SAS® GLOBAL FORUM 2020 #SASGF

Paper 4230-2020

PARSING-Using SAS® When the Data Are Hiding in a Non-Standard Format

Andrew T. Kuligowski, Independent Consultant

ABSTRACT / INTRODUCTION

Sequential files? Spreadsheets? Databases? There are numerous tutorials that instruct the SAS® user in techniques to extract data from standard sources. Sometimes, however, the desired data is hidden inside a non-standard source; information may be found within the flow of a text document, for example.

This presentation will address some techniques that can be used when not dealing with cleanly formatted data, through use of an example where data are found within a free-form text file. It will deal with identifying what can be considered useful data and what can be discarded, then tackle techniques to extract the data for further analysis, reporting, or whatever is the desired result.

Please note that this paper covers many of the items covered in “Parsing Useful Data Out of Unusual Formats Using SAS”, which the author first wrote and presented approximately 10 years ago. This paper uses a new example for use as a learning tool. It has been modified to use a new approach; this paper introduces concepts and commands when they are needed to deal with the example, rather than the prior technique of providing numerous commands, and then utilizing some of them to solve the example.

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 SASLOG 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 SASLOG following some “wrapper” information from a calling routine, searching for the word "Cary" should bring the user to the start of the SASLOG. Similarly, finding the next (or last) occurrence of the word should take us to the end of the SASLOG, assuming the string is not used within the routine itself! When running batch jobs in Z/OS, the output was displayed immediately after the SASLOG; using FIND in an online editor and searching for “Cary” twice would take us past the SASLOG and to the reports. [See Figures 1 and 2 for an example.] 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 1 - 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 2 - Searching for the word "Cary" at the bottom of a SASLOG

Note that while this example was taken from an IBM mainframe, the code demonstrated in this paper has been developed and written for the Windows environments. The concepts illustrated are independent of platform and can be applied to any platform on which SAS currently resides.

IDENTIFYING UNIQUE CHARACTER STRINGS

The most important aspect of writing a routine to parse data from text 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 (typically referred to as “signal”), and
- identifiers that help to determine the difference between signal 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.

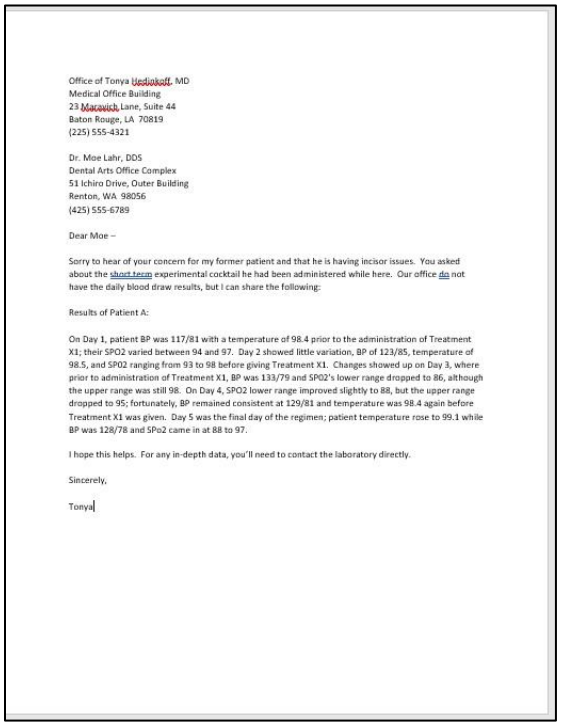
Let us re-examine the “Cary” example cited above to clarify this point. In our earlier example, the search could be handled simply by passing in the string “Cary” – a find against this string would locate the address for SAS as printed at the top and bottom of the SASLOG. This assumes that the word does 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.)

EXAMINING PARSING TECHNIQUES VIA EXAMPLE

In order to explore some of the techniques required for parsing out data from non-standard formats, we will be taking a simulated example and writing a routine to deal with the data contained within. Our example is a simulated letter between two physicians in which a patient's set of daily vital signs are exchanged. (See Figure 3.)

