

SASLint: A SAS® Program Checker

Igor Khorlo, Syneos Health™

ABSTRACT

Linters are programs that carry out static source code analysis, detecting certain bugs, coding rule deviations, and other issues. Linters also detect redundant or error-prone constructs that are nevertheless, strictly speaking, legal. Potential performance optimizations can also be checked by linters. This paper covers creating a modular linter for the SAS® language consisting of a parser module, an analysis module that includes a list of rules, and a reporting module that displays issues found. It is possible to include and exclude rules, as well as develop your own rules, all of which makes the linter very flexible for any team with its own list of requirements regarding the source code and programming standards. The parser for SAS language grammar is based on the ANTLR Java parser. The tool is written in Java and SAS, which is why it can be integrated into any SAS environment.

INTRODUCTION

So what is a linter? A linter is a program which parses your source code and builds a tree of objects which reflects what is written in the source code (which statements are used, how many spaces are used for indentation, etc). This is called an abstract syntax tree (later AST). And afterwards it analyses the AST for a set of rules. So, the main idea is that the linter doesn't run the input program, it directly analyses how the program has been written. In other words, it performs a static analysis of the source code.

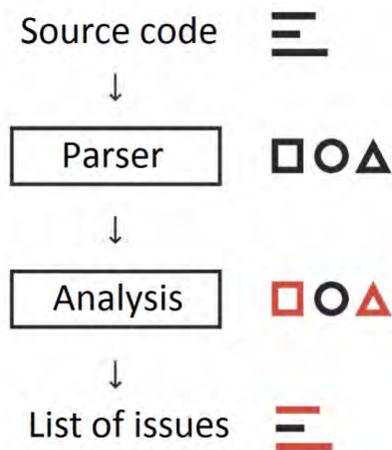


Figure 1: Linter architecture

OK. But what benefits does this bring, you ask. Let's consider several real-life examples.

PROGRAMMING RULES

I guess most people here have their own team rules which are written out in some document. The main disadvantage is that you have to check all these rules manually and force yourself to comply with them otherwise someone else may find a deviation from the rules in your code and I would say that is a situation to be avoided. Linters can fix this. Let's make machines do the work. Migrate all your rules to algorithms which will detect rule deviations in your code, and it's done. Now the machine tells you what is wrong with your program. And this saves much time and nothing will ever be missed.

For example you have a reporting framework for graphics in your company and you must always use it, otherwise deviations from this rule must be marked with a comment in your program and well explained and argued. We can write a check for this: if a program creates a graphic and contains no calls of macros from the graphical macro library, then it must contain a comment in a specific place with, say, a minimum length of 200 explaining why.

DANGEROUS BUT LEGAL CODE

Suppose we have the following condition:

where `pcng < 90`;

The issue here is that the condition may work now as expected because there are no missings in the `pcng` variable, but tomorrow, when new data arrives, it will work incorrectly without generating any error or warning and you may spend several hours identifying the issue. The safe way is:

where `. < pcng < 90`;

Writing a check for this sort of situation will definitely save you time and nerves in the future as well as protect you from errors.

STYLEGUIDE

Let's consider two pieces of code:

Poorly aligned:

```
DATA CLASS CLASS18;
SET SASHELP.CLASS;BMI=WEIGHT/HEIGHT**2;
IF AGE > 18 THEN DO; OUTPUT CLASS18; END;
ELSE OUTPUT CLASS;
PROC FREQ DATA = CLASS18;
TABLES BMI;
```

and well-aligned:

```
data class class18;
  set sashelp.class;
  bmi = weight / height ** 2;
  if age > 18 then do;
    output class18;
  end;
  else do;
    output class;
  end;
run;

proc freq data = class18;
  tables bmi;
run;
```

I think most of you will agree that the second one looks better. It is even hard to understand what is going on in the first piece of code. And we can also force users to correct their indentation as this will be reported by the linter.

But if you don't need it you can disable this rule or modify it for your needs.

KNOWLEDGE SHARING PROBLEM

Let's consider that you have several teams in your company. You faced a situation in your code and want to share this *lesson learned* with others. Of course, you can send an email to everyone and even if it was read most people would forget about this quickly or even have no time to read it. Linters can help here too. You can have a global configuration file for your linter which will be sharable across all teams. If you faced a *lesson learned* and you want to share it and warn other people, write a rule for it and it will automatically be distributed across all teams. So, you can use this global config for knowledge sharing.

The same thing concerns SAS pitfalls, dangerous code, and similar sort of things which you can find in many papers. Unfortunately, the reality is that you cannot keep all these things in mind, but having a tool which will automatically check your source code will resolve this problem — we should outsource to machines this boring work.

