Data Step Hieroglyphics

Harry Droogendyk, Stratia Consulting Inc., Lynden, Ontario

Abstract

Have you found that most SAS data step code is quite readable? SAS statements, functions, keywords and SI supplied routines are most often given meaningful names. If the developer has been even somewhat verbose while naming data step variables, the final product can "read" quite easily.

Sure it reads easy …, until you encounter the {*&%~(.:@|? special characters littered throughout the code!! What in the world do those things mean? Do we really need the funny characters?

This paper will examine 10 implementations of special characters, each of them specially placed in the SAS language for your coding pleasure. Special characters, properly utilized, allow you to write code more quickly, more efficiently, and, believe or not, more simply.

Introduction

Unabashed SAS fans extol the flexibility and usefulness of the SAS data step language for solving real-world problems. It doesn't seem to matter how goofy the input data is, or how complicated the processing requirements are, SAS with its "toolbox" of functionality, will often provide multiple solutions. In addition, SAS code written on one platform often runs without modification on other platforms.

SAS-L subscribers often see this played out. A question is posted and three or four solutions are offered, each of them solving the problem differently. As the level of solution sophistication and efficiency rises, more of SAS's less obvious data step "tools" are invoked and special characters begin to litter the code! New users usually find this flexibility somewhat daunting and as a consequence, stay with the same old methods they encountered early in their SAS life.

This paper is an attempt to remove *some* of the mystery around those strange data step characters. It is certainly not an exhaustive treatment of any of the characters or functionality involved. See the SAS-L archives or SAS Online Documentation for all the intricacies and nuances of the subject matter.

Let's begin with several characters generally seen hanging around the often-used INPUT statement.

1. Trailing @

The @ sign is a familiar sight in the INPUT statement, most often seen *before* variable names when text files with fixed-width fields are processed. The @ is used to position the input pointer at a specific column of data, e.g.

```
INPUT @23 record_type  $3.;
      or
INPUT @numeric_variable record_type $3.;
```

It may be that the input file contains different types of data records. From the **single_at.txt** data file below, we want to keep only those records with a transaction code of 'A' (add) and discard those with a transaction code of 'D' (delete):

Single_at.txt

```
xaction   given
   id   _name  surname  dept start_date
------------------------------------
A 0013 Michael Card      12 23May2001
A 0028 John    MacArthur 15 20Apr2002
D 0002
D 0004
A 0014 Ravi    Zacharias 15 11Dec2001
A 0016 James   Dobson    16 03Jan2002
A 0018 RC      Sproul    AA 29Feb2000
D 0005
A 0020 Michael Horton    11 23Apr2003
```

As the following code illustrates, we could read in the entire record, allow SAS to parse the input line appropriately, applying the specified (or default) informats and then output only the records we really want. If our input file is comprised of 6 fields and 9

records it probably wouldn't matter from an efficiency standpoint.. However, if our input dataset contains 500 variables and is 4 gig in size …. efficiency matters!

```
data adds ( drop = xaction );
  infile  'C:\stratia\sesug\data\single_at.txt' pad missover;

  input @1  xaction     $1.
        @3  id          $4.
        @7  given_name  $10.
        @17 surname     $10.
        @27 dept
        @31 start_date  date9. ;

  if xaction eq 'A';  /*  Keep only additions  */
run;
```

To make this process more efficient, we must be able to read the **xaction** field first, interrogate the value and *only then* input the rest of the line if required. In order to input only part of a line and retain the rest of the line for future input, we must use the trailing @ to *hold the input pointer on the current data line*. Note the @ sign at the *end* of the third line of code:

```
data adds ( drop = xaction );
  infile  'C:\stratia\sesug\data\single_at.txt' ;

  input @1  xaction     $1. @;  /*  Trailing @  */

  if xaction eq 'A';  /*  Keep only additions  */

  input @3  id          $4.
        @7  given_name  $10.
        @17 surname     $10.
        @27 dept
        @31 start_date  date9. ;
run;
```

What happens if the value of xaction is not equal to 'A'? The observation is not included in the output dataset and the automatic iteration of the data step "resets" the trailing @.

