Paper 190-2008

Using SAS® to Parse External Data

Andrew T. Kuligowski, The Nielsen Company

ABSTRACT

Most “Introduction to Programming” courses will include a section on reading external data; the first assumption they make will be that the data are stored in some sort of documented and consistent format. Fortunately, in the “real world”, a lot of the data we deal with has the same basic assumption of occurring in a documented, consistent format – a *lot* of it, but not *all* of it.

This presentation will address some techniques that can be used when we are not dealing with cleanly formatted data, when the data we want is in a less-than-ideal format, perhaps intermingled or seemingly buried with unnecessary clutter. It will discuss the principles of using the SAS System to parse a file to extract useful data from a normally unusable source. This will be accomplished by citing examples of unusual data sources and the SAS Code used to parse it

PARSERS - A PRIMER

The word *parser* will normally cause a computer-minded individual to think of a compiler or interpreter. Both must include a parser, which determines the syntactic structure of a string of characters coded in a high-level language. However, while correct, this is too specific a definition for our purposes. Let us use a more generic (read: non-computer specific) definition of ***parse*** as ***the analysis of a string of characters and subsequent breakdown into a group of components.***

To illustrate this definition, let us cite an example which will be familiar to many SAS users. The Copyright statement which appears at the beginning of a SAS Log contains a character string which is unlikely to be used elsewhere in a routine: "Cary". When using an on-line editor to browse a listing which contains a SAS Log following some “wrapper” information from a calling routine, searching for the word "Cary" should bring the user to the start of the SAS Log. Similarly, finding the next (or last) occurrence of the word should take us to the start of our actual listing – assuming the string is not used within the routine itself! [See Figure 1 for an example from Z/OS MVS.] This basic example shows how a unique character string can be used to identify and isolate a specific section of a text file for further processing.

```
BROWSE ----- USERID.SASRUN.LISTING  ---------- CHARS ' Cary' FOUND
COMMAND ===>                                     SCROLL ===> CSR


 NOTE: Copyright (c) 2002-2003 by SAS Institute Inc., Cary, NC, USA.
 NOTE: SAS (r) 9.1 (TS1M3)
       Licensed to COMPANY NAME, Site SITENUM.
 NOTE: This session is executing on the z/OS   V01R07M00 platform.

                                  …      …      …
```

Figure 1A - Searching for the word "Cary" at the top of a SASLOG

```
BROWSE ----- USERID.SASRUN.LISTING  ---------- CHARS ' Cary' FOUND
COMMAND ===>                                     SCROLL ===> CSR

 NOTE: The address space has used a maximum of 620K below the line and
 NOTE: SAS Institute Inc., SAS Campus Drive, Cary, NC USA 27513-2414
1                                                 The SAS System

                              Obs     Index     Random1     Random2
                                1         1     0.79940          90
                                2         2     0.58974          53
                               …       …       …
```

Figure 1B - Searching for the word "Cary" at the bottom of a SASLOG

Please note that the examples used in this paper were taken from both the MVS and Windows environments. The concepts illustrated are independent of platform, and can be applied to any platform on which the SAS System currently resides.

IDENTIFYING UNIQUE CHARACTER STRINGS

The most important aspect of writing a data parser is the analysis performed by the humans making the request and writing the routine. They must determine exactly what information in the file to be processed should be considered useful, and what is to be categorized as noise. "Noise" can be rejected, while "useful" data falls into two categories:

- data to be kept and subsequently processed, and
- identifiers that help to determine the difference between data and noise.

In many situations, the identifiers are easy to recognize – for the human(s) analyzing the problem, and for the routine they end up creating. In others, the process of determining what will identify useful data can be quite complex. Further, once a rule is defined, it is often punctuated with exceptions that must be dealt with via extra code.

Let us re-examine the "Cary" example cited above to clarify this point. In our earlier example, the search could be handled simply by passing in the string "Cary" – a find against this string would locate the address for SAS as printed at the top and bottom of the SASLOG. This assumes that the word does not occur anywhere else in the output under examination, which is not always accurate – "Cary" could be used in a comment string, or as a variable name, or within the output data, or virtually anywhere. In those cases, it might be necessary to expand the string and look for "Cary, NC", or even "Cary, NC, USA". (The latter version of the expanded string introduces yet another problem – the precise string is not universal. The version of this string at the top of the SASLOG has a comma and space between "NC" and "USA", while the one at the end has a space but no comma. This is yet another challenge to be dealt with when determining the appropriate identifiers.)

INPUT STATEMENT: NAMED INPUT

The first technique to be discussed is one that has come "bundled" in base SAS for years – **Named Input**. Named Input is one of the four types of input described in the SAS documentation, and is probably the most rarely used of those four. With Named Input, the data to be read must be in the form *fieldname=value*. (An aside: SAS documentation used to state that this technique could not be combined with other input styles – list, formatted, and column – in the same INPUT statement. That has been proven false and has been corrected; other styles can be used, providing they occur before the first instance of Named Input. Once the first *fieldname=* is encountered on the INPUT statement / input data, all other variables on the statement must follow that same *fieldname=value* pattern.)

Named Input can be used as the basis of a data parser in that it specifically examines a line of input for a given string, and then uses that string to read in a requested piece of data. (See Figure 2 for a simple example of Named Input.)

```
DATA Example2;
   INPUT  RANDOM3=;
   DATALINES4;
RANDOM3=94
RANDOM3=334
RANDOM3=966
RANDOM3=809
RANDOM3=934
RANDOM3=889
;;;;
proc print uniform; run;
```

→

```
        The SAS System

        Obs     RANDOM3
        1         94
        2        334
        3        966
        4        809
        5        934
        6        889
```

Figure 2 – Simple example of Named Input

As an aside, it is worth mentioning that Named Input has one big advantage over other forms of input – the variables on the INPUT statement do not need to be in the same order as they occur in the actual data. This can result in an entire line being processed

It should also be noted that variables do not necessarily even have to be explicitly listed on the INPUT statement in order to be read in using Named Input. Any variable previously defined within the DATA Step, such as with a LENGTH or INFORMAT statement, will also be identified and processed when SAS deals with Named Input. This may be precisely what the coder wanted. However, since the traditional "field name / value" pair relationship is being augmented, it may also result in additional data being unexpectedly read and stored. The coder is advised to be aware of this when writing a DATA step using Named Input.

