PharmaSUG 2012 - Paper AD04

How Readable and Comprehensible Is a SAS® Program? A Programmatic Approach to Getting an Insight into a Program

Rajesh Lal, Experis, Portage, MI, USA
Raghavender Ranga, Vertex, Cambridge, MA, USA

ABSTRACT

In any programming language, there are general guidelines for writing "better" programs. Well-written programs are easy to re-use, modify, and comprehend. SAS programmers commonly have guidelines for indentation, program headers, comments, dead code avoidance, efficient coding, etc.

It would be great if we could gauge a SAS program's quality by quantifying various factors and generalize for individual programming styles.

This paper aims to quantify various qualitative characteristics of a SAS program using Perl regular expressions, and provide insight into a SAS program. This utility, when used across a large project, can serve as a tool to gauge the quality of programs and help the project lead take any corrective measures to ensure that SAS programs are well written, easily comprehensible, well documented and efficient.

INTRODUCTION

One of the key stages of the software development life cycle is the development of the code and the fundamental principle while developing the code is using the best programming practices. A plethora of information is available on the guidelines of using best programming practices and these guidelines can be applied to any programming language. Following these guidelines during program development ensures readability of the programs, ease of re-use across the projects, minimal errors during development, minimal effort for future enhancements, and increased efficiency in the program. Not following these guidelines during development stages usually leads to error prone programs, limited use of the program for any further enhancements, or inability to replicate across the projects because of its poor readability and comprehensibility.

In practice, not all the principles of these guidelines are applied to the programs. Although, every programmer strives to use these guidelines, often only some of them are followed or some of them get missed. In this paper we try to quantify some of these qualitative characteristics of a SAS program using SAS Perl regular expressions. The program is rated based upon these characteristics and is given a score.

OVERVIEW

Some characteristics of a good SAS program are listed below. The following sections describe in detail how the utility quantifies each of these characteristics.

- Program header
- Indentation
- Comments
- Data set naming convention
- Dead code (commented out code) avoidance
- Line size limitation
- Optimum code usage

The utility reads in the SAS program along with Pattern Definition Files (files which define SAS Perl regular expressions that are used to identify patterns in the input SAS program). The utility identifies and extracts single and multi-line comments, calculates comments expectancy, compares actual versus expected comments, calculates actual versus expected indentation, identifies dead code (commented out code) by searching for SAS keywords, compares the standard program header template with actual header, identifies code that may be optimized, searches for and identifies data set names that are not following standard conventions and generates a summary of the calculated information.

The flowchart shown in Figure 1 below summarizes the process flow when the utility we're presenting is run on a SAS program.