2. Double Trailing @

We've just said the trailing @ is forgotten on the next iteration of the data step. What happens if our input data has more than one observation per line of data or if the data for one observation spans more than one input line?

double_at.txt

```
A 0013 Michael Card 12 23May2001 A 0028 John MacArthur
15 20Apr2002 D 0002 D 0004
A 0014 Ravi Zacharias 15 11Dec2001
A 0016 James Dobson 16 03Jan2002
A 0018 RC Sproul AA 29Feb2000
D 0005
A 0020 Michael Horton 11 23Apr2003
```

We could write some messy looping logic to handle data of this "structure" and deal with the LOST CARD message. Or, we can save ourselves a great deal of grief and let SAS figure it out. Double trailing @ signs instruct SAS to hold the input line pointer *even across* data step iterations. Since SAS manages the looping, there's no LOST CARD message and no confusion. Note the @@ on the `/*  Hang on!!  */` lines.

```
data adds    ( drop = xaction )
     deletes ( keep = id );

  length xaction     $1
```

```
            given_name
               surname $10
                    id  $4
                  dept   4
            start_date   8
    ;

    infile  'C:\stratia\sesug\data\double_at.txt' ;

    input xaction
       id       @@;                /*  Hang on!!  */

    if xaction eq 'D' then
        output deletes;
    else do;
        input given_name
              surname
              dept
               start_date date9. @@;  /*  Hang on!!  */
        output adds;
    end;
run;
```

From the log and output shown below, it's apparent that though our input data had only 7 records, SAS dealt with the misshapen data and created 9 observations in 2 different datasets in one pass.

Log:

```
NOTE: 7 records were read from the infile 'C:\stratia\sesug\data\double_at.txt'.
      The minimum record length was 6.
      The maximum record length was 54.
NOTE: SAS went to a new line when INPUT statement reached past the end of a line.
NOTE: The data set WORK.ADDS has 6 observations and 5 variables.
NOTE: The data set WORK.DELETES has 3 observations and 1 variables.
```

Output:

Delete Transactions

id
0002
0004
0005

Add Transactions

given_name	surname	id	dept	start_date
Michael	Card	0013	12	2001/05/23
John	MacArthur	0028	15	2002/04/20
Ravi	Zacharias	0014	15	2001/12/11
James	Dobson	0016	16	2002/01/03
RC	Sproul	0018	.	2000/02/29
Michael	Horton	0020	11	2003/04/23

3. Question Mark ?

Based upon the informat specified (or defaulted), SAS converts the input data to an internal format during input. If the field is defined as numeric, non-numeric data will cause an error. Dates must also be in the format specified (better automatic date parsing functionality available in version 9).

If the data value is not in the correct format, a number of things happen:

- the field value is set to missing (or the value defined in the INVALIDDATA system option)
- **NOTE: Invalid data message …** is raised, giving the field name, input record number and the offending columns
- the input record is dumped to the log
- the automatic _ERROR_ variable is set to 1
- this sequence is repeated each time bad data is found until the system option ERRORS= limit is met.

The complete log for the previous example showed that the 5th data line had an invalid numeric dept value, 'AA', hence the missing value in the "**RC Sproul**" observation.

```
NOTE: Invalid data for dept in line 5 18-19.
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6
5         A 0018 RC Sproul AA 29Feb2000 29
xaction=A given_name=RC surname=Sproul id=0018 dept=. start_date=14669 _ERROR_=1 _N_=7
```

The default behavior can be problematic if you *know* the source data for certain fields are consistently bad. If I root through my log looking for *unexpected* invalid data, the many messages for the consistently bad fields may result in a genuine problem being missed. Set the system option ERRORS to zero, that'll suppress the messages! NO!! That'll result in *legitimate* input errors being missed. In a perfect world it may be possible to simply push the file back to the originating department and have them recreate pristine data. But what about those of us who live in more realistic situations?

