Applied Go code similarity analysis

Шарипов Искандер (ВКонтакте) @quasilyte

Golang Conf 2019

Профессиональная конференция для Go-разработчиков

Note: original gopher design by Renee French

VK backend infra team

VK backend infra team

Let’s start with a premise

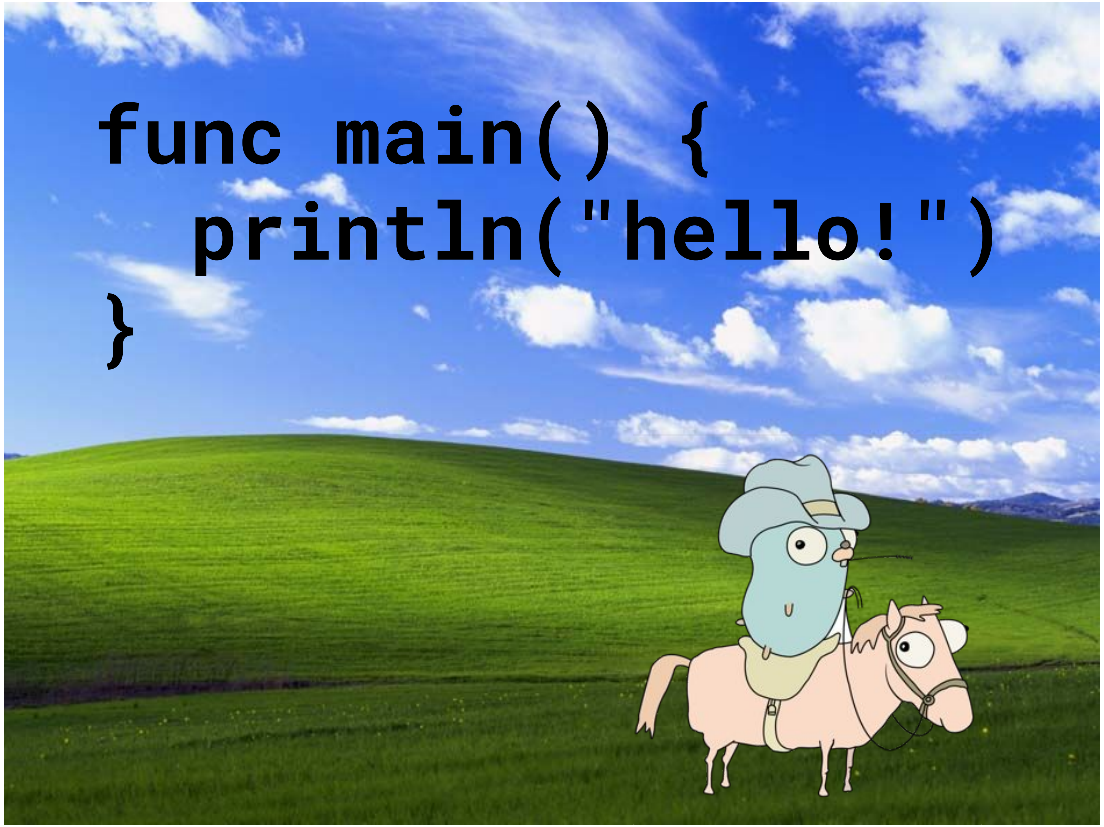
I’ll tell you some stories

Story-1

Intel days, intrinsics and append-combine, gogrep

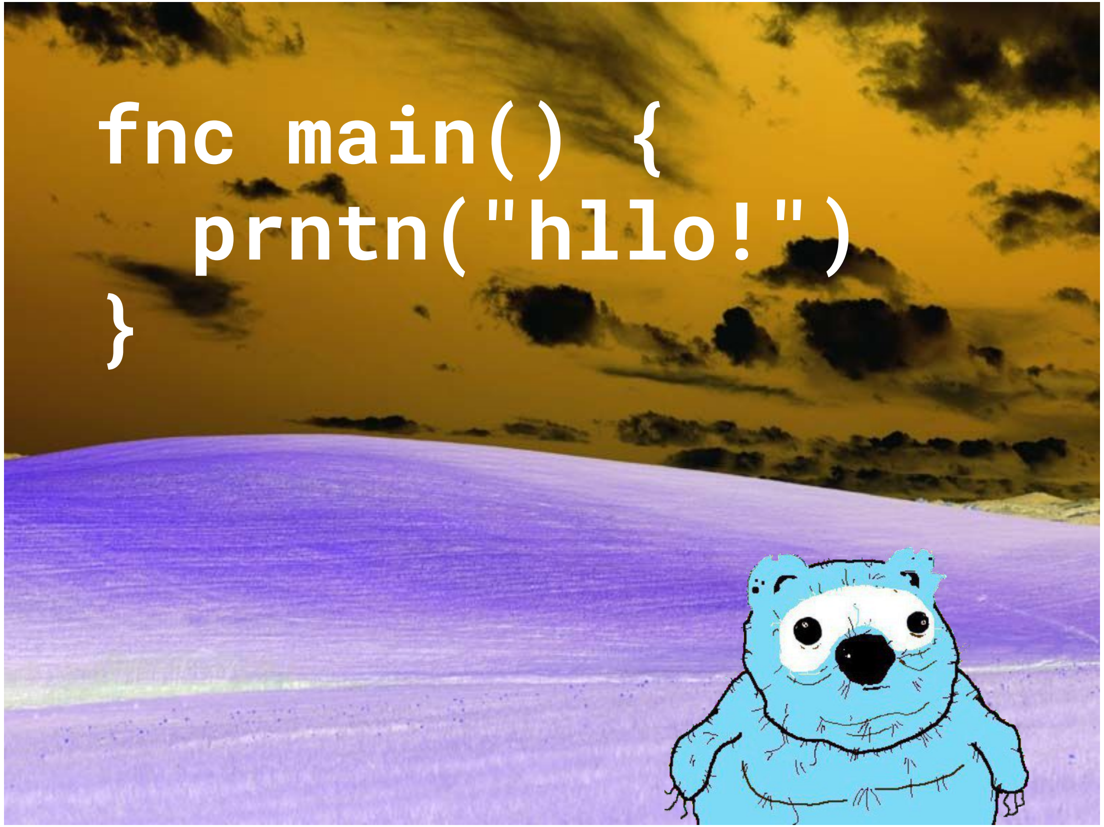
Story-2
Linters, normalization ideas, VK hackathon

Story-3
Ideas finalization, efficient algorithms, GolangConf



```
func main() {
  println("hello!")
}
```

How analyzers see code before optimizations



```
fnc main() {
  prntn("hllo!")
}
```

How analyzers after code before optimizations

