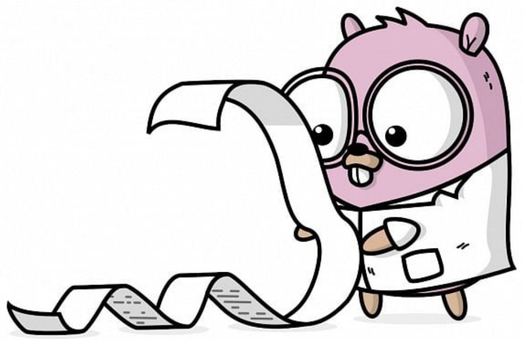
Writing Linters in Go



Josef Podaný, Software Engineer in Δ team, Jamf Security Cloud

Outline

Introduction

- Linting in theory
  - What's a linter, motivation
  - How to read & analyse source code?
- Linting in Go
  - Libraries to get us going
  - Quick demo
- Conclusion

Linters in Theory

Linter

Linters in Theory

- In general, linters...
  - **analyse programs** without running them
  - and try to show that programs have (or have not) certain *structural* properties
- Why should we lint the source code?
  - catch errors before running (or compiling) the code
  - to improve readability, maintainability, efficiency, security...

Linters in Relation to Static Analysis

Linters in Theory

- Static analysis helps with
  - type checking, correctness (program adheres to a specification)
- Linting is a subset of static analysis, a branch of formal verification
  - *in general*, checking for any (non-trivial) *semantic* property of a program is *undecidable*[1]

[1] Rice's theorem on Wikipedia

High-Level Overview of Linting

Linters in Theory

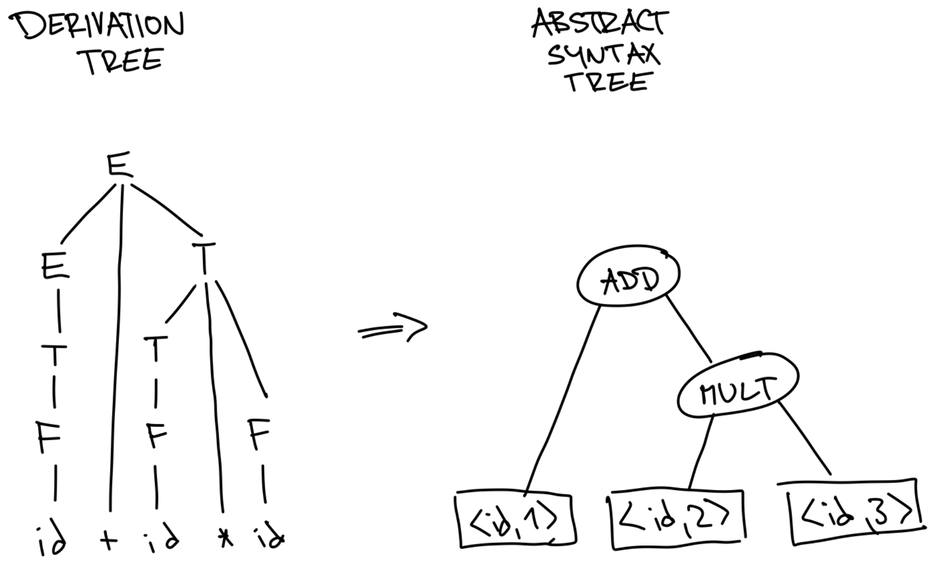
1. Read the source code
2. **Transform** it **into** some more **computer-appropriate representation**
3. **Evaluate the linting rules** on such representation
4. Report the results to the user

Abstract Syntax Tree

Linters in Theory

- AST is a rooted directed-acyclic graph
- AST nodes carry *semantically important* information about the code
- **AST is a convenient abstraction**; hides the clutter of the concrete syntax