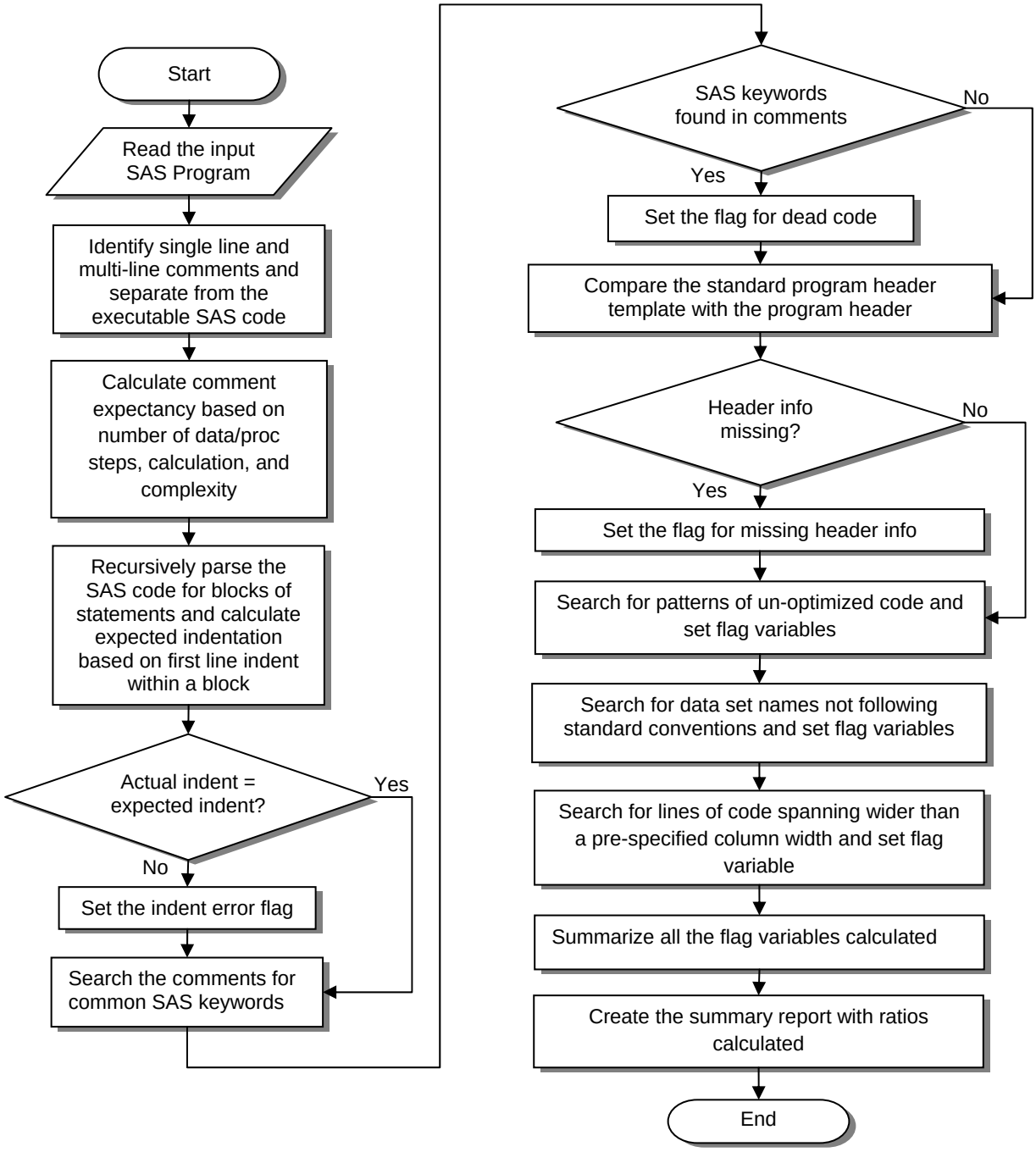


Figure 1. Flow chart for overall processing of the utility

When this utility is used across a large project, it can serve as a tool to gauge the quality of the programs. By looking at the scores provided by this utility, a project lead can take any corrective measures, if required, to ensure that SAS programs are well written, easily comprehensible, well documented and efficient.

Figure 2 shows how the utility reads multiple SAS programs from a project, reads in the Pattern Definition Files, processes the SAS programs and generates the reports.

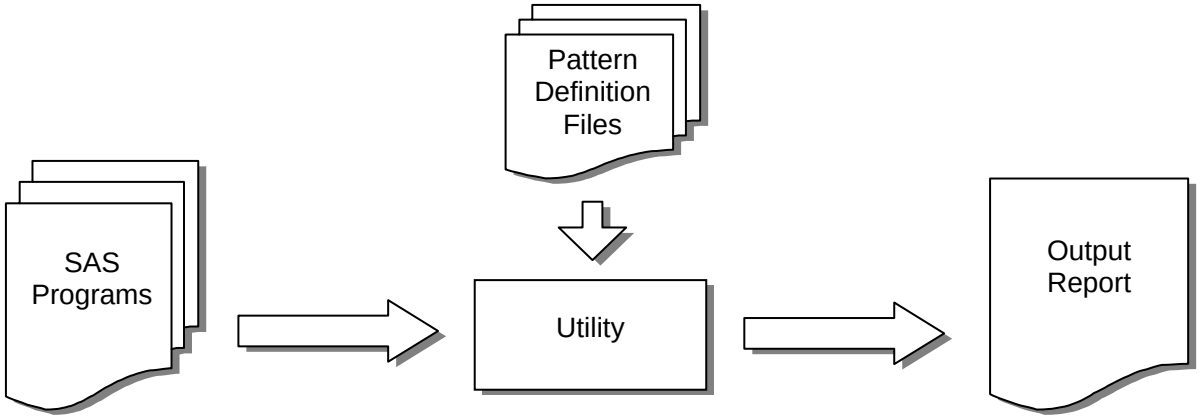


Figure 2. Overall data flow diagram of the utility when used on a large project

ASSUMPTIONS AND LIMITATIONS

For checking the program header completion, the utility expects a blank standard header template as an input, and that every program should use the same standard header.

Quantification of the comments has been done based on only a certain number of factors, including the number of DATA or PROC steps, important decision making or logical SAS statements, complexity of calculations etc. More factors can be identified and added in the Pattern Definition Files.

Indentation algorithm follows a scalable and recursive approach to detect actual versus expected indentation levels and adapts to individual preferences of number of spaces used for indentation. It only works for a limited set of commonly used SAS statement blocks, which can be expanded by adding more patterns to the Pattern Definition Files.

Code optimization is gauged using some of the basic SAS language patterns and other patterns can be identified and plugged into the utility to expand the scope.

To simplify the flow chart in Figure 3 and Figure 4, it is assumed that there is at least one comment and one DATA/PROC step in the input SAS program.