TASK 1 – IDENTIFY USEFUL DATA, DROP NOISE

Looking through the sample letter, we can see that all of the useful patient information can be found in a single paragraph. The paragraph is preceded by the line "Results of Patient A" and is concluded by a blank line. Everything prior to that line of data and following the blank line at the end of the paragraph – our "Indicators" – can safely be ignored. (This is illustrated in Figure 4; Figure 5 extracts and expands the signal part of our data.)



Office of Tonya Uedlakoff, MD
Medical Office Building
23 Macavitch Lane, Suite 44
Baton Rouge, LA 70819
(225) 555-4321

Dr. Moe Lahr, DDS
Dental Arts Office Complex
51 Ichiro Drive, Outer Building
Renton, WA 98056
(425) 555-6789

Dear Moe –

Sorry to hear of your concern for my former patient and that he is having incisor issues. You asked about the shoot team experimental cocktail he had been administered while here. Our office do not have the daily blood draw results, but I can share the following:

Results of Patient A:

On Day 1, patient BP was 117/81 with a temperature of 98.4 prior to the administration of Treatment X1; their SPO2 varied between 94 and 97. Day 2 showed little variation, BP of 123/85, temperature of 98.5, and SP02 ranging from 93 to 98 before giving Treatment X1. Changes showed up on Day 3, where prior to administration of Treatment X1, BP was 133/79 and SP02's lower range dropped to 86, although the upper range was still 98. On Day 4, SPO2 lower range improved slightly to 88, but the upper range dropped to 95; fortunately, BP remained consistent at 129/81 and temperature was 98.4 again before Treatment X1 was given. Day 5 was the final day of the regimen; patient temperature rose to 99.1 while BP was 128/78 and SPo2 came in at 88 to 97.

I hope this helps. For any in-depth data, you'll need to contact the laboratory directly.