The way: simplify, analyze, map back

Simplify before analysis,
but refer to original code inside warnings

The scope

The scope

❏ Code similarity analysis

The scope

- ❑ Code similarity analysis
    - ❑ Code duplication detection

The scope

- Code similarity analysis
  - Code duplication detection
    - => Function/type suggestions

The scope

- Code similarity analysis
    - Code duplication detection
        - => Function/type suggestions
- Code normalization

The scope

- Code similarity analysis
  - Code duplication detection
    - => Function/type suggestions
- Code normalization
  - => Simpler code analysis

“Applied”

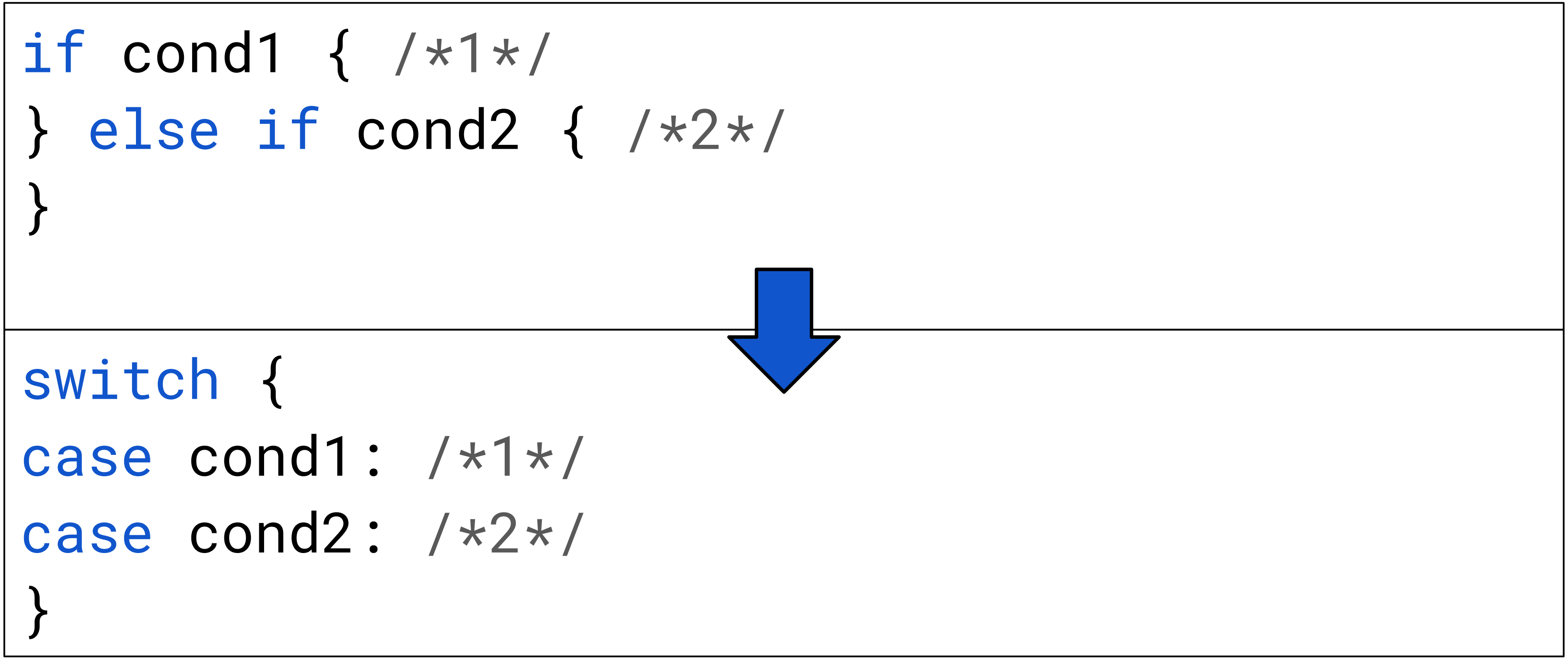
- Not hard to implement
- Works for the most cases
- Solves pragmatic problems

