PharmaSUG 2025 - Paper OS-122

Streamlining Validation Review and SAS® Program Management with R

Huei-Ling Chen
Merck & Co., Inc., Rahway, NJ, USA

ABSTRACT

Unlike SAS, R treats nearly everything as an object that can be easily modified. In addition to familiar objects such as datasets, the R language can also work with lists, vectors, and matrices. This flexibility in handling various objects dramatically expedites the workflow. This paper illustrates how to leverage R language to import large SAS files into a list for rapid collective processing within seconds. It highlights three specific applications that optimize post-processing tasks: (1) examining large-scale SAS validation output files, (2) automating the trimming of SAS syntax text, and (3) adding SAS syntax text to append data attributes from a central location for all analysis datasets.

INTRODUCTION

Advanced SAS programmers have worked to enhance the automation of the SAS program writing process. For example, Gao (2004) showed how to use SAS code to generate another SAS program. Timusk (2017) explained a method for importing a SAS program into a dataset that contains a single character variable, which programmers then manipulate to create a new program. Zhuo (2019) presented an automated approach where SAS generates code to apply data attributes to datasets.

Key techniques utilized in these papers include the INFILE statement and the PUT statement. The INFILE statement imports SAS programs into a SAS dataset, while the PUT statement writes SAS code within a data step. Both statements treat each line of text as a record, meaning that a SAS program becomes a dataset comprising hundreds of records and each record containing at least a single character variable storing the SAS syntax. While this programming approach reduces coding work for the programmer, it does come with the downside of extended processing time.

In recent years, programmers have also shared experience using R functions like **`list.files`** to identify and manage files and batch-run multiple scripts. For example, Huang (2023) demonstrated how to manage programs and execute tasks in R, while Mengelbier (2024) implemented audit functionalities using R.

This paper utilizes R functions **`lapply`**, **`read_file`**, and **`list.files`** to read large-scale files into a list and process them collectively for various applications. All files in the list can be edited and processed together in just a few seconds. This R approach not only automates the SAS program writing process but also demonstrates greater efficiency through a rapid processing time.

R VS. SAS COMPARISON

	R	**SAS**
Retrieve file names in a folder	Use **`list.files`** to read the file names in a directory or folder.	Use the **FILENAME** statement along with the functions DOPEN, DNUM, DREAD, and DCLOSE to read the file names in a directory or folder.
Handling large-scale files	Import large-scale files (.sas, .log, .lst) into a single list.	Import large-scale files (.sas, .log, .lst) into multiple SAS datasets (.sas7bdat).
Reading file content	Read the content of all files using **`lapply(list, read_file).`**	Read the content of a file using the **INFILE** statement. Repeat this process for each file.

Data structure	The content of a file becomes a character element in a list.	The content of a file becomes multiple records in a SAS dataset.
Processing file content	Use **`stringr`** package to process the content of all files.	Use **PUT** statement to write SAS code in a data step. Process the content of all files through data manipulation steps. Repeat this process for each file.
Write SAS codes to a SAS program	Use **`writeLines`** to write the SAS syntax text to a file with extension of '.sas' for all files.	Use the **FILENAME** statement along with the data step of PUT statement to write the SAS syntax text to a file with extension of '.sas'. Repeat this process for each file.

R FUNCTIONS RETRIEVING LARGE-SCALE FILES INTO A LIST

The key R functions reading files into a list are **`list.files`**, **`lapply`**, and **`read_file`**. They are from package **`base`** and **`readr`**.

```
## specify the current working location
setwd("/mk9999/prot001/validate/dbl-pgm/val0outlist")

## retrieve file names in the folder and build a list
## with each file name as an element
lf = list.files(path=getwd(), pattern='.lst', all.files=FALSE,
                full.names=FALSE)

## read the content of all files
a <- lapply(lf,read_file)
```

First, use **`setwd`** to specify the current working location.