Sincerely,

Tonya

Figure 3 – Sample Letter

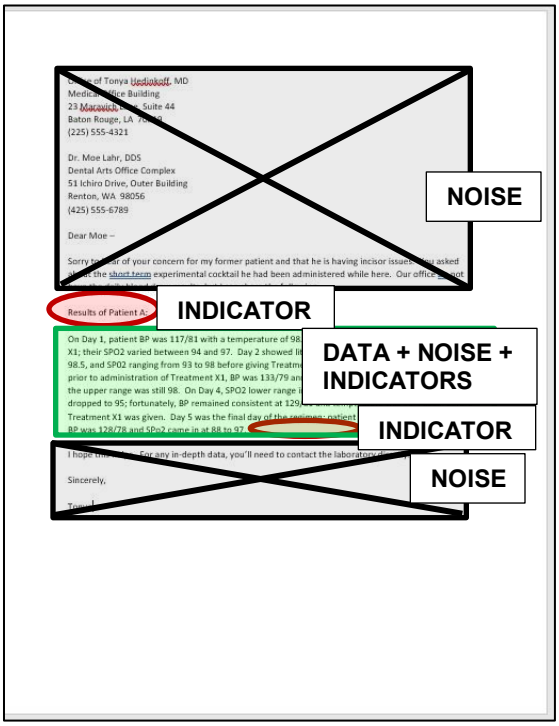


Figure 4 – Sample Letter - Annotated

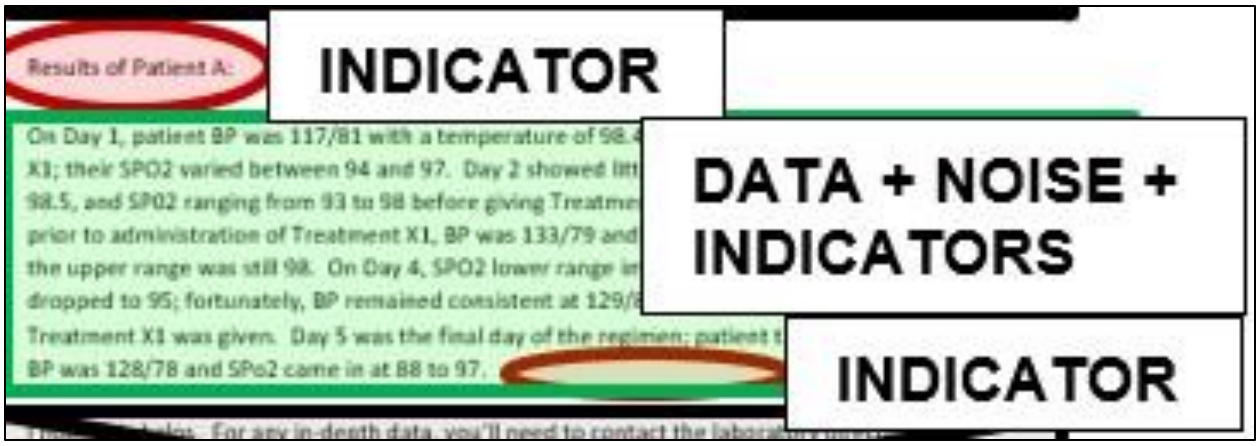


Figure 5 – Sample Letter - Annotated

The first task is to read in the file – for the sake of this example, let us assume that we have it in a .TXT file in ASCII format. The **`INFILE`** and **`INPUT`** statements are basic commands within a DATA step used to define the source of input data and to define the data which we wish to bring in from that data source. We will not take the time or space to go into detail on their exact syntax or options – those who wish more detailed information should refer to the "Recommended Reading" for detailed presentations on those commands.

We will look at on aspect of INPUT – the "NULL input". A NULL input statement contains no variable definitions and no options; it simply states the word INPUT. The traditional rationale for using a NULL input is to terminate a "training @", that is, to tell SAS that it should go back to its default behavior of dropping to the next line automatically after reading the current one. However, there is one other thing that a NULL input will do – it will populate an automatic temporary SAS variable called **`_INFILE_`**. (For the record, *every* INPUT statement will populate this variable.) The entire line of input data will be stored in this temporary SAS variable, meaning that it is available for use within the DATA step but will not be written to the output dataset.

Our code needs to parse through each line of data, looking through each line for the string "Results of Patient". We want to drop all data prior to that string and keep all data after that string until the blank line at the end of the paragraph is found. At that point, we can drop all subsequent data. We can use the **`INDEX`** function to search each line for the string. The INDEX function takes the string provided in the first parameter, and searches it for the string specified in the second parameter. If found, it will return the starting position of the second string within the first string. Otherwise, it will return a 0.

We will track whether the indicator phrase has been found and is active through a True/False variable, which we have called Output_IND. (Aside: It has been my practice to use the "_IND" suffix on all of my True/False variable names.) If the variable is False, as it is initialized at the beginning of the routine in the **`RETAIN`** statement, the line of data can safely be ignored. If True, then we should keep the data being read, and store it in our work file for further processing. Once the blank line at the end of the paragraph is encountered, the routine can stop writing out data once more. The sample code watches for the blank line before writing to the output dataset, but there should be no subsequent issue if it is also written out.

In order to keep the input data, it is necessary to transfer the data from the _INFILE_ variable to one that is written to the output dataset. As stated, _INFILE_ is a temporary SAS variable that is automatically dropped at the end of the Data step execution. Adding it to a KEEP statement or KEEP option will not override this behavior. This is handled with a simple "equation", <permvar> = _INFILE_. (The code to perform this functionality is shown in Figure 6.)

```
DATA  Letter_Contents;
   RETAIN  Output_IND 0;
   INFILE "<file name & location goes here>";
   INPUT ;
   IF  INDEX( _INFILE_, "Results of Patient")  THEN  Output_IND = 1;

   IF  Output_IND  THEN DO;
      OUTPUT_LINE = _INFILE_;
      IF  OUTPUT_LINE = ""  THEN  Output_IND = 0;
      ELSE  OUTPUT;
   END;
RUN;
```

Figure 6 – Code to separate signal from noise – first pass

As with many coding exercises, this first pass of our routine may compile correctly, but it will not correctly perform the task we set out to accomplish. Our output dataset will contain only one line, which is "Results of Patient A. A re-examination of our input shows that there is a blank line following this indicator. Since the code is designed to stop storing data once a blank line is encountered, our indicator line is the only one that will be stored.