OPTIMIZATIONS

During my programming experience, I've seen many strange and inexplicable things from a logical point of view. Some of them are:

- Sorting a dataset without need to do so.
- Changing dataset attributes like variable labels, formats and informats using DATA step. However, this can be done using the DATASETS procedure without wasting system resources rewriting a whole dataset.
- Using IF subset statement when it is possible to use WHERE.
- One more tip about performance and the SQL procedure using DICTIONARY Tables. You can use a WHERE clause to help restrict which libraries are searched. However, the WHERE clause will not process most function calls such as UPCASE. For example, if where upcase(libname) = 'WORK' is used, the UPCASE function prevents the WHERE clause from optimizing this condition. All libraries assigned within the SAS session are searched. Searching all the libraries could cause an unexpected increase in search time, depending on the number of libraries assigned within the SAS session. All librefs and SAS table names are stored in upper case. If you supply values for LIBNAME and MEMNAME in upper case, and you remove the UPCASE function, the WHERE clause will be optimized and performance will be improved. In the previous example, the code would be changed to where libname='WORK'. And you will be surprised that even many papers contain non-optimized upcase(libname) or similar WHERE clause with a SAS function. More information about this can be found in [SAS® 9.4 SQL Procedure User's Guide, Fourth Edition](#).

The good news is that this too can be detected by linters and even more — corrected using AST transformations.

ANTLR INTRODUCTION

The main goal of this section is to give a general overview of ANTLR's capabilities and to explore the language application architecture. Once we have the big picture, we'll continue to build an ANTLR grammar for SAS.

WHAT IS ANTLR AND HOW DOES IT HELP IN BUILDING A LINTER?

What is ANTLR? - ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar perspective, ANTLR generates a parser that can build and walk parse trees.

Programs that recognize languages are called *parsers* or *syntax analyzers*. *Syntax* refers to the rules governing language membership. A grammar is just a set of rules, each one expressing the structure of a

phrase. The ANTLR tool translates grammars to parsers that look remarkably similar to what an experienced programmer might build by hand.

The process of grouping characters into words or symbols (tokens) is called lexical analysis or simply tokenizing. We call a program that tokenizes the input a lexer. The lexer can group related tokens into token classes, or token types, such as INT (integers), ID (identifiers), FLOAT (floating-point numbers), and so on.

The second stage is the actual parser which feeds off these tokens to recognize the sentence structure, in this case an assignment statement. By default, ANTLR-generated parsers build a data structure called a parse tree or syntax tree that records how the parser recognized the structure of the input sentence and its component phrases. The following diagram illustrates the basic data flow of a language recognizer (Figure 2):

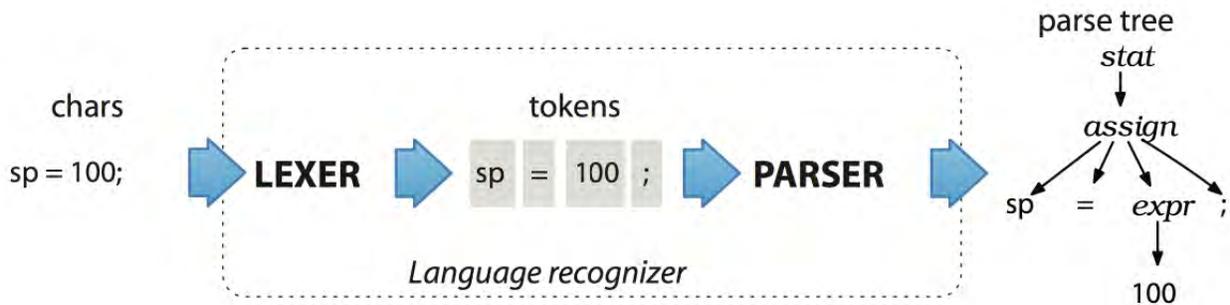


Figure 2: ANTLR produces AST

By producing a parse tree, a parser delivers a handy data structure to the rest of the application (in our case this is a linter) containing complete information about how the parser grouped the symbols into phrases. Trees are easy to process in subsequent steps and are a default data structure for this kind of task.

EXAMPLE OF PARSING SAS EXPRESSIONS

The first step towards building a linter application is to create a grammar that describes a language's syntactic rules (the set of valid sentences). In this section, we will discover ANTLR in more detail by building a grammar for a simplified, generic SAS expression. We'll build a grammar which will recognize expressions like:

- 3
- 'Tom'
- a + b
- a**2 - 4 * a * c
- 'foo' || "bar"
- rfendt - rfstdt + (rfendt >= rfstdt)
- x in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
- trim(month) || left(put(year, 8.))

