
PVS-Studio

Go vet не поможет... Как сделать свой анализатор кода для Go?

PVS-Studio

Глеб Асламов

Senior Developer Advocate

Глеб Асламов

Senior Developer Advocate

- Разрабатываю статический анализатор
- Пишу про статический анализатор
- Говорю про статический анализатор:
  SQA Days , Стачка, DotNext, UIC Dev, DevFest , etc.

PVS-Studio

Статический анализ в мире go

Статический анализ в Go

Статический анализ в Go

- Статический анализ это процесс выявления недочетов, ошибок и потенциальных уязвимостей в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода.

Статический анализ в Go

- Статический анализ это процесс выявления недочетов, ошибок и потенциальных уязвимостей в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода.
- Но кажется вы и так с ним знакомы, даже если не знали об этом

Немного базовой-базы

- `go vet` *как* часть тулчейна

Немного базовой-базы

- `go vet` ***это*** часть тулчейна

  Нет, не вашего
  Самого тулчейна Go, это не отдельный линтер

Немного базовой-базы

- **`go vet` *`это`*** часть тулчейна

  Нет, не вашего
  Самого тулчейна Go, это не отдельный линтер

- Был **`go build`**? **`go test`**?
  Поздравляю, вы уверенный пользователь статического анализатора!

Немного базовой-базы

- `go vet` ***это*** часть тулчейна

  Нет, не вашего

Вы его уже используете
(но могли просто не замечать)

- Был `go build`? `go test`?
  Поздравляю, вы уверенный пользователь статического анализатора!

Пример. Синтетический


```
if A == B {
   if A == B {
      ....
   }
}
```

Пример. Синтетический


```
if A == B {
    if A == B {
        ....
    }
}
```

Сообщение PVS-Studio:

V8020. Recurring check. This condition was already verified on a previous line.

- Ха-ха, ошибка слишком простая! Вот я точно так не ошибусь!

Пример. Синтетический


```
if A == B {
    if A == B {
        ....
    }
}
```

Сообщение PVS-Studio:

V8020. Recurring check. This condition was already verified on a previous line.

- Ха-ха, ошибка слишком простая! Вот я точно так не ошибусь!

...Ведь...так?

Пример. Из проекта Incus


```
err = p.Save(pidPath)
if err != nil {
    err2 := p.Stop()
    if err != nil {
        return fmt.Errorf("...: %s: %s", err, err2)
    }
    ...
}
```

Пример. Из проекта Incus


```
err = p.Save(pidPath)
if err != nil {
    err2 := p.Stop()
    if err != nil {      // <= похоже, должно быть err2
        return fmt.Errorf("...: %s: %s", err, err2)
    }
    ...
}
```

Сообщение PVS-Studio:

V8020 Recurring check. The 'err != nil' condition was already verified on line 407

Пример. Из проекта Incus


```
err = p.Save(pidPath)
if err != nil {
    err2 := p.Stop()
    if err != nil {       // <= похоже, должно быть err2
        return fmt.Errorf("...: %s: %s", err, err2)
    }
    ...
}
```

Так это и go vet поймает!
Зачем что-то свое\другое?

Хватит ли...?


```
func rgb1(r float32, g float32, b float32) {
   if r > 1 || g > 1 || r > 1 {
....
   }
}
```

Хватит ли...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || r > 1 {
....
  }
}
```

Конечно хватит!

```
go vet: redundant or: r > 1 || r > 1
```

Хватит ли...?


```
func rgb1(r float32, g float32, b float32) {
   if r > 1 || g > 1 || 1 < r {
....
   }
}
```

А если так...

Хватит ли...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || 1 < r {
....
  }
}
```

А если так...

Конечно хватит!

Хватит ли...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || 1 < r {
....
  }
}
```

А если так...

Конечно хватит!

Ведь так?


go vet не поможет

Хватит ли...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || 1 < r {
....
  }
}
```

А если так...

Конечно **не** хватит!

go vet: All good! 👍

А если не синтетика?

- Nuclei

```
func NewEntityParser(dir string) (*EntityParser, error) {
  cfg := &packages.Config{
    Mode:
                  packages.NeedName   | packages.NeedFiles |
                  packages.NeedImports| packages.NeedTypes |
                  packages.NeedSyntax | packages.NeedTypes |
                  packages.NeedModule | packages.NeedTypesInfo,
    ....
  }
}
```

А если не синтетика?

- Nuclei

```
func NewEntityParser(dir string) (*EntityParser, error) {
  cfg := &packages.Config{
    Mode:
                  packages.NeedName   | packages.NeedFiles |
                  packages.NeedImports| packages.NeedTypes |
                  packages.NeedSyntax | packages.NeedTypes |
                  packages.NeedModule | packages.NeedTypesInfo,
    ....
  }
}
```

```
go vet: All good!
```

Ну это наверное единичный случай...

