PhUSE 2017

Paper AD08

Lint, a SAS Program Checker

Igor Khorlo, inVentiv Health Clinical, Berlin, Germany

ABSTRACT

Linters are programs which carry out static source code analysis, detecting certain bugs, coding rule deviations and other issues. Linters also detect redundant or error-prone constructs which are nevertheless, strictly speaking, legal. This paper will cover creating a modular linter for the SAS ® language consisting of parser module, an analysis module which includes a list of rules and a reporting module which displays issues found. It is possible to include/exclude rules as well as develop your own rules which make 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 ANLTR ® Java parser. The tool is written in Java and SAS which is why it can be integrated into almost 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.

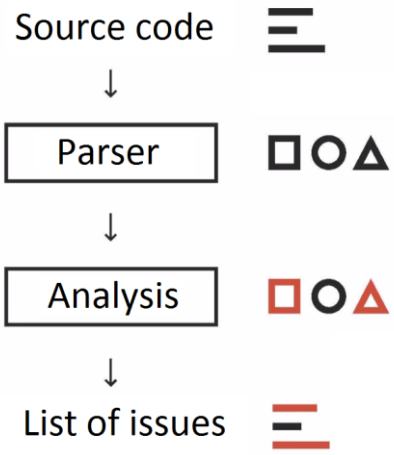

Figure 1: Linter architecture

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 example;
set sashelp.class;
if Name='John' then do;
%do_something;
end;
proc freq data=example;
tables type;
```

and well-aligned:

```
data example;
    set sashelp.class;
    if Name = 'John' then do;
        %do_something;
    end;
run;

proc freq data = example;
    tables type;
run;
```

I think most of you will agree that the second one looks better. 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.

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):

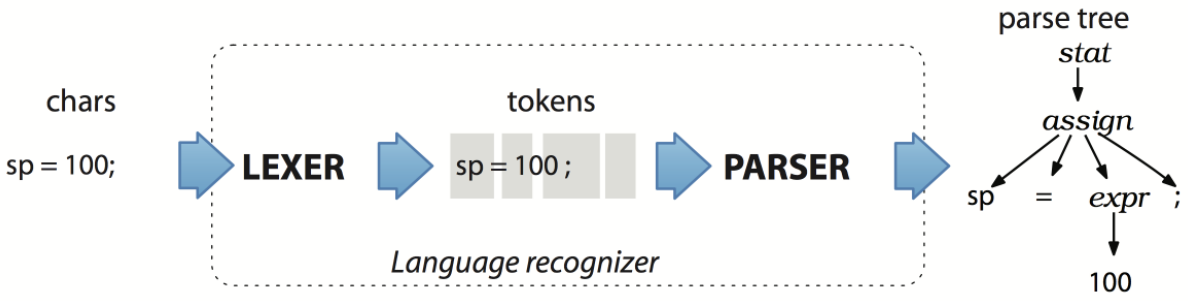

Figure 2: ANTLR produces AST

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

In this section we will discover ANTLR in more detail by building a grammar for a generic SAS expression. 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). We'll build a grammar which will recognize expressions like:

- `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.)).`

ANTLR grammar is a plain text file, with a `.g4` extension, which contains a set of rules. First let's consider simple expressions with `+, -, *, /` and with integers and variables names only (to keep things simple for now). We will use a choice pattern, where the 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]+ -> skip ;         // toss out whitespace
```

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/antlr-4.7-complete.jar:$CLASSPATH"
alias antlr4='java -jar /usr/local/Cellar/antlr/4.7/antlr-4.7-complete.jar'
alias grun='java org.antlr.v4.gui.TestRig'
```

For more details on how to set ANTLR up please see [2] and the documentation - Getting Started with ANTLR v4.

The result is the following parse tree for the example expression `a + b * (var3 / 2)` (Figure 3):

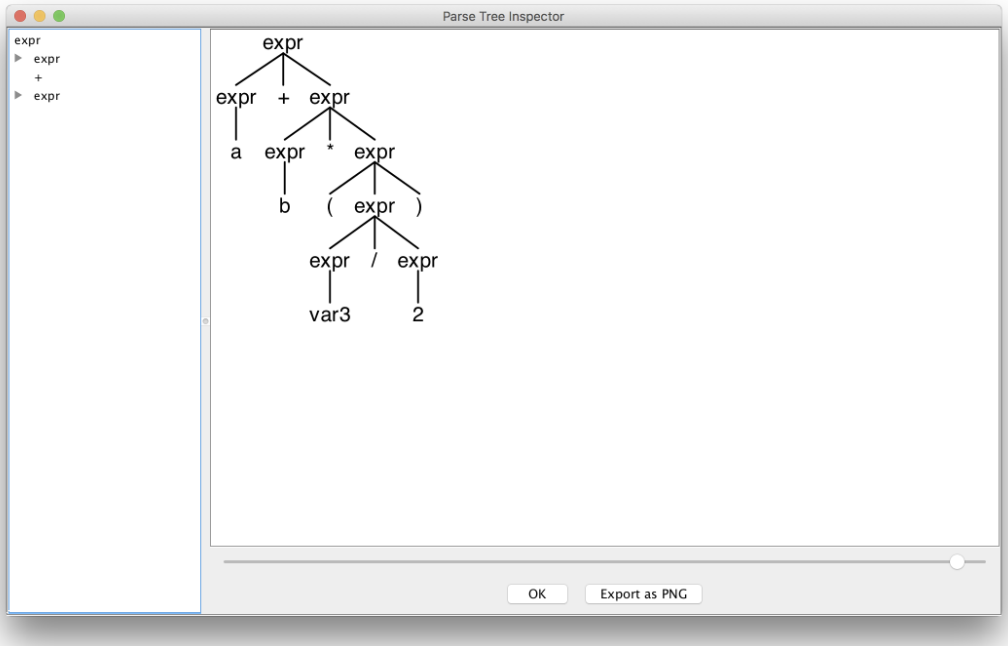

Figure 3: Parse tree

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

Now let's add function support.

