PhUSE 2016

Paper CC03

Macro-free parsing of multiple text files with SAS

Adam Amborski, inVentiv Health Clinical, Warsaw, Poland

ABSTRACT

Statistical programmers are well aware how powerful a tool SAS® software is when it comes to data processing. This paper goes beyond the manipulation of data contained in data sets and describes how smart use of the data step can also be used to modify text files, including SAS programs themselves. The proposed approach uses no macro programming and can be data or user driven, for example to define the set of files to be modified or the modifications required. The technique described can be used in any environment and, unlike the Unix® SED command, does not rely upon regular expressions to find and replace text (although they can certainly be used). An important consideration too is that no unwanted changes are made (for example, trailing blanks are not lost) – essential in a version-controlled environment – and this is also provided for in the proposed method.

INTRODUCTION

Modifying multiple text files in base SAS efficiently and accurately without any unwanted changes may pose a challenge. There are tools for text processing such as the Unix or GNU SED command which allows the magic of regular expressions and much more, and which is very robust. The difficulty is, not all programmers using SAS software are fluent with regular expressions, plus, SED is difficult to use if changes depend on multiple data sources or other factors, and it is seldom available in Windows® environments at all.

With SAS software, it is easy to define changes based on multiple factors, obtaining information to put into files from external databases or even from the user - from the choice of files to process to the changes you actually want to apply. If one is in a version-controlled environment, an important consideration is that no unwanted changes are made (for example, trailing blanks are not lost). SAS software offers data step tools to achieve this, and with smart use of them, one can achieve all the above without resorting to the macro language.

THE PROCESS

The best way to explain the process is by giving an example. Please mind that the example is simplified to show just the key elements. The simplicity of the example is meant to show you how easy it is to include a complicated process without sacrificing the general effect of changing only what is meant to be changed.

LIST OF FILES TO BE PROCESSED

The first thing to do is to create a list of files to be processed. At this stage you could collect specific information on how the files are to be processed later, however to keep the example clear, the following code just gathers program files present in a folder:

```
1  %let dir=h:/phuse2016/codes/;
2  data files (keep=filename full_path label="All SAS programs in a directory");
3    length filename full_path $1000;
4    rc=filename("dir","&dir");
5    did=dopen("dir");
6    if did ne 0 then do;
7      do i=1 to dnum(did);
8        filename=dread(did,i);
9        full_path="&dir"||filename;
10       if lowcase(scan(filename, -1, "."))="sas" then output;
11     end;
12   end;
13   else do;
14     put "ERROR: Failed to open the directory &dir";
15     stop;
16   end;
17 run;
18 proc sort data=files;
19   by full_path;
20 run;
```

The collecting of file names is done by opening the required directory using the DOPEN function (line 5), which opens a directory and assigns a positive identifier value to it (if successful), then a loop goes from the first element to the last (using the DNUM function – line 7). The DREAD function reads subsequent elements of the directory, returning the names to the `filename` variable (line 8). Then we only keep files with "sas" (case insensitive) as the extension.

Full paths are also stored and the data set sorted by path for ease of processing later. If you process files in only one directory, as in the example, there is no need to store it in a separate variable – you may just include it later in the output code and thus save some resources. On the other hand, if you are processing files from multiple folders, it may be reasonable to store such information.

VIEWTABLE: All SAS programs in a directory

	filename	full_path
1	program_a.sas	h:/phuse2016/codes/program_a.sas
2	program_b.sas	h:/phuse2016/codes/program_b.sas
3	program_dont_modify.sas	h:/phuse2016/codes/program_dont_modify.sas

Figure 1 Data set listing files to be processed.

Although the paper presents a "macro-free" technique, you may have noticed a macro variable being used above (dir). This is simply to avoid the directory path being hard-coded twice in the code, and it will not be used anywhere later in the example.

READING THE FILES

Once a data set containing the list of files to be processed is ready, the next step is to read them all in using the INFILE statement together with the FILEVAR and LENGTH options. The FILEVAR= option enables you to read multiple files within one data step. The LENGTH= option allows you to find the actual length of every line that has been read, including trailing blank characters which are otherwise lost in the input process, as the LENGTH will include them. As variables assigned to the FILEVAR= and LENGTH= options are treated as automatic variables and not written to the resulting data set, extra steps are needed to preserve the information they carry:

```
21 data contents (label="Original files' full contents"
22                rename=(full_path_tmp=full_path));
23   set files end=last;
24   length full_line $&lrecl;
25   infile codes filevar=full_path truncover lrecl=&lrecl
26                length=line_length_tmp end=all;
27   full_path_tmp=full_path;
28   line_number=0;
29   do while(not all);
30     input;
31     full_line=_infile_;
32     line_length=line_length_tmp;
33     line_number+1;
34     output;
35   end;
36 run;
```

The `full_path` variable is copied into `full_path_tmp` (line 27) which is subsequently renamed to the name of the former (line 22), as it carries the same information. The `line_length_tmp` is copied to `line_length` and the latter is stored in the resulting data set.