For demonstration purposes we will simplify this and consider only integers as numeric literals, also we won't cover here all operator combinations supported by SAS, like:

- 1 =>: 2
- 1 ~ ne 2
- 1 not <=: 2
- 1 ~>=: 2
- ?-y = 2 with [CHARCODE](#) SAS System Option.

More detailed documentation for SAS expressions grammar can be found [here](#).

ANTLR grammar is a plain text with .g4 extension which contains a set of rules. First let's consider simple expression with +, -, *, / and with integers and variables names only (for a simplification purpose for a now). We will use a choice pattern when parser can choose from a set of rules:

```
// Expr.g4

grammar Expr;

expr: expr ( '*' | '/' ) expr
    | expr ( '+' | '-' ) expr
    | INT
    | ID
    | '(' expr ')'
    ;

ID : [A-Za-z_][A-Za-z_0-9]* ; // match identifiers
INT: [0-9]+ ;                // match integers
NL : '\r'? '\n' ;           // newlines
WS : [ \t\r\n\f]+ -> skip ; // toss out whitespace
```

Note the precedence here — the rule with multiplication and division comes first, so will be matched first.

The next step is to compile this grammar to a parser which is basically a set of Java classes, then compile the generated Java code to bytecode and then test the result:

```
antlr4 Expr.g4
javac *.java
echo 'a + b * (var3 / 2)' | grun Expr expr -gui
```

To make the above commands work, you need to create the aliases for antlr4 and grun and update the CLASSPATH for Java:

```
export CLASSPATH=".:usr/local/Cellar/antlr/4.7.1/antlr-4.7.1-complete.jar:$CLASSPATH"
alias antlr4='java -jar /usr/local/Cellar/antlr/4.7.1/antlr-4.7.1-complete.jar'
alias grun='java org.antlr.v4.gui.TestRig'
```

The path /usr/local/Cellar/antlr/4.7.1/antlr-4.7.1-complete.jar should be replaced by the path where your ANTLR download is located. For more details on how to set ANTLR up please see the documentation — [Getting Started with ANTLR v4](#).

The result is the following parse tree for the example expression `a + b * (var3 / 2)`:

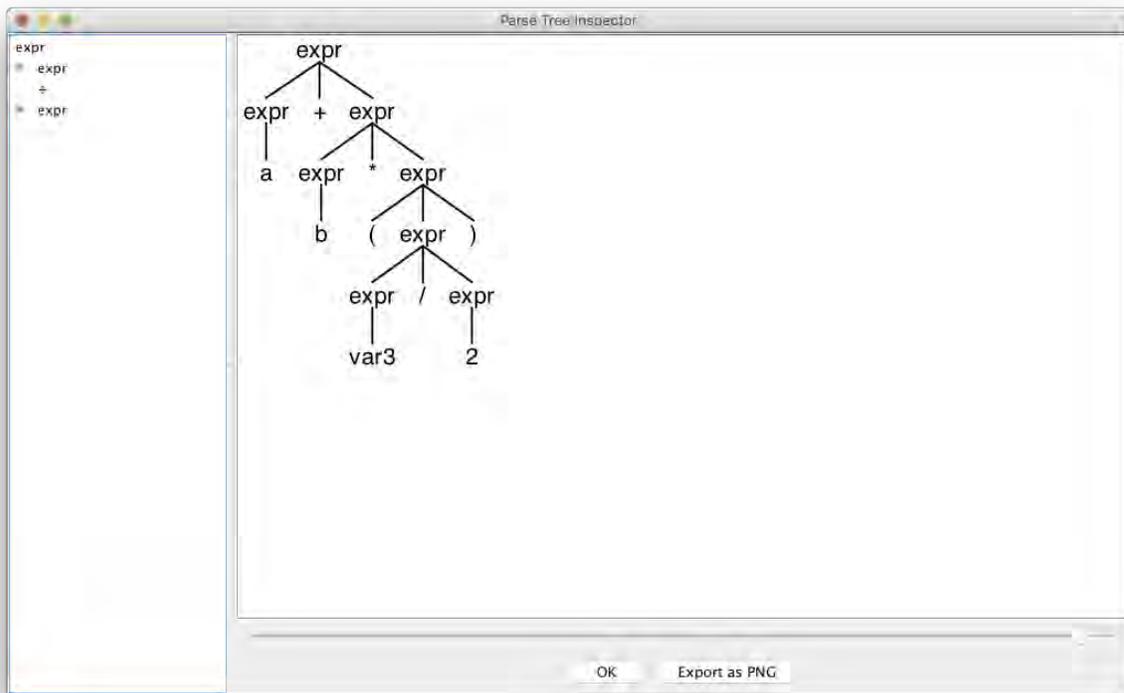


Figure 3: Parse tree

As you may already realize from the grammar, the parser rule for `expr` is recursive, therefore we can split the expression into a set of atomic rules and ANTLR will do its work and match the code accordingly.