It will be necessary to add logic to watch for the presence of two blank lines. The first can safely be ignored, while the second is the true indicator that informs the routine that it can stop storing data. This can be handled using a second variable, which we have called Blankline_Cnt. (Aside: As with _IND, I have used the suffix _Cnt as a quick reference that a variable contains a counter.) The first blank line increments Blankline_Cnt from 0 to 1, while the second blank line represents the true end of our input data. This resets both in both Blankline_Cnt and Output_IND being set to 0.

```
DATA  Letter_Contents;
   RETAIN  Output_IND    0
           Blankline_Cnt 0;
   INFILE "<file name & location goes here>";
   INPUT ;
   IF  INDEX( _INFILE_, "Results of Patient")  THEN  Output_IND = 1;
   IF  Output_IND  THEN DO;
      OUTPUT_LINE = _INFILE_;
      IF  _INFILE_ = ""  THEN  DO;
         Blankline_Cnt + 1;
         IF  Blankline_Cnt = 2  THEN DO;
            Blankline_Cnt = 0;
            Output_IND = 0;
         END;
      END;
      ELSE  OUTPUT;
   END;
RUN;
```

Figure 7 – Code to separate signal from noise – second pass

IDENTIFYING INDIVIDUAL DATA ITEMS FROM EACH INPUT LINE

Once this second pass has been run, we should have a dataset containing only those lines with useful data – ten of them in this example. Our next challenge is to pull out individual data items from within these lines.

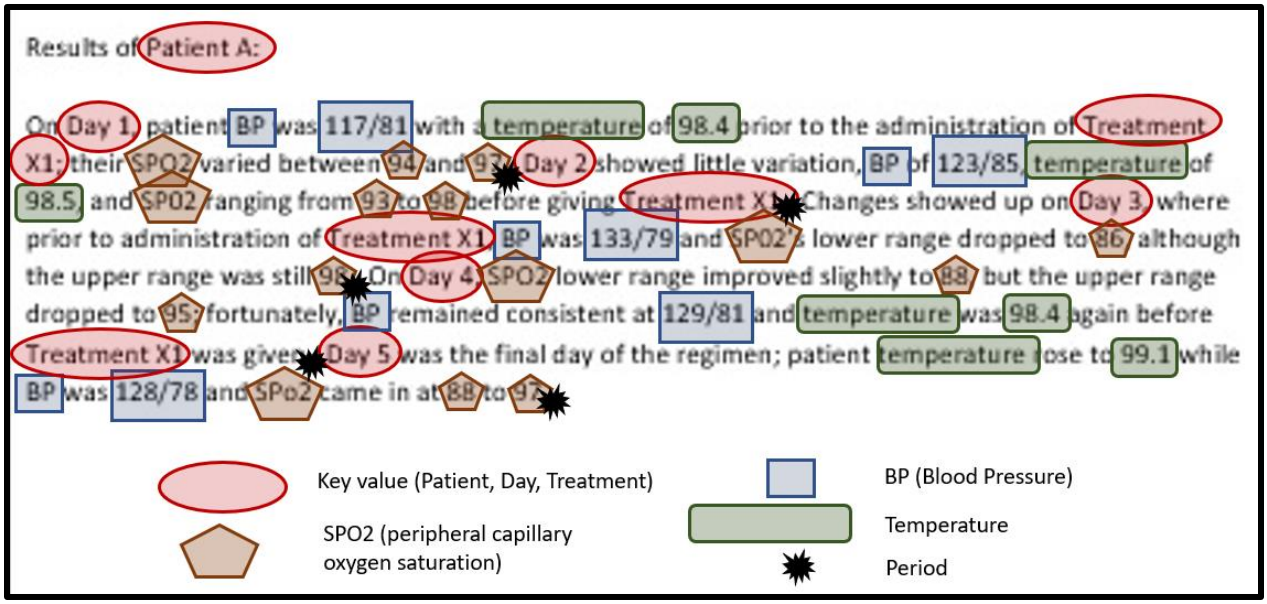


Figure 8 – Sample Letter – Detail Annotated

Figure 8 illustrates the portion of the letter that was retained after our initial parsing. It has been color-coded to denote the information that we wish to extract. A few facts that will be important to our processing:

- All data refers to the Patient identified on the original indicator line. This information is not repeated; the value should be identified and applied to all records.
- Each sentence contains one day's worth of information for said patient. The order of the data is not consistent from sentence to sentence. Due to the presence of decimal values, it will be necessary to use the presence of a period followed by a space to denote the end of a sentence. It is possible for the period to be the last character of a line, as well, in which case it would not have a space following it. Our example does not include any situations where this occurs.
- We do not have true dates provided by our source, BUT we have a set of whole numbers preceded by the word "Day". In our example, this is always the first key piece of information in each sentence. Further, every sentence contains information regarding patient results for that single day identified up front.

DEVELOPING A PARSING ROUTINE

The first strong caution that should be put on the record is let the routine evolve, *don't try to write the entire thing in one sweep*! Debugging every functionality of a freshly-written routine at the same time is a recipe for frustration – a lesson that the author needs to re-learn every time he attempts to write such a routine.

Task # 1 will be to take the lines that were input from the letter and convert them to sentences, since we have already determined that each sentence contains one day's worth of data. (See Figure 9 for the code. Note that the shaded code was added as a Task #1A – identify the Patient ID and ensure it is added to each observation.)

It will be necessary to store the data in two separate character variables. Data that is in the process of being gathered from each "Output_Line" that we read in earlier will be stored in "Partial_Sentence" Once we have encountered the period indicating the end of the sentence, we will shift the field into "Complete_Sentence". At this point, "Partial_Sentence" will be re-initialized so that it can be re-populated.

```
DATA  Letter_Contents_Sentences(Keep=Patient_ID Complete_Sentence);
   SET  Letter_Contents;
   LENGTH  Partial_Sentence  $ 1000.
           Complete_Sentence $ 1000. ;
   RETAIN  Period_Found_LOC  0
           Partial_Sentence " "
           Patient_ID ;
   IF  _N_ = 1  THEN  DO;
      Partial_Sentence = Output_Line ;
   END ;

   IF  INDEX( Output_Line, "Results of Patient")  THEN DO;
      Patient_ID = COMPRESS( SUBSTR( Output_Line,
                                     LENGTH("Results of Patient")+1 ), ":");
      Partial_Sentence = " ";
   END;
   Period_Found_LOC = FIND( Output_Line, ". " );
   IF  Period_Found_Loc THEN  DO;
      Complete_Sentence = TRIM( Partial_Sentence ) || " " ||
                          SUBSTR( Output_Line, 1, Period_Found_Loc ) ;
   ***Complete_Sentence = CATX( " ", Partial_Sentence,
   ***                          SUBSTR( Output_Line, 1, Period_Found_Loc ) );
      OUTPUT  Letter_Contents_Sentences;
      Partial_Sentence = SUBSTR( Output_Line, Period_Found_Loc + 2 ) ;
      Period_Found_LOC = FIND( Output_Line, ". ", Period_Found_Loc + 2 );
   END;
   ELSE
      Partial_Sentence = TRIM( Partial_Sentence ) || Output_Line ;
 RUN;
```

Figure 9 – Convert lines of text into complete sentences

We will also look for the position of the period, if one exists in the line. This will be accomplished by searching for a "." In the input line using the **FIND** function. FIND is similar to INDEX in that it searches for a target string in source string. If found, it will also return the starting position of the second string within the first string, with a 0 representing "not found". However, FIND allows options that can start the search from any point within the source string, and to work from left-to-right OR right-to-left.