This "feature" has an unfortunate side effect – the inverse is also true. If the Input statement is processing Named Input and a "fieldname=value" pair is encountered that is not identified on the INPUT statement or elsewhere in the DATA step, SAS will report that absence via a NOTE: In addition, if the requested field is not present on a given input line, SAS will report that fact, as well. (Please see Figure 3 for an example of this interaction. The input data has been slightly enhanced from Figure 2; an additional value has been added to each line. In addition, a header line inserted at the top to illustrate another condition found when parsing input data – not every line may contain valid data.)

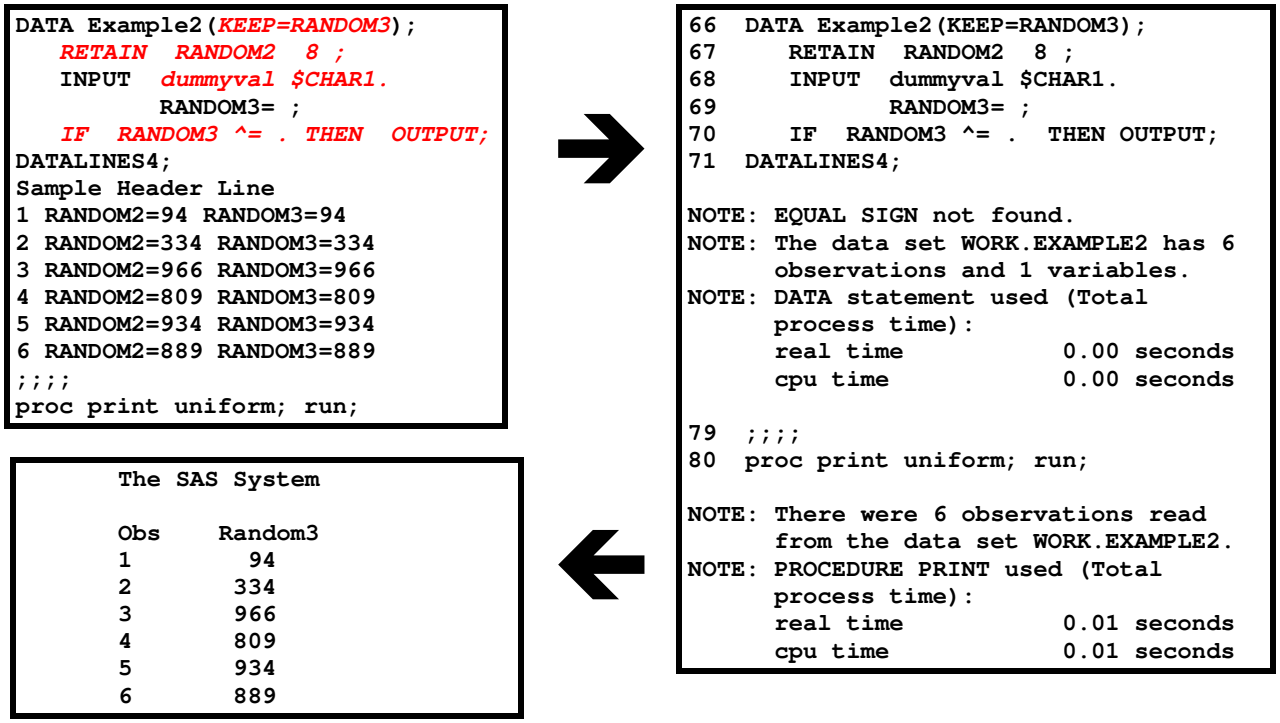
```
DATA Example2;
   INPUT  RANDOM3= ;
DATALINES4;
Sample Header Line
1 RANDOM2=94 RANDOM3=94
2 RANDOM2=334 RANDOM3=334
3 RANDOM2=966 RANDOM3=966
4 RANDOM2=809 RANDOM3=809
5 RANDOM2=934 RANDOM3=934
6 RANDOM2=889 RANDOM3=889
;;;;
proc print uniform; run;
```

```
47  DATA Example2;
48     INPUT  RANDOM3= ;
49  DATALINES4;

NOTE: EQUAL SIGN not found.
NOTE: NAME, '1 RANDOM2' is not defined.
RULE:      ----+----1----+----2----+---
951        1 RANDOM2=94 RANDOM3=94
RANDOM3=94 _ERROR_=1 _N_=2
NOTE: NAME, '2 RANDOM2' is not defined.
952        2 RANDOM2=334 RANDOM3=334
RANDOM3=334 _ERROR_=1 _N_=3
NOTE: NAME, '3 RANDOM2' is not defined.
953        3 RANDOM2=966 RANDOM3=966
RANDOM3=966 _ERROR_=1 _N_=4
NOTE: NAME, '4 RANDOM2' is not defined.
954        4 RANDOM2=809 RANDOM3=809
RANDOM3=809 _ERROR_=1 _N_=5
NOTE: NAME, '5 RANDOM2' is not defined.
955        5 RANDOM2=934 RANDOM3=934
RANDOM3=934 _ERROR_=1 _N_=6
NOTE: NAME, '6 RANDOM2' is not defined.
956        6 RANDOM2=889 RANDOM3=889
RANDOM3=889 _ERROR_=1 _N_=7
NOTE: The data set WORK.EXAMPLE2 has
      7 observations and 1 variables.
                …      …
```

```
      The SAS System

      Obs     RANDOM3
      1          .
      2         94
      3        334
      4        966
      5        809
      6        934
      7        889
```