Next, use the **`list.files`** function to retrieve the file names in the folder and create a list where each file name becomes an element. The '**`pattern`**' argument allows you to choose the file type. In the example code, the script selects the SAS output listing files with the file type '.lst' and stores them in a list object called '**`lf`**'.

The R function **`read_file`** reads an entire file at once, capturing the whole content as a single character string. The **`lapply`** function in R applies **`read_file`** across a list. You can treat a SAS output listing file as a string, where each file serves as an element composed of characters.

Name	Type	Date modified
dbl0adpkqt.lst	LST File	7/9/2024 4:08 PM
dbl0qtcf0chg0time0plot.lst	LST File	7/9/2024 8:19 PM
dbl0qtcf0cmax0est.lst	LST File	7/9/2024 6:05 PM
dbl0qtcf0cmax0est0plot.lst	LST File	7/9/2024 6:05 PM
dbl0qtcf0cmax0model0outdata.lst	LST File	7/9/2024 6:05 PM
dbl0qtcf0conc0dose0overlay0plot.lst	LST File	7/9/2024 6:05 PM
dbl0qtcf0conc0dose0plot.lst	LST File	7/9/2024 6:05 PM
dbl0qtcf0conc0reg0plot.lst	LST File	7/9/2024 6:05 PM

Display 1: Files in a directory/folder

```
> lf
 [1] "dbl0adpkqt.lst"                       "dbl0qtcf0chg0time0plot.lst"
 [3] "dbl0qtcf0cmax0est.lst"                "dbl0qtcf0cmax0est0plot.lst"
 [5] "dbl0qtcf0cmax0model0outdata.lst"      "dbl0qtcf0conc0dose0overlay0plot.lst"
 [7] "dbl0qtcf0conc0dose0plot.lst"          "dbl0qtcf0conc0reg0plot.lst"
 [9] "dbl0qtcf0conc0time0plot.lst"          "dbl0qtcf0model0outdata.lst"
[11] "dbl0qtcf0param0est.lst"               "dbl0qtcf0pred0obs0plot.lst"
[13] "dbl0qtcf0quantile0plot.lst"           "dbl0qtcf0quantile0sim0plot.lst"
```

Display 2: Files in a directory/folder becomes elements of a list

R FUNCTIONS WORKING ON STRINGS EASILY

The `stringr` package provides a set of functions designed to make working with strings as easy as possible.

Package	Function
`base`	**`lapply`** returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.
`readr`	**`read_file`** reads a complete file into a single object: either a character vector of length one, or a raw vector.
`stringr`	**`str_detect`** returns a logical vector with TRUE for each element of string that matches pattern and FALSE otherwise.
	`str_locate` returns the start and end position of the first match.
	`str_locate_all` returns the start and end position of each match.
	`str_sub` extracts or replaces the elements at a single position in each string.
	`str_replace_all` replaces all matches.

APPLICATION 1 – QC OF SAS VALIDATION OUTPUTS

The SAS programming team uses a double programming approach to validate datasets, tables, listings, and plots. This is a regular task to ensure the accuracy of study results.

PROC COMPARE is a procedure in SAS that compares the content and structure of two datasets. Validation of analysis datasets requires a clean PROC COMPARE report proving that the double programmer's analysis dataset matches the developer's analysis dataset. When validating tables, listings, and figures, every double programmer should also have a clean PROC COMPARE report proving that the double program can produce an identical output dataset that directly generates the table or figure. Therefore, ensuring that a clean PROC COMPARE report is available in every double programming output file is important.

```
                         The COMPARE Procedure
                  Comparison of WORK.DEV with WORK.DBL
                            (Method=EXACT)

                           Data Set Summary

    Dataset             Created          Modified  NVar    NObs

    WORK.DEV  16DEC24:23:08:24  16DEC24:23:08:24     5      26
    WORK.DBL  16DEC24:23:08:24  16DEC24:23:08:24     5      26


                           Variables Summary

              Number of Variables in Common: 5.



                          Observation Summary

                   Observation      Base  Compare

                   First Obs           1        1
                   Last  Obs          26       26

 Number of Observations in Common: 26.
 Total Number of Observations Read from WORK.DEV: 26.
 Total Number of Observations Read from WORK.DBL: 26.

 Number of Observations with Some Compared Variables Unequal: 0.
 Number of Observations with All Compared Variables Equal: 26.

 NOTE: No unequal values were found. All values compared are exactly
       equal.
```

Display 3: A clean PROC COMPARE report.