a + b * c ⇒



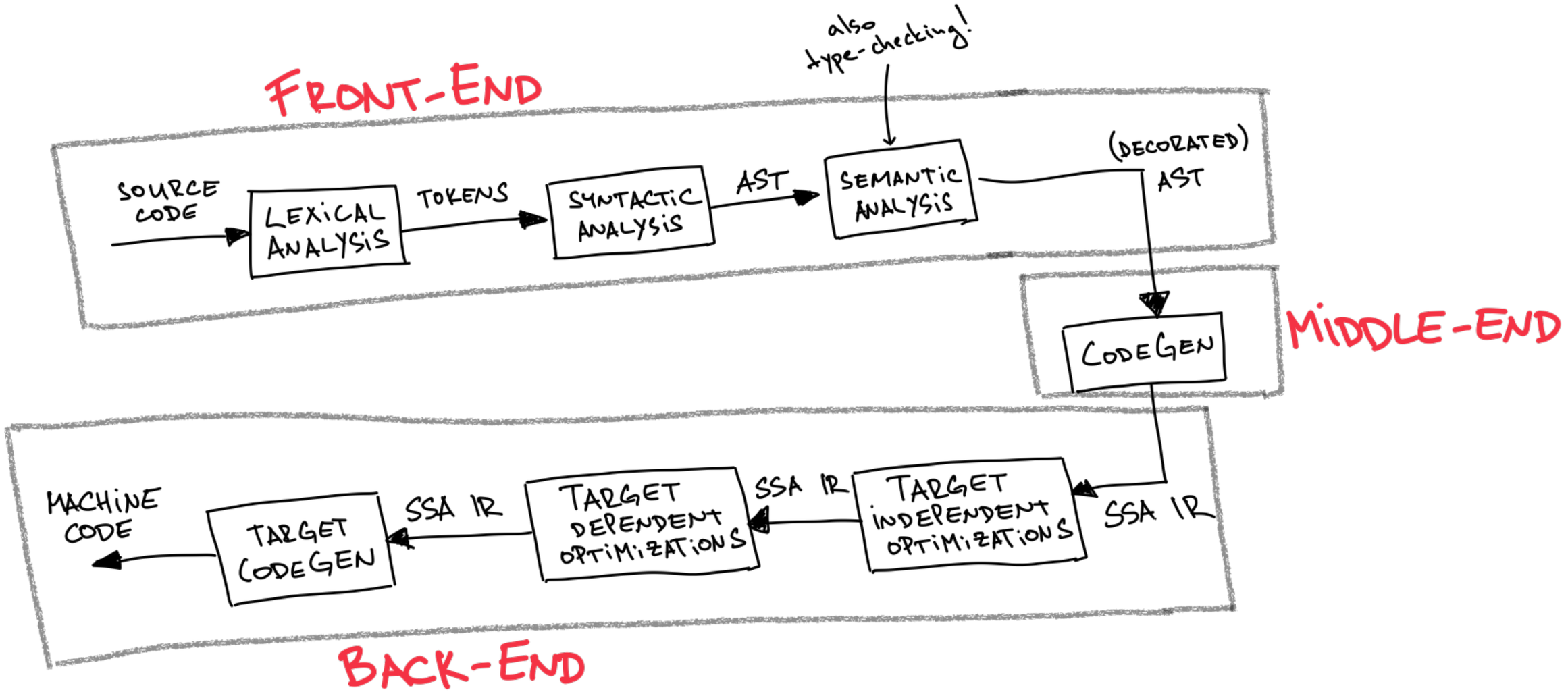
AST Construction

Linters in Theory

- AST can be built either from the
  - derivation tree (concrete syntax tree),
  - stream of lexemes.
- Various techniques to translate source code to AST depending on the type of grammar:
  - *top-down parsing*: recursive descent, Pratt parsing...
  - *bottom-up parsing*: LR, LALR, CYK...

AST Construction: Compiler Side Note

Linters in Theory



Linters in Go

Linting Go Code

Linters in Go

- Actually, knowledge of the aforementioned theory is **not** required! :)
- The standard library does all the heavy lifting for us:
  - go/token: provides lexical tokens of Go
  - go/ast: provides types representing AST nodes
  - go/parser: parses source code to AST
  - x/tools/go/analysis: bells and whistles

go/token

Linters in Go

- Defines basic lexical tokens of the Go language
  - identifiers
  - keywords
  - operators (+, -, *, /, ...)
  - brackets, braces, parentheses
  - ... and many more
- A complete list of tokens in Go can be found here

go/ast

Linters in Go

- Defines types of the AST nodes
  - BlockStmt, BinaryExpr, Comment, InterfaceType...
- Convenience functions for working with ASTs, mainly ast.Walk

go/parser

Linters in Go

- Parser for Go source code
- Offers only 4 functions:
  - ParseDir, ParseExpr, ParseExprFrom, ParseFile

x/tools/go/analysis

Linters in Go

- Toolkit for composing individual linters
- Gives bells and whistles on top of go/{token,ast,parser} packages
- Contains interesting predefined passes[2]
- inspect.Analyzer: filter out the unimportant AST nodes
- Suggestions

[2] golang.org/x/tools/go/analysis/passes

quasilyte/go-ruleguard

Linters in Go

- DSL/library for writing linting rules in Go
- Built on top of x/tools/go/analysis
- Turns out it's really expressive and convenient!

Demo Time!