Figure 3 – Enhanced example of Named Input (with issues)

The resulting dataset will most likely be close to what is expected – except for the missing value stored in the first observation. However, the cautionary NOTEs in the SASLOG could – and this author argues that they *should* – cause some concern for the analyst upon execution. The coder could decide to simply suppress the NOTEs – but this noncreative and/or rushed approach could prevent other, unexpected NOTEs from being displayed, as well! The coder could add some processing to address these issues – perhaps an OUTPUT statement to prevent undesirable lines from affecting the output dataset, plus some allowance for the additional unspecified variable at the beginning of each input line. (The resulting code can be found in Figure 4.) However, this modification does not address the biggest drawback with using Names Input in this fashion – this approach only works if the text string is in "fieldname=value" form! It appears that a more robust mechanism to input this data would be appropriate.



```
DATA Example2(KEEP=RANDOM3);
   RETAIN  RANDOM2  8 ;
   INPUT  dummyval $CHAR1.
          RANDOM3= ;
   IF  RANDOM3 ^= . THEN  OUTPUT;
DATALINES4;
Sample Header Line
1 RANDOM2=94 RANDOM3=94
2 RANDOM2=334 RANDOM3=334
3 RANDOM2=966 RANDOM3=966
4 RANDOM2=809 RANDOM3=809
5 RANDOM2=934 RANDOM3=934
6 RANDOM2=889 RANDOM3=889
;;;;
proc print uniform; run;
```

```
66  DATA Example2(KEEP=RANDOM3);
67     RETAIN  RANDOM2  8 ;
68     INPUT  dummyval $CHAR1.
69            RANDOM3= ;
70     IF  RANDOM3 ^= .  THEN OUTPUT;
71  DATALINES4;

NOTE: EQUAL SIGN not found.
NOTE: The data set WORK.EXAMPLE2 has 6
      observations and 1 variables.
NOTE: DATA statement used (Total
      process time):
      real time            0.00 seconds
      cpu time             0.00 seconds

79  ;;;;
80  proc print uniform; run;

NOTE: There were 6 observations read
      from the data set WORK.EXAMPLE2.
NOTE: PROCEDURE PRINT used (Total
      process time):
      real time            0.01 seconds
      cpu time             0.01 seconds
```

```
        The SAS System

        Obs     Random3
        1         94
        2        334
        3        966
        4        809
        5        934
        6        889
```

Figure 4 –Enhanced example of Named Input with additional "clean-up" coding applied

INPUT STATEMENT: "@ <string>

Most coders learn how to use column pointers on INPUT statements early in their SAS coding careers. A command line such as "**INPUT @ 1 variable_name ;**" is an elementary component of Base SAS; it causes SAS to move the column pointer to position 1 of the current line and read the information that begins in that location, populating "variable_name". However, many SAS coders – including a good number of veteran SAS coders – do not realize that SAS added a parallel syntax that can be used to move the column pointer based on character strings.

"**INPUT @ "*string*" variable_name ;**" will search the current input line for the value specified by "*string*". It will move the column pointer to the position immediately to the right of the value in "string", and will begin reading the value to be stored in "variable_name" from that location forward. Please note that the command will accept either a hard coded literal quoted string or a character variable. The variable provides greater flexibility, but the examples cited in this work will use the hard coded quote for clarity.

It should be stressed that the value contained in "*string*" will not actually be read in and stored by the INPUT statement (unless the coder clearly – and often cumbersomely – causes that to occur). For example, a routine looking for 4 digit years in the current century might contain the line:

```
INPUT @ "20"  the_year ;
```

The values contained in "the_year" will contain only two digits each – remember, the column pointer was positioned to the right of the string "20" and does not actually process it. (This particular piece of code may also lead to confusion since the string "20" may appear to be a valid numeric value. The quotation marks around "20" show that the code is clearly not positioning the column pointer at position 20, but that might not be clear if the code is only given a quick cursory glance. A well-placed comment may assist those who have to test, debug, and maintain this particular code – and the reader is reminded the person responsible for testing and subsequent maintenance is often the original author of the routine!)

The example specified in Figure 3 (bypassing the enhancements added for Figure 4) can be modified to take advantage of the INPUT statement's ability to position the pointer with a character value. This is a trivial change to type in, as demonstrated in Figure 5 below. The "fieldname=" will be preceded by an "@" to denote a line pointer, the string we are searching out will be surrounded by quotation marks, and a new

reference to the variable to be process must be inserted. (In our earlier Named Input example, "Random3" served both as the string to be located and the variable to be populated.)

```
DATA Example2;
   INPUT  @ "RANDOM3=" Random3 ;
DATALINES4;
Sample Header Line
1 RANDOM2=94 RANDOM3=94
2 RANDOM2=334 RANDOM3=334
3 RANDOM2=966 RANDOM3=966
4 RANDOM2=809 RANDOM3=809
5 RANDOM2=934 RANDOM3=934
6 RANDOM2=889 RANDOM3=889
;;;;
proc print uniform; run;
```

```
66   DATA Example2;
67      INPUT  @ "RANDOM3=" Random3 ;
68   DATALINES4;

NOTE: SAS went to a new line when INPUT
      @'CHARACTER_STRING' scanned past
      the end of a line.
NOTE: The data set WORK.EXAMPLE2 has
      6 observations and 1 variables.
NOTE: DATA statement used (Total
      process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

76   ;;;;
77   proc print uniform; run;

NOTE: There were 6 observations read
      from the data set WORK.EXAMPLE2.
NOTE: PROCEDURE PRINT used (Total
      process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

