The Missing Semicolon™

End of Line for Different ASCII Files

When reading in a text file, it is important to match up the operating system reading in the file with the way that the file was created. Although DOS, UNIX and MACS all use ASCII files, one key difference can cause considerable problems if one is unaware of it. For text files in Windows editors, including notepad, the default end of line is a carriage return and a line feed. This translates to the hex codes 0D 0A. UNIX expects just a line feed or hex 0A, while MACS use only a carriage return or hex 0D. Therefore, each of these file types may need to be read in a slightly different way.

If a program in UNIX, for example, is trying to read a text file that was created on a Microsoft operating system without converting the file, the program will not recognize the end of line in the text file correctly. The end of line is read as only the 0A character, so the 0D character appears as a part of the data. It can be very difficult to detect, as the 0A and 0D characters are not printable characters, so unless one is using a text editor with a hex code option, it is not readily apparent which character is being used. This is not an issue when the program reads specific columns from the line. It only occurs when an input statement is open ended, reading all of the characters remaining in a particular line.

Fortunately there are some relatively simple solutions. Certain text editors such as UltraEdit have options for file handling and can save text files in specific formats corresponding to the system that is being used for the job. When saving a program, make sure that it is saved in the proper format. By doing this, the data should be read correctly. SAS also provides some tools and options to read the data correctly.

Here is an example of a file, created on Windows, that is transferred to UNIX and read there.

In Notepad the file appears as three simple lines of text with each line containing 3 characters.

ABC
DEF
GHI

After transferring the file to UNIX, we hoped the following program would run and select the first record. But as you can see, no records are selected.

```
1         data test;
2          infile'testfile.dat'lrec l=8;
3          input Name $;
4          if name='ABC';
5         run;
```

```
NOTE: 3 records were read from
the infile 'testfile.dat'.
The minimum record length was 4.
The maximum record length was 4.
NOTE: The data set WORK.TEST has
0 observations and 1 variables.
```

So, why didn't the program work and how do we solve the problem? One thing to note is that the minimum and maximum record lengths were four, not the three we expected. Secondly, we can use the SAS LIST statement to display the input buffer. LIST shows the input record and displays hexadecimal as needed. Thirdly, the SAS PUTLOG statement can display the problematic variable.

Looking at the new SAS log, one can see that there is an extra character, the carriage return (hex 0d), that Windows uses but UNIX doesn't.

```
8         data test;
9          infile 'testfile.dat' lrecl=8;
10         input Name $;
11         putlog name= $hex10.;
12        list;
13        if name='ABC';
14        run;
```

```
Name=4142430D20
```

SYSTEMS SEMINAR CONSULTANTS, INC.

**2997 Yarmouth Greenway Drive
Madison, WI 53711**

www.sys-seminar.com
train@sys-seminar.com
1-800-997-7081

In This Issue

run;

Letter From the President

Dear SAS User:

"We Have Computers, Let's Use Them."

It has been busier than ever at our office. We have been involved in large data warehouse projects, as well as major platform conversions.

When faced with moving hundreds of programs, it is important to use the best tools available. Not only are we moving the SAS applications, we use SAS as the tool to do much of the conversion. A client once told me, "We have computers; let's use them." For example, rather than manually converting Z/OS JCL, we have written programs to convert the JCL into appropriate SAS commands.

Another time-saving utility we have written is a SAS log-reading program that digs out errors and also finds record counts for all files in and out. This is useful because your log can be summarized in a way that is easier and more effcient to read, analyze, and debug.

As always, SAS has proven itself to be an excellent utility program that saves an incredible amount of manual work. We have always prided ourselves on automating processes and procedures that save our clients' time.

Please let us know of your conversion or data warehousing needs, and we will help you leverage the power of SAS to reduce your manual workload.

Thanks for Reading,

Steve Fust

run;

SYSTEMS SEMINAR CONSULTANTS, INC.