False Positive

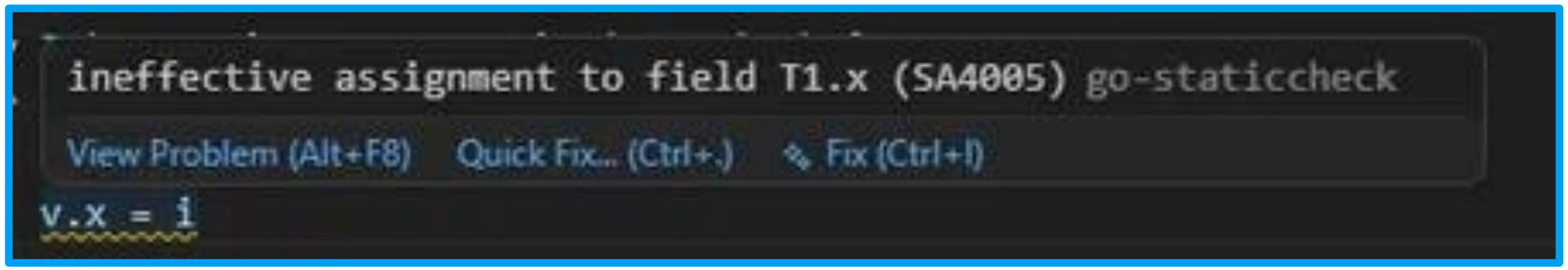
- Staticchek

```
func (v T1) negative_range_string(s string) {
    for i := range s {
        fmt.Println(v.x)
        v.x = i
    }
}
```

False Positive

- Staticchek

```
func (v T1) negative_range_string(s string) {
    for i := range s {
        fmt.Println(v.x)
        v.x = i
    }
}
```



Но ошибка ли это?

Нет!

False Negative

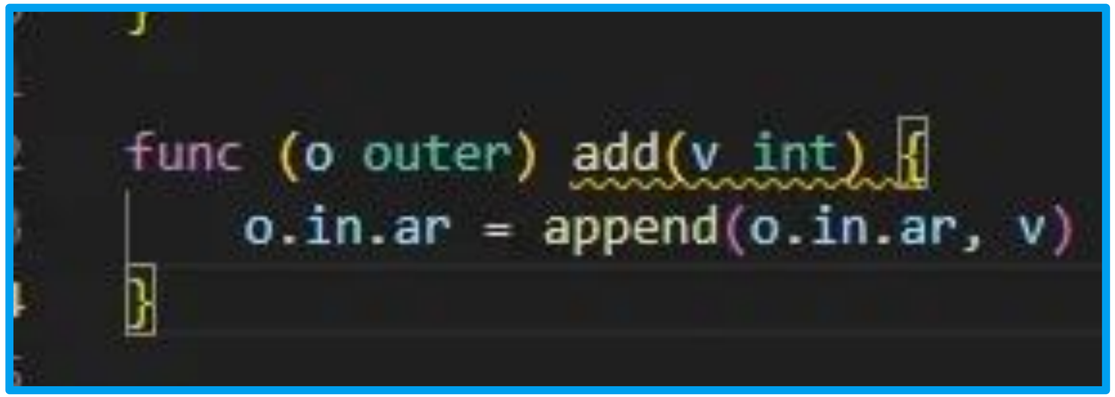
- Еще Staticchek

```
func (o outer) add(v int) {
    o.in.ar = append(o.in.ar, v)
}
```

False Negative

- Еще Staticchek

```
func (o outer) add(v int) {
    o.in.ar = append(o.in.ar, v)
}
```



Но ошибка ли это?

Да!

Ставим задачу

Ставим задачу

Попробуем найти эту ошибку:

Метод меняет поле получателя, переданного по значению. Правки уходят в копию, оригинал остаётся прежним.

Ставим задачу

Попробуем найти эту ошибку:

Метод меняет поле получателя, переданного по значению. Правки уходят в копию, оригинал остаётся прежним.

Так не работает

```
type Buffer struct {
  data []byte
}

func (b Buffer) Append(value byte) {
  b.data = append(b.data, value)
}
```

Ставим задачу

Попробуем найти эту ошибку:

Метод меняет поле получателя, переданного по значению. Правки уходят в копию, оригинал остаётся прежним.

Так не работает

```
type Buffer struct {
  data []byte
}

func (b Buffer) Append(value byte) {
  b.data = append(b.data, value)
}
```

А так работает

```
type Buffer struct {
  data []byte
}

func (b *Buffer) Append(value byte) {
  b.data = append(b.data, value)
}
```

Зачем делать что-то своё

+ Плюсы

- Не нужно выбирать
- Настраиваем под себя
- Полный контроль
- Учитывается специфика проекта

Зачем делать что-то своё

+ Плюсы

- Не нужно выбирать
- Настраиваем под себя
- Полный контроль
- Учитывается специфика проекта

– Минусы

- Мини-проект
- Нужно поддерживать и обновлять
- FP и FN придётся ловить самому

Надеюсь это всё... (нет)