```
       The SAS System

       Obs     Random3
       1         94
       2        334
       3        966
       4        809
       5        934
       6        889
```

Figure 5 – Example of Input statement : column pointer using character value

The reader will note the offending error messages that were encountered using Named Input have been eliminated with this method. However, a new message has replaced them, advising that the INPUT statement moved to a new line to look for the string. Again, this message may be acceptable to the coder who wrote the code, the testers who validated the code, and the analysts who run their code. If so, no further modifications would be needed and the job could be considered complete. However, it is likely that one or more of those individuals – or those above them in the hierarchy – would be bothered by the presence of any NOTE over and above the ones specifying dataset name, size, and execution time! In that case, yet more robust code would be required.

USEFUL CHARACTER FUNCTIONS

Long-time SAS users are most likely familiar with INDEX, a common function that is used to explore character strings. **INDEX(*source-string*, *search-string*)** will return a number denoting the first occurrence of a search string within a source string, or a 0 if the search string is not present. This function can be used as the basis for logic to augment the INPUT @ "string" logic discussed earlier. Figure 6 shows how the code in Figure 5 could be slightly enhanced with the INDEX function to eliminate the extra NOTEs in the SASLOG.

```
DATA Example2C;
  INPUT @ ;
  IF INDEX(_INFILE_,"RANDOM3=")
        > 0  THEN DO;
    INPUT @ "RANDOM3=" Random3 @;
    OUTPUT ;
  END;
DATALINES4;
    … <data inserted here> …
;;;;
proc print uniform; run;
```

```
90 DATA Example2C;
91   INPUT @ ;
92   IF INDEX(_INFILE_,"RANDOM3=")
93         > 0  THEN DO;
94     INPUT @ "RANDOM3=" Random3 @;
95     OUTPUT ;
96   END;
97 DATALINES4;

NOTE: The data set WORK.EXAMPLE2C has
      6 observations and 1 variables.
```

Figure 6 – Using Character Function INDEX to enhance character search

Notice that the extraneous NOTEs are no longer present. (The output from PROC PRINT was omitted intentionally – the reader is probably sick of seeing that 6 line PROC PRINT listing by this point!)

Before moving on, the _INFILE_ variable used in the example should be discussed. This is a special automatic (internal only, not written to the output dataset) SAS variable that contains the line most recently read from the input file. You can obtain a similar result by specifying the **_INFILE_=**<variable> option on the INFILE statement. It is a useful device to process the contents of the current input line without having to know or validate anything specific about the contents of that line.

The INDEX function cited is a valuable tool in this case, but it may not be sufficient in other examples. The function returns the specific position in the search string where the *first* occurrence of the target string is located. What if the target being sought is contained multiple times within a string? Or what if there are two targets – the code only looking for the second target if the first is found? Older SAS code handled this type of problem using DO loops. Nowadays, the coder is encouraged to expand their horizon and look at the newer SAS character functions; in this case, the **FIND()** function. Like INDEX, FIND will search for a given target string within a source string. However, FIND can start at any position within the source string, and can search forwards or backwards within that source string!!

Veteran coders who have not looked into the changes and enhancements to Base SAS are missing out on a number of useful updates, including many functions that can be used to handle character strings. Let us list out just a few of them:

- **ANYALPHA(*source-string <, startpos>*)** Returns the position of the first *alphabetic* character in the source string (by default, starting at position one, but that can be overridden), or 0 if there are no alphabetic characters in the source string.
- **ANYXDIGIT(*source-string <, startpos>*)** Returns the position of the first valid *hexadecimal numeric* character in the source string (by default, starting at position one, but that can be overridden), or 0 if there are no hexadecimal characters in the source string.
- To conserve space in this presentation, the reader is encouraged to refer to the online SAS Language Reference: Dictionary manual to explore similar functions, such as **ANYALNUM**, **ANYDIGIT**, **ANYUPPER**, and others – there are 13 total functions in the "ANY" family defined under SAS 9.1.3!
- In addition, each of the 13 "ANY" functions has a mirror image "NOT" function. For example, **NOTALPHA** returns the position of the first *non-alphabetic* character in the source string

Of course, there are several other functions – let's maintain terminology and call them "veteran functions" since they have been a part of the language for decades – to deal with character values. These may prove useful when parsing lines of input data, and can also be a valuable part of the parser's toolkit:

- **SUBSTR(*source-string, startpos <, length>*)** returns the portion of the source string starting with the position specified by the second parameter "startpos" for a length of "length" characters. If length is not specified, then the entire remaining portion of the source string is used.
- **VERIFY(*source-string, search-string*)** is sort of (but not quite) the mirror image of INDEX. VERIFY scans the source string and returns the position of the first character that is not present in the search string, or 0 if all characters are present.
- **SCAN(*source-string, word-num<, delimiters>*)** separates the source string into words, as defined by the default (or overridden) delimiters, and returns the Nth word as specified by "word-num".
- **COMPARE(*string1, string2<, modifiers>*)** returns a 0 if the two strings match, or the position of the 1st character where the strings differ. Optional modifiers can be used to ignore case, leading blanks, etc.

In addition, the reader may find the character handling capabilities of the Perl Regular Expressions (PRX) to be quite useful. These are outside the scope of this presentation. The reader is encouraged to check out one of the several fine works on the topic that were presented at earlier conferences; a partial list has been included in the "References / For Further Reading" section at the end of this paper.

INPUT STATEMENT: # x <buffer>

On occasion, the coder writing a data parser may find it desirable to read multiple lines from their data source. For example, the data necessary to populate a single observation may exist on multiple lines of the input file. For these situations, the INPUT statement provides the coder access to the line pointer. The coder simply follows the INPUT keyword with "**# *value***", where *value* is either a positive integer, a numeric variable assigned to a positive integer, or an expression whose result works out to be a positive integer. This is known as the **absolute line pointer**. (Figure 7 shows an example where each of these approaches is used.) As an aside, readers may be familiar with the relative line pointer, represented by the slash "/". The relative line pointer can also be used, but has the disadvantage of only allowing the INPUT statement to advance to the next line, not to return to an earlier one.

