Lint, a SAS program checker

AD08 – Phuse 2017

Igor Khorlo

inc Research®

inVentiv Health

Standards

I guess most people here have their own team rules or styleguides which are written out in some document. Many programming languages have community/company/etc driven standards:

- NASA C style Guide - 10 rules that allow NASA to write millions of lines of code with minimal errors
- A community-driven Ruby coding style guide
- Linux kernel coding style
- Google C++ Style Guide

Static Code Analyzers

Static code analyzers are programs which check your source code without running it. This can be achieved in many ways. The most common one is an AST (Abstract Syntax Tree) generation from a raw source code and its analysis. E.g.:

- Lint, a C Program Checker
- rubocop – A Ruby static code analyzer, based on the community Ruby style guide
- PVS-Studio analyzer – Static code analyzer for C, C++, and C#

SAS Examples

In the next slides, we will consider several examples of the static code analyzer for SAS. How this is implemented we will discuss later.

SAS Strings Limitations Checks

SAS has many limitations for string length:

- Most of you familiar with `LRECL` option for an `%include` and similar statements. But, there is a pitfall – macro compiled via `sasautos` will use `LRECL=256` – this is **unchangeable**.
- Maximum titles/footnotes statements length is **255**, entrytitle/entryfootnote for GTL is **256** (SAS 9.2).
- general recommendation for a maximum line length of 80 symbols (pep8, rubocop).

Question

Any thoughts what is wrong here?

```
if length(str) > 0 then do;
    /* ... */
end;
```

Hidden-true Expressions

Let's consider a common expression `length(str)` $>$ `0`. The problem here is that:

> *LENGTH Function – Returns the length of a non-blank character string, excluding trailing blanks, and **returns 1** for a blank character string.*

Therefore, this expression is always true, what does not seem so from the first glance.

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 a new data arrives, it will work incorrectly without generating any error/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

Many languages have a styleguide – a set of recommendations how you should write/align/format your source code. Some of them even have a build-in functionality which force source code formatting. E.g. `Go` language with `gofmt`.
Let's consider two pieces of code:
Poorly aligned:

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

Styleguide – Well-aligned

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

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

Optimizations

```
data adlb_lymph;
    set adlb;
    if paramcd = 'LYMPH';
run;
```

Can be optimized to:

```
data adlb_lymph;
    set adlb;
    where paramcd = 'LYMPH';
run;
```

Hardcode Detection

Check for an undocumented hardcoded code parts.

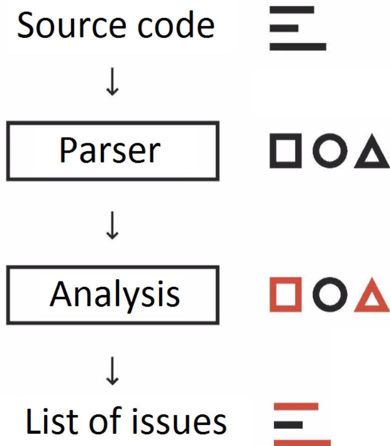
Bad:

```
where (&itt_cond and ctd1 ^= 2.4 and paramcd = 'ALLCSMRT'
       and parcat1n = 1)
      or usubjid = '99999-9999-9999';
```

Good:

```
where (&itt_cond and ctd1 ^= 2.4 and paramcd = 'ALLCSMRT'
       and parcat1n = 1)
      /* cf. Section x.x of the data handling document */
      or usubjid = '99999-9999-9999';
```

Linter Architecture



Linter architecture

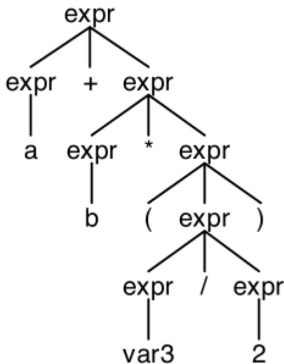
inc Research®
inVentiv Health

Abstract Syntax Tree – AST

Abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is “abstract” and not representing every detail appearing in the real syntax.

Abstract Syntax Tree for an Expression

```
a + b * (var3 / 2)
```



Parse tree

ANTLR

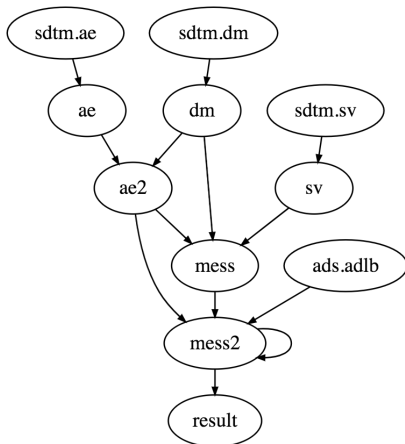
```
// Expr.g4

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
```

Generating Data Flow Graph Using DOT Language.



Data flow graph for the example program

SAS Integration

In the end, we are expecting a tool which can be easily integrated into SAS. ANTLR generates a parser in Java, logic for linter rules/checks are written in Java as well. Since everything is written in Java we can this easily integrate this in SAS and to call linter from SAS program. This can be done using Data Step component Java Object.

Final Look

```
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
...
```

Conclusion

Why do you need to lint?

- Speed
- Less error prone
- Faster code reviews
- Machines should do the work
- Scalability
- Rules and checks are based on the algorithms

Thank You

Company: INC Research/inVentiv Health

Email: igor.khorlo@inventivhealth.com