```
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):

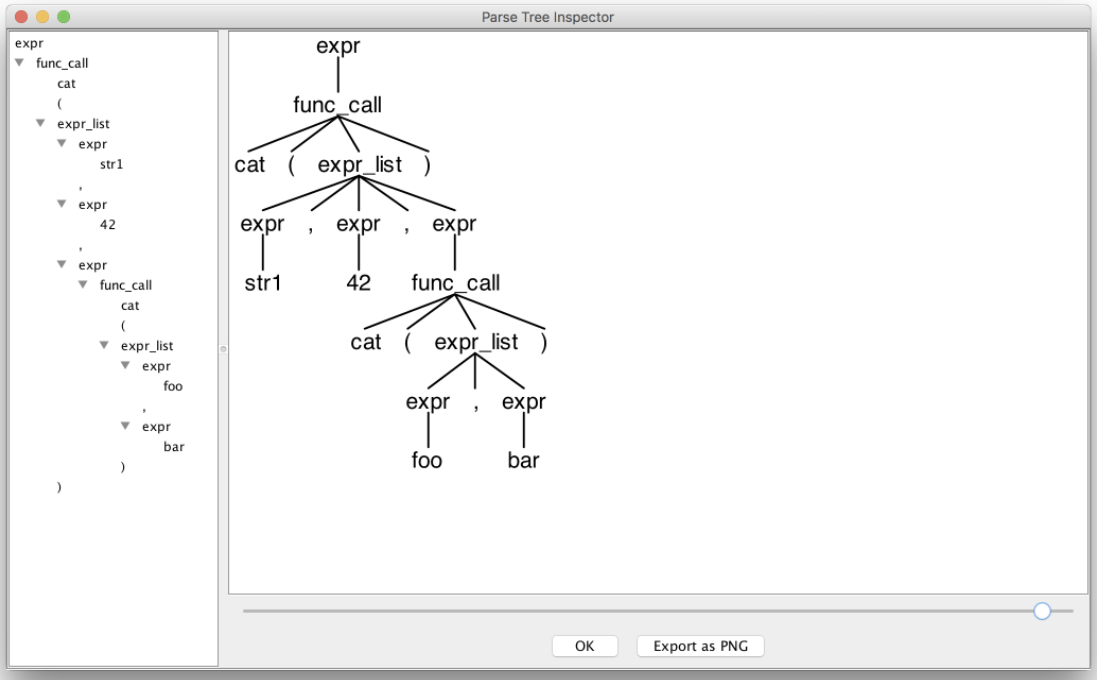

Figure 4: Parse tree

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, where you can easily see the flow of the data and which datasets were used for the creation of others. As a result we obtain the picture below (Figure 5):

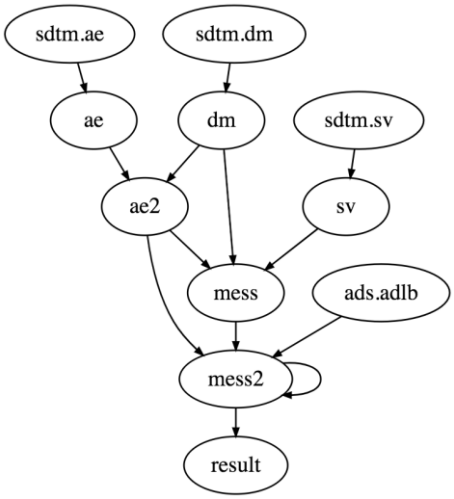

Figure 5: Data flow graph for the example program

Suppose that we have a SAS Grammar (see the Source code section). Let's see how it works on a simple SAS program like this:

```
data class;
    set sashelp.class end=eof;
    name = catx('-', name, eof);
run;

proc sort data=class;
    by name;
run;
```