Now let's add function support:

```
// Expr.g4
```

```
expr: ID '(' expr_list? ')'  
    | expr ('*' | '/') expr  
    | expr ('+' | '-') expr  
    | INT  
    | ID  
    | '(' expr ')'  
    ;
```

```
expr_list: expr ( ',' expr)* ;
```

The `expr_list` rule uses a sequence pattern, so it will match a sequence of expressions separated by a comma inside a function call. Here's the result of parsing `cat(str1, 42, cat(foo, bar))` (Figure 4):

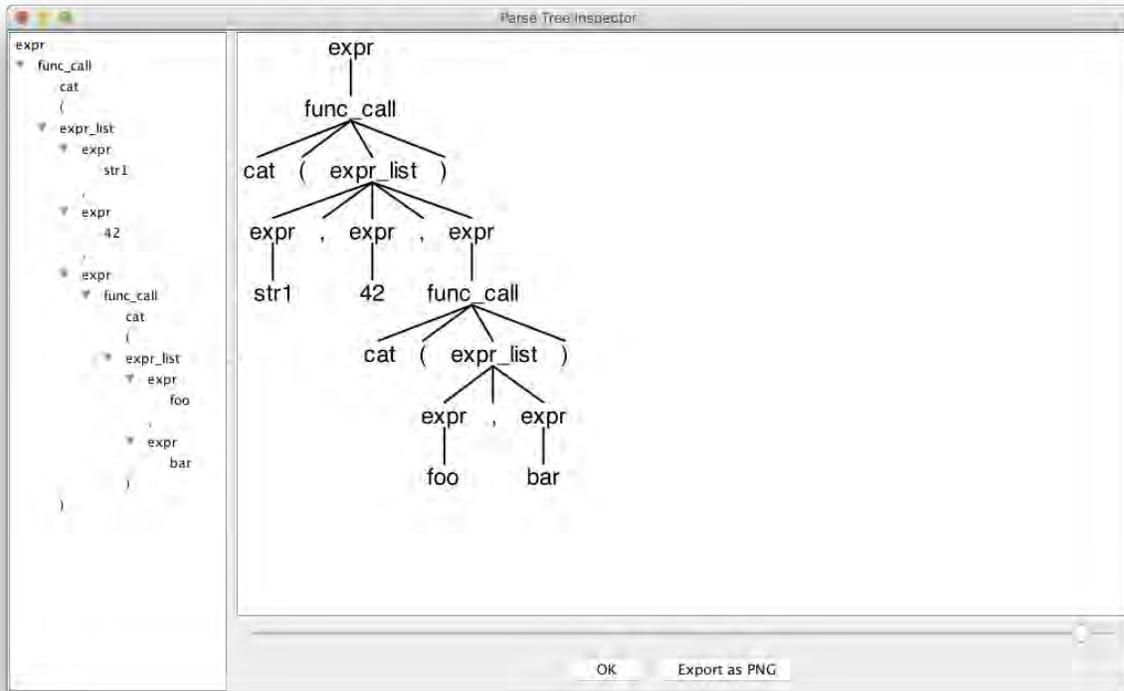


Figure 4: Parse tree

As you can see, ANTLR is a really powerful tool for parsing and building grammars.

In the following sections, we will consider how to use the generated parse tree, and how to extract useful information from static code.

SAS GRAMMAR

In this section we will consider several cases for using SAS grammar together with a walking method as provided by ANTLR for AST.

DATA FLOW GRAPH

I'm sure many of us have been faced with the problem of needing to become familiar with a complex program in a short amount of time. The program may be hard to navigate — it may be very big, or you cannot run it (data is not available, or it is legacy code). We will build a data flow graph for the program,

In our case we will use the listener pattern. First, we'll generate a graph using the language called [DOT](#).

DOT is a declarative language for describing graphs such as network diagrams, trees, or state machines. (DOT is a declarative language because we say what the graph connections are, not how to build the graph.) It's a generically useful graphing tool, particularly if you have a program that needs to generate images.