```
DATA TEMP;
   RETAIN  pnt 2 ;
   INFILE DATALINES4;
   INPUT # 1   X
         # pnt Y
         # (pnt**2 - 1) Z ;
DATALINES;
1
2
3
4
5
6
;;;
;proc print uniform; run;
```

```
                The SAS System
      Obs    pnt    X    Y    Z
       1      2     1    2    3
       2      2     4    5    6
```

Figure 7 – INPUT statement with absolute line pointer

It is not possible to discuss the absolute column pointer on the INPUT statement without also referencing the N= option on the INFILE statement. The N= option controls the number of lines that are available to the INPUT statement; the default is 1. This is not an issue when reading in sequential order, but can quickly become one if the reading process ceases to be sequential! Figure 8 contains a slight modification of the code in Figure 7 – note how a small change in the line order causes the routine to react negatively upon execution!

```
DATA TEMP;
   RETAIN  pnt 3 ;
   INFILE DATALINES4;
   INPUT # 1   X
         # pnt Y
         # (pnt-1) Z ;
DATALINES;
1
2
3
4
5
6
;;;;
proc print uniform; run;
```

```
55   DATA TEMP;
56      RETAIN  pnt 3 ;
57      INFILE DATALINES4;
58      INPUT # 1   X
59            # pnt Y
60            # (pnt-1) Z ;
61   DATALINES;
ERROR: Old line 63 wanted but SAS is
       at line 64.
       Use: INFILE N=X; , with a
       suitable value of x.
RULE:      ----+----1----+----2----+-
64         3
pnt=3 X=1 Y=3 Z=. _ERROR_=1 _N_=1
NOTE: The SAS System stopped processing
      this step because of errors.
WARNING: The data set WORK.TEMP may be
         incomplete.  When this step
         was stopped there were 0
         observations and 4 variables.
WARNING: Data set WORK.TEMP was not
         replaced because this step
         was stopped.
```

Figure 8 – INPUT statement with absolute line pointer

LOOPING / CONDITIONAL EXECUTION

The main thing that separates parsing from other types of input is the decision-making process that must occur during reads. The questioning starts with "Does this line contain valid pieces of information – directly or indirectly – or can it be rejected outright?" It could continue with "Do we want to keep the information in this line, or is it simply a marker to denote some other information that we want?" "Then, it might further branch to "How much valid information is contained in this line? Where in the line is it, and what format does it fall in?" etc. etc.

There are two types of conditional clauses that can prove useful in this effort:

- **IF *expression*– THEN / ELSE** will evaluate the results of the expression. If true, the code following the THEN loop will be executed. If not, the code following the ELSE expression will be invoked. ELSE is optional; if not present, then there will be no "special" code executed should the expression be evaluated as false, and the routine will continue unconditionally from that point. NOTE: The results of a Boolean true condition will be stored as a 1, and a false stored as 0. However, any non-zero value will be considered to be "true" during evaluation, while zero and missing values are treated as "false".
- **SELECT *<expression> / WHEN <expression / … / OTHERWISE*** can be used when dealing with several potential alternatives. (IF-THEN / ELSE IF / … ELSE can also be used.) At least one WHEN clause is required; only the first one deemed to be true will be executed. (Most of the time, the expressions are coded such that they are mutually exclusive – only one WHEN clause will resolve to true – but this is not a coding requirement. As with ELSE above, the OTHERWISE is optional. Its absence will result in no conditional code being executed should none of the expressions be resolved to be true, and the routine will continue from that point.

The default for each of these is to execute a single command. This can be extended with a **DO** command, which will cause all subsequent statements to be bundled together until terminated by an **END** command.

There are also two types of conditional loops that can be employed in a data parser:

- **DO WHILE *expression;*** will evaluate the results of the expression, and if true, execute the next set of statements until a closing END statement is encountered. The expression is evaluated before the loop is executed, so it is possible that the loop may never execute if the expression is immediately evaluated to be false.
- **DO UNTIL *expression;***, like DO WHILE, will evaluate the results of the expression, and if true, execute the next set of statements until a closing END statement is encountered. Unlike DO WHILE, the expression is evaluated at the completion of the loop (the END statement), so the loop will always execute at least once.

It is common coding practice to have code in the expression or within the loop that will potentially cause the expression to be altered to false. For example, the expression may contain an iterative that will increment a counter up to a specified limit. Failing this, it is also possible to include an IF-THEN statement within the loop that will execute a **LEAVE** statement when evaluated to be true – LEAVE will terminate the execution of the current DO / END block and leave the block. On occasion, both constructs will be used, with the loop scheduled to occur for a set number of iterations, but with the loop potentially terminating early should some second condition be judged to have occurred.

PUTTING IT ALL TOGETHER

The most challenging – and the author would argue, rewarding – aspect of a parser is putting all of the requirements and tools together to solve a given problem. Of course, not every coding trick and function will prove necessary or useful in solving any individual problem, just as a mechanic would not use every tool in his toolbox to fix each malfunctioning automobile! The experienced and creative coder will select the most appropriate methods to resolve the situation at hand, knowing that an equally talented coder might select a different subset in the same situation and produce an equally correct conclusion!