We're about to discuss...

- ❏ Code similarity evaluation
  - ❏ Code duplication detection
    - ❏ => Function/type suggestions
- ❏ Code normalization
  - ❏ => Simpler code analysis

Code normalization

Transforming code into its canonical form (what is canonical is up to us)



```
if cond1 { /*1*/
} else if cond2 { /*2*/
}
```

```
switch {
case cond1: /*1*/
case cond2: /*2*/
}
```

Statements and control flow rewrite

```
x := 0xff
y := int(30)
z := `s\`
```

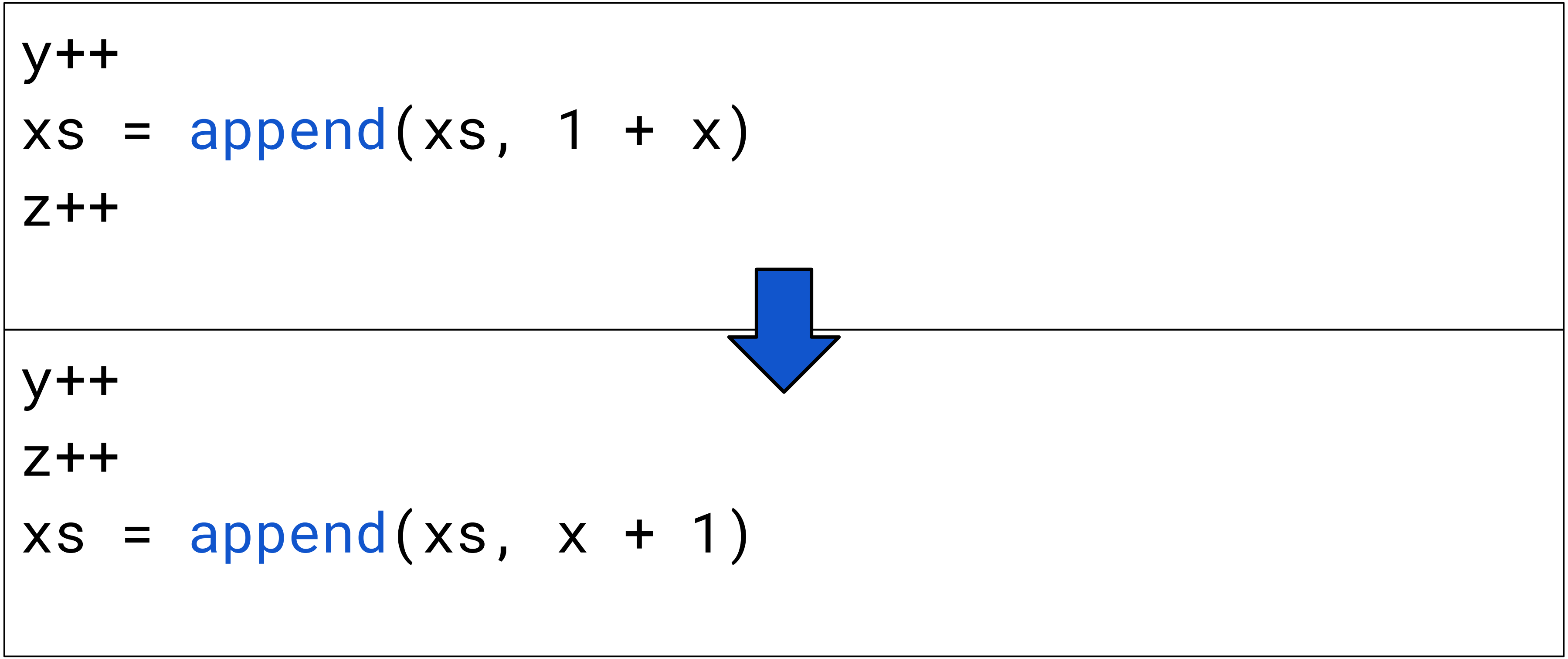
```
x := 256    // Base-10
y := 30     // Removed redundant type convert
z := "s\\" // Replace raw literals
```