PROGRAM HEADER

Every program should start with a program header section. The goal of this component of the utility is to check the completeness of the program header. Program header labels are passed into the utility and using these labels as the starting point the utility checks if each of these individual sections is completed. The header is divided into two parts: mandatory and optional sections. In the mandatory section, utility expects all the fields are completed whereas in the optional section individual fields can either be completed with the relevant text or entered with 'NA' where applicable. Program Name, Author, SAS version and Purpose are classified as mandatory fields and Program Dependency, History, Input and Output files are treated as optional fields. The utility counts the empty fields and the total number of fields present in the header section using regular expressions. If the below header is passed into the utility, it counts the total number of non-missing fields. In the example header below, there are 6 non-missing fields and 2 blank fields. So the value of HDCNT is 6 and since HDCNT is not equal to the total number of fields (TOTCNT = 8) the utility calculates percent of non-missing fields in the header section.

```
/*************************************************************************
Program: example.sas
Programmer: Author Name
Creation Date: 2011-11-14
Purpose: Program for table report
Input: All Input parameter need to run the program if none put NA
Output: Output Report
Program Dependency:
History:
*************************************************************************/
```

The following code snippet calculates the number of missing and non-missing fields:

```
data header;
  set pgm;
  retain hdcnt 0 totcnt 8;
```

```
  if prxmatch("/(Program) *(:) *(\w\W)*/",txt) then hdcnt +1;
  if prxmatch("/(Programmer) *(:) *(\w\W)*/",txt) then hdcnt +1;
  if prxmatch("/(Creation Date) *(:)([\d\D])*/",txt) then hdcnt +1;
  **** repeat for Purpose, Input, output, etc.;
  percomp = (hdcnt/totcnt)*100;
run;
```

Display 1 shows how the utility calculates the number of non-missing fields.

	txt	hdcnt	totcnt
1	/**********	0	8
2	Program: example.sas	1	8
3		1	8
4	Programmer: Author Name	2	8
5		2	8
6	Creation Date: 2011-10-30	3	8
7		3	8
8	Purpose: Program for table report	4	8
9		4	8
10	Input: All Input parameter need to run the program if non put NA	5	8
11		5	8
12	Output: Output Report	6	8
13		6	8
14	Program Dependency:	6	8
15		6	8
16	History:	6	8
17	**********	6	8
18		6	8

Display 1. Display showing recursive indentation calculations

INDENTATION

By using proper indentation in a SAS program, the programmer ensures that various blocks in the program are easy to distinguish and it's easy to follow the program logic. The utility uses a scalable and recursive algorithm described in Figure 3 to calculate the actual and expected indentation and adapts to the individuals' preferences for number of spaces used for indentations. For each recursion, a 'level starter' is identified along with an 'end of statement'. Depending upon whether or not the 'end of statement' is on the same line as the 'level starter' further lines are checked to see the presence of 'end of statement'. Once 'end of statement' is reached, the starting position of next statement is noted and is compared with starting position of the current level. Further statements within the same block are checked for indentation level equal to that of the first statement. If any of the statements are not indented as expected, a flag is set. The above logic is used recursively for multiple levels of indentation. The flag is used to gauge the overall indentation compliance of the program. The flow chart in Figure 3 below explains the process.

Below is an example of Pattern Definition File for indentation:

```
%let rxst1=/(data )[\w\W]*/; *** pattern for DATA step;
%let rxct1=%str(/(data )[\w\W]*;/);
%let rxes1=%str(/[\w\W]*;/);
%let rxen1=%str(/\s*run/);

%let rxst2=/(proc )[\w\W]*/; *** pattern for PROC step;
%let rxct2=%str(/(proc )[\w\W]*;/);
%let rxes2=%str(/[\w\W]*;/);
%let rxen2=%str(/run|quit/);

%let rxst3=%str(/(if )[\w\W]*do;/); *** pattern for IF statement with DO-END;
%let rxct3=%str(/(if )[\w\W]*do;/);
%let rxes3=%str(/[\w\W]*;/);
%let rxen3=%str(/end/);
```