ASCII Files

```
RULE:        ----+----1----+----2----+

1    CHAR    ABC. 4
     ZONE    4440
     NUMR    123D
Name=4445460D20

2    CHAR    DEF. 4
     ZONE    4440
     NUMR    456D
Name=48494A0D20

3    CHAR    HIJ. 4
     ZONE    4440
     NUMR    89AD
NOTE: 3 records were read from the infile
'testfile.dat'.
The minimum record length was 4.
The maximum record length was 4.
NOTE: The data set WORK.TEST has 0 observations
and 1 variables.
```

How do we read this in successfully? In this case, we could add columns to our input to only read three characters. Although this can work, a better technique is to tell SAS what kind of record terminator we are using. In the UNIX section of the SAS Documentation, we find that INFILE has an option to specify using both the carriage return and the line feed characters as terminators. Adding that option solves our problem and produces the record we were hoping for.

```
15        data test;
16          infile 'testfile.dat' lrecl=8 termstr=crlf ;
17          input Name $;
18          if name='ABC';
19        run;

NOTE: 3 records were read from the infile
'testfile.dat'.
The minimum record length was 3.
The maximum record length was 3.
NOTE: The data set WORK.TEST has 1 observations
and 1 variables.
```

There are other values for TERMSTR available for jobs running under all platforms to specify the terminators from a different creating platform.

In summary, reading text files correctly is difficult enough when only one platform is involved; when a file is created on a different platform, it becomes even more difficult. In any case, with a little background and some tools, we should be able to read all files correctly.

run;

Sudoku and SAS Arrays, Part 2

In the last issue of *The Missing Semicolon*, I presented part of my Sudoku-solving program. The program first defines eighty-one variables, representing the cells in the sudoku, then sets up three arrays so that the cells can be referenced in terms of the rows, columns and quadrants they comprise. Next, the mainline portion of the program cycles through the rows, columns, and quadrants, sequentially feeding them to two algorithms that update the possible values for each cell. Last time, I described the flow of the mainline portion but did not reveal the algorithms. The algorithms are the main focus of this second installment.

Solving Sudoku

Before we plunge straight into the code, some background on how computers solve sudoku and how humans solve sudoku is necessary.

Sudoku are a special case of the "exact cover" problem in set theory. Donald Knuth devised an algorithm to solve the problem via manipulation of a 729 x 324 matrix. Another way to solve the exact cover problem is by brute force—checking different combinations of numbers until a solution is found. When implemented correctly, both Knuth's algorithm and this less elegant method are guaranteed to generate a solution to any sudoku. Many people have written sudoku-solving programs based on either Knuth's algorithm or the brute force method.

Humans typically solve sudoku by looking for patterns. Various websites describe some of these patterns. Two of the most universal are "Naked Singles" and "Hidden Singles."

Naked Singles
Each row, column, or quadrant (hereafter, "structure") may contain each digit only once. Therefore, if a digit already appears in a structure, it may be eliminated as a possibility for all other cells in that structure. When all digits but one have been eliminated for a cell, the remaining digit, known as a naked single, must be the correct value for that cell.

Hidden Singles
Each digit appears exactly once in each structure. Therefore, if a digit has been eliminated for every cell but one in a structure, it must belong in that cell. This line of reasoning is known as finding a hidden single.

My Program
My program works by cycling through the structures, searching for naked singles and hidden singles, then updating the possible values for cells accordingly. This process continues until iteration through every structure produces no more updates. Unfortunatley, this method does not generate a solution to every sudoku like Knuth's algorithm or the brute force method. In fact, only the most simple sudoku are solvable through identification of naked singles and hidden singles alone. However, many human beginners also focus on finding naked singles and hidden singles. This program, therefore, models the "human" method of solving a sudoku better than most sudoku-solving computer programs.

The Code
When the naked singles or hidden singles subroutine is called, the work81 variable contains data from a particular structure (that is, a particular row, column or quadrant). The work81 variable is comprised of nine, nine-character substrings for a total length of 81, so that it can store up to nine potential values for each of the nine cells in the structure. Each of the nine characters in a cell's substring is either a digit or a blank. If nothing is known about what values a cell may contain, the substring looks like this: '123456789.' If some digits have been eliminated, the substring contains blanks in place of those digits: e.g., '1 345 789.' If the value of the cell is known, the substring contains eight blanks and one digit: e.g., ' 7 '.

```
/**************************************************************/
/* if cell contains a single candidate,then
   eliminate candidate from all cells in structure."/
/**************************************************************/

naked_singles:
 do cell=1 to 9;
  fposition=((cell-1)*9)+1;
  search=left(substr(work81,fposition,9));
  if length(search)=1 then                 /* single dig? */
    do;
     work81=translate(work81,' ',search);
                                 /* change all cells inc itself */
     substr(work81,fposition,9)=search;
                                 /* set single digt back to search*/
    end;
 end;
return;
```