Literals normalization

```
const n = 8
offset := n + unsafe.Sizeof(PairInt64{})
return size / n + offset
```

```
offset := 24
return size / 8 + 24
```

Constant folding and inlining

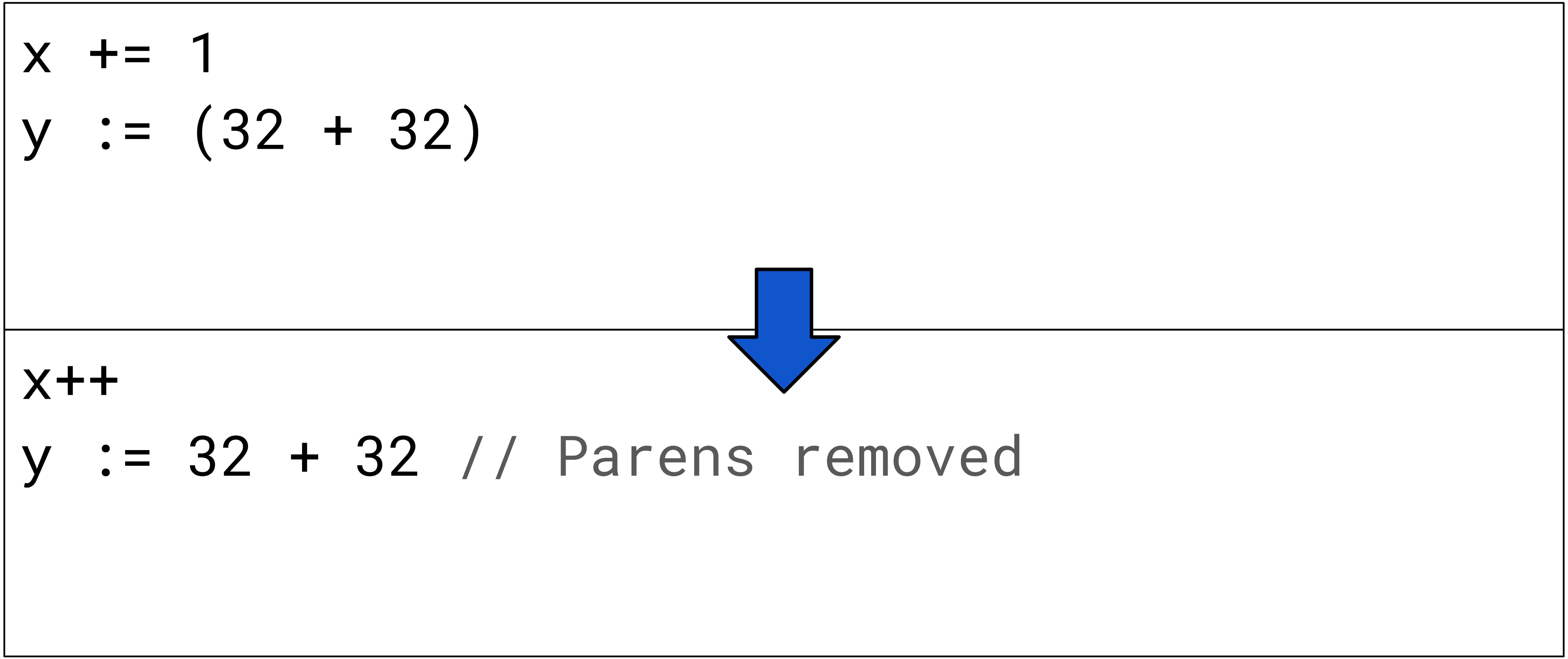


```
y++
xs = append(xs, 1 + x)
z++
```

```
y++
z++
xs = append(xs, x + 1)
```

Reordering and grouping

Syntax simplifications



```
x += 1
y := (32 + 32)
```

```
x++
y := 32 + 32 // Parens removed
```

And many more...