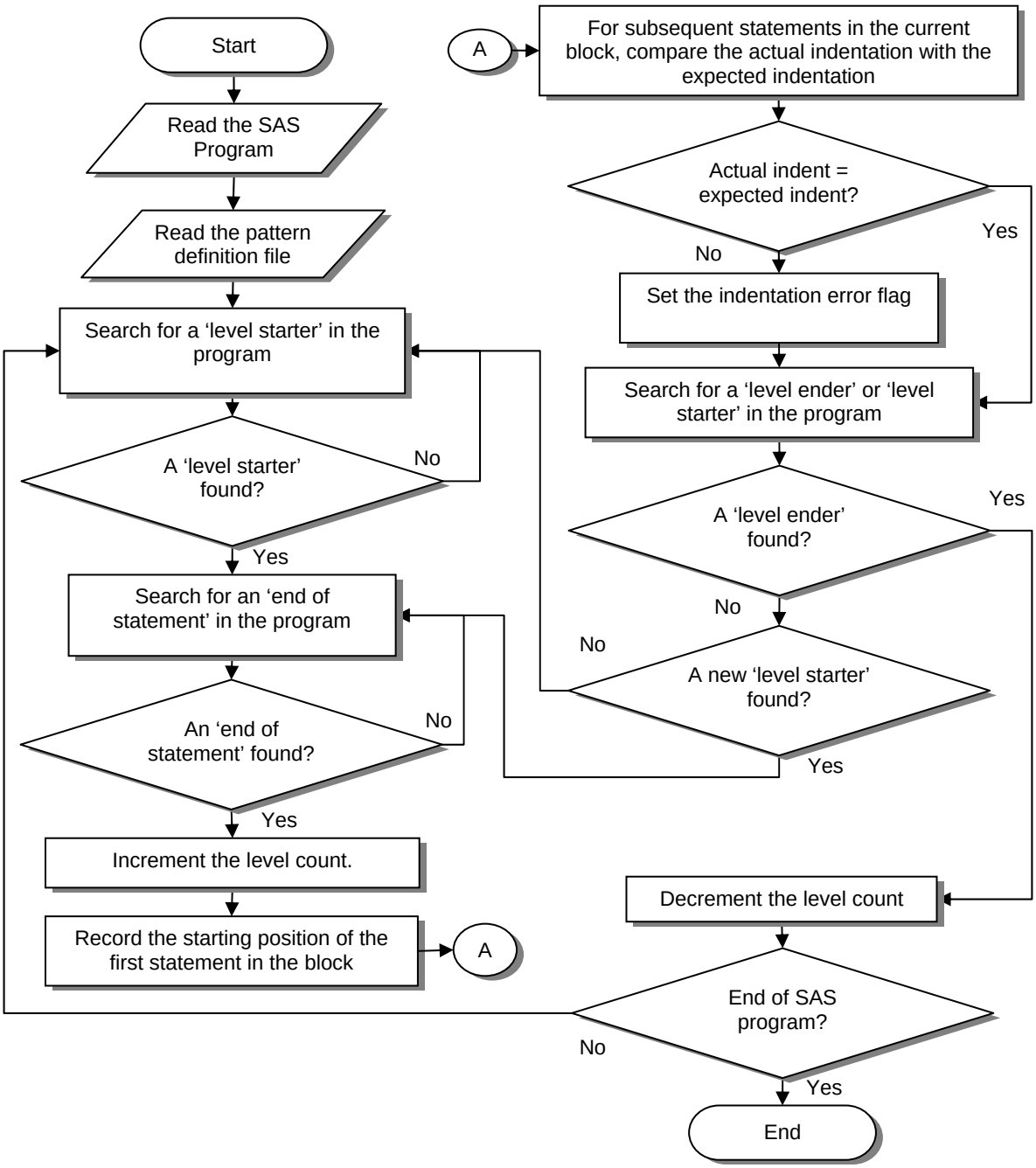


Figure 3. Flow chart of the utility for indentation calculations

Below is a snippet of SAS code that calculates actual and expected indentation levels recursively.

```
*** read in the SAS program file ***;
data pgm2;
   set pgm1(keep=txt_ where=(not missing(txt_)));
   *** flags reqd for each indentation level;
   array lvl_indtA[10] lvlinda01-lvlinda10;
   array lvl_indtB[10] lvlindb01-lvlindb10;
   retain lvl_cnt 1 stmt_cont 0 _1st indx lvlinda01-lvlinda10 lvlindb01-lvlindb10;

   if _1st then do; *** if this is the first statement after level start;
      lvl_indtB(lvl_cnt) = prxmatch("/\S/",txt_); *** calculate user indent;
     _1st = 0;
   end;
```

```
      *** if a statement is not already continuing, parse for a level starter;
      if not stmt_cont then do i=1 to &nst;
         rxst = symget("rxst"||strip(put(i,8.)));
         rxct = symget("rxct"||strip(put(i,8.)));
         *** if found, mark the start of a level ;
         if prxmatch(rxst,txt_) then do;
            indx=i;
            lvl_indtA(lvl_cnt)=prxmatch(rxst,txt_);
            *** if the statement does not end on the same line, mark it as continuing;
            if not prxmatch(rxct,txt_) then stmt_cont = 1;
            else do;
               *** otherwise, increment the level counter and set flag so that the;
               *** user given indent for the next statement in the block can be;
               *** caclulated;
               lvl_cnt = lvl_cnt + 1;
               _1st = 1;
            end;
         end;
      end;
      *** if a statement is continuing and end of that statement is found;
      else if prxmatch(symget("rxes"||strip(put(indx,8.))),txt_) then do;
         *** reset the statement continuing flag;
         stmt_cont = 0;
         *** increment the level counter and set flag so that the user given;
         *** indent for the next statement in the block can be caclulated;
         lvl_cnt = lvl_cnt + 1;
         _1st = 1;
      end;

      *** if an end of statement is found;
      if lvl_cnt and not stmt_cont and
         prxmatch(symget("rxen"||strip(put(indx,8.))),txt_) then do;
         *** decrement the level counter;
         lvl_cnt = lvl_cnt - 1;
      end;

      *** put a description of the current lvl e.g. DATA/PROC step, IF statement etc;
      lvl_starter = put(indx,lvls.);

      *** calculate indentation errors;
      if not _1st and lvl_indtB(lvl_cnt) ne prxmatch("/\S/",txt_) then indt_err = 1;
   run;
```