DOT code looks like:

```
digraph G {  
  a -> b  
  b -> c  
  c -> a  
}
```

Here's the resulting diagram created using the DOT visualizer, [graphviz](#):

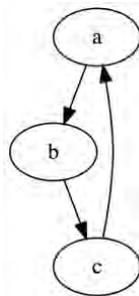


Figure 7: DOT example

To visualize a data flow, we need to read in a SAS program and create a DOT file (and then view it with graphviz). Our strategy is straightforward. When the parser finds a DATA step or a SORT procedure, our application will create two lists with input and output datasets. When the parser sees a dataset reference in input statements (SET/MERGE/PROC SORT DATA= option) or output statements (DATA statement/PROC SORT OUT= option) it will add it to the appropriate list. Upon exiting a DATA step or a procedure the application will dump all input and output datasets into a DOT output file in the following way: suppose we have datasets a, b at input and c at output. Then next lines will be generated to DOT file:

```
...  
a -> c  
b -> c  
...
```

We will override the enter and exit methods for GlobalStatement and enter methods for DatasetInput and DatasetOutput nodes:

```
// DataflowGraph.java
```

```
static class DataflowListener extends SASBaseListener {  
  Graph graph = new Graph();  
  Set<String> input = new OrderedHashSet<String>();  
  Set<String> output = new OrderedHashSet<String>();  
  
  public void enterGlobalStatement(SASParser.GlobalStatementContext ctx) {  
    input.clear();  
    output.clear();  
  }  
}
```

```

public void exitGlobalStatement(SASParser.GlobalStatementContext ctx) {
    for (String i: input) {
        graph.nodes.add(i);
        for (String o: output) {
            graph.edge(i, o);
        }
    }
}

public void enterDatasetOutput(SASParser.DatasetOutputContext ctx) {
    output.add(ctx.dataset().ID().toString().replaceAll("\\[|\\]", "").replaceAll(
ll(", ", "."));
}

public void enterDatasetInput(SASParser.DatasetInputContext ctx) {
    input.add(ctx.dataset().ID().toString().replaceAll("\\[|\\]", "").replaceAll(
l(", ", "."));
}
}

```

The entry point of our program will be a main method of the DataflowGraph class:

```
// DataflowGraph.java
```

```

public class DataflowGraph {
    ...
    public static void main(String[] args) throws Exception {
        // Read input file if present otherwise take use input
        String inputFile = null;
        if ( args.length > 0 ) inputFile = args[0];
        InputStream is = System.in;
        if ( inputFile!=null ) {
            is = new FileInputStream(inputFile);
        }

        // Prepare
        ANTLRInputStream input = new ANTLRInputStream(is);
        SASLexer lexer = new SASLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        SASParser parser = new SASParser(tokens);
        parser.setBuildParseTree(true);

        // Parse specifying the input parse rule file
        ParseTree tree = parser.file();

        // create a dataflow graph
        ParseTreeWalker walker = new ParseTreeWalker();
        DataflowListener collector = new DataflowListener();
        walker.walk(collector, tree);

        // print the result
        System.out.println(collector.graph.toDOT());
    }
}

```

So let's try this on the following program:

```
/* dataflow.sas */

proc sort data=sdtm.dm out=dm;
  by usubjid;
run;

proc sort data=sdtm.ae out=ae;
  by usubjid aestdct;
run;

proc sort data=sdtm.sv out=sv;
  by usubjid;
run;

data ae2;
  merge dm ae;
  by usubjid;
run;

data mess;
  set dm ae2 sv;
  s = '
    data abc;
      set a b c;
      set d;
      ss = ''
        data aabbcc; /* / this is a */
          set aa bb cc; /* <=| string not */
        run; /* \ a DATA step */
      '';
    run;
  ';
run;

proc sort data=ae2;
  by usubjid aestdct aeendct;
run;

data mess2;
  merge mess ae2;
  by usubjid;
  set ads.adlb(where=(paramcd='BACT' and visit = 'Screening')) key=keyvar / unique;
run;

data mess2;
  set mess2;
  paramcd = 'BACTERIA';
run;

proc sort data=mess2 out=result;
  by usubjid svdct;
run;
```

Note that I specifically included the string with the DATA step code where the `ss = variable` is assigned in the `mess` DATA step. This was done intensively to show that this text won't be recognized as a SAS code and will be considered as a normal string by the parser — this behavior is what we expect from the parser.