❑ Small functions inlining

❑ Loops de-unrolling

❑ Call substitution to another equivalent

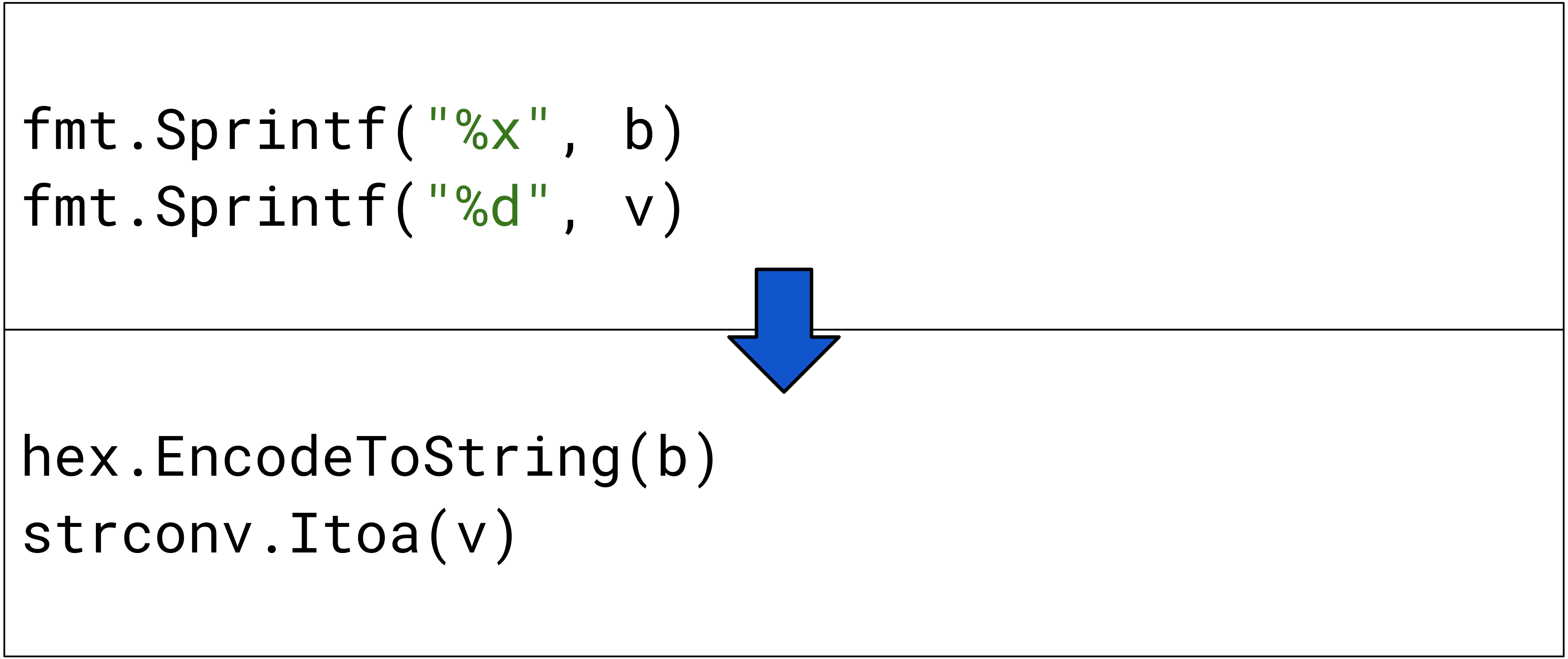
We want to make **identically behaving** code **syntactically identical**.

Normalization levels

To do normalization right,
we need to make some assumptions.

Not every code rewrite is valid in all contexts.

(Example with a call substitution follows.)



```
fmt.Sprintf("%x", b)
fmt.Sprintf("%d", v)
```

```
hex.EncodeToString(b)
strconv.Itoa(v)
```

Call substitution (only when enabled)

We’re about to discuss...

- ❑ Code similarity evaluation
  - ❑ Code duplication detection
    - ❑ => Function/type suggestions
- ❑ Code normalization
  - ❑ => Simpler code analysis

Improved AST analysis

Make static code analyzers find more bugs in your code without modifying their implementation

```go
func NotEqual(x1, x2 int) bool {
  return (x1) != x1
}
```

Improved AST analysis

```
// staticcheck gives no warnings.
func NotEqual(x1, x2 int) bool {
  return (x1) != x1
}
```

Improved AST analysis

```
// Now it does report “duplicated sub-expr”.
func NotEqual(x1, x2 int) bool {
  return x1 != x1
}
```

Improved AST analysis

```
if err == nil {
  return err
}
```

Improved AST analysis

```
// (A) Typo? Maybe != is intended.
if err == nil {
  return err
}
```

Improved AST analysis

```
// (B) Could just return nil.
if err == nil {
  return err
}
```

Improved AST analysis

```
// Code with the same semantics, but
// with less idiomatic syntax.
switch {
case err == nil:
  return err
}
```

Improved AST analysis

Generics will make things harder

(For Go language tools developers)

AST normalization experiment

github.com/quasilyte/astnorm

astnorm is an experimental library that provides normalization functions.