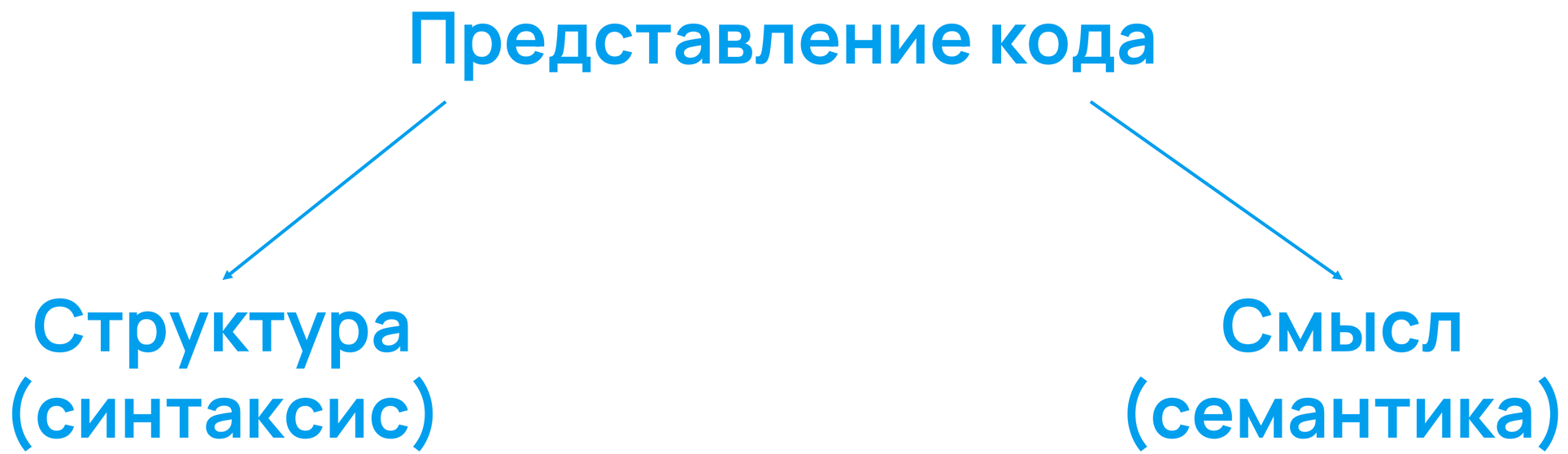
Одна из причин

ГОСТ

- ГОСТ 71207 и 56939
- Для компилируемых языков
- Список направлений на которые он распространяется пополняется
- Go vet из коробки не прокатит

Обзор технологий и пакетов Go для анализа

Представление кода



Синтаксис


```
func Foo() {
    if Condition {
        CallExpressionX()
        CallExpressionY()
    } else {
        fmt.Println()
    }
}
```

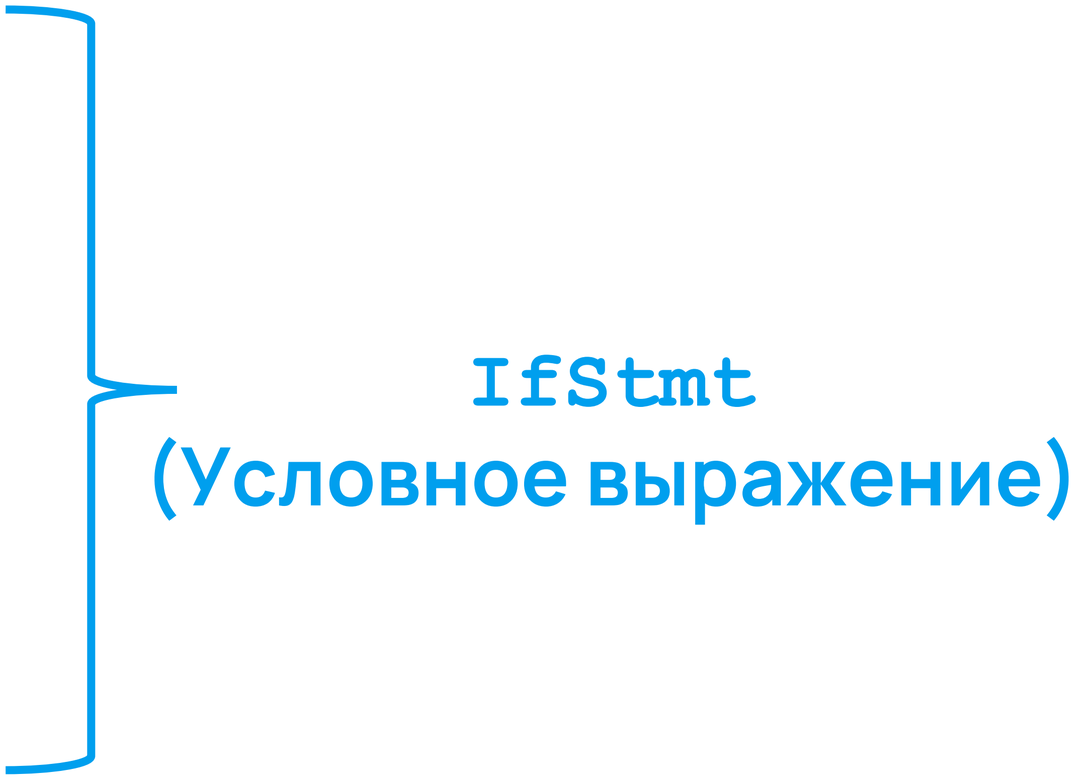
Синтаксис


```
func Foo() {
    if Condition {
        CallExpressionX()
        CallExpressionY()
    } else {
        fmt.Println()
    }
}
```