After running the application we get the following output:

```
$ javac *.java
$ java DataflowGraph dataflow.sas
digraph G {
  ranksep=.25;
  edge [arrowsize=.5]
  "sdtm.dm"; "sdtm.ae"; "sdtm.sv"; "dm"; "ae"; "ae2"; "sv"; "mess"; "ads.adlb"; "mess
2";
  "sdtm.dm" -> "dm";
  "sdtm.ae" -> "ae";
  "sdtm.sv" -> "sv";
  "dm" -> "ae2";
  "dm" -> "mess";
  "ae" -> "ae2";
  "ae2" -> "mess";
  "ae2" -> "mess2";
  "sv" -> "mess";
  "mess" -> "mess2";
  "ads.adlb" -> "mess2";
  "mess2" -> "mess2";
  "mess2" -> "result";
}
```

In the output you will get the DOT source code of the data flow graph. To obtain the Figure 6 you can use `graphviz` and pipe the result to it to obtain the image. Or you can use this website - <https://dreampuf.github.io/GraphvizOnline/>.

Data flow graph using the SCAPROC procedure or RTRACE System Option

Similar results may be obtained using the output from [the SCAPROC procedure](#) or using the RTRACE System Option for [UNIX](#) or [Windows](#). This approach is easier and does not involve SAS source code parsing but you have to **adapt** your program and **run** it to obtain the output from these procedures or options.

SOURCE CODE FORMATTING

The topic of source code formatting, automatic alignment, and AST transformations attracted programmers from the earliest stages of programming development. Everyone would agree it would be great if you didn't need to bother about for code alignment and your code became well-structured at the press of a button.

The following screenshots (Figure 8, 9) are taken from *Computerworld* magazine from the 80s and I was not even born at that time. These are source code formatting tools for the COBOL language, used at that time:

Turns Spaghetti Code COBOL Into Structured COBOL Automatically.

SUPERSTRUCTURE takes your unstructured COBOL programs and automatically produces structured COBOL programs that are easy to understand and maintain. SUPERSTRUCTURE provides a simple and cost-effective alternative to manually rewriting these unstructured programs that are a maintenance nightmare. Of course you can't believe it. Let us show SUPERSTRUCTURE work using your programs at your location. SUPERSTRUCTURE...the breakthrough you've been seeking by Call today. Marketing Director - SUPERSTRUCTURE

Group Operations, Incorporated
1115 Vermont Avenue, N.W.
Washington, D.C. 20005
(202) 887-5420
Offices in Boston,
Chicago, Dallas,
Los Angeles, and New York.




Figure 8: *Computerworld* May 14, 1984, p. 24

Cobol restructuring engines clean up spaghetti code

By GIRISH PARIKH

Program restructuring is just beginning to enter the vast and relatively unexplored world of software maintenance. The restructuring engine is likely to become one of the most practical tools for DP personnel because of its ongoing usefulness in program maintenance.

Ever since researcher Guy de Balhais developed a restructuring engine in the early '70s to restructure Fortran programs, the concept of automated software restructuring has excited managers who must deal with old, problematic unstructured software. However, in spite of the popularity of structured programming in the '70s and early '80s, automated restructuring did not catch on.

With the advent of Cobol restructurers, especially IBM's recent entry into the marketplace, the idea of automated restructuring seems to be surfacing. In the world of Cobol restructuring, Peat, Marwick, Mitchell & Co.'s Structured Retrofit offered the first commercial service and, in time, the first restructuring tool.

Later entrants in the Cobol restructuring race include Group Operations, Inc.'s Superstructure, developed by Bill Morgan; Language Technology, Inc.'s Recoder, developed by Eric Blush; and IBM's Cobol/SP developed by Rick Linger and his team.

In this discussion, these four products and services are presented chronologically based on the dates of their introduction into the marketplace.

Structured Retrofit

TER statements, PERFORM-THRU and FALL-THRU logic, eliminates dead code; restricts GOTOs to local loops; repackages the program into a hierarchical structure; and isolates and consolidates all I/O.

Structured Retrofit uses an automated assembly line process, consisting of the following steps:

Review. A static analysis and management reporting tool called Pathvu highlights unexecutable code, use of ALICEs and GOTOs, levels of logical nesting and software structure and so on.

The old program is analyzed by Pathvu, which determines if manual intervention is required. For example, if there is a runaway path, it should be fixed before restructuring.

Compile. Retrofit takes a working but unstructured Cobol program and produces a functionally equivalent structured Cobol program.

Review. If the first steps have highlighted compile errors, significant structural issues or dead code, they should be reviewed and fixed immediately.

Format. Using the formatting software tools, the readability of the code is improved.

Recompile. The new structured program is then recompiled to ensure that it is clean of diagnostics.