For example, if the following SAS code in Display 2 is passed into the utility, it calculates line 6 as indentation error. After calculating the flag indt_err for each line, overall indentation compliance is calculated as ratio of lines of code having indentation as expected to the total lines of code. So, a SAS program containing the following DATA step would have an indentation compliance ratio of 9/10 = 90%.

	txt_	lvl_cnt	stmt_cont	_1st	indx	lvl_starter	indt_err
1	data demo;	2	0	1	1	Data Step	.
2	set antcrt.demo;	2	0	0	1	Data Step	.
3	length bsacat_ $20;	2	0	0	1	Data Step	.
4	if trtacd eq 5 then do;	3	0	1	3	If stmt with do;	.
5	trtacd_ = 2;	3	0	0	3	If stmt with do;	.
6	trta_ = "Treatment Arm A";	3	0	0	3	If stmt with do;	1
7	end;	2	0	0	3	If stmt with do;	.
8	if bsacat = 0 then bsacat_ = "<Median";	2	0	0	3	If stmt with do;	.
9	else if bsacat = 1 then bsacat_ = ">=Median";	2	0	0	3	If stmt with do;	.
10	run;	2	0	0	3	If stmt with do;	.

Display 2. Display showing recursive indentation calculations

COMMENTS

Comments are required for comprehending the flow of a program. The utility calculates the 'comment expectation' based on a number of factors, including the number of DATA or PROC steps, important decision making or logical SAS statements, complexity of calculations etc., which is read in as a list of pre-defined patterns from a Pattern Definition File. This comment expectation is compared with actual comments and a compliance flag is set. Using this flag the overall actual versus expected comments ratio is calculated.

Display 3 shows comments calculations for a sample piece of code. Figure 4 shows the process flow for comment calculations.

Below is an example of a Pattern Definition File, using DATA step, PROC step, IF statement, assignment statement, CASE statement as key steps in a SAS program for which a comment is expected. More patterns can be identified and added to this list.

```
%let ptrn1=/(data )[\w\W]*/; * start of a DATA step;
%let ptrn2=/(proc )[\w\W]*/; * start of a PROC step;
%let ptrn3=%str(/(if )[\w\W]*;/); * IF statement;
%let ptrn4=%str(/[\w\W]*(=)[\w\W]*;/); * assignment statement;
%let ptrn5=%str(/(case )[\w\W]*/); * CASE statement;
```

	txt	cmnts	cntdstar	cntdslash	
1		0	0	0	
2	/* this is a multiline	0	0	1	
3	comment example	0	0	1	
4	*/	1	0	0	
5		1	0	0	
6	data demo;	1	0	0	da
7	set antcrt.demo;	1	0	0	se
8	*** this is a single line comment that ends on the same line;	2	0	0	
9	length bsacat_ $20;	2	0	0	
10	*** this is a single line comment that spans	2	1	0	
11	multiple lines;	3	0	0	
12	if trtacd eq 5 then do;	3	0	0	if t
13	trtacd_ = 2;	3	0	0	
14	trta_ = "Treatment Arm A";	3	0	0	
15	end;	3	0	0	en
16		3	0	0	
17	/*else if trtacd in (1 3 7 9) then do;	3	0	1	
18	trtacd_ = 1;	3	0	1	
19	trta_ = "Treatment Arm B";	3	0	1	
20	end; <<<<< this block is dead code*/	4	0	0	
21	if bsacat = 0 then bsacat_ = "<Median";	4	0	0	
22	else if bsacat = 1 then bsacat_ = ">=Median";	4	0	0	
23	run;	4	0	0	ru
24		4	0	0	