The resulting parse tree is as follows (Figure 6):

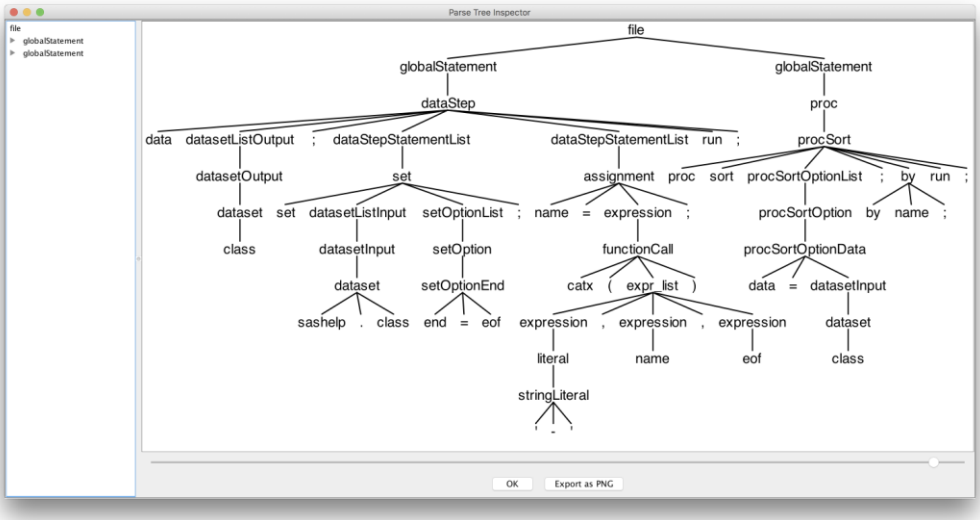

Figure 6: Parse tree for the program.sas

ANTLR provides you two methods to walk an AST: the listener and the visitor pattern. The listener pattern implements the idea that for each node like a set statement, variable reference or expression two methods (functions) will be triggered - when the parser enters the node and when it exits. So basically everything is done automatically, whereby the appropriate methods are triggerered by the parser during the tree walk, and you just need to write the code you want to execute in the methods you need. The visitor pattern acts differently, in that you manually call a method (function) to visit a specific node.

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:

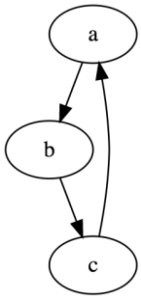

Figure 7: DOT example

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 SAS global statement like those in a data step or proc sort, 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 the global statement 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
...
```

So, we will override the `GlobalStatement` method for enter and exit and provide `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(", ",".."));
    }

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

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 our code on the following program:

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

```
run;

proc sort data=sdtm.ae out=ae;
    by usubjid aestdtc;
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;
                    set aa bb cc;
                run;
            '';
        run;
    ';
run;

proc sort data=ae2;
    by usubjid aestdtc aeendtc;
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 svdtc;
run;
```

Note that I specifically included the special string in the program which has the data step text. So, 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";
"mess2";
  "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 image (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/.

FORMAT THE SOURCE CODE

For the source code formatting, as explained in the Styleguide section above, we will need to increase or decrease the indentation. The amount of indentation is stored in a variable the population of which is triggered on entering and exiting the parse rule nodes.

CHECK FOR SPECIFIC RULES

To check a program for specific rules you need to write a logic code which will visit the appropriate nodes of the AST and check them. For example you may wish to define rules to ensure *global* options like FMTERR are not overwritten in the program, or for dangerous but legal code such as the `where pcng < 90;` condition mentioned above:

```
$ java SASLint lint.sas
   line 1:9 SAS option PS was changed `ps=50`
   line 1:15 SAS option LS was changed `ls=100`
   line 1:22 SAS option FMTERR was changed `nofmterr`
   line 5:5 warning: possible missings in the `height` variable may cause the
condition `pcng < 90` to work not as expected
```

INTEGRATING INTO SAS

In the end we are expecting a tool which can be easily integrated into SAS. Since everything is written in Java we can this easily integrate this into SAS and call our linter from a SAS program. This can be done using Data Step component Java Object. See the relevant SAS documentation for how to do this.

SOURCE CODE