Optimize. Using object code analyzers, the hardware runtime performance of the new code is improved.

Validate. After completing the retrofit process, the restructured program is validated to ensure that identical inputs produce the same outputs as the old program did.



People put U.S. companies into the software maintenance mess, and people will clean it up. Cobol restructuring engines, which can reduce maintenance costs by as much as 50%, may be the catalysts.

Figure 9: *Computerworld* March 31, 1986, p. 59

For source code formatting, as explained in the [styleguide](#) section, we need to write formatters for the parse tree nodes. This allows us to control the case of keywords, indentation, and general code style. Particularly, the indentation is stored in a variable the population of which is triggered on entering and exiting the parse rule nodes like IF-THEN/ELSE, DO-WHILE, DO-END statements.

SAS LOG PARSING AS A MACRO PARSING WORKAROUND

SAS Macro is an old language from the 80s and it's intensively used nowadays. The problem with parsing it is that it is basically a text preprocessing language. And as a normal preprocessing language it should not be used in such intense *mixins* within the language it's accompanying. Unfortunately, for SAS the situation is the opposite and it is impossible to imagine a flexible program in SAS without SAS Macro language. Parsing SAS Macro language is problematic. In SAS you can code as follows:

```
%let d = dat;
%let end_token_pointer = end_token;
%let end_token = n;
%let if = __if;

%macro __if(c);if &c%mend;

&d.a smthng;
  set sashelp.class;
  %&if(%str(age > 18))then do;
    bmi = weight / height ** 2;
  e&&&end_token_pointer..d;
run;
```

As can be seen here, you can use macro variables as a part of SAS keywords: &d.a => data or e&&&end_token_pointer..d => end. But you cannot use macro calls as a part of the keywords — SAS will treat the macro call bound as a separator and not recognize the resulting keyword.

In this paper we will consider a workaround which will release us from the need to parse actual SAS Macro code. The idea is to parse the SAS Log produced with the SOURCE, SOURCE2, and MPRINT System Options and extract the underlying SAS Program.

Using the simplified SAS log grammar, our main focus will be in identifying the following statements from a SAS log: errors, warnings, notes, SOURCE, SOURCE2, MPRINT. All other statements will be ignored by the tool. The grammar and extracting tool can be found in the source code attached (SASLog.g4, LogExtractor.java, LogExtractorListener.java). The tool interface is quite simple:

```
$ java LogExtractor -source2 -mprint sas-log-extractor-demo.log
OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
options ls=64 mprint source2;
%include "~/source2.sas";
data _null_;
set sashelp.class;
...
```

What I found interesting during SAS log grammar compose is how SAS marks wrapped lines for SOURCE, SOURCE2, and MPRINT:

```
2 1      OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
3 72
4 73      options ls=64 mprint source2;
5 74
6 75      %include "~/source2.sas";
7 NOTE: %INCLUDE (level 1) file ~/source2.sas is file
8      /home/sspkmd/source2.sas.
9 76      +data _null;
10 77      +set sashelp.class;
11 78      +if Name = 'John' then do;
12 79      +put "Lorem Ipsum is simply dummy text of the
13 79      !+printing and typesetting
14 80      +industry. Lorem Ipsum has been the industry's
15 80      !+standard dummy text ever
16 81      +since the 1500s, when an unknown printer took a
17 81      !+galley of type and scrambled
18 82      +it to make a type specimen book. It has survived not
19 82      !+only five centuries, but
```

Figure 10: SOURCE and SOURCE2 wrapping algorithm by SAS

```
95 102      %useless_loop(50)
96 MPRINT(USELESS_LOOP): data class50;
97 MPRINT(USELESS_LOOP): set sashelp.class sashelp.class
98 sashelp.class sashelp.class sashelp.class sashelp.class
99 sashelp.class sashelp.class sashelp.class sashelp.class
100 sashelp.class sashelp.class sashelp.class sashelp.class
101 sashelp.class sashelp.class sashelp.class sashelp.class
102 sashelp.class sashelp.class sashelp.class sashelp.class
103 sashelp.class sashelp.class sashelp.class sashelp.class
104 sashelp.class sashelp.class sashelp.class sashelp.class
105 sashelp.class sashelp.class sashelp.class sashelp.class
106 sashelp.class sashelp.class sashelp.class sashelp.class
107 sashelp.class sashelp.class sashelp.class sashelp.class
108 sashelp.class sashelp.class sashelp.class sashelp.class
109 sashelp.class sashelp.class sashelp.class sashelp.class ;
```

Figure 11: MPRINT wrapping algorithm by SAS

Interesting is what will happen if the SAS log line number for SOURCE exceeds 9 digits...