FuncDecl
(Декларация функции)

Синтаксис


```
func Foo() {
    if Condition {
        CallExpressionX()
        CallExpressionY()
    } else {
        fmt.Println()
    }
}
```



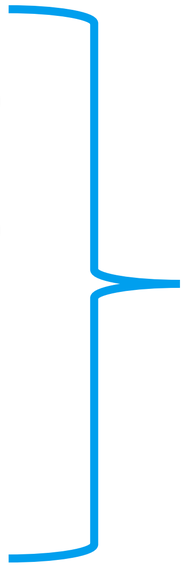
Синтаксис


```
func Foo() {
    if Condition {
        CallExpressionX()
        CallExpressionY()
    } else {
        fmt.Println()
    }
}
```

BlockStmt

Синтаксис


```
func Foo() {
    if Condition {
        CallExpressionX()
        CallExpressionY()
    } else {
        fmt.Println()
    }
}
```



CallExpr
(Вызов функции)

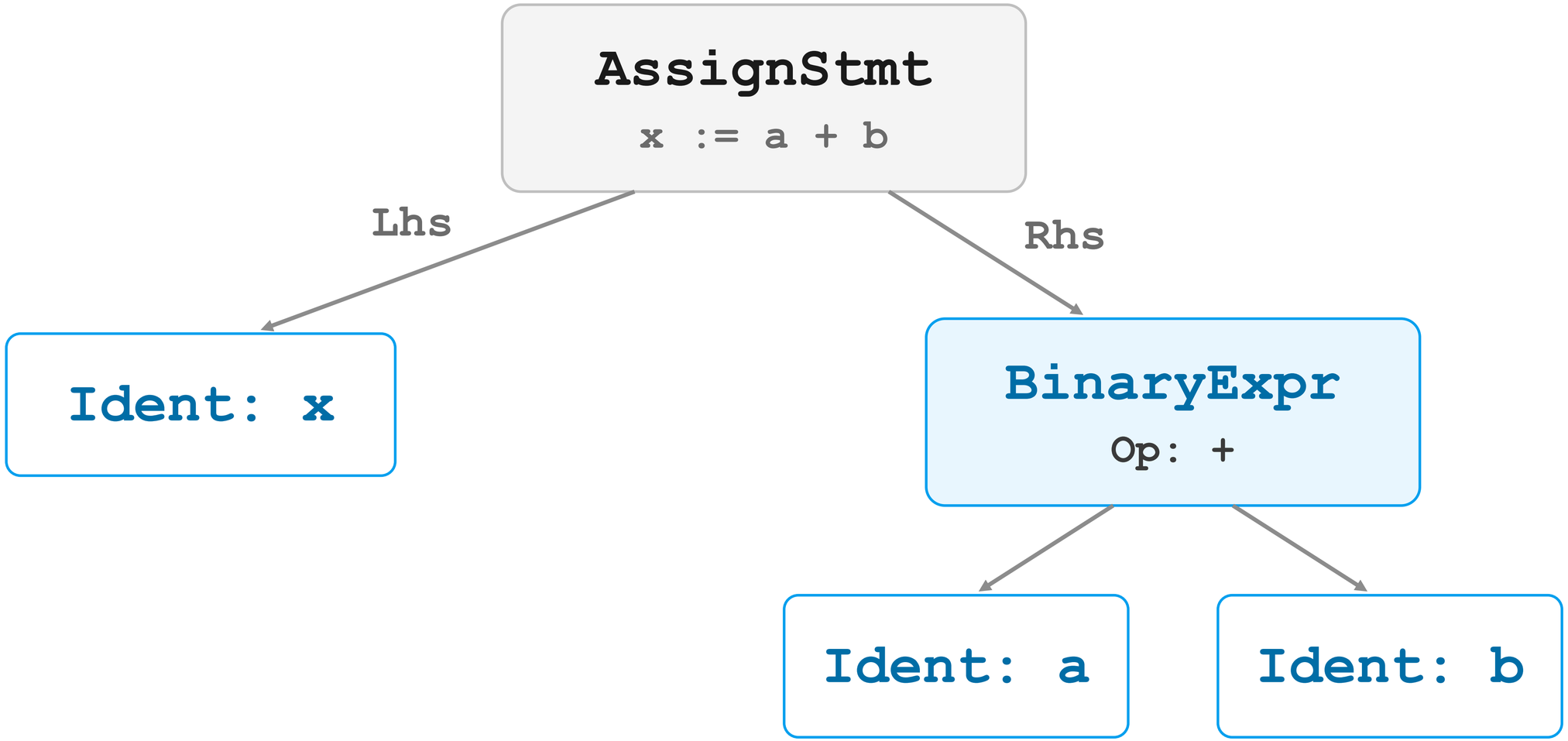
Синтаксическое дерево (AST)

Дерево


```
x := a + b
```

Дерево


```
x := a + b
```



Поиск ошибок с помощью синтаксиса

- WOX

```
func(manifest StorePluginManifest) bool {
    return manifest.Id == manifest.Id
}
```

```
go vet: All good!
```