Display 3. Display showing comment calculations

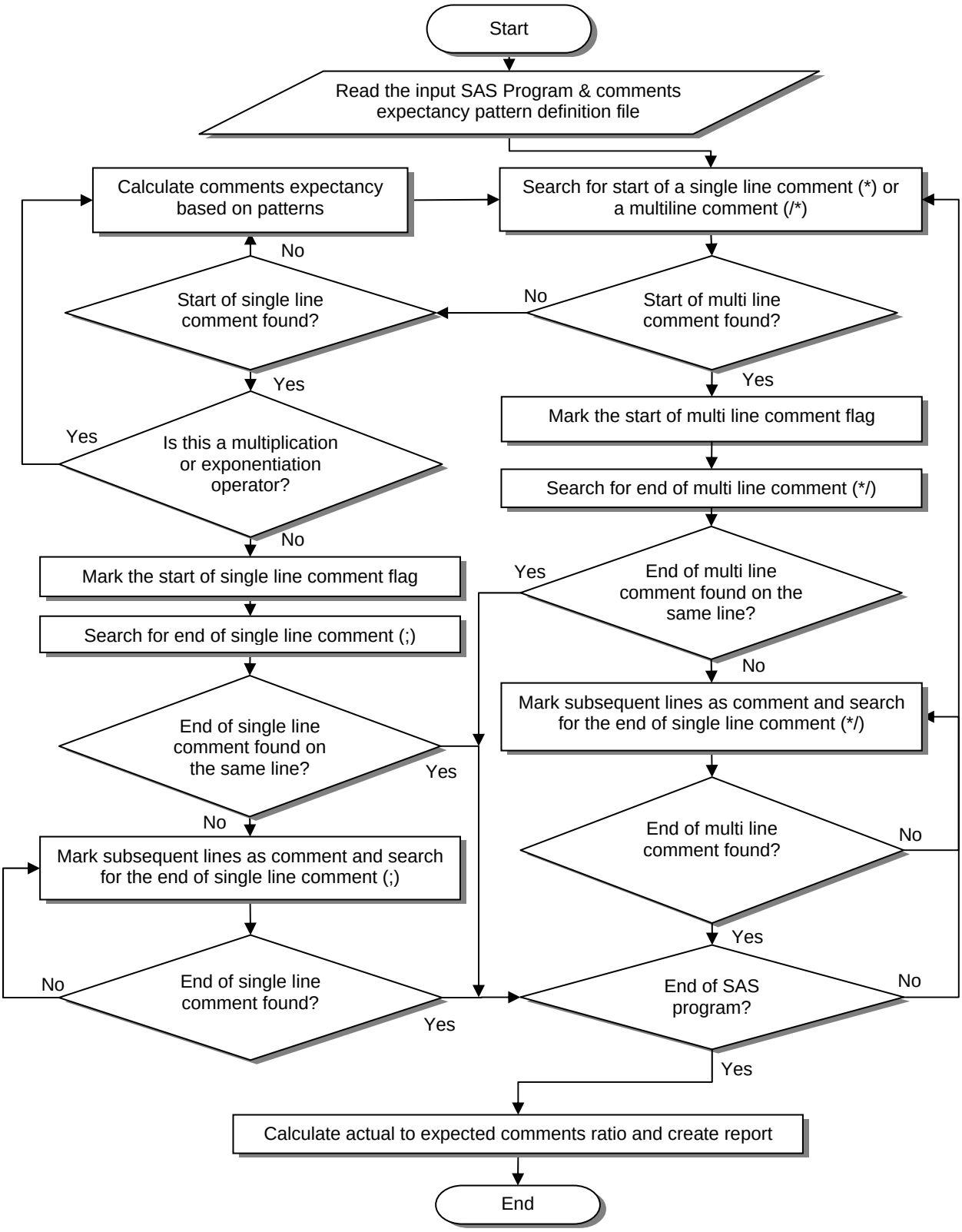


Figure 4. Flowchart showing process flow for comment calculations

DATA SET NAMING CONVENTION

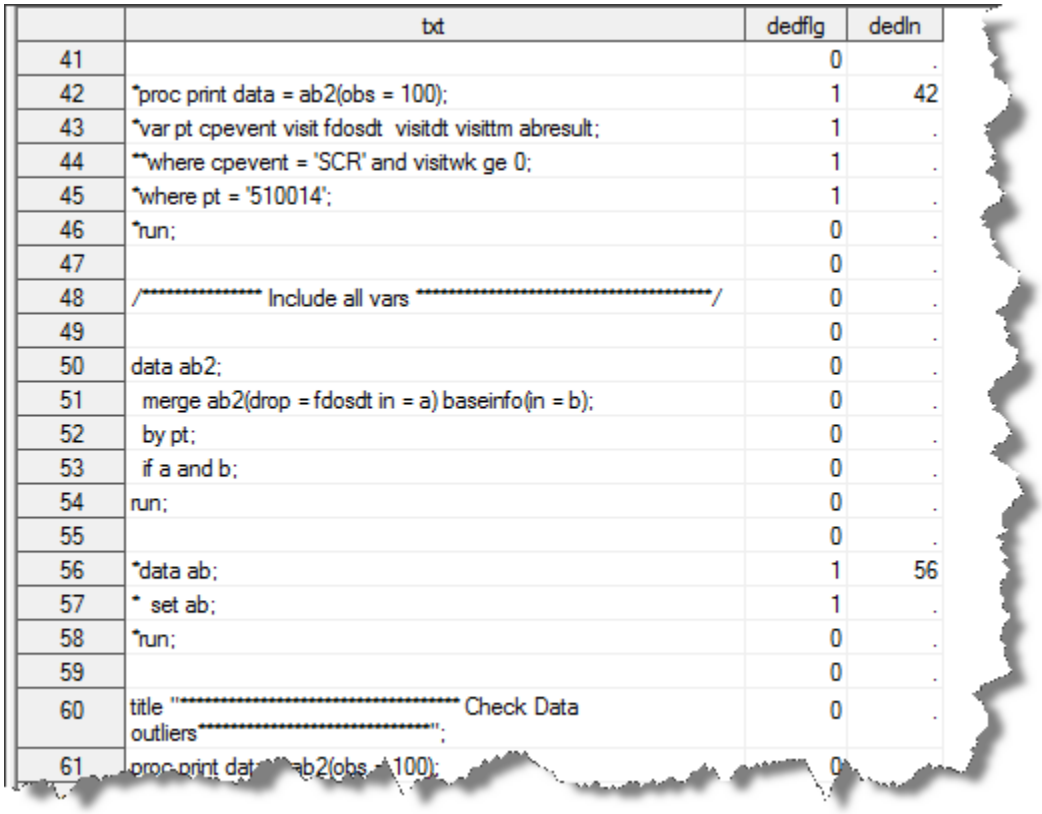
One of the important considerations in naming a data set or a variable is to fully represent the entity. Names that are too short do not convey any meaning. This utility counts the number of instances where the data set names are created using single character or single character followed by a number of digits or data set names larger than 8 characters long. Although variables names larger than 8 characters may be just fine in some industries, they are not expected to be more than 8 characters in Clinical/Pharmaceuticals industries as dictated by CDISC standards or FDA requirements. This utility checks for data set naming patterns like X, X1 etc. Display 4 below shows the screen shot of the output of the utility when run to check the length of data set names. In the display, the variable “txt” stores each line of the program as individual record and variable “DTNM” extracts and stores every instance of the output data set name. Once, the data set names are extracted, algorithm counts the instances which do not meet the above mentioned criteria of length of data set name. In this particular example, “DTNMFLG” variable has a value of 2 that means there are two data set names which do not meet the above described criteria.