SAS permits the insertion of a question mark between the field name and field informat. This allows the programmer to be pro-active and acknowledge that a field value is often of the wrong type and suppress the "**NOTE: Invalid data for …**" message for *only* the designated field. Note the line flagged with `/* ? often invalid data */` comment.

```
data adds ( drop = xaction );
  infile  'C:\stratia\sesug\data\single_at.txt' pad missover;

  input @1  xaction     $1.
        @3  id          $4.
        @7  given_name  $10.
        @17 surname     $10.
        @27 dept      ?       /*  ?  often invalid data  */
        @31 start_date  date9. ;

  if xaction eq 'A';  /*  Keep only additions  */
run;
```

Log – note the absence of the invalid data message.

```
NOTE: The infile 'C:\stratia\sesug\data\single_at.txt' is:
      File Name=C:\stratia\sesug\data\single_at.txt,
      RECFM=V,LRECL=60

RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6
7         A 0018 RC        Sproul    AA 29Feb2000
xaction=A id=0018 given_name=RC surname=Sproul dept=. start_date=14669 _ERROR_=1 _N_=7
```

Note as well that the input record is still dumped and the _ERROR_ variable is set.

4. Double Question Mark ??

In a real world situation, the raw data input phase might be the first step in a longer, production process which runs overnight in a batch environment. If real errors occur in the batch process, a home-grown SAS macro job processing mechanism will not allow the process to continue past the step in which the error occurred. In datasteps, the non-zero value of _ERROR_ will cause the job to stop. This scenario poses a problem for this input data since the value of the **dept** field is often invalid.

Well, there must be a solution, or else it wouldn't be in this paper!! SAS allows the specification of **??** (double question marks) in the INPUT statement to suppress the log notes and setting of _ERROR_. Note the **??** on the **dept** input.

```
%let error = 0;

data adds ( drop = xaction );
  infile  'C:\stratia\sesug\data\single_at.txt' pad missover lrecl=60;

  input @1  xaction     $1.
        @3  id          $4.
```

```
        @7  given_name  $10.
        @17 surname     $10.
        @27 dept      ??       /*  ??  often garbage, not an error!!  */
        @31 start_date  date9. ;

  if xaction eq 'A';  /*  Keep only additions          */

  if _error_ then     /*  Records the real errors      */
      call symput('rc',put(_error_,1.));
run;

%put Data adds step error switch is &rc;
```

Log – note absence of log errors and value of macro variable &error on datastep completion.

```
NOTE: The infile 'C:\stratia\sesug\data\single_at.txt' is:
      File Name=C:\stratia\sesug\data\single_at.txt,
      RECFM=V,LRECL=60

NOTE: 9 records were read from the infile 'C:\stratia\sesug\data\single_at.txt'.
      The minimum record length was 6.
      The maximum record length was 39.

Data adds step error switch is 0
```

As a bonus, both ? and ?? are valid in the INPUT **function** as well and have the same effect as noted above.

e.g.
```
input(dept,? 4.);
input(dept, ?? 4.);
```

5. Ampersand &

The ampersand does triple duty in SAS.

a) In keeping with the raw data theme, the first use to be discussed involves the INPUT statement. The default behavior for "list input" (e.g. as we saw in the double trailing @ example, fields are not fixed width and do not begin in specific columns) is to use a single space delimiter when inputting values. In the event that the data contains spaces between words, for example, a person's full name, the **&** format modifier must be used. SAS expects that fields will then be delimited by at least 2 spaces.

```
data list;
   length fullname $20;
   /*  specify & to get entire name */
   input fullname $ & age;
cards;
Josephus     45
Paul Martin   67
Ron Smith  14
run;
```

```
Format Modifier &
Obs     fullname      age

 1     Josephus        45
 2     Paul Martin     67
 3     Ron Smith       14
```

b) Second, ampersands are most often used in macro variable references. At the risk of gross over-simplification, macro variables contain text generated by a prior process or definition to be used by a subsequent process. In the code example from **4. Double Question Mark**, the macro variable &rc was created by the datastep which processed the input file. Subsequent macro code or data steps could interrogate the value in the &rc macro variable and react appropriately.