Once the period indicating the end of the sentence is located, the "Complete_Sentence" is assembled by taking whatever sentence component was identified earlier in the processing and appending the initial portion of the current line to it. In our example, we show the older technique to handle this logic – using the **TRIM** function to suppress the training blanks of the partial string, appending a single blank to that string, then tacking on a substring of the current line using the **SUBSTR** function – starting at the first position of the current line and going to the position where the period was found. Note that the more recent **CATX** function can be used to perform the TRIM and the concatenation functions, as well as inserting separated by a specified parameter, for multiple strings. The equivalent CATX function is commented in the sample code.

Before moving on to process the next line of the letter, it is necessary to reset the partial sentence to contain the unused portion of the original string. This is accomplished with a SUBSTR function. Our sample code also checks to see if there might be a second period in the line. Since our example does not include any such examples, and in the interest of time and space in this paper, the DO loop logic necessary to perform multiple checks on the same line has been omitted. The code may also encounter problems if the period falls on the absolute last character of a line, not leaving any room for the trailing blank – again, this deficiency is noted, but not included in our example.)

When the code is executed, the SASLOG warns:

```
NOTE: Variable "Patient_ID" was given a default length of 32767
      as the result of a function call.  If you do not like this,
      please use a LENGTH statement to declare "Patient_ID".
```

SAS's handling of default lengths has been improved over the years. However, certain functions – in this case, the open-ended SUBSTR function, without the LENGTH parameter – cause SAS to play it safe and define the variable length with a "safe" (read: "excessive") length. For a small amount of data, the wasted storage space is not worth worrying over. For larger files, storing what turns out to be a 2-byte substring in 32,767 bytes will add up fast. As the message suggests, use the LENGTH statement to properly manage the definition.

The next task is to parse each individual line. The DATA step in Figure 10 has been set up to identify and extract each individual data item contained within the source letter. (Aside: The goal in this example was to use a different approach for each extracted variable in order to utilize a wider selection of techniques, and in particular, of SAS character functions. In the real-world, this sort of coding philosophy would complicate development, debugging, testing, and any subsequent support, and is not recommended.)

NOTE: DUE TO LOOMING (actually "missed") DEADLINES, AND DUE TO THE AUTHOR'S UNDERESTIMATION OF JUST HOW LONG DEVELOPING THE CODE FOR THIS EXAMPLE WOULD TAKE, THIS INITIAL VERSION OF THE PROGRAM WILL NOT HAVE THE CORRESPONDING TEXT WITH AN IN-DEPTH EXPLANATION OF THE APPROACHES AND FUNCTIONS USED IN THIS DATA STEP. THEY WILL BE COVERED IN THE IN-PERSON LECTURE / HANDS-ON WORKSHOP VERSION OF THIS PAPER.
A .pdf FILE CONTAINING THE REMAINING INFORMATION IN POWERPOINT FORMAT CAN BE REQUESTED FROM THE AUTHOR.