Поиск ошибок с помощью синтаксиса

- WOX

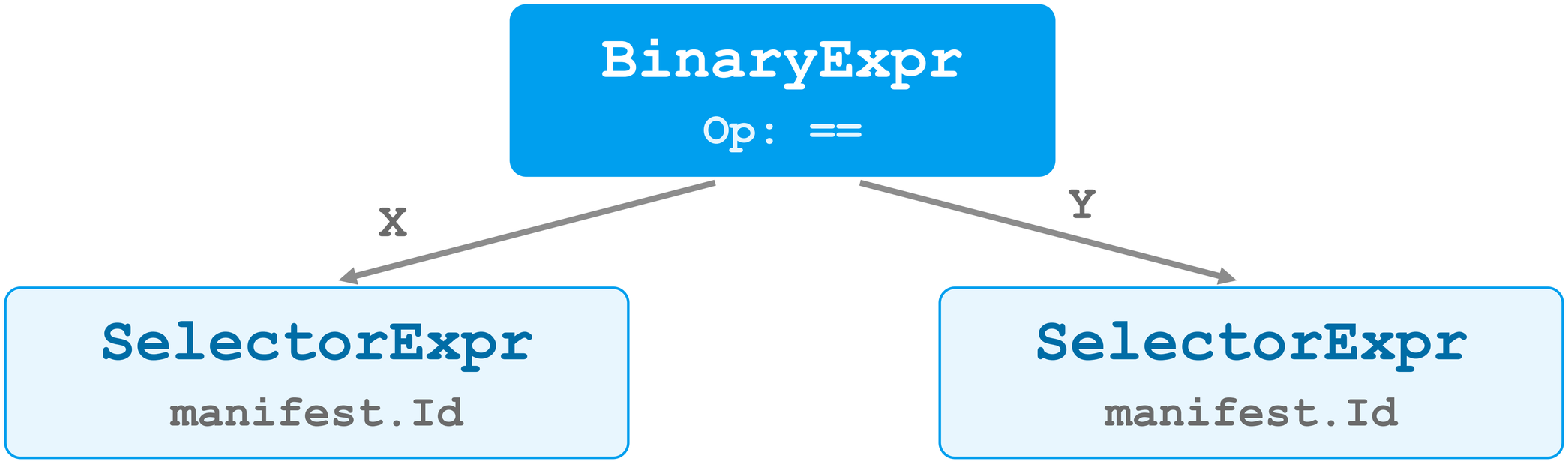
```
for _, manifest := range pluginManifest {

   func(manifest StorePluginManifest) bool {
        return manifest.Id == manifest.Id
}
```

```
go vet: All good!
```

Поиск ошибок с помощью синтаксиса


```
manifest.Id == manifest.Id
```



Слева и справа одно и то же поддерево.
Типы (семантика) не понадобились

В нашей диагностике

Исходник

```
func (b Buffer) Append(value byte) {
    b.data = append(b.data, value)
}
```

В нашей диагностике

Исходник

```
func (b Buffer) Append(value byte) {
    b.data = append(b.data, value)
}
```

Что видит go/ast

```
FuncDecl
  Recv: FieldList
    Field
      Type: Ident "Buffer"
  Name: Ident "Append"
  Body: BlockStmt
```

В нашей диагностике

Исходник

```
func (b Buffer) Append(value byte) {
    b.data = append(b.data, value)
}
```

Что видит go/ast

```
FuncDecl
  Recv: FieldList
    Field
      Type: Ident "Buffer"
  Name: Ident "Append"
  Body: BlockStmt
```

Смотрим:
Звёздочки нет = = значит **по значению**.

На этом синтаксис всё…
В другом не поможет.

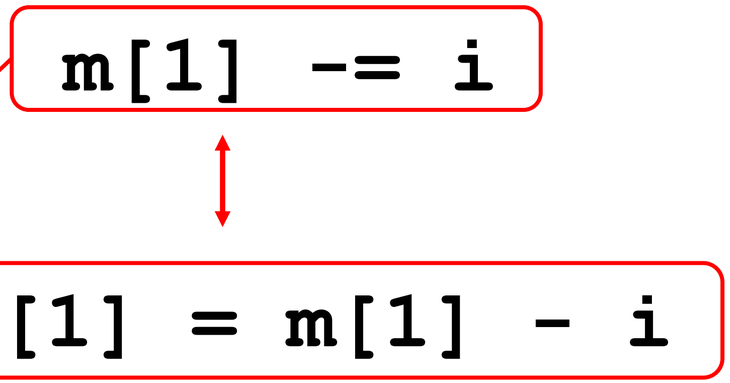
Семантика

Gitea


```
if m != nil {
    lastChar := line[m[1]-1]
    if lastChar == '.' {
      m[1]--
    } else if lastChar == ')' {
      ....
    } else if lastChar == ';' {
      i := m[1] - 2
        for ; i >= m[0]; i-- {
          if util.IsAlphaNumeric(line[i]) {
            continue
          }
          break
        }
        if i != m[1]-2 {
          if line[i] == '&' {
            m[1] -= m[1] - i
          }
        }
      }
```

