82  
SON DICK 15  D  66  175  110  85  85  
SON HARRY 14  D  66  135  105  65  80  
FAMILY JONES 3  
MOTHER SELMA 62  D  110  145  110  75  
DAUGHTER LISA 12  D  55  75  106  65  60  
DAUGHTER JUDY 43  D  60  180  140  130  90  
FAMILY BROWN 2  
FATHER DARYL 45  D  69  170  120  75  65  
MOTHER BRENDA 40  D  60  130  130  100  85  
PROC PRINT;  

Figure 21

RELATIONAL DATA

What are Relational Data
• A collection of flat, 2-way tables
• Items in tables relate to items in other tables
• For each table, the following holds  
  • Each entry equals 1 data item
  • Column homogeneous
  • Row homogeneous
  • Distinct rows
  • Rows/columns can be viewed in any sequence, at any time, without affecting information content, or other functions using the table
• The tables are related by keys
  • Unique and nonredundent
  • Primary and Secondary
• ALL FILES CAN BE REPRESENTED AS FLAT FILES

Manipulations:
• Concatenate
• Interleave
• Subset Vertically
• Subset Horizontally
• Join

Example: Relational Data

INFILE EX7A  
1401  WIDGETS  3.25  
1402  HANDLES  4.00  
1403  NORBITS  2.50  
1404  NAPARTS  1.75  
1405  LODERTS  0.95  

INFILE EX7B  
1401  10/22/86  3  
1401  10/23/86  4  
1402  10/10/85  5  
1403  11/18/85  1  
1403  11/19/85  6  
1406  12/25/85  2  

Figure 22

Figure 23
DATA MERGE;
  RETAIN NAME PRICE LASTNUM;
  DROP NUM1 LASTNUM;
  INFILE EX7B;
  INPUT NUM DATE $ NUMSOLD;
  IF LASTNUM ^= NUM THEN DO;
    INFILE EX7A;
    DO UNTIL(NUM1=NUM);
      INPUT NUM1 NAME $ PRICE;
    END;
  END;
  LASTNUM=NUM;
PROC PRINT;
OBS NAME PRICE NUM DATE NUMSOLD
1 WIDGETS 3.25 1401 10/22/85 3
2 WIDGETS 3.25 1401 10/23/86 4
3 HANDLES 4.00 1402 10/10/85 5
4 NORBITS 2.50 1403 11/18/85 1
5 NORBITS 2.50 1403 11/19/85 6
6 LODERTS 0.95 1405 12/25/85 2
Figure 24