We consider the PROC COMPARE report clean when the following four text strings do not appear in the output:

- **"Listing of Common Variables with Differing Attributes"**
- **"Variables with Unequal Values"**
- **"but not in"**
- **"Variables with Conflicting Types"**

A lead programmer frequently needs to review multiple PROC COMPARE outputs. Manually opening each output listing can become tedious, especially since re-runs are quite common in clinical trials. This paper demonstrates how to use R to check whether a PROC COMPARE report exists in an output listing file with the '**.lst**' file extension and to determine if the PROC COMPARE report is clean by verifying that the aforementioned four text strings are absent.

Working Flow

1. Read all SAS output listing files into a list named '*outlist*'.

```
outlist = list.files(path=getwd(), pattern='.lst', all.files=FALSE,
                full.names=FALSE)

a <- lapply(outlist,read_file)
```

2. Create an empty list to iterate later.

```
lfchk1 = {}
lfchk2 = {}
…
```

3. Verify if there are any PROC COMPARE results in the output listing files.

```
for (i in 1:length(a)){lfchk1[i] <- str_detect(a[i],'The COMPARE Procedure')}
```

This loop checks for the presence of the phrase 'The COMPARE Procedure' in each element of the list and stores the logical results, either TRUE or FALSE, in a list format.

4. Check if the highlighted text strings appear in the output listing files.

```
for (i in 1:length(a))
{lfchk2[i] <- str_detect(a[i],
          'Listing of Common Variables with Differing Attributes|
           Variables with Unequal Values|
           but not in|
           Variables with Conflicting Types')}
```

This loop iterates through the list containing PROC COMPARE outputs and checks whether each element includes specific text strings pertaining to unmatched datasets using the `str_detect` function. If any of these text strings appear, the result will be TRUE; otherwise, it will yield FALSE. We store the results in a list format.

5. Summarize the results of the above checks and generate a summary report.

```
b <- list(filename=outlist, proccompare =lfchk1, unequal =lfchk2 )
c <- as.data.frame(do.call(cbind,b))
d <- c %>% mutate (checks=ifelse(unequal==TRUE | proccompare==FALSE ,'X','')) %>%
  select (filename, checks)

# Create checking result to excel file
library(openxlsx)
write.xlsx(d, file = "chk_compare_result.xlsx")
```

A summary report will include the SAS output listing filename along with the results of the checks. A flag variable, marked with a cross ('**X**'), indicates the presence of suspicious messages or situations where no PROC COMPARE results exist in the SAS output listing file. The R syntax exports the summary report to an Excel file.

Output

A	B
filename	checks
dbl0adpkqt.lst	
dbl0qtcf0chg0time0plot.lst	X
dbl0qtcf0cmax0est.lst	
dbl0qtcf0cmax0est0plot.lst	
dbl0qtcf0cmax0model0outdata.lst	
dbl0qtcf0conc0dose0overlay0plot.lst	
dbl0qtcf0conc0dose0plot.lst	
dbl0qtcf0conc0reg0plot.lst	
dbl0qtcf0conc0time0plot.lst	
dbl0qtcf0model0outdata.lst	
dbl0qtcf0param0est.lst	
dbl0qtcf0pred0obs0plot.lst	

Display 4: output excel file

```
                    The COMPARE Procedure
           Comparison of WORK.DEV with WORK.DBL
                      (Method=EXACT)

                Variables with Unequal Values

Variable  Type  Len   Label                  Ndif   MaxDif

MEANCHG   NUM     8   Change from Baseline     18    1.908
LLM       NUM     8   Change from Baseline     18    3.437
ULM       NUM     8   Change from Baseline     18    2.254
```

Display 5: An unclean PROC COMPARE report.