The naked singles algorithm loops through the nine cells in the structure once. It locates the substring representing the cell and checks whether it contains only one digit—that is, exactly one non-blank character. If the substring contains more than one digit, the next cell is checked. When the subroutine finds a one-digit cell, it replaces all other occurrences of the digit in the full work81 string with a blank. The blank indicates that the digit it replaced is no longer a possibility for the cell. In this way, any digit that is the known value of a cell is eliminated for all other cells in the same structure.

```
run;
```

```
/**************************************************************/
/* a digit occurs only once in a structure, but may have other
   digits with it.  When found, set cell and eliminate in rest of
                                             structure*/
/**************************************************************/

hidden_singles:
 do digit=1 to 9;
  digit_char=put(digit,1.);
  cnta{digit}=count(work81,digit_char);
   if cnta{digit}= 1 then
     do;
      fposition=index(work81j,digit_char);
      fposition=(int()fposition-1)/9)*9)+1;
      substr(work81,fposition,9)= digit_char!!' ';
     end;
end;
```

The hidden singles algorithm loops through the digits one through nine once. It counts the number of times the digit occurs in the work81 string. If the digit occurs more than once, it continues to the next digit. If the digit occurs only once, then the cell substring that contains the digit is set to the digit followed by 8 blanks. This indicates that the cell value is now known to be that digit, since it has been eliminated for all other cells in the structure.

```
unloadwork81:
                                /* reload structure with changed values */
   do cell=1 to 9;
     fposition=((cell-1)*9)+1;
     if structure="Q" then
         quada{struct_no,cell}=substr  (work81,fposition,9);
         else
           if structure="R" then
           rowa{struct_no,cell}=substr (work81,fposition,9);
               else
               if structure="C" then
cola{struct_no,cell}=substr(work81,fposition,9);
   end;
   if work81 ne save_work81          /* was any change made?*/
     then change='Y';                    /* yes, set flag */
return;
```

The unloadwork81 subroutine moves data from the work81 variable into the main data structures of the program. These data structures are three arrays corresponding to the three types of structure in the sudoku: rows, columns, and quadrants. The arrays all reference the same 81 variables, which represent the 81 cells in the sudoku. The cell variables therefore form a master data set that may be updated every time a structure goes through the naked singles and hidden singles algorithms.

```
report:
                                         /* graphic report */
 if "&graphics" ne "y" then
     goto report_text;
pageno+1;
pagenochar='Page_' !! put(PAGENO,Z2.);
%DCLANNO ;

 x=3;                                    /* title */
 y=32;
 text=rtitle;
 style='Swissb';
 size=1.5;
 position='6';
 output;
 DO Y=1 TO 28 by 3;
    if mod(y-1,9)= 0 then size=10;
       else
         if mod(y-1,3)= 0 then size=3;
            else size=1;
    if size=3 then linetype=2;
       else
           linetype=1;
    %LINE(1,Y,28,Y,BLACK,linetype,size);
 END;
 DO X=1 TO 28 by 3;
    if mod(x-1,9)= 0 then size=10;
      else
        if mod(x-1,3)= 0 then size=3;
           else size=1;
    if size=3 then
        linetype=2;
            else
            linetype=1;
    %LINE(X,1,X,28,BLACK,linetype,size);
 END;

Do row=1 to 9;
  do col=1 to 9;
    x=3*col - .5;
               /* trans to lower left origin on 27*27 grid */
    y=-3*row + 29.5;
  savex=x;
  savey=y;
    function='label';
    Color='Black';

style='swissb';
   position='+';
```

```
     search=left(all81{row,col});
 if length(search)=1 then                /* single dig? */
     do;
        text=substr(search,1,1);
        size=2;
          output;
     end;
   else
      do fposition=1 to length(search);
         text=substr(search,fposition,1);
         size=.8;
      style='swiss';
      color='Red';
         x=savex;
      y=savey;
        if text in ('1','2','3') then
          y=y+1;
        if text in ('7','8','9') then
          y=y-1;
        if text in ('1','4','7') then
          x=x-1;
        if text in ('3','6','9') then
          x=x+1;
        output;
        end;
   end;
end;
report_text:                             /*text report*/
file print;
put _page_@
put rtitle;
    put @1 110*'*';
    do row=1 to 9;
       do col=1 to 9;
     if col in(1,4,7) then
       put '*'@;
      else
       put '|'@;
     put +1 all81{row,col} $char9. +1 @;
     end;
    put @110 '*';
    if row in (3,6,9) then
      put @1 110*'*';
    else
       do;
         put 3*'*--------- | --------- | --------- '@;
         put @110 '*';
       end;

end;
put ///;
return;
 run;
```

Continued on next page

```
PROC GREPLAY NOFS IGOUT=WORK.GSEG;
 DELETE _ALL_; QUIT;
PROC Ganno ANNOTATE=annods
        name=pagenoCHAR;TITLE;run;
```

The reporting subroutine outputs a grid that shows the current state of the sudoku. The mainline portion of the program periodically calls the reporting subroutine as the sudoku is solved. Large black numbers are known values. Small red numbers are remaining potential digits.