Gitea


```
if m != nil {
    lastChar := line[m[1]-1]
    if lastChar == '.' {
      m[1]--
    } else if lastChar == ')' {
      ....
    } else if lastChar == ';' {
      i := m[1] - 2
        for ; i >= m[0]; i-- {
          if util.IsAlphaNumeric(line[i]) {
            continue
          }
          break
        }
        if i != m[1]-2 {
          if line[i] == '&' {
            m[1] -= m[1] - i
          }
        }
      }
```



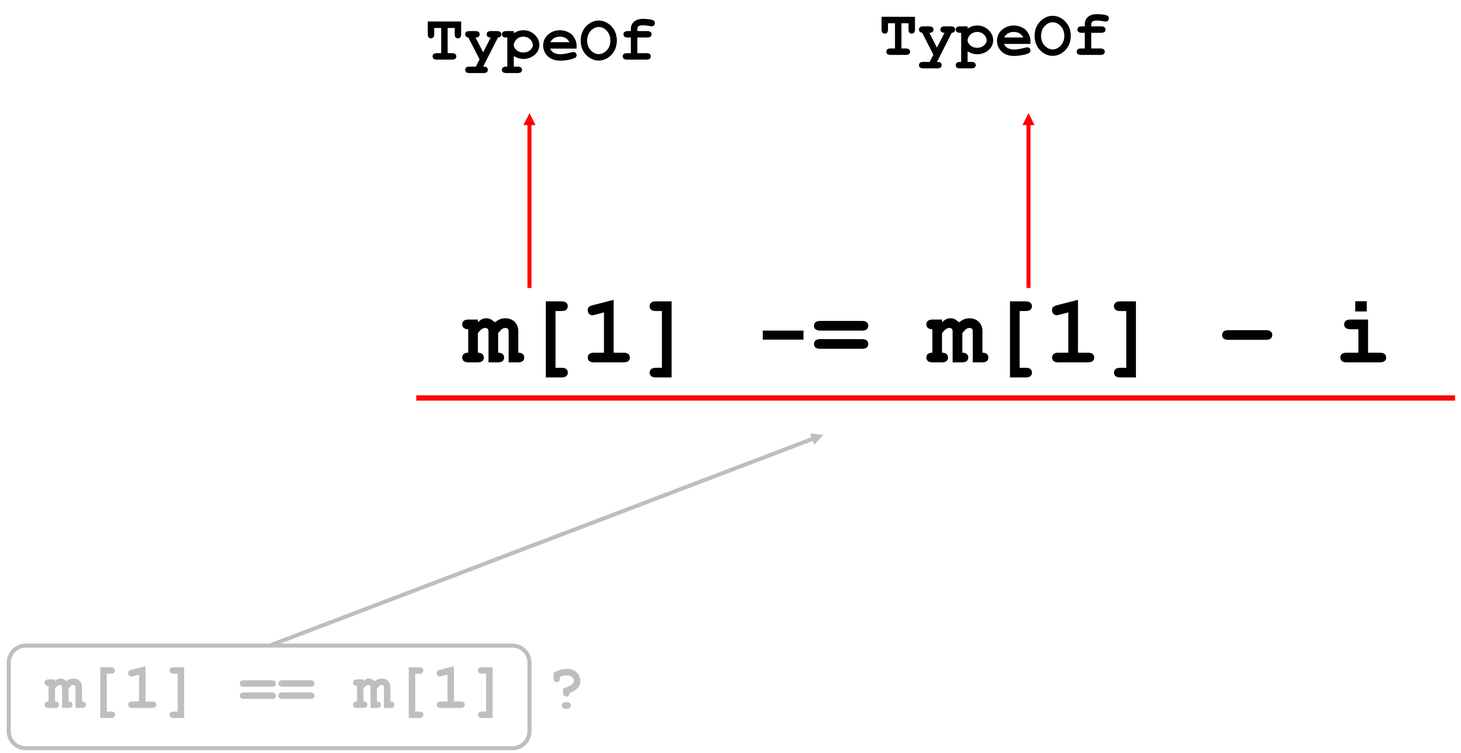
Поиск ошибок при помощи семантики


```
m[1] -= m[1] - i
```

```
m[1] == m[1]
```

?

Поиск ошибок при помощи семантики



В нашей диагностике

Внутри анализатора поле это номер

```
FieldAddr
   X:      t0
   Field: 1
```

В нашей диагностике

Внутри анализатора поле это номер

```
FieldAddr
   X:      t0
   Field: 1
```

Имя достаём из типа

```
elem := ptr.Elem().Underlying()
st   := elem.(*types.Struct)
st.Field(1).Name()
// "metricType"
```

В нашей диагностике

Внутри анализатора поле это номер

```
FieldAddr
   X:      t0
   Field: 1
```

Без `go/types` в предупреждении был бы **`field1`** вместо понятного пути до поля.