```
// SAS.g4

grammar SAS;

file: globalStatement* ;

globalStatement
    : dataStep
    | proc
    //| anywhereOptions
    ;

dataStep
    : 'data' ( DATA_NULL | datasetListOutput ) ';'
      dataStepStatementList*
      'run' ';'
    ;

datasetListOutput
    : datasetOutput+
    ;
```

```
datasetListInput
    : datasetInput+
    ;

datasetList
    : dataset+
    ;

datasetOutput
    : dataset
    ;

datasetInput
    : dataset
    ;

dataset
    : (ID '.')? ID ('(' datasetOptionList ')')?
    ;

datasetOptionList
    : datasetOption*
    ;

datasetOption
    : 'where' '=' '(' expression ')'
    ;

dataStepStatementList
    : assignment
    | set
    | merge
    ;

set : 'set' datasetListInput setOptionList? ';' (by)?
    ;

setOptionList
    : setOption
    ;

setOption
    : setOptionKey
    | setOptionEnd
    ;

setOptionKey
    : 'key' '=' ID ( '/' 'unique' )
    ;

setOptionEnd
    : 'end' '=' ID
    ;

merge
    : 'merge' datasetListInput ';' (by)?
    ;

by  : 'by' 'descending'? ID ('descending'? ID)* ';'
    ;

//proc:;
```

```
//anywhereOptions:;

assignment
    : ID '=' expression ';'
    ;

expression
    : functionCall
    | expression ('='|'*'|'/'|'+'|'-'|'||'|'!!'|'and'|'or') expression
    | literal
    | ID
    | '(' expression ')'
    ;

functionCall
    : ID '(' expr_list? ')'
    ;

expr_list: expression ( ',' expression)* ;

literal
    : stringLiteral
    | numberLiteral
    ;

stringLiteral
    : '""'
    | '\'\''
    | '"' ( '""' | ~'"' )* '"'
    | '\'' ( '\'\'' | ~'\'' )* '\''
    ;

numberLiteral
    : INT
    | FLOAT
    ;

proc: procSort
    ;

procSort
    : 'proc' 'sort' procSortOptionList ';'
      by
      'run' ';'
    ;

procSortOptionList
    : procSortOption*
    ;

procSortOption
    : procSortOptionData
    | procSortOptionOut
    ;

procSortOptionData
    : 'data' '=' datasetInput
    ;

procSortOptionOut
    : 'out' '=' datasetOutput
    ;
```

```
DATA_NULL
    : '_null_';

ID   : [A-Za-z_][A-Za-z_0-9]* ; // match identifiers
INT  : DIGIT+ ;                 // match integers
DIGIT: [0-9] ;
FLOAT: DIGIT+ '.' DIGIT*        // match 1. 39. 3.14159 etc...
     | '.' DIGIT+               // match .1 .14159
     ;
NL   : '\r'? '\n' -> skip;      // newlines
WS   : [ \t]+     -> skip;      // toss out whitespace

// DataflowGraph.java

import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.misc.MultiMap;
import org.antlr.v4.runtime.misc.OrderedHashSet;
import org.antlr.v4.runtime.tree.ParseTree;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import org.stringtemplate.v4.ST;

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.Set;

public class DataflowGraph {
    static class Graph {
        Set<String> nodes = new OrderedHashSet<String>();
        MultiMap<String, String> edges =
                new MultiMap<String, String>();
        public void edge(String source, String target) {
            edges.map(source, target);
        }
        public String toString() {
            return "edges: "+edges.toString()+", functions: "+ nodes;
        }
        public String toDOT() {
            StringBuilder buf = new StringBuilder();
            buf.append("digraph G {\n");
            buf.append("  ranksep=.25;\n");
            buf.append("  edge [arrowsize=.5]\n");
            buf.append("  ");
            for (String node : nodes) { // print all nodes first
                buf.append(node);
                buf.append("; ");
            }
            buf.append("\n");
            for (String src : edges.keySet()) {
                for (String trg : edges.get(src)) {
                    buf.append("  ");/**/
                    buf.append(src);
                    buf.append(" -> ");
                    buf.append(trg);
                    buf.append(";\n");
                }
            }
            buf.append("}\n");
            return buf.toString();
        }
    }

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

    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());
    }
}
```

CONCLUSION

As you can see, linters which check for a set of defined rules can be really helpful and save you a lot of time during programming. They also ensure consistency within all your or your team's programs. And the greatest thing is that this boring work is done for you by a machine (computer)!

ACKNOWLEDGMENTS

I would like to thank to Terence Parr for creating ANTLR, a great tool for parsing anything which works fast and the results are impressive. The original idea of creating a parser was inspired by Andrey Sitnik, see: WebCamp:Front-End Why you need to lint CSS.

REFERENCES

1. ANTLR
2. The Definitive ANTLR 4 Reference
3. DOT
4. Data Step component Java Object
5. Stylelint Why and How to Lint CSS

CONTACT

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

Igor Khorlo
INC Research/inVentiv Health
Berlin
igor.khorlo@inventivhealth.com