To illustrate the point, let us examine a canned example – a form letter containing an embedded table with US government-mandated nutritional information in the format standard when reporting that information on product packaging. (See Figure 9 for an annotated illustration of our input data.)

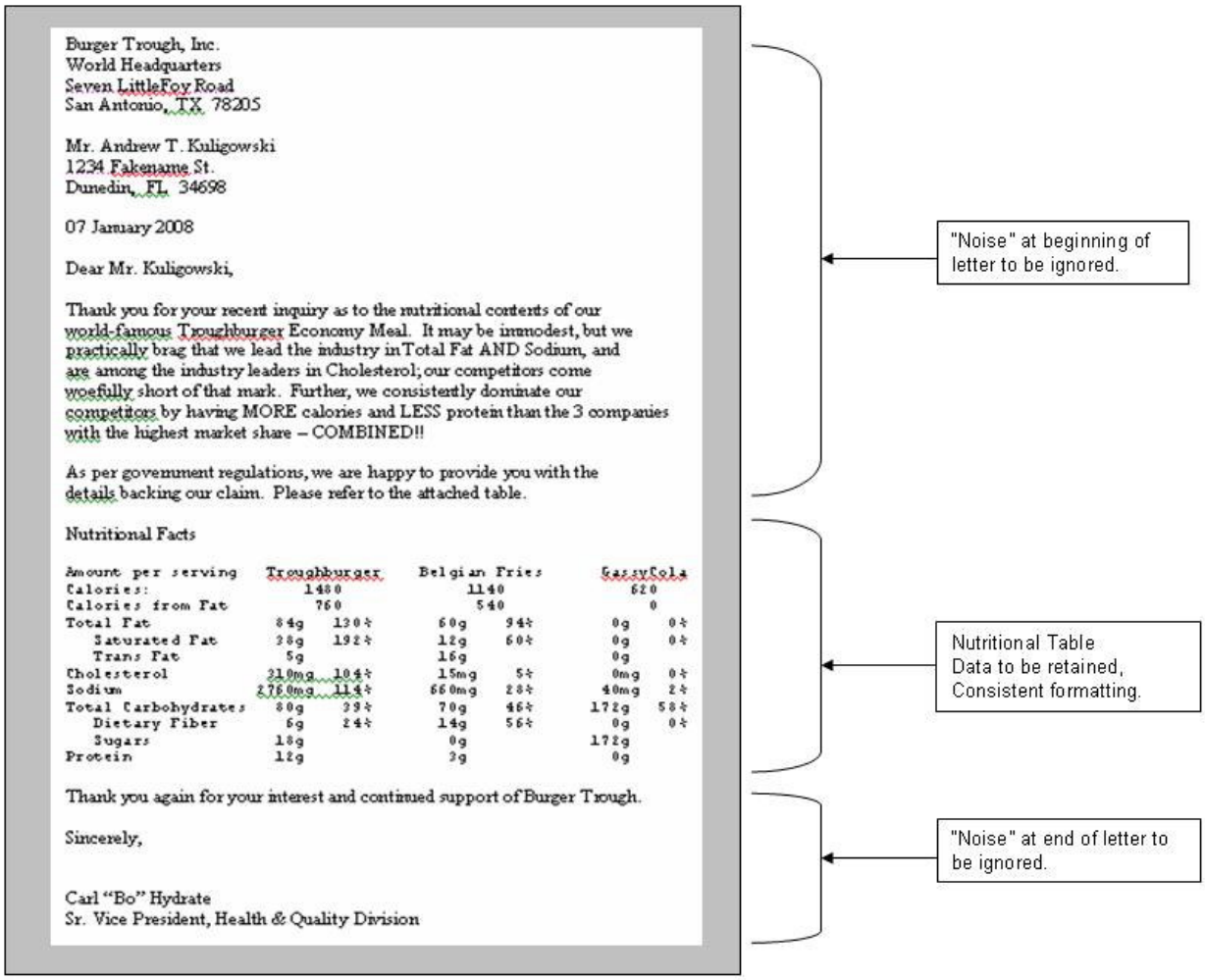


Figure 9 – Sample form letter with embedded table

Using the definitions at the beginning of this presentation, we can determine that the first half of the document and the last few lines can be considered noise, which can safely be ignored. The line "Nutritional Information" is an identifier – it is not necessary to retain this line, but it signifies the separation between noise and actual data. The blank line at the end of the table performs the inverse function – it separates the data from remaining noise. The remaining lines between those two identifiers contain the table of nutritional information that is to be processed by the routine.

An examination of the table that we have identified reveals that there is one banner line at the top, which can be parsed to obtained product names. This row is clearly identified by the "Amount per serving" label. However, there is an additional "trick" – some column headers contain a single word, while one of them contains 2 words separated by a single blank character. There are several methods that could be used to resolve this issue; the technique used in our example was to examine the line character by character. Any non-blank character is assumed to be part of a label, while the presence of multiple consecutive blanks is assumed to denote the conclusion of a label. (Single blank characters could be part of the label.) Of course, we could handle it with an INPUT statement and column pointers if the table size was fixed; however, the whole point of a parser is that this is a luxury that cannot be assumed!

The remainder of the table has a few other challenges. To start with, some components have two values in each column – one denoting the value as a number of units, and the other reflecting it as a percentage. Other items only have one value in the column. Further, the position of the number for those "one-value" components is inconsistent, with some centered and others positioned on the left side of the column. Our

routine “cheats” – it has a list of the line items that only contain one value for each food item – the “Calorie” lines – and handles those separately from the rest that are assumed to have two values each.

A further complication is that some of the values have a unit symbol following them while others do not. We handle this by checking each number for the presence of an alpha character – if one is found, we separate it from the number and store it separately, calling it “unit”. (Those numbers without units are assumed to be percentages and handled appropriately – unless their label reflects “Calories” , in which case an appropriate unit is assigned.)

The resulting code, which can be examined in Figure 10A and 10B, reflects one technique to solve this problem. (The resulting SAS dataset is displayed in Figure 11.) Please note that it is not the intention of the author to thoroughly walk through the code in this text. Rather, the prose covers the sample code at a high level; and the reader is encouraged to examine the code in detail. Certain constructs were selected for the purpose of illustrating selected points from the text; it is possible that by re-examining the original problem at a later date, an entirely different solution may emerge! Every coder’s experience and personal preferences will result in their own unique resolution to the situation. Some will be more elegant than others; some may use newer or more advanced coding techniques than others. It should be noted that our example is quite small; as such, there was no emphasis placed on efficiencies. Parsers written for larger tables, especially ones that will execute on multiple occasions, should not take efficiencies for granted. Part of testing should include an analysis on execution time; any results in this area that are deemed unacceptable should be revisited. Of course, it may prove out that the problem is so complex that the “inefficient” code could run faster than any other alternatives that are examined!

WARNING - Do You Really Need to Write a Parser?

This paper would be incomplete without a warning: There are often better alternatives to a data parser. The challenge of identifying the proper pattern(s) and satisfaction in successfully writing a routine to locate / use them can sometimes blind even the most diligent developer (or manager) from pursuing a less intensive coding approach.

Finally, do not eliminate manual intervention from consideration. In some cases, it would be quicker to have someone manually type the data into a sequential dataset and proofread it than it would be to write and test a parsing routine to automate the process. This is especially true for "one-time-only" requests; if the request is for an ongoing process, a parser is more likely to be cost justified.

WHAT WILL THE FUTURE HOLD?

This presentation was prepared prior to the official release of SAS Version 9.2, currently scheduled for First Quarter 2008. It is anticipated that there will be new options and constructs available that will provide additional options to the individual wanting to write a data parser. For example, it is well documented that the default delimiter on a non-CSV input file is a single blank ‘ ‘, and that it is possible to override that default with the **`DELIMITER=' '`** (usually shortened to **`DLM=' '`**) option on the INFILE statement. Version 9.2 will introduce a new option, **`DLMSTR='  '`**, that will permit the presence of the specified set of multiple characters to act as a delimiter!