APPLICATION 2 – TRIMMING SAS SYNTAX TEXT IN BATCH EDITING OF SAS PROGRAMS

Suppose there is a need to edit a batch of SAS ADaM programs by trimming SAS syntax text after the step of the final dataset. In the example below, Display 6, we would like to remove the redundant comments and unwanted syntax at the end of an SAS program ‘adex.sas’.

```
run;
proc sort data=_adex out=_ex1;
by usubjid extrt astdt exseq;
run;
data _adex;
retain nvacnum 0;
set _ex1 ;
by usubjid extrt;
if first.usubjid or first.extrt then nvacnum = 0;
if exdose <= 0 then exvacnum = .;
else do;
nvacnum +1;
exvacnum=nvacnum;
end;
run;
*-----------------------------------------------------------------
*- Generate SAS code (end) ;
*-----------------------------------------------------------------
filename mprint clear ;
```

Display 6: SAS program ‘adex.sas’ – BEFORE trimming version

Above SAS programs need to trim their syntax text after the final data step. One way to do it is to find the desired position of the SAS code should be sliced.

Working Flow

1. Read all SAS programs into a list named '*saspgm*'.

```
saspgm = list.files(path=getwd(), pattern='.sas', all.files=FALSE,
                full.names=FALSE)

aaa <- lapply(saspgm,read_file)
```

2. Create an empty list to iterate later.

```
bbb = {}
ccc = {}
…
```

3. Find the desired position where the SAS code should be sliced.

```
# loop 1: find the position of the last data
for (i in 1:length(aaa)){bbb[i] <-
                         last(unlist(as.list(str_locate_all(aaa[i],'data ')))+1)}

# loop 2: keep the sas syntax after the last data
for (i in 1:length(aaa)){ccc[i] <- str_sub(aaa[i],start=bbb[i])}

# loop 3: find the position of the first 'run' end
for (i in 1:length(aaa)){fff[i] <-
                         first(unlist(as.list(str_locate_all(ccc[i],'run')))+3) }
ggg <- unlist(bbb)+unlist(fff)

# final loop: keep the sas syntax up to the end of data statement
for (i in 1:length(aaa)){hhh[i] <- str_sub(aaa[i],start=1,end=ggg[i]) }
```

The first loop identifies where the last occurrence of the string "**data** " is in each string, which is the SAS syntax. The second loop extracts the SAS syntax that follows the *last* occurrence of "**data** ". The third loop searches for the *first* occurrence of "**run**" in the extracted syntax and calculates the end position for extracting the final SAS syntax by combining the *last* "**data** " position and the *first* "**run**" position. The last loop extracts the SAS commands from the start to the calculated end position.

4. Save the new SAS code back to the original file.

```
# replace the sas program
for (i in 1:length(aaa)){writeLines(hhh[i],saspgm[i]) }
```

This R code replaces the SAS programs by writing new lines into specified files, the original file names stored in the list '***saspgm***'. The **`writeLines`** function writes character vectors to a text file, saving the changes made to each SAS program file.

Output

Trimming the SAS syntax text during batch editing of SAS programs takes only a few seconds.

```
run;
proc sort data=_adex out=_ex1;
by usubjid extrt astdt exseq;
run;
data _adex;
retain nvacnum 0;
set _ex1 ;
by usubjid extrt;
if first.usubjid or first.extrt then nvacnum = 0;
if exdose <= 0 then exvacnum = .;
else do;
nvacnum +1;
exvacnum=nvacnum;
end;
run;
```

Display 7: SAS program 'adex.sas' – clean version

APPLICATION 3 – BATCH EDITING SAS PROGRAMS TO APPEND DATA ATTRIBUTES TO FINAL DATASETS

A common approach for creating ADaM datasets is to employ metadata to maintain variable names, labels, types, origins, and the logic used for variable attributes. We save the metadata in an Excel file and convert it to a SAS dataset.