A Puzzle That Program Can Solve

A level 2 Sudoku.

```
..5.9.8.6
42.......
.3.6....5
.7.9.....
..97521..
.....8.7.
6....4.5.
.......29
8.7.2.6..
```

Step1 — initial load

Interation   1 R Structure Report

Continued on next page

Interation 1 Q Structure Report

Interation 2 C Structure Report

Continued on next page

Interation 2 R Structure Report

Interation 2 Q Structure Report

7	1	5	2	9	3	8	4	6
4	2	6			3	9	7	
9	3	8	6	4	7	2	1	5
	7		9			5		
3	8	9	7	5	2	1		4
	6		4		8	9	7	
6	9	2			4	7	5	8
1		3			4	2	9	
8		7		2	9	6	3	1

Interation 3 C Structure Report

7	1	5	2	9	3	8	4	6
4	2	6	1	8		3	9	7
9	3	8	6	4	7	2	1	5
2	7	4	9	6		5	8	
3	8	9	7	5	2	1	6	4
5	6	1	4		8	9	7	
6	9	2	3		4	7	5	8
1	5	3	8	7		4	2	9
8	4	7	5	2	9	6	3	1

Interation 3 R Structure Report

7	1	5	2	9	3	8	4	6
4	2	6				3	9	7
9	3	8	6	4	7	2	1	5
		7	4	9		5	8	
3	8	9	7	5	2	1	6	4
5	6	1	4		8	9	7	
6	9	2			4	7	5	8
1	5	3	8	7		4	2	9
8	4	7	5	2	9	6	3	1

QUICK TIP

If you need to figure out time intervals, try using the intnx function with multipliers and shift indexes. SAS provides date, time, and datetime intervals for counting different periods of elapsed time. By using multipliers and shift indexes, you can create multiples of intervals and shift their starting point to produce more complex interval terms.

The general form of an interval name is

name<multiplier><.shift-index>

The *multiplier* and the *shift-index* arguments are optional and default to 1.

Example: Find the Monday of the week for a given date.

```
data _null_;
     Dec1st = '01Dec07'd;
     Monday = intnx( 'week.2', Dec1st, 0 );
     put Monday= date7.;
  run;

Monday=26NOV07
```

The "week.2" means the interval is by week starting on the second day. Monday, "0" moves you to the beginning of the interval.

More Examples

Interval	Description
WEEK6.11	Specifies six-week intervals starting on second Wednesdays
DAY3	Three-day intervals starting on Sundays
TENDAY4.2	Four ten-day periods starting at the second TENDAY period
MONTH2	January-February, March-April, May-June, July-August, September-October, November-December
QTR3.2	Three-month intervals starting on April 1, July 1, October 1, and January 1
SEMIYEAR.3	Six-month intervals, March-August and September-February
YEAR.10	Fiscal years starting in October
HOUR8.7	Eight-hour intervals starting at 6 a.m., 2 p.m., and 10 p.m. (might be used for work shifts).

run;

Placement Services

SAS Recruitment Expertise ◆ Personalized Approach ◆ Quality Candidates and Exciting Opportunities

Need a career change?

We will provide:

◆ Suggestions for resume improvement
◆ Interview tips
◆ Guidance throughout the process

Need additional staff ?

We will:

◆ Utilize SAS community resources
◆ Thoroughly screen candidates
◆ Assist throughout the process

Receive Regular Email Updates on Our Open Positions

Send your current resume and email address to careers@sys-seminar.com

Call our recruiters at 1-800-997-7081, ext. 313

Steve First
President

Katie Ronk
Director of Operations
Editor

Jennifer First
Office Manager
Editor

Ines Bowers
Administrative Assistant
Editor

Andrew First
Consultant

Kelly Kieler
Sr. Consultant

Kathleen Nosal
Programmer

Erin Ward
Placement Coordinator

Elizabeth First
Consultant
Editor