In an environment where directory or dataset names change regularly, it's a very tedious and error-prone process to ensure the dynamic information is changed correctly, particularly in a long program. It's counter productive to be enslaved to your programs (http://www.prowerk.com/prowerk_V3/presentations/Sugi27_P65.zip), there's more interesting and valuable things to do with your time! Deliverance may be as simple as defining macro variables to hold the dynamic information at the top of your program and using the macro variables references wherever the value is needed throughout the program.

```
%let month = 200309;
```

```
. . . many lines later

filename data "f:\projects\campaigns\&month\data";
```

Note that the string is wrapped in double quotes. If single quotes had been used, the macro variable would not be resolved.

While the SAS Macro Processing / Compile process is quite complicated, for the purposes of this discussion it may be distilled to the following: macro variables are (hopefully) resolved before the statements that reference them are compiled. It is helpful to remember that SAS macro is really only text generation or text replacement. When the macro processor runs, it attempts to "resolve" all the macro and macro variable references it finds. The resolved values replace the original reference and the SAS code is compiled. In the filename statement above, the SAS compiler will see:

```
filename data "f:\projects\campaigns\200309\data";
```

Ampersand = freedom from the world of error and the mundane!!

c) Third, where an ampersand is *not followed by non-blank* character, it is used as a Boolean "AND"

```
if amt > 1000 & date > "20Feb2003"d then ..
```

6. Underscore _

There's absolutely nothing magical about the underscore, but it's the only special character allowed in a variable name. For that reason, it's a natural character to use when creating *meaningful* variable names. In the not so distant past, SAS variables were limited to only 8 characters. This limitation led to very cryptic variable names:

cynslspc = cynetsls / pynetsls;

With the increase to 32 character variable names and the use of the underscore, understandable variable names became possible:

curr_yr_net_sales_pct = curr_yr_net_sales / prev_yr_net_sales

Use long names!! It's a great aid for those who follow you and provides a modicum of documentation. Write readable code!!

The data step generates automatic variables as well. These variables are created by SAS and there's nothing you can do about it except use them gratefully. Automatic variables are automatically dropped and will not appear in the output dataset. Examples:

_n_	data step iterations, often useful as a counter
error	error flag
infile	variable that contains the *entire* INPUT buffer, not just the variables INPUT thus far
iorc	I/O return code, set during modify / update operations test using %sysrc macro and SAS supplied mnemonics (all of which begin with an underscore as well, e.g. _sok, _dsenmr etc..)

Underscores also frame useful, special words. (also useful in PROCs) e.g.

all	all variables in the datastep	put _all_;
character	all character variables defined to data step thus far	
numeric	all numeric characters defined to data step thus far	

7. Colon :

The subtle colon is another multi-purpose character. You may have to peer closely to distinguish this little gem from the ubiquitous semi-colon.

a) The colon is most widely used as a wildcard character. Where field names have a common prefix, they may be referenced collectively using the colon character.

Suppose a dataset contains 12 months of purchase data, number and dollar amounts of purchases, by month. The fields are named:

```
num_purch01, num_purch02, num_purch03 … num_purch12 and
amt_purch01, amt_purch02, amt_purch03 … amt_purch12.
```

To refer to the purchase amt fields collectively and reference all 12 variables:

```
amt_purch:
```

If the dataset really did contain only the 24 fields identified above, even `a:` would suffice, grabbing all fields starting with the letter 'a'. That'll save a bunch of typing!