METADATA

A metadata of ADaM datasets is in place.

(see Display 9)

R

Use the metadata to write SAS syntax. Append the SAS syntax to the existing SAS programs, specifically for ADaM datasets.

SAS PROGRAM

Use the ATTRIB statement with the LENGTH and LABEL options to define variable attributes.

Utilize the KEEP statement to select the variables in the dataset.

Apply the PROC SORT procedure to define the variable order in the ADaM dataset.

(See Display 10)

Suppose a batch of SAS programs for ADaM datasets lacks proper data attributes. There is a need to add data attributes to the final dataset.

```
set scplus;
where sctestcd='STRATUM';
astrap=scorres;
keep usubjid astrap;
run;
proc sort data=_astrap ;
by usubjid;
run;
data adsl_final_1;
length PLNTHPY prthpy dcsreasp $100.;
merge adsl_final(in=in1) _tuside _assign _livmeta _intmeta _prthpy _pgf _invso
by usubjid;
...
run;
filename mprint clear ;
```

Display 8: adsl.sas – BEFORE adding data attribute statements

DOMAIN	VARIABLE	LABEL	TYPE	LENGTH	VAR_ORD
ADSL	STUDYID	Study Identifier	Char	12	1
ADSL	USUBJID	Unique Subject Identifier	Char	30	2
ADSL	SUBJID	Subject Identifier for the Study	Char	10	3
ADSL	SITEID	Study Site Identifier	Char	20	4
ADSL	SITENUM	Study Site Number	Char	10	5
ADSL	INVNAM	Investigator Name	Char	200	6
ADSL	AGE	Age	integer	8	7
ADSL	AGEU	Age Units	Char	6	8
ADSL	AGEGR1	Pooled Age Group 1	Char	20	9
ADSL	AGEGR1N	Pooled Age Group 1 (N)	integer	8	10
ADSL	AGEGR8	Pooled Age Group 8	Char	50	11
ADSL	AGEGR8N	Pooled Age Group 8 (N)	integer	8	12
ADSL	AGEGR9	Pooled Age Group 9	Char	50	13
ADSL	AGEGR9N	Pooled Age Group 9 (N)	integer	8	14
ADSL	SEX	Sex	Char	2	15
ADSL	SEXN	Sex (N)	integer	8	16
ADSL	ASEX	Analysis Sex	Char	20	17
ADSL	ASEXN	Sex (N)	integer	8	18
ADSL	RACE	Race	Char	41	19
ADSL	RACEN	Race (N)	integer	8	20

Display 9: ADSL attributes saved in a metadata in SAS dataset.

Working Flow

1. Read all SAS programs into a list named '*saspgm*'. Create an empty list to iterate later. Determine the desired position where the SAS code should be appended.
The program flow is like Application 2.

2. Use the metadata to write SAS syntax text.
The following R code uses the metadata 'adam0spec' for the ADaM datasets to generate SAS syntax text that defines dataset attributes and the order of variables in each of the ADaM datasets. Note that the regex (regular expression) line break character string (**\n**) creates line breaks in the syntax.

```
## read adam datasets information
adamvar <- haven::read_sas("/documents/specs/adam0spec.sas7bdat")
keepvar <- adamvar %>%
           mutate(type1=ifelse(TYPE %in% c('Char','Text'), "$", ''),
                  attrib1=paste0("\nattrib ",VARIABLE,
                                 ' label="',LABEL,
                                 '" length=',type1,LENGTH,
                                 ";") ) %>%
           group_by(DOMAIN) %>%
           mutate(VARLIST=paste(VARIABLE, collapse = " "),
                  ATTRIB=paste(attrib1, collapse = " ")) %>%
           . . .

keeplst <-as.list (keepvar$VARLIST)
attrlst <-as.list (keepvar$ATTRIB)

for (i in 1:length(aaa)){iii[i] <- paste0("\ndata lptda.",
                                  str_trim(datalst[i],"left"),";",
                                  "\n",str_trim(attrlst[i],"left"),
                                  "\nset ",gsub('=','',eee[i]),";",
                                  "\nkeep ",keeplst[i],";",
                                  "\nrun;",
                                  "\nproc sort;",
                                  "\nby ", sortlst[i], ";",
                                  "\nrun;")
                        }
```