"I love deadlines. I like the whooshing sound they make as they fly by."
— Douglas Adams
https://www.brainyquote.com/quotes/douglas_adams_134151

```
DATA  Letter_Contents_Detail(Keep=Patient_ID  SPO2_Range1  SPO2_Range2
                                Day  BP  Temperature  Complete_Sentence);
   SET Letter_Contents_Sentences;
   /** Day         **/
   Day_Pos = FINDW( Complete_Sentence, "Day" ) ;
   Day     = SCAN( SUBSTR( Complete_Sentence, Day_Pos ), 2);
   /** Treatment   **/
   LENGTH  Treatment     $ 10.
           Treatment_Lag $ 10.;
   RETAIN  Treatment_Lag ;
   Treatment_Pos = FINDW( LOWCASE( Complete_Sentence ), "treatment" ) ;
   IF   Treatment_Pos = 0   THEN  Treatment = Treatment_Lag ;
   ELSE   Treatment = SCAN( SUBSTR( Complete_Sentence, Treatment_Pos ), 2);
   Treatment_Lag    = LAG( Treatment );
   /** BP          **/
   BP_Pos  = FINDW( Complete_Sentence, "BP");
   BP_WIP  = SUBSTR( Complete_Sentence,
                     ANYDIGIT(Complete_Sentence, BP_Pos ));
   BP_WIP2 = SUBSTR( BP_WIP, 1, FINDC( BP_WIP, "/")+3 ) ;
   BP      = COMPRESS( BP_WIP2, "/", "DK");
   /** temperature **/
   Temp_Pos = FINDW( UPCASE( Complete_Sentence ), "TEMPERATURE" ) ;
   IF  Temp_Pos > 0  THEN DO;
      Temp_DP_Pos    = FINDC( Complete_Sentence, ".", Temp_Pos );
      Temp_Dec_Pos   = FIND( Complete_Sentence, " ", Temp_Pos );
      Temp_Rvse_Sntc = COMPRESS( REVERSE(
                         SUBSTR( Complete_Sentence, 1, Temp_DP_Pos ) );
      Temp_Digit     = SUBSTR( Temp_Rvse_Sntc, 1,
                           NOTDIGIT( COMPRESS( REVERSE(
                           SUBSTR( Complete_Sentence, 1, Temp_DP_Pos-1) ))));
      Temperature    = REVERSE( COMPRESS( Temp_Digit ) ) ||
                       CHAR( Complete_Sentence, Temp_DP_Pos+1 ) ;
   END;
   /** SPO2        **/
   Sentence_SPO2 = TRANSLATE( UPCASE( Complete_Sentence ), " ", "'" );
   Sentence_SPO2 = TRANWRD( UPCASE( Sentence_SPO2 ), "SP02", "SPO2" );
   SPO2_Pos = FINDW( UPCASE( Sentence_SPO2 ), "SPO2" ) ;
   Sentence_SPO2 = SUBSTR( Sentence_SPO2, SPO2_Pos) ;
   RETAIN  Keep_SPO2_Cnt 0;
   Keep_SPO2_Cnt = 0 ;
   Word_Cnt      = COUNTW( Sentence_SPO2, " " );
   DO Indx = 1 TO Word_Cnt;
      Keep_the_Word_Q = SCAN( Sentence_SPO2, Indx, " " );
      IF  NOTDIGIT(COMPRESS(Keep_the_Word_Q, " .,;:"))  THEN /* ignore it */;
      ELSE DO;
         Keep_SPO2_Cnt+1;
         IF  Keep_SPO2_Cnt = 1  THEN
            SPO2_Range1 = COMPRESS( Keep_the_Word_Q, " .,;:" )  ;
         ELSE IF Keep_SPO2_Cnt = 2  THEN DO;
            SPO2_Range2 = COMPRESS( Keep_the_Word_Q, " .,;:" ) ;
            Indx = Word_Cnt;
         END;
      END;
   END;
 RUN;
```

Figure 10 – Identify and store the value for each desired data component

AN IMPORTANT CAUTION

Parsers can provide a technical challenge for a programmer who is used to more traditional technical requirements. They can also, frankly, be quite fun to tackle. However, in a professional environment, the time required to complete the routine may not be the most effective way to obtain the necessary data. Before investing the effort to create a data parser, investigate whether the data can be provided in a more traditional format. If that is not possible, it may actually be more desirable to simply perform the task manually – read the source and either cut-and-paste or simply type in the data. (Then, perform a double-check, as this can be error prone.)

Further, this approach assumes that the data will arrive in some semblance of a consistent format. Experience has shown that this is not always the case. Different vendors may provide their data in different formats; in fact, occasionally, the SAME vendor may prove to be quite variable in their data format.

Personal experience has shown three instances where a data parser justifies itself:

1) When the data turnaround must be instantaneous.
2) When the data volume causes the amount of time required to perform manual entry to be so excessive as to be non-viable.
3) When the process recurs over time.