For example:

```
proc print data = datalib.purchase_data ;
     var amt_purch:;
run;

data yearly_sums ( drop = amt_purch: num_purch: ) ;
    set datalib.purchase_data;

    yr_amt_purch = sum( of amt_purch:  );
    yr_num_purch = sum( of num_purch:  );
run;
```

Here's a handy tip picked up from Ian Whitlock of SAS-L fame. Ensure all the "working" variables in your data step begin with the underscore. For example, DO loop counters, intermediate result fields etc.. Rather than having to spell out a long keep or drop statement to ensure only the good stuff is retained, use the colon modifier in the DROP option on the DATA statement to discard all variables beginning with underscore.

```
data datalib.my_new_data_set ( drop = _: );

    . . . .
```

Simple, effective, smiley.

b) A second popular use of the colon is as a wildcard in character *value* comparisons. To extract all employees with a surname beginning with "SMIT" we could code an IF statement or WHERE clause with the SUBSTR function:

```
where substr(surname,1,4) = 'SMIT';
```

Or, the colon could be used to denote "begins with":

```
where surname =: 'SMIT';
```

Same results, less code, more efficiency. Crude benchmarking on my struggling laptop showed a 50% execution time savings using the colon wildcard over the SUBSTR method. In a similar manner, other comparison operators can take advantage of the same shortcut:

```
if surname >: 'SMIT';
if description in: ('Hi','Zoo','Comp');
if name =: pattern;  /*  length of pattern must be < length of name  */
```

Note: the colon wildcard will <u>not</u> work in SQL. Another set of operators must be used to achieve the same effect, see EQT.

c) Third, the colon may be used when inputting list data (extracting data from a non-blank column to the next blank column) that requires an informat. Normally the informat length would prevail and cause all sorts of problems if the field value was shorter than the specified length. Smith's age was included with his name and Jones was skipped entirely when SAS went to a new line in an effort to find Smith's age.

```
data age;
   input name $10. age ;
datalines;
smith 49
jones 67
droogendyk 29
```

```
Colon Format Modifier - Informat
Obs       name       age

 1     smith 49        .
 2     droogendyk      29
```

```
;run;
```

After the addition of the colon modifier, input data is read correctly:

```
input name : $10. age ;
```

```
Colon Format Modifier - Colon

Obs    name          age

 1     smith          49
 2     jones          67
 3     droogendyk     29
```

8. Dash single - or double --

The single or double dash character is used to denote a range of variables by specifying the first and last variable name in the range desired. Before the dash will make any sense, one must have some understanding of SAS's loading of the PDV (program data vector). As the compiler chews through the program, top down, variables are added to the PDV in the order in which they are *encountered* in the data step. In the example below, the order of variables in the PDV will be:

amt1 num1 amt2 num2 amt3 num3 single double

```
data _null_;
    amt1 = 100;  num1 = 10;
    amt2 = 200;  num2 = 20;
    amt3 = 300;  num3 = 30;
    single = sum( of amt1 - amt3 );
    double = sum( of amt1 -- amt3 );
    put single=;
    put double=;
run;
```

a) The single dash is a *numbered range list* and requires that the alphabetic prefixes of the start and end variable *be the same* and that the numeric portion of the first and last variable refer to a *consecutive* range of numbers.

```
single = sum( of amt1 - amt3 );
```

is the same as

```
single = sum( amt1, amt2, amt3 );
```

If the amt2 field did *not* exist, `'of amt1 – amt3'` would result in an error.

b) The double dash refers to a *named range list* that selects the variables based on their position in the PDV.

```
double = sum( of amt1 -- amt3 );
```

is the same as

```
double = sum( amt1, num1, amt2, num2, amt3 );
```

The double dash may be restricted to only numeric or character variables within the specified PDV range:

```
sum_nums = sum( lo_numeric_field –numeric- hi_numeric_field );

array chars{*}  lo_character_field –character- hi_character_field  );
```

c) There's one more very handy use of the dash character. Numeric formats right-justify and character formats left-justify results by default. To change the default behavior, follow the format specification with a dash and one of the justification characters, l, c, r:

```
put amt comma10.2-c;      /*  Center amount field  */
```

**9. Asterisk *

We've seen a couple of short cuts that will allow us to very quickly define a list of variables *without* knowing how many variables might be returned by the list, the wildcard colon and the dash(es). Suppose we want to define an array using a variable list or wildcard, but do not want to hard-code the expected number of variables. If we do specify a number and we're wrong, i.e. the array size is not equal to the number of variables, the data step will not compile.