3. Save the new SAS code back to the original file.
The program flow is like Application 2.

Output

```
data adsl_final_1;
length PLNTHPY prthpy dcsreasp $100.;
merge adsl_final(in=in1) _tuside _assign _livmeta _intmeta _prthpy _pgf _invsoc _astrap
by usubjid;
...
run;

data lptda.ADSL;
attrib STUDYID label="Study Identifier" length=$12;
attrib USUBJID label="Unique Subject Identifier" length=$30;
attrib SUBJID label="Subject Identifier for the Study" length=$10;
attrib SITEID label="Study Site Identifier" length=$20;
attrib SITENUM label="Study Site Number" length=$10;
attrib INVNAM label="Investigator Name" length=$200;
attrib AGE label="Age" length=8;
attrib AGEU label="Age Units" length=$6;
attrib AGEGR1 label="Pooled Age Group 1" length=$20;
attrib AGEGR1N label="Pooled Age Group 1 (N)" length=8;
. . .
set adsl_final_1;
keep STUDYID USUBJID SUBJID SITEID SITENUM INVNAM AGE AGEU AGEGR1 AGEGR1N AGEGR8 AGEGR8N
run;
proc sort;
by USUBJID;
run;
```

Display 10: adsl.sas - AFTER adding data attribute statements

CONCLUSION

Clinical trial deliverables involve many analysis datasets and numerous tables, listings, and plots. Daily tasks, such as reviewing logs and outputs, can be quite labor-intensive. Each time SAS programs are executed, several post-processing activities must be repeated, which places a significant burden on programmers. This paper demonstrates how the R language can import large-scale SAS files into a list, allowing for collective processing in just seconds.

Although there are existing SAS applications tailored for post-processing tasks, the techniques presented using R offer a distinct advantage in terms of processing speed. Given the considerable time savings illustrated through these examples, it is advantageous for the clinical programming team to investigate the application of similar R methodologies to other tasks in clinical trial studies.

REFERENCES

Proceedings (SAS)

- Gao, L. (2004). “Write SAS® Code to Generate Another SAS® Program A Dynamic Way to Get Your Data into SAS®” Proceedings of SUGI 29. https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/175-29.pdf
- Timusk, P. (2017). “Writing SAS® Code on the Fly Using SAS Code as Character Variables” Proceedings of SAS Global Forum 2017. https://support.sas.com/resources/papers/proceedings17/1436-2017.pdf
- Zhuo, Y. (2019). “End of Computing Chores with Automation: SAS® Techniques That Generate SAS® Codes” Proceedings of PharmaSUG 2019. https://www.pharmasug.org/proceedings/2019/BP/PharmaSUG-2019-BP-255.pdf

Proceedings (R)

- Mengelbier, M. (2024). “Adding The Missing Audit Trail to R” Proceedings of PharmaSUG 2024. https://www.lexjansen.com/pharmasug/2024/AP/PharmaSUG-2024-AP-424.pdf
- Huang, C.-H. (2023). “A Light-Weight Framework to Manage Programs and Run All the TLFs in R” Proceedings of PharmaSUG 2023. https://www.lexjansen.com/pharmasug/2023/SD/PharmaSUG-2023-SD-185.pdf

ACKNOWLEDGMENTS

The authors would like to thank Lily Zhang, Christine Teng, Xinhui Zhang, and Xiaohui Wang from Merck & Co., Inc., Rahway, NJ, USA, for their advice on this paper/presentation.

CONTACT INFORMATION

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

Huei-Ling Chen, Ph.D.
Merck & Co., Inc., Rahway, NJ, USA
e-mail: huei-ling_chen@merck.com

TRADEMARK