We're about to discuss...

- ❑ Code similarity evaluation
  - ❑ Code duplication detection
    - ❑ => Function/type suggestions
- ❑ Code normalization
  - ❑ => Simpler code analysis

Function suggestion

Normalize, compare,
match & suggest an appropriate
function

```
strings.Replace(s, old, new, -1)
^^ Suggest strings.ReplaceAll

func ReplaceAll(s, old, new string) string {
  return Replace(s, old, new, -1)
}
```

Function suggestion

```
if (len(k) >= len(h) && k[:len(h)] == h) {
    ^^ Suggest strings.HasPrefix

func HasPrefix(s, prefix string) bool {
  return len(s) >= len(prefix) &&
         s[:len(prefix)] == prefix
}
```

Function suggestion

Sounds cool, but how?

github.com/mvdan/gogrep

Searching Go code by syntax patterns

gogrep pattern example

```
if $*_; $x == nil {
    $*_
}
```

Optional “init statement”

gogrep pattern example

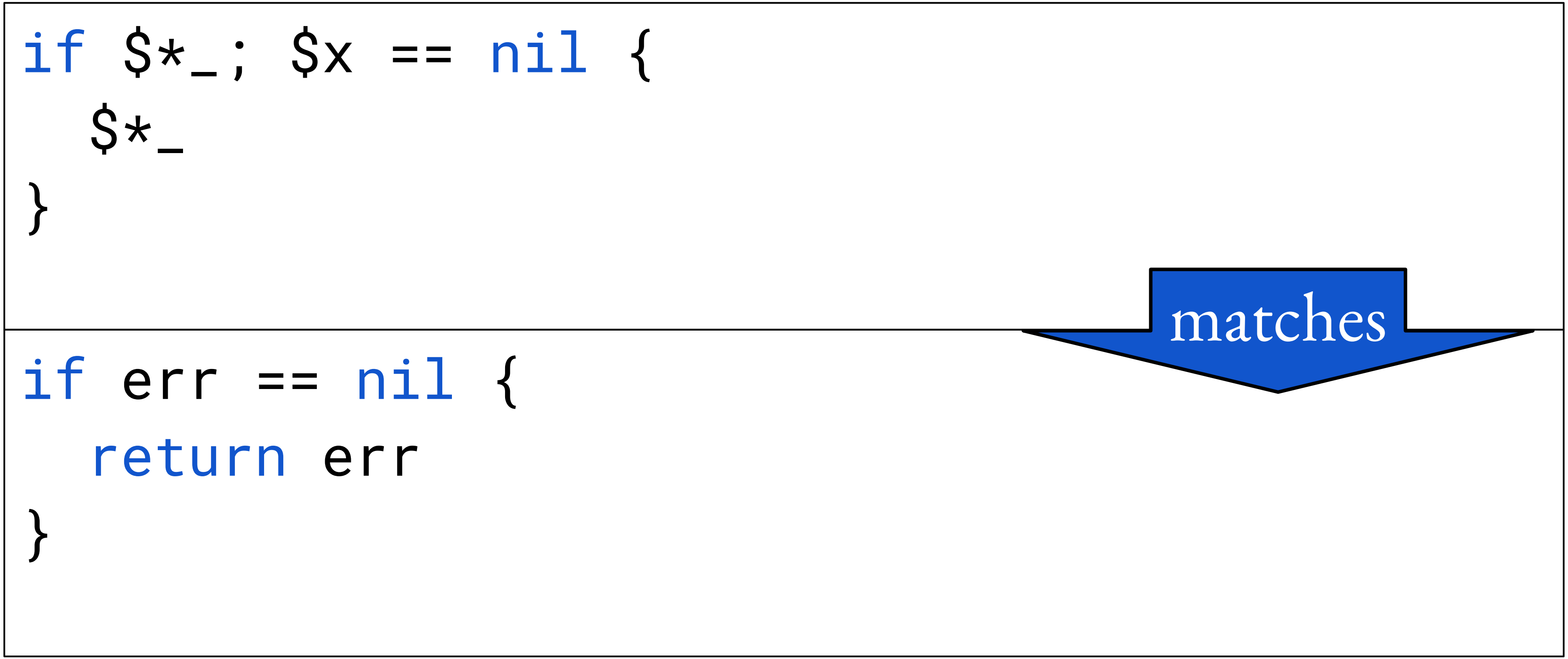
```
if $*_; $x == nil {
  $*_
}
```

Any expression

gogrep pattern example

```
if $*_; $x == nil {
   $*_
}
```

Zero or more statements

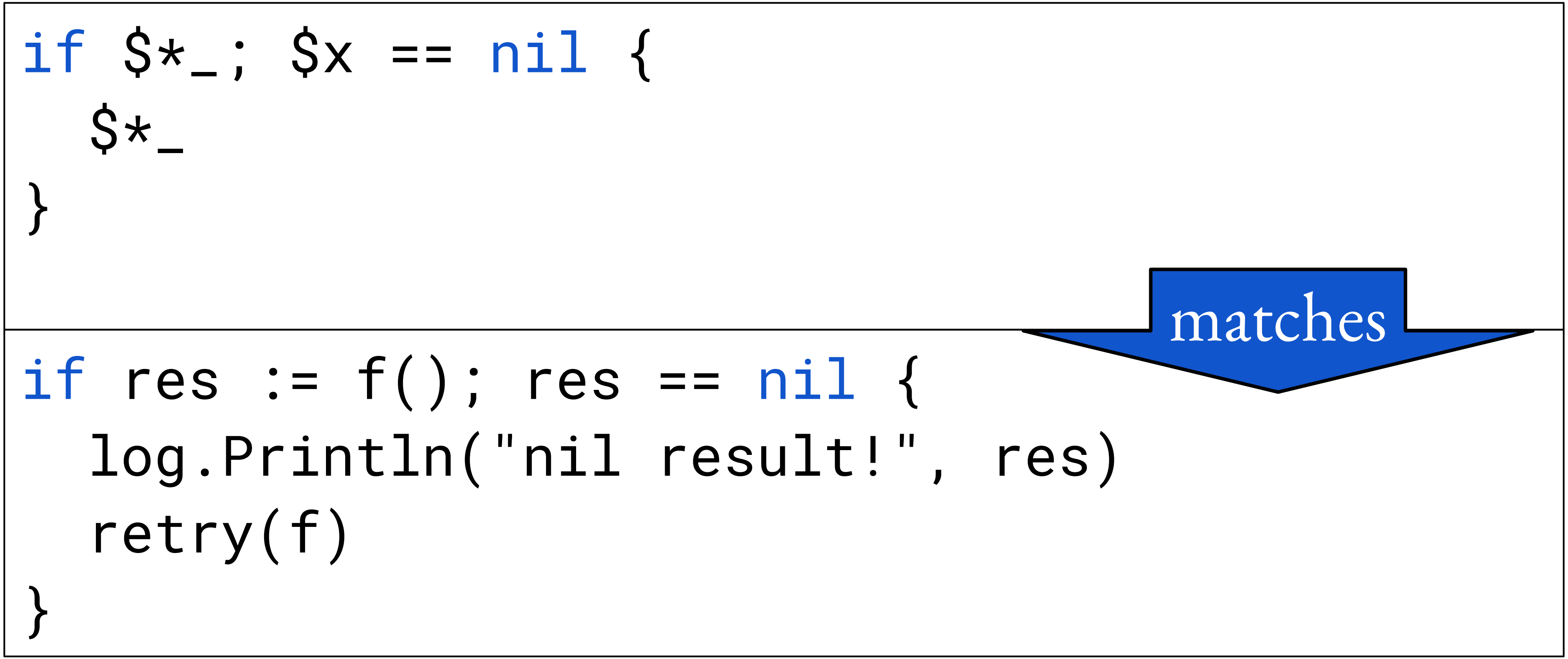


```
if $*_; $x == nil {
  $*_
}
```

matches

```
if err == nil {
  return err
}
```

Matching by patterns!



```
if $*_; $x == nil {
  $*_
}
```

matches

```
if res := f(); res == nil {
  log.Println("nil result!", res)
  retry(f)
}
```

Matching by patterns!

```
// gogrep pattern:
len($s) >= len($p) && $s[:len($p)] == $p

// matches:
len(k) >= len(h) && k[:len(h)] == h
len(f()) >= len(p) && f()[:len(p)] == p
len(x.a) >= len(x.b) && x.a[:len(x.b)] == x.b
```