	txt	procflg	dtnmflg	dtnm
69	proc sort data = ab out =ab1234567;	1	1	ab1234567
70	by pt;	1	1	
71	run;	0	1	
72		0	1	
73	proc freq;	1	1	
74	tables period cpevent*visitwk*analysis / list missing out = t;	1	2	t
75	run;	0	2	
76		0	2	
77	proc means data = tansbi noprint;	1	2	
78	var pbflg;	1	2	
79	output out = mbflg mean = meanfl sum = sumfl;	1	2	mbflg
80	run;	0	2	
81		0	2	
82	/**********Derive all the derived vars Identify positive subjects ********/	0	2	
83		0	2	
84	%macro abctrans(var=,ds=,typ=);	0	2	
85		0	2	
86	data ab&ds;	0	2	ab&ds
87	set ab2;	0	2	
88	where strip(upcase(abasaytp))="&typ";	0	2	
89	run;	0	2	

Display 4. Screen shot output for checking data set naming convention compliance

DEAD CODE (COMMENTED OUT CODE) AVOIDANCE

Inside the single or multi-line comments, utility checks for any SAS keywords and tries to identify if it is dead code or commented out code. If the commented line is identified as dead code, a flag is set and this flag is used to gauge the overall occurrences of dead code in the SAS program. In the Display 5 below, “TXT” variable contains each line of the input program as record on which the utility is run. When this utility is run on the program, “DEDLN” variable identifies the starting line number of a block of DATA or PROC step which are commented or line number of commented individual statements. Thus, by counting the number of non missing values in the in the “DEDLN” variable the utility gives the overall occurrences of the dead code in the program with associated line numbers.

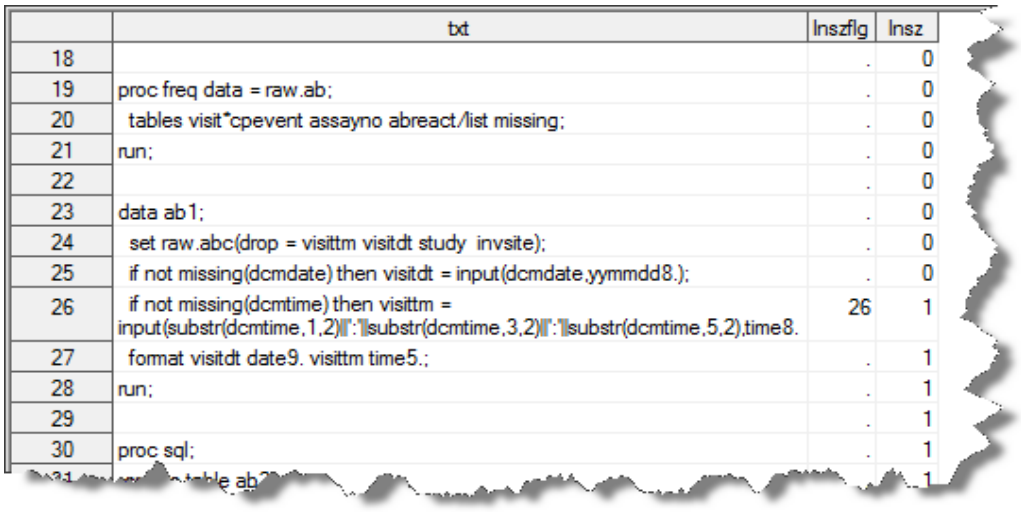


	txt	dedflg	dedln
41		0	.
42	*proc print data = ab2(obs = 100);	1	42
43	*var pt cpevent visit fdosdt visitdt visittm abresult;	1	.
44	**where cpevent = 'SCR' and visitwk ge 0;	1	.
45	*where pt = '510014';	1	.
46	*run;	0	.
47		0	.
48	/********** Include all vars **********/	0	.
49		0	.
50	data ab2;	0	.
51	merge ab2(drop = fdosdt in = a) baseinfo(in = b);	0	.
52	by pt;	0	.
53	if a and b;	0	.
54	run;	0	.
55		0	.
56	*data ab;	1	56
57	* set ab;	1	.
58	*run;	0	.
59		0	.
60	title "********** Check Data outliers**********";	0	.
61	proc print dat ab2(obs = 100);	0	