CONCLUSION

The final tool, called SASLint, has a command line interface and reports the issues in the following way (for all SAS programs in the current directory):

```
$ saslint
Inspecting 10 files
..E.W..N..

Offenses:

test-errors.sas:3:26: E:22-322: Missing ';' at the end of the assignment statement.
    f = cat(3, 5, 3 + 5)
                        ^

test-errors.sas:4:26: E:22-322: Extra closing ')'.
    f = cat(3, 5, 3 + 5));
                        ^

test-errors.sas:5:25: E:79-322: Missing closing ')'.
    f = cat(3, 5, 3 + 5;
                ^

test-warnings.sas:4:9: W: Expression is always true.
    if length(str) > 0 then do;
        ^^^^^^^^^^^^^^^^^^^

test-notes.sas:7:5: C: This DATA step uses no executable statements and can be replaced by the DATASETS procedure.
data class;
^^^^

10 files inspected, 5 offenses detected.
```

The further tool development and more demos can be found on the SASLint website <https://saslint.com>.

REFERENCES

ANTLR (ANOther Tool for Language Recognition). (Accessed March 2018) Available at <http://www.antlr.org/>

Terence Parr. 2012. *The Definitive ANTLR 4 Reference*. Dallas, Texas, Raleigh, North Carolina: The Pragmatic Bookshelf. <https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference>

Wikipedia contributors. 2018. *DOT (graph description language)*. (Accessed March 2018) Available at [https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language))

Andrey Sitnik. 2017. *Stylelint: Why and How to Lint CSS*. (Accessed March 2018) Available at <http://slides.com/ai/stylelint/#/>

Computerworld Vol. 18, No. 20. May 14, 1984. IDG Enterprise. (Accessed March 2018) Available at <https://books.google.de/books?id=BrEo9KtAQH4C&pg=PA24&dq=turns+spaghetti+code+cobol+into+structured+cobol+automatically&hl=en&sa=X&ved=0ahUKEwi3ufiP-9rZAhVHy6QKHVvNDk04FBDoAQg1MAM#v=onepage&q&f=false>

Computerworld Vol. 20, No. 13. March 31, 1986. IDG Enterprise. (Accessed March 2018) Available at https://books.google.de/books?id=nvgwHu6zOfQC&pg=PA59&lpg=PA59&dq=superstructure+cobol&source=bl&ots=Tug9PazyYb&sig=N8ylbjPleyby8f3td0whqTPEANE&hl=en&sa=X&ved=0ahUKEwjn29_S3NnZAuUosaQKHAY-CvwQ6AEILzAC#v=onepage&q&f=false

John MacFarlane. *Pandoc*. (Accessed March 2018) Available at <http://pandoc.org/index.html>

Rob Krajcik "Why Does SAS® Run Clockwise?". NESUG 2009. (Accessed March 2018) Available at <https://www.lexjansen.com/nesug/nesug09/bb/BB10.pdf>

William S. Calvert, Malla R. Rao "A SAS® Code Formatter". SUGI-92. (Accessed March 2018) Available at <http://www.sascommunity.org/sugi/SUGI92/Sugi-92-30%20Calvert%20Rao.pdf>

SAS Institute Inc. *SAS® 9.4 SQL Procedure User's Guide, Fourth Edition*. Cary, NC: SAS Institute Inc. (Accessed March 2018) Available at <http://documentation.sas.com/?docsetId=sqlproc&docsetTarget=n02s19q65mw08gn140bwfdh7spx7.htm&docsetVersion=9.4&locale=en>

SAS Institute Inc. *SAS® 9.4 Language Reference: Concepts, Sixth Edition*. Cary, NC: SAS Institute Inc. (Accessed March 2018) Available at <http://documentation.sas.com/?docsetId=lrcn&docsetTarget=titlepage.htm&docsetVersion=9.4&locale=en>

ACKNOWLEDGMENTS

I'd like to thank [Terence Parr](#) and all [other contributors](#) for their work on [antlr](#) as well as Rowland Hale who gave me some hints for the rules which I used or plan to implement in the linter. Also, all other authors listed in the [References](#) section. The original idea of creating a linter for SAS was inspired by [Andrey Sitnik](#)'s talk [WebCamp:Front-End Why you need to lint CSS](#).

CONTACT INFORMATION

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

Igor Khorlo
Syneos Health™
{ igor.khorlo } at syneoshealth dot com
{ igor.khorlo } at gmail dot com