CONCLUSION

This paper was designed to show the use of some of SAS's various techniques to handle character data, through their use in a fictitious example. The reader may or may not ever encounter a situation similar to the example in this paper. They are far more likely to have to deal with SOME situation where data are stored in a non-standard format and are virtually guaranteed to have to deal with a number of character data situations over the course of their careers.

REFERENCES AND RELATED READING

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.
https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/135-2007.pdf

Cassels, David L. (2007). "The Basics of the PRX Functions". *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute, Inc.
https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/223-2007.pdf

Childress, Spencer. (2014). "Express Yourself! Regular Expressions vs SAS® Text String Functions". *Proceedings of the PharmaSUG 2014 Conference*.
https://www.lexjansen.com/pharmasug/2014/BB/PharmaSUG-2014-BB08.pdf

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.
https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/110-31.pdf

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. https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/228-31.pdf

Kuligowski, Andrew T. (2007). "Easy Come, Easy Go — Interactions between the DATA Step and External Files". *Proceedings of the SAS® Global Forum 2007 Conference*. Cary, NC: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/220-2007.pdf

Kuligowski, Andrew T. (2015). "Using INFILE and INPUT Statements to Introduce External Data into SAS®". *Proceedings of the PharmaSUG 2015 Conference*. https://www.lexjansen.com/pharmasug/2015/HT/PharmaSUG-2015-HT07.pdf

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. https://www.lexjansen.com/sesug/1994/SESUG94021.pdf

Kuligowski, Andrew T. (2009) "Using SAS® to Parse External Data". *Proceedings of the SAS® Global Forum 2009 Conference*. Cary, NC: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/190-2008.pdf

Kunwar, Pratap S. and Erinjeri, Jinson. (2019) "Quick Tips and Tricks: Perl Regular Expressions in SAS®". *Proceedings of the SAS® Global Forum 2019 Conference*. Cary, NC: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/190-2008.pdf

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

SAS Institute Inc. (2017). *Base SAS® 9.4 Procedures Guide, Seventh Edition*. Cary, NC: SAS Institute Inc. https://documentation.sas.com/api/docsets/proc/9.4/content/proc.pdf

SAS Institute, Inc. (2016) *SAS 9.4 Companion for Windows, Fifth Edition*. http://support.sas.com/documentation/cdl/en/hostwin/69955/PDF/default/hostwin.pdf

SAS Institute, Inc. (2019) *SAS® 9.4 Functions and CALL Routines: Reference, Fifth Edition*. https://documentation.sas.com/api/docsets/lefunctionsref/9.4/content/lefunctionsref.pdf

Wicklin, Rick. (2016). "Break a Sentence Into Words in SAS". *The DO Loop.*. https://blogs.sas.com/content/iml/2016/07/11/break-sentence-into-words-sas.html

This list is meant to whet the appetite. The reader is heartily encouraged to search for other fine papers on www.lexjansen.com

ACKNOWLEDGMENTS

This paper evolved from one I did on the topic approximately 10 years ago. At the time, Peter Eberhardt and Sue Douglass thought it might be something worth researching, documenting, and presenting at SAS Global Forum. It had been presented at various user group meetings and conferences over the following years, in both lecture and Hands-On format. PharmaSUG China 2019 chairperson Kriss Harris attended one of those presentations and thought that the material would be beneficial and interesting to the people attending that conference, as well. However, Kriss thought that maybe a tweak or two would be appropriate in order to make it relevant to his industry-specific audience. That "tweak or two" ended up resulting in a complete rewrite of the original paper and presentation. (For the record, two paragraphs from the original paper survived relatively intact.) He must have had a point, as other 2019 fall conferences also have requested that I give this presentation at their events. I appreciate all the hard-working individuals who have invited me to presentation over the years.

Everyone who attended and provided feedback on the original paper and presentation contributed in some way to the re-presentation of the concept in this form, as well, and would like to acknowledge and thank them.

CONTACT INFORMATION

In the event of any questions, comments, or whatever, you can contact the authors via email:

Andrew T. Kuligowski
KuligowskiConference@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.