```
DATA Example4;
  RETAIN  KeepRec_Ind 0;
  LENGTH  Product_Label  $ 80. ;   *** MANDATORY DUE TO 'DO UNTIL' stmt!  ***;
  LENGTH  NextChar       $  1. ;
  INFILE 'C:\HOW\Kuligowski\Example4.txt';
  INPUT ;
  *  Indentify start of the nutritional table.                          ;
  IF  INDEX( UPCASE(_INFILE_), 'NUTRITIONAL FACTS') > 0  THEN  DO;
    PUTLOG 'Start of Nutritional Table identified' ;
  KeepRec_Ind = 1 ;
  END;
  *  Identify body of the nutritional table.                            ;
  ELSE  IF  INDEX( UPCASE(_INFILE_), 'AMOUNT PER SERVING') > 0  THEN  DO;
    PUTLOG 'Body of Nutritional Table identified' ;
    KeepRec_Ind = 2 ;
    LengthStr = LENGTH( TRIM(_INFILE_) ) ;
    ScanPos = LENGTH('AMOUNT PER SERVING');
    DO WHILE( ScanPos < LengthStr);
      ScanPos = ScanPos + 1 ;
    NextChar = SUBSTR( _INFILE_, ScanPos, 1 );
    IF  NextChar = ' '  OR  ScanPos = LengthStr  THEN  DO;
      IF  ScanPos = LengthStr  THEN  DO;
        OneBlankInd = 1;
        Product_Label = TRIM( Product_Label ) || NextChar ;
      END;
      IF  OneBlankInd  AND  Product_Label ^= ' '  THEN  DO;
        Product_Cnt + 1;
        OUTPUT;
        OneBlankInd = 0;
        Product_Label = '';
      END;
        ELSE  OneBlankInd = 1;
      END;
      ELSE  DO;
        IF  OneBlankInd  THEN  DO;
          Product_Label = TRIM( Product_Label ) || ' ' || NextChar ;
          OneBlankInd   = 0;
        END;
        ELSE  Product_Label = TRIM( Product_Label ) || NextChar ;
      END;
    END;
  KeepRec_Ind = 3;
  END;
  *  Identify termination of the nutritional table.                     ;
  ELSE IF  KeepRec_Ind = 3  THEN  DO;
    IF  _INFILE_ = " "  THEN  DO;
      PUTLOG 'End of Nutritional Table identified' ;
    KeepRec_Ind = 0 ;
  END;
  ELSE  DO;
     Category = PUT( _INFILE_, $20.) ;
     IF  UPCASE( Category )  IN ( 'CALORIES:', 'CALORIES FROM FAT',
                '   TRANS FAT', '   SUGARS', 'PROTEIN')     THEN DO;
      ParseCnt = 3;
      ParseDivideInd = 0;
    END;
    ELSE  DO;
      ParseCnt = 6;
      ParseDivideInd = 1;
    END;
                                        …   …      …
```

Figure 10A– Sample code to parse table out of form letter (Part 1 of 2)