Имя достаём из типа

```
elem := ptr.Elem().Underlying()
st   := elem.(*types.Struct)
st.Field(1).Name()
// "metricType"
```

Интеграция со стандартными инструментами

Пакеты Go

`go/ast`	`go/types`	`go/token`	**`analysis`**
синтаксическое дерево	семантика	позиции в файле	`golang.org/x/tools/go/` фреймворк правил

go/analysis — фреймворк для анализаторов

- Правило — это analysis.Analyzer
- Парсинг и типизацию берёт на себя
- singlechecker, multichecker, готовые passes

Пакеты Go

`go/ast`
синтаксическое дерево

`go/types`
семантика

`go/token`
позиции в файле

`analysis`
`golang.org/x/tools/go/`
фреймворк правил

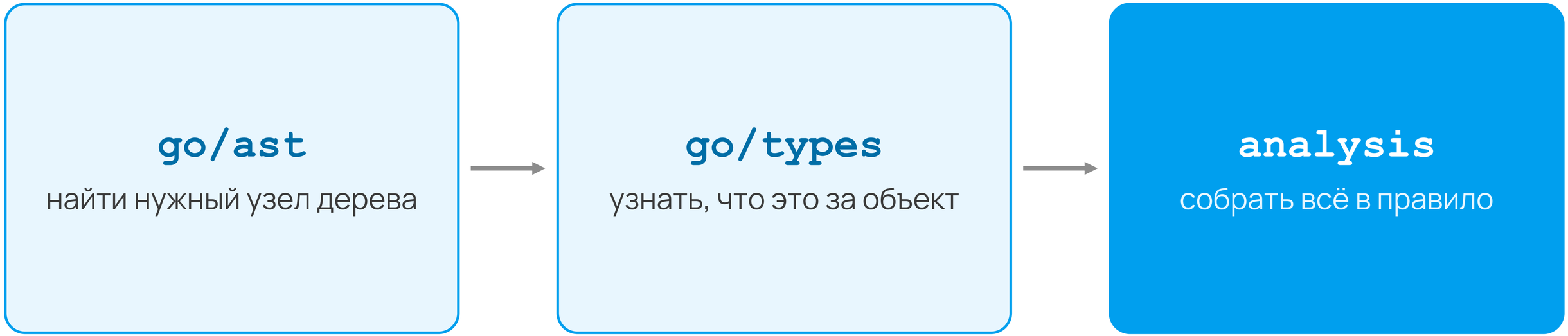
Опционально, когда нужно больше

`go/parser`
ручной парсинг без фреймворка

`go/packages`
загрузка пакетов с типами

`go/ssa`
dataflow и межпроцедурный анализ

Как всё сочетается



Дерево говорит, **как записано**. Типы — **что это значит**

Как устроен проход анализа

фреймворк делает это сам

здесь живёт ваше правило

go/analysis

go/analysis

- Каждое правило – объект типа ***`analysis.Analyzer`***

go/analysis

- Каждое правило – объект типа ***`analysis.Analyzer`***
  - название и описание диагностики
  - зависимости от других анализаторов
  - функция **`Run`**
  - способ вывода ошибок

go/analysis

- Каждое правило – объект типа ***analysis.Analyzer***

```
var Analyzer = &analysis.Analyzer{
  Name: "mycheck",
  Doc:  " Description of the rule",
  Run:  run,
}

func run(pass *analysis.Pass) (any, error) {
  // логика проверки
  return nil, nil
}
```

go/analysis

- Каждое правило – объект типа ***analysis.Analyzer***

```
var Analyzer = &analysis.Analyzer{
  Name: "mycheck",
  Doc:  " Description of the rule",
  Run:  run,
}


func run(pass *analysis.Pass) (any, error) {
  // логика проверки
  return nil, nil
}
```

go/analysis

- ***analysis.Pass***

go/analysis

- ***`analysis.Pass`***
  - *Files* – список AST-файлов *(**`ast.File`**)*, готовых к обходу

go/analysis

- ***`analysis.Pass`***
  - *Files* – список AST-файлов *(**`ast.File`**)*, готовых к обходу
  - ***`TypesInfo`*** - информация о типах (***`types.Info`***) для выражений в дереве

go/analysis

- ***analysis.Pass***
  - *Files* – список AST-файлов *(**ast.File**)*, готовых к обходу
  - ***TypesInfo*** - информация о типах (***types.Info***) для выражений в дереве
  - ***Pkg*** – текущий пакет

go/analysis

- ***analysis.Pass***
  - *Files* – список AST-файлов *(**ast.File**)*, готовых к обходу
  - ***TypesInfo*** - информация о типах (***types.Info***) для выражений в дереве
  - ***Pkg*** – текущий пакет
  - ***Reportf*** и ***Report*** — методы для вывода ошибок

go/analysis

- ***analysis.Pass***
  - *Files* – список AST-файлов *(**ast.File**)*, готовых к обходу
  - ***TypesInfo*** - информация о типах (***types.Info***) для выражений в дереве
  - ***Pkg*** – текущий пакет
  - ***Reportf*** и ***Report*** — методы для вывода ошибок
  - ***Fset*** — таблица позиций (***token.FileSet***)

go/analysis

- ***analysis.Pass***
  - *Files* – список AST-файлов *(**ast.File**)*, готовых к обходу
  - ***TypesInfo*** - информация о типах (***types.Info***) для выражений в дереве
  - ***Pkg*** – текущий пакет
  - ***Reportf*** и ***Report*** — методы для вывода ошибок
  - ***Fset*** — таблица позиций (***token.FileSet***)
  - ***ResultOf*** — данные от других анализаторов, если у правила есть зависимости

go/analysis

- ***singlechecker*** и ***multichecker***

go/analysis

- ***singlechecker*** и ***multichecker***

```
package main
import (
  "golang.org/x/tools/go/analysis/singlechecker")
func main() {
  singlechecker.Main(Analyzer)
}

go vet -vettool=./mycheck ./...
```

go/analysis

- ***singlechecker*** и ***multichecker***

```
multichecker.Main(
   rule1.Analyzer,
   rule2.Analyzer,
   rule3.Analyzer,
   ...
)
```

go/analysis

- Запуск через ***go vet***

```
go build -o mycheck ./cmd/mycheck
```

```
go vet -vettool=./mycheck ./...
```