Display 5. screen shot for identifying dead code in the program

LINE SIZE LIMITATION

A program is difficult to read and comprehend if it spans over more columns than are readable on a screen without repeatedly scrolling horizontally, unless the wide line is a part of 'DATALINES' or otherwise necessary to be on the same line and cannot be broken down to multiple lines. The utility checks to see if any of the qualifying statements are longer than a pre-defined line size, and if so, flags it as a wide line and then gauges the presence of wide lines in the program. For example, in the Display 6 below, the utility reads the input program and stores each line of program in the "TXT" variable. "LNSZFLG" variable identifies the line number of the input program which is wider than the predefined line size and by counting the number of the non-missing values in the "LNSZFLG" variable the utility gives the total number of instances where the input program is wider than the predefined line size.



	txt	lnszflg	lnsz
18		.	0
19	proc freq data = raw.ab;	.	0
20	tables visit*cpevent assayno abreact/list missing;	.	0
21	run;	.	0
22		.	0
23	data ab1;	.	0
24	set raw.abc(drop = visittm visitdt study invsite);	.	0
25	if not missing(dcmdate) then visitdt = input(dcmdate,yymmdd8.);	.	0
26	if not missing(dcmtime) then visittm = input(substr(dcmtime,1,2)||':'||substr(dcmtime,3,2)||':'||substr(dcmtime,5,2),time8.	26	1
27	format visitdt date9. visittm time5.;	.	1
28	run;	.	1
29		.	1
30	proc sql;	.	1
31	table ab2	.	1

Display 6. screen shot for identifying lines beyond the predefined size

OPTIMUM CODE USAGE

In this feature, utility checks for some of the programming styles that would render some insight about the efficiency of the program. Although each programmer has his/her own style of coding, however, there are certain programming steps that are unnecessarily duplicated and if avoided, would decrease the processing time and will be more efficient. This utility does not try to tackle the logic but only checks for some of the basic programming techniques which are deemed unnecessary. The utility looks for patterns as shown below in the example 1, a DATA step is followed by a SORT procedure without any processing done in the DATA step. The DATA step could have been avoided by using PROC SORT with OUT= option. Also, as shown in example 2, the utility checks the DATA steps where 'IF' statement is used and can be replaced with the 'WHERE' statement. The possibilities are endless if we want to characterize the program efficiency.

```
***example 1;
data new;
  set old;
run;
proc sort data = new;
  by var1 var2;
run;

***example 2;
data new;
  set old;
  if var1 > 0;
run;
```

UTILITY SAMPLE OUTPUT

```
Report for program test.sas:
Program header compliance: 6/8 (75%)
Program Indentation: 9/10 (90%)
Comments: 4/7 (57%)
Data set naming convention incompliance: 4
Dead code instances: 3
Line size incompliance: 1
Un-optimized code instances: 1
```

FUTURE DEVELOPMENTS

Additional patterns can be plugged in to the utility for indentation checking, commented out code recognition, expectancy of comments and program efficiency.

CONCLUSION

Once we calculate the scores from the individual sections, the utility gives the summary report for each program for which the utility was run. Quantification of various qualitative characteristics of a SAS program provides an important insight into the program. When this utility is used across a large project, it can serve as a tool to monitor the quality of programs and take any corrective actions, if required. It can help ensure that SAS programs are well written, easily comprehensible, well documented and efficient.

REFERENCES

- SAS Online Documentation
- An Introduction to Perl Regular Expressions in SAS 9 by Ron Cody, Robert Wood Johnson Medical School, Piscataway, NJ – SUGI 29
- Steve McConnell, Code Complete, 1993, Microsoft Press.

ACKNOWLEDGMENTS

We would like to thank Chuck Kincaid, Scott Davis, Jack Fuller, Dave Polus and Brian Varney for reviewing this paper and providing valuable comments.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. For any questions or sample codes described in this paper, please contact the authors at:

Name: Rajesh Lal
Enterprise: Experis
Address: 5220 Lovers Lane, STE 200
City, State ZIP: Portage, MI 49002
Work Phone: 269-553-5147
Fax: 269-553-5101
E-mail: rajesh.lal@experis.com
Web: www.experis.com

Name: Raghavender Ranga
Enterprise: Vertex Pharmaceuticals
Address: 130 Waverly Street
City, State ZIP: Cambridge, MA 02139
Work Phone: 617-444-7639
Fax: 617-444-6766
E-mail: raghavender_ranga@vrtx.com
Web: www.vrtx.com