Linters in Go

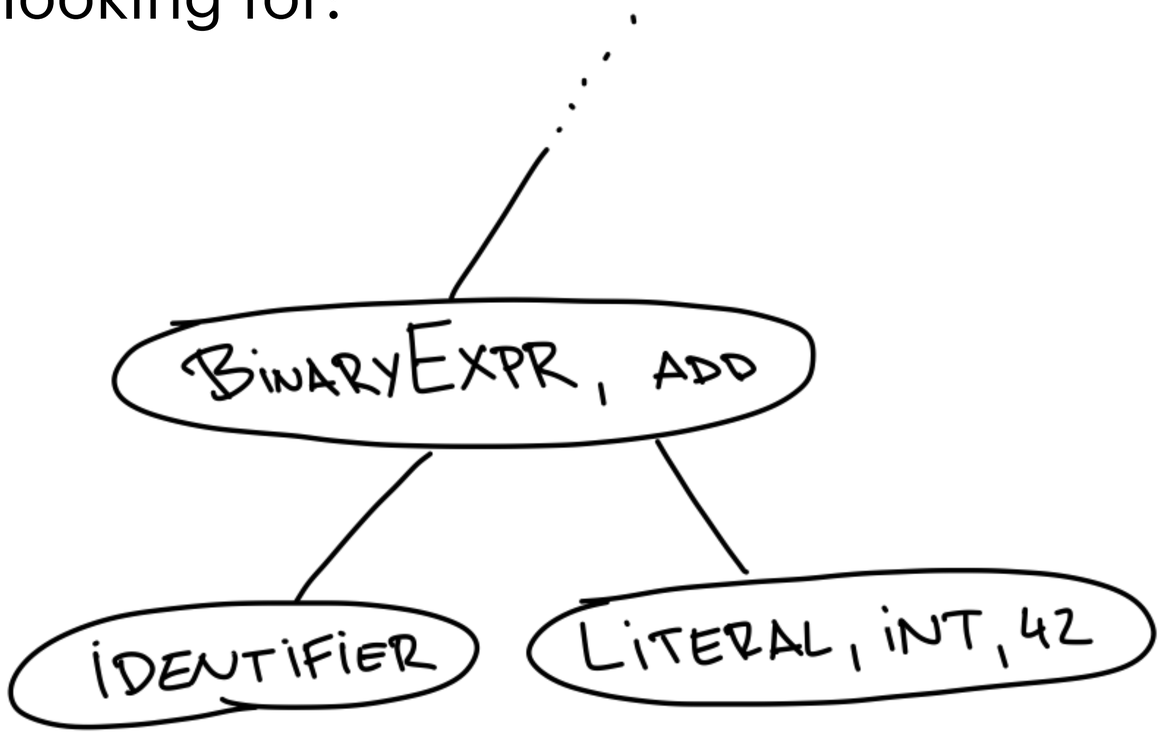
- Example problem: warn about expressions in the form of *'<variable> + 42'*
- 3 different implementations:
  1. go/{ast, token, parser} implementation
  2. go/{ast, token, parser} + x/tools/go/analysis implementation
  3. quasilyte/go-ruleguard implementation

[3] Complete source code available here: https://github.com/jsfpdn/linting_examples

Demo Time!

Linters in Go

These are the subtrees we're looking for:



Integrating into golangci-lint

Linters in Go

- go/analysis package must be used
- Adding a linter to golangci-lint is really easy (~200 lines including documentation, reference configuration and tests![4])

[4] E.g., tagalign: https://github.com/golangci/golangci-lint/commit/134f2e049134a96b5b137a5b376cfdae27126ea3

Bonus: Getting a Formatter "For Free"

Linters in Go

- Thanks to *strict formatting rules*, *no maximum line length*, and *a great standard library*, core gofmt functionality has (only) 594 lines![5]
- Conceptually pretty simple and straightforward (IB002): *"just traverse the (AST) tree and pretty-print it."*
- Take a look at the Dart formatter for comparison[6] 😵‍💫

[5] Have a look at go/src/fmt/format.go

[6] Bob Nystrom: The Hardest Program I've Ever Written. The story of Dart formatter. A great read!

Other Possible Approaches

Linters in Go

- Github CodeQL: https://codeql.github.com/
- Semgrep: https://semgrep.dev/docs/writing-rules/overview/

Conclusion

- We have showed...
  - the fundamentals of linting
  - and how (relatively) easy it is to lint & analyse Go programs in Go

Thank you for your attention!

Any questions?

Resources

- AST vs CST: https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees#id6
- https://lia.mg/posts/writing-go-linters/
- https://eli.thegreenplace.net/2021/rewriting-go-source-code-with-ast-tooling/
- https://disaev.me/p/writing-useful-go-analysis-linter/

Resources

- https://github.com/quasilyte/go-ruleguard
- https://go-ruleguard.github.io/
- https://cheikhhseck.medium.com/create-a-linter-rule-with-go-ruleguard-66804c3a5c9b
- https://developer20.com/custom-go-linter/
- https://arslan.io/2019/06/13/using-go-analysis-to-write-a-custom-linter/