Line numbers are also stored in the above data step. This may be very helpful in further processing, e.g. identifying parts of code and their relationships, defining standardized rules for changes, etc., although it is not used in the example presented in this paper.

VIEWTABLE: Original files' full contents

	filename	full_line	full_path	line_number	line_length
1	program_a.sas	data pies;	h:\phuse2016\codes\program_a.sas	1	10
2	program_a.sas	a="1"; /* 2 blanks follow here */	h:\phuse2016\codes\program_a.sas	2	39
3	program_a.sas	run;	h:\phuse2016\codes\program_a.sas	3	4
4	program_b.sas	data pies;	h:\phuse2016\codes\program_b.sas	1	10
5	program_b.sas	a="1"; /* 3 blanks follow here */	h:\phuse2016\codes\program_b.sas	2	40
6	program_b.sas	run;	h:\phuse2016\codes\program_b.sas	3	4
7	program_b.sas		h:\phuse2016\codes\program_b.sas	4	0
8	program_dont_modify.sas	data pies;	h:\phuse2016\codes\program_dont_mo	1	10
9	program_dont_modify.sas	a="1"; /* Three blanks follow here */	h:\phuse2016\codes\program_dont_mo	2	44
10	program_dont_modify.sas	run;	h:\phuse2016\codes\program_dont_mo	3	4

Figure 2 Data set with contents of files to be processed.

In the contents above, you can see the number of trailing blanks causes differences in `line_length` variable values for otherwise identical second lines (`line_number=2`) between program_a, program_b, and program_dont_modify (rows 2, 5 and 9, respectively).
Another macro variable, LRECL, is used here to highlight places where values should remain constant throughout the process. This one represents the maximum line length of all the files that are processed. It will also be referred to when we output processed files.

PROCESSING

This part depends on the reason for amending the files in question. For the sake of clarity, the code below just adds a line of text at the end of each file processed, with an exception for files that have "dont_modify" included anywhere in their name:

```
37 data processed (label="Manipulate file contents");
38   set contents;
39   by full_path;
40   output;
41   if last.full_path and index(filename,"dont_modify")=0 then do;
42     full_line=catx(" ", filename, "is stored on",
43                    scan(full_path, 1, "\"), "drive.");
44     line_length=length(full_line);
45     line_number=line_number+1;
46     output;
47   end;
48 run;
```

This causes the last file not to be modified at all (see below).

VIEWTABLE: Manipulate file contents

	filename	full_line	full_path	line_number	line_length
1	program_a.sas	data pies;	h:\phuse2016\codes\program_a.sas	1	10
2	program_a.sas	a="1"; /* 2 blanks follow here */	h:\phuse2016\codes\program_a.sas	2	39
3	program_a.sas	run;	h:\phuse2016\codes\program_a.sas	3	4
4	program_a.sas	program_a.sas is stored on h: drive.	h:\phuse2016\codes\program_a.sas	4	39
5	program_b.sas	data pies;	h:\phuse2016\codes\program_b.sas	1	10
6	program_b.sas	a="1"; /* 3 blanks follow here */	h:\phuse2016\codes\program_b.sas	2	40
7	program_b.sas	run;	h:\phuse2016\codes\program_b.sas	3	4
8	program_b.sas		h:\phuse2016\codes\program_b.sas	4	0
9	program_b.sas	program_b.sas is stored on h: drive.	h:\phuse2016\codes\program_b.sas	5	39
10	program_dont_modify.sas	data pies;	h:\phuse2016\codes\program_dont_modify.sas	1	10
11	program_dont_modify.sas	a="1"; /* Three blanks follow here */	h:\phuse2016\codes\program_dont_modify.sas	2	44
12	program_dont_modify.sas	run;	h:\phuse2016\codes\program_dont_modify.sas	3	4

Figure 3 Data set with modified contents of files.

The `line_length` variable here is calculated in order to write the new line with no truncation or trailing blanks. The `line_number` variable is maintained only to show that it can be kept accurate to facilitate further processing, if necessary.

IDENTIFYING CHANGES

You may wish to flag the files which were changed so as to minimize the IO burden on unchanged files. This could have been done during the above processing, but can also be done in a separate step as below:

```
49 data updatedfiles (label="Files that where actually changed");
50   merge contents  (rename=(full_line=orig_line line_length=orig_length)
51                    in=orig)
52         processed (in=processed);
53   by full_path line_number;
54   retain changed;
55   if first.full_path then changed="";
56   if        orig ne processed   or
57        orig_line ne full_line   or
58      line_length ne orig_length
59        then updatedfl="Y";
60   if last.full_path and updatedfl="Y" then output;
61   keep full_path filename;
62 run;
63
64 data tosave;
65   merge processed updatedfiles (in=updated);
66   by full_path;
67   if updated;
68 run;
```

The `line_number` variable is used in the BY statement as a sanity check as the order of the records should not have changed as a result of the processing and the variable is not referenced outside the BY statement (SAS would write an error to the log if `line_number` was not in sorted order). It is of course possible to introduce other sanity checks in the code above.
Only two files are changed, and their contents are ready to be saved (see below).