Написание собственного анализатора

Попробуем сделать свою диагностику

Диагностика должна находить пустые **`if`** блоки вида:

```
if cond {
}
```

Правило использует пакет **`go/ast`** для обхода синтаксического дерева.

Попробуем сделать свою диагностику

Код в файле **main.go**:

```
package main
import (
   "emptyif"
   "golang.org/x/tools/go/analysis/singlechecker"
)
func main() {
   singlechecker.Main(emptyif.Analyzer)
}
```

Попробуем сделать свою диагностику

Код самого правила в файле **emptyif.go**

```
package emptyif

import (
  "go/ast"

  "golang.org/x/tools/go/analysis"
)

var Analyzer = &analysis.Analyzer{
  Name: "emptyif",
  Doc:  "reports empty if statements",
  Run:  run,
}
```

```
func run(pass *analysis.Pass) (any, error) {
  for _, file := range pass.Files {
    ast.Inspect(file, func(n ast.Node) bool {
      if stmt, ok := n.(*ast.IfStmt); ok {
        if stmt.Body != nil && len(stmt.Body.List) == 0 {
          pass.Reportf(stmt.Pos(), "empty if block")
        }
      }
      return true
    })
  }
  return nil, nil
}
```

Из чего состоит это правило

- Обход AST
- Проверка `IfStmt`
- Анализ тела блока
- Формирование отчёта

Из чего состоит это правило

- **Обход AST**

Используется **`ast.Inspect`**, позволяющий рекурсивно просматривать узлы дерева.

```
for _, file := range pass.Files {
   ast.Inspect(file, func(n ast.Node) bool {
```

- Проверка `IfStmt`
- Анализ тела блока
- Формирование отчёта

Из чего состоит это правило

- Обход AST
- **Проверка `IfStmt`**

Узел сравнивается через приведение типа:
n.(`*ast.IfStmt`)

```
if stmt, ok := n.(*ast.IfStmt); ok {
```

- Анализ тела блока
- Формирование отчёта

Из чего состоит это правило

- Обход AST
- Проверка `IfStmt`
- **Анализ тела блока**

`stmt.Body.List` содержит список операторов. Если он пустой, то выдаём предупреждение.

```
if stmt.Body != nil && len(stmt.Body.List) == 0 {
```

- Формирование отчёта

Из чего состоит это правило

- Обход AST
- Проверка `IfStmt`
- Анализ тела блока
- **Формирование отчёта**

`pass.Reportf` выводит сообщение с привязкой к точке в исходниках.

```
pass.Reportf(stmt.Pos(), "empty if block")
```

И что мы получим?

Если вы сделали всё правильно, то после сборки сможете запустить ваш анализатор:

```
go build -o emptyif.exe ./cmd/emptyif
go vet -vettool=D:\emptyif\emptyif.exe ./...
```

И что мы получим?

Вывод будет примерно таким:

```
tests\emptyif_test\file1.go:12:2: empty if block
```

Возвращаемся к нашей задаче

Логика диагностики

1. Метод, у которого получатель передан по значению
2. Внутри есть запись в поле этого получателя
3. Дальше по коду это поле никто не читает
4. Значит, запись потеряна. Выдаём предупреждение

Одного дерева будет мало. Смотрим на порядок выполнения и SSA

То же самое, но кодом


```
recv := fn.Receiver()
// 1. интересен только получатель по значению
if recv == nil || isPointer(recv) {
    return
}
// 2. все записи в поля этой копии
for _, w := range writesToFields(recv) {
    // 3. читают ли это поле после записи?
    if !hasReadAfter(w) {
        // 4. не читают, значит запись потеряна
        report(w, "поле копии, запись потеряна")
    }
}
```

Пример того, что мы найдем

- Milvus

```
func (gi GenericIndex) WithMetricType(metricType MetricType) {
    gi.baseIndex.metricType = metricType
}
```

- go vet и staticcheck молчат

Пример того, что мы найдем

- Milvus

```
func (gi GenericIndex) WithMetricType(metricType MetricType) {
    gi.baseIndex.metricType = metricType
}
```

- go vet и staticcheck молчат

Что выведет диагностика:

Modifying the 'gi.baseIndex.metricType' field of the 'gi' value receiver has no effect. Consider using a pointer receiver.

Меняется поле копии — оригинальный объект остаётся прежним

PVS-Studio 8.0

- Поддержка анализа Go, Js, Ts
  (помимо анализаторов C\C++\C#\Java)
- По 40 новых диагностик для каждого языка
  (помимо более 1000 существующих)
- Новые IDE и обновление классических
  (масштабная переработка VS Code)

Q / A