```
                                        …  …      …
    ParseStr = SUBSTR( _INFILE_, 21 );
    DO ParseLoop = 1 TO ParseCnt ;
      Value       = SCAN( ParseStr, ParseLoop );
      IF  ParseDivideInd  THEN  Product_Cnt = ROUND( ParseLoop / 2 );
      ELSE                      Product_Cnt = ParseLoop ;
        Unit_Start = ANYALPHA( Value );
        IF  Unit_Start  THEN  DO;
          Value_Num  = INPUT( SUBSTR( Value, 1, Unit_Start - 1 ), 10. );
          Value_Unit = SUBSTR( Value, Unit_Start );
        END;
        ELSE  DO;
          Value_Num  = INPUT( Value, 10. );
          IF  Category =: "Calorie"  THEN  Value_Unit = 'cal' ;
          ELSE                             Value_Unit = "%";
        END;
        OUTPUT;
      END;
    END;
  END;
RUN;
PROC SORT  DATA=Example4;
   BY  Product_Cnt ;
RUN;
DATA  Example4;
  LENGTH  Product_Name $ 44. ;
  RETAIN  Product_Name ' ';
  SET  Example4;
  BY  Product_Cnt  ;
  IF   FIRST.Product_Cnt  THEN  Product_Name = TRIM( Product_Label );
  IF  ^FIRST.Product_Cnt  THEN  OUTPUT;
RUN;
PROC PRINT uniform;
  var Product_Cnt   Product_Name  Category   Value  Value_Num  Value_Unit ;
RUN;
```

Figure 10B– Sample code to parse table out of form letter (Part 2 of 2)

Obs	Product_ Cnt	Product_Name	Category	Value	Value_ Num	Value_ Unit
1	1	Troughburger	Calories:	1480	1480	cal
2	1	Troughburger	Calories from Fat	760	760	cal
3	1	Troughburger	Total Fat	84g	84	g
4	1	Troughburger	Total Fat	130	130	%
5	1	Troughburger	Saturated Fat	38g	38	g
6	1	Troughburger	Saturated Fat	192	192	%
7	1	Troughburger	Trans Fat	5g	5	g
8	1	Troughburger	Cholesterol	310mg	310	mg
9	1	Troughburger	Cholesterol	104	104	%
10	1	Troughburger	Sodium	2760mg	2760	mg
			… … …			
42	3	GassyCola	Cholesterol	0mg	0	mg
43	3	GassyCola	Cholesterol	0	0	%
44	3	GassyCola	Sodium	40mg	40	mg
45	3	GassyCola	Sodium	2	2	%
46	3	GassyCola	Total Carbohydrates	172g	172	g
47	3	GassyCola	Total Carbohydrates	58	58	%
48	3	GassyCola	Dietary Fiber	0g	0	g
49	3	GassyCola	Dietary Fiber	0	0	%
50	3	GassyCola	Sugars	172g	172	g
51	3	GassyCola	Protein	0g	0	g

Figure 11– Output from Sample code to parse table out of form letter

CONCLUSION

A data parser can be an effective tool to extract useful data from a normally unusable source. The prospective author of a data parser using SAS should develop fluency with the syntax and options for the **`INFILE`** and **`INPUT`** statements. They should become well acquainted with identifying and using the various character string functions available in SAS; this is especially true for veteran SAS coders who may not have kept pace with changes and enhancements to the language. Further, coders should also be prepared to make judicious use of the **`IF-THEN/ELSE`** and **`SELECT/WHEN/OTHERWISE`** constructs and the **`DO UNTIL`** and **`DO WHILE`** statements. Additionally, before commencing on a potentially complex coding project, they should be certain that a simpler solution might not be available. However, when necessary and done properly, a data parser can be a blessing, providing availability to data which might have been thought inaccessible.

It is not possible to cover all aspects of this topic in a short paper or presentation. It is hoped that the information contained in this paper will serve to stimulate the curiosity of the reader, and that they will continue their education by researching the appropriate manuals and technical papers devoted to the specific topics discussed within this paper. Ultimately, however, it will be through real-life trial and error that true comprehension and retention of this knowledge will be attained.

REFERENCES / FOR FURTHER INFORMATION

Burlew, Michele M. (2002). *Reading External Data Files Using SAS®: Examples Handbook. Cary, NC.* SAS Institute, Inc.

Borowiak, Kenneth W.. (2007). "Perl Regular Expressions 102". *Proceedings of the SAS Global Forum 2007 Conference*.. Cary, NC: SAS Institute, Inc.

Cassels, David L. (2007). "The Basics of the PRX Functions". *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute, Inc.

Cody, Ron (2006). "An Introduction to Perl Regular Expressions in SAS 9". *Proceedings of the Thirty-First Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc
.
Cody, Ronald (2007). *Learning SAS® by Example – A Programmer's Guide*. Cary, NC: SAS Institute, Inc.

Cody, Ron (2004). *SAS® Functions by Example*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (2001). *Class Notes: Turning External Data into SAS® Data*. Dunedin, FL. Self-published.

Kuligowski, Andrew T. (2006). "DATALINES, Sequential Files, CSV, HTML and More – Using INFILE and INPUT Statements to Introduce External Data into the SAS® System". *Proceedings of the Thirty-First Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (2005). "Getting Data Into SAS®: INFILE and INPUT". *Proceedings of the Eighteenth Annual NorthEast SAS Users Group Conference*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (1994). "IDCAMS™ to SAS® – The Parser Two-Step". *Proceedings of the Second Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (1995). "Writing a Data Parser Using the SAS® System". *Proceedings of the Third Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute, Inc.

Langston, Rick (2007). "*What's Coming in Version 9.2 of Base SAS Software".* Unpublished speech presented at the Opening Session of PNWSUG 2007, 16 September 2007.

Mason, Phil. (2006). *In the Know … SAS® Tips and Techniques from Around the Globe, Second Edition*. Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1999). *SAS® Online Documentation, Version 8.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (2006). *SAS® Online Documentation, Version 9.* Cary, NC: SAS Institute, Inc.

Wikimedia Foundation, Inc. (2008). "Parsing". *http://en.wikipedia.org/wiki/Parser*.

Wikimedia Foundation, Inc. (2007). "Parsing". *http://en.wiktionary.org/wiki/parse*.

The author can be contacted via e-mail as follows:
A_Kuligowski@msn.com
Andy.Kuligowski@Nielsen.com

ACKNOWLEDGMENTS

Many people have contributed to my understanding of the topic over the years. In particular, for this presentation, Peter Eberhardt and Sue Douglass thought it might be something worth researching and documenting. In addition, there are all of those folks "behind the scenes" at SAS who helped ensure that the author's initial concept for a "hands on" workshop magically manifested itself on 300 machines at the conference. The author expresses his sincere thanks.