VIEWTABLE: Files that where actually changed

	filename	full_path
1	program_a.sas	h:/phuse2016/codes/program_a.sas
2	program_b.sas	h:/phuse2016/codes/program_b.sas

Figure 4 Data set listing files that were actually modified.

VIEWTABLE: Work.Tosave

	filename	full_line	full_path	line_number	line_length
1	program_a.sas	data pies;	h:\phuse2016\codes\program_a.sas	1	10
2	program_a.sas	a="1"; /* 2 blanks follow here */	h:\phuse2016\codes\program_a.sas	2	39
3	program_a.sas	run;	h:\phuse2016\codes\program_a.sas	3	4
4	program_a.sas	program_a.sas is stored on h: drive.	h:\phuse2016\codes\program_a.sas	4	39
5	program_b.sas	data pies;	h:\phuse2016\codes\program_b.sas	1	10
6	program_b.sas	a="1"; /* 3 blanks follow here */	h:\phuse2016\codes\program_b.sas	2	40
7	program_b.sas	run;	h:\phuse2016\codes\program_b.sas	3	4
8	program_b.sas		h:\phuse2016\codes\program_b.sas	4	0
9	program_b.sas	program_b.sas is stored on h: drive.	h:\phuse2016\codes\program_b.sas	5	39

Figure 5 Data set with contents that will be saved to files.

WRITING THE FILES OUT

The next step is to output the changed content, using the FILEVAR= option again, but now using the FILE statement. We need to use the $VARYING. format in conjunction with a variable which contains the length of each line so as to ensure that any trailing blanks are preserved.

```
69 data _NULL_;
70   set tosave;
71   file sascode filevar=full_path lrecl=&lrecl;
72   put full_line $varying. line_length;
73 run;

NOTE: The variable full_path exists on an input data set, but was also specified in an
      I/O statement option.  The variable will not be included on any output data set.
NOTE: The file SASCODE is:
      Filename=h:\phuse2016\codes\program_a.sas,
      RECFM=V,LRECL=1256,File Size (bytes)=0,
```

```
      Last Modified=29Aug2016:22:59:21,
      Create Time=24Aug2016:21:34:17

NOTE: The file SASCODE is:
      Filename=h:\phuse2016\codes\program_b.sas,
      RECFM=V,LRECL=1256,File Size (bytes)=0,
      Last Modified=29Aug2016:22:59:21,
      Create Time=24Aug2016:21:34:17


NOTE: 4 records were written to the file SASCODE.
      The minimum record length was 4.
      The maximum record length was 39.
NOTE: 5 records were written to the file SASCODE.
      The minimum record length was 0.
      The maximum record length was 40.
NOTE: There were 9 observations read from the data set WORK.TOSAVE.
NOTE: DATA statement used (Total process time):
```

The $VARYING. format works well even outputting empty (zero length) records where necessary (program_b). If the length variable's value is larger than the length of the string to output, the line is padded with trailing blanks, which is exactly the desired behavior. In the log output above you can see the one byte difference in maximum record length between program_a and program_b that line two of the files is responsible for.

CONCLUSION

Once a data set containing the list of files to be processed is ready, the next step is to read them all in using the INFILE statement together with the FILEVAR= and LENGTH= options. It is important to preserve the contents of the two variables defined therein as they are both temporary variables. Then comes the actual processing – and that depends on the purpose of the whole exercise. We may wish to flag the files which were changed so as to minimize the IO burden on unchanged files. The next step is to output the changed content, using the FILEVAR= option again, but now using the FILE statement. We need to use the $VARYING. format with a variable which contains the length of each line to ensure that any trailing blanks are preserved. The result is a set of correctly amended files achieved with straightforward data step programming!

Overall, with this technique we achieve full control of the contents of the files without having to resort to other systems or even use macro programming. We preserve precisely the original state of lines we are not changing and furthermore we have full access to the data step's data manipulation functionality for more complex tasks.

REFERENCES

SAS Institute, "SAS 9.2 Language Reference: Dictionary"

ACKNOWLEDGMENTS

I would like to thank Rowland Hale for kind criticism and for questioning the solution presented herein. This allowed needless parts to be purged, which greatly helped the clarity of the paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Adam Amborski
inVentiv Health Clinical Poland
ul. Nowogrodzka 68
02-014 Warszawa
Email: Adam.Amborski@inventivhealth.com
Web: www.inventivhealth.com