Instead of living on the edge, the SAS documentation suggests the use of an asterisk to indicate that the array size is to be defined according to the number of variables indicated by the list or shortcut:

```
array all_nums{*} _numeric_;
array amts{*}      amt:     ;
array counts{*}    count1 - count6;
```

In actual fact, if the {*} is omitted from each of those lines, v8.2 code will compile and run exactly the same. Do you then jettison the hieroglyphics? I vote for keeping the asterisk around so that it's more apparent that the array size definition is dynamic, another of those subtle self-documenting occasions.

The data step asterisk is helpful in other contexts. Use {*} to refer to all elements of an array in PUT, INPUT and applicable SAS functions, e.g:

```
input amts{*} ;
sum(of amts{*});
```

We're familiar with the asterisk and its use as the multiplier operator in mathematical statements. In somewhat the same fashion, the asterisk is used to *multiply* definitions or values. Arrays can be initialized at definition time:

```
array total_amts{12} (12*0);    /*  Fill all 12 buckets with nuthin' */
```

Another *multiplier* function is available as well, this time as part of the INPUT statement. If a series of input fields share the same informat, the following syntax applies the informats appropriately:

```
input (amt_purch01 - amt_purch12 ) ( 9*10.2 3*12.2 );
```

10. Parenthesis ( )

The parenthesis are a handy "container" for all sorts of things in the dataset.

Function arguments:

```
sum(jan_sales, feb_sales, mar_sales );
```

Data set options appear in parenthesis as well. There are many occasions when programmers write code to deal with stuff that could be more efficiently dealt with via a dataset option.

Do you know the data are in sorted order? Use the dataset option and tell SAS about it to avoid unnecessary sorts in a subsequent process:

```
data datalib.account_data ( drop = _: sortedby=account );
```

Create an index on the fly rather than executing an additional step to do so (note, your mileage may vary with this technique). DROP variables on the SET statement to avoid bringing them in just to throw them out.

```
data test ( keep   =   account amt: num01 -- num12
            index  = ( account ) );

   set dataset (  drop  = old:
                  rename = ( account_num = account )
                  where  = ( account_status = 'O' ) );
   . . .
```

Conditional statements:

Are you still using massive if – then – else statements ? Try the select with its parenthesis!

```
select (rectype);
    when (1)      output header;
    when (2)      output detail;
    otherwise     delete;
end;
```

UNTIL / WHILE clauses:

```
do until(last.account);
    set test;
     by account;
end;
```

Conclusion

The next time you're confronted with a programming challenge, explore a little, search the SAS-L archives, peruse the Online Docs, dive into the toolbox and see if some of the lesser-known SAS hieroglyphics will help you solve the problem quickly and efficiently.

There's no reason to ignore the {*&%~(.:@|? special characters, they'll likely make you and your code more productive!

References

SAS Institute Inc.
SAS OnlineDoc®, Version 8
February 2000
Copyright ©2000, SAS Institute Inc.

A zillion SAS-L posts over the years.

Trademark Information

Acknowledgements

SAS-L, thanks for teaching me SAS. Prowerk Consulting LLC (www.prowerk.com) has my gratitude for giving me opportunity to use my SAS skills, especially over the last few years. Thank you to Marje Fecht, who floated the idea of a paper, then cajoled me into accepting the invitation, and then read it a number of times, offering helpful suggestions and ideas. Thanks as well to proof-readers Laura House and Uncle Bill Droogendyk.

Author

The author welcomes comments, suggestions, questions, corrections (!!) by email at harry@stratia.ca

Harry Droogendyk
PO Box 145,
Lynden, ON
L0R 1T0

www.stratia.ca