Function suggestion with gogrep

```
// gogrep pattern:
len($s) >= len($p) && $s[:len($p)] == $p

// matches:
len(k) >= len(h) && k[:len(h)] == h
len(f()) >= len(p) && f()[:len(p)] == p
len(x.a) >= len(x.b) && x.a[:len(x.b)] == x.b
```

Function suggestion with gogrep

```
// gogrep pattern:
len($s) >= len($p) && $s[:len($p)] == $p

// matches:
len(k) >= len(h) && k[:len(h)] == h
len(f()) >= len(p) && f()[:len(p)] == p
len(x.a) >= len(x.b) && x.a[:len(x.b)] == x.b
```

Function suggestion with gogrep

```
// gogrep pattern:
len($s) >= len($p) && $s[:len($p)] == $p

// matches:
len(k) >= len(h) && k[:len(h)] == h
len(f()) >= len(p) && f()[:len(p)] == p
len(x.a) >= len(x.b) && x.a[:len(x.b)] == x.b
```

Function suggestion with gogrep

```
// gogrep pattern:
len($s) >= len($p) && $s[:len($p)] == $p

// matches:
len(k) >= len(h) && k[:len(h)] == h
len(f()) >= len(p) && f()[:len(p)] == p
len(x.a) >= len(x.b) && x.a[:len(x.b)] == x.b
```

Function suggestion with gogrep

Pattern-based matching benefits

Pattern-based matching benefits

❏ Makes variables/const names insignificant

Pattern-based matching benefits

❑ Makes variables/const names insignificant

❑ Can express “optional” parts

Pattern-based matching benefits

❏ Makes variables/const names insignificant

❏ Can express “optional” parts

gogrep also supports types information checking during matching.

grep=text search

gogrep=syntax search

normalize+gogrep=operation search

Type suggestion

Generalized function suggestion,
method set based

Type suggestion example

If someone implements **bytes.Reader** with a buffer, maybe they don’t know about **bufio.Reader**.

We're about to discuss...

- ❑ Code similarity evaluation
  - ❑ Code duplication detection
    - ❑ => Function/type suggestions
- ❑ Code normalization
  - ❑ => Simpler code analysis

Function duplicate matching

Finding funcs/methods that have duplicated bodies

Function duplicates detection

Function duplicates detection

1. Normalize the entire code base

Function duplicates detection

1. Normalize the entire code base
2. Build a map of {hash}=>{decl}

Function duplicates detection

1. Normalize the entire code base
2. Build a map of {hash}=>{decl}
3. For every collision report code duplication

```
// astHash computes a hash value for
// the given AST node.
func astHash
accepts (*token.FileSet, ast.Node)
returns (string, error)
```

AST hashing - function signature

```go
h := md5.New()
err := format.Node(h, fset, n)
if err != nil {
  return "", err
}
s := hex.EncodeToString(h.Sum(nil))
return s, nil
```

AST hashing - implementation

```
code := normalize(fn.Body)
key := astHash(fset, code)
if _, ok := funcTab[key]; ok {
  // Found duplicated code.
}
funcTab[key] = fn
```

Finding function duplicates

Function duplicates detection

It’s O(1) duplicate check and O(n) indexing!

Function duplicates detection

It’s O(1) duplicate check and O(n) indexing!

Won’t work for partial/local matching. :(

Function duplicates detection

It's O(1) duplicate check and O(n) indexing!

Won't work for partial/local matching. :(

Can ignore logging/printing/etc code while computing a hash, but it's dubious.

github.com/mibk/dupl

Code duplication detection tool, included in golangci-lint

dupl linter approach

❏ Syntax-based suffix tree

❏ Ignores most nodes values

Can tolerate value differences.

Can't tolerate non-normalized code.

Local matching

Intra-function code matching and function suggestions

```
if len(k) >= len(p) {
  return false
}
if data == nil {
  return false
}
return check(data) && k[:len(p)] == p
```

Intra-function suggestions

```
if len(k) >= len(p) {
  return false
}
if data == nil {
  return false
}
return check(data) && k[:len(p)] == p
// Related code is highlighted.
```

Intra-function suggestions

```
if len(k) >= len(p) {
  return false
}
if data == nil {
  return false
}
return check(data) && k[:len(p)] == p
// Unrelated code is highlighted.
```

Intra-function suggestions

```
return strings.HasPrefix(k, p) &&
       data != nil && check(data)

// Just use strings.HasPrefix!
```

Intra-function suggestions

We're about to discuss...

- ❑ Code similarity evaluation
  - ❑ Code duplication detection
    - ❑ => Function/type suggestions
- ❑ Code normalization
  - ❑ => Simpler code analysis

Partial matching

Matching similar, but not completely equal code chunks f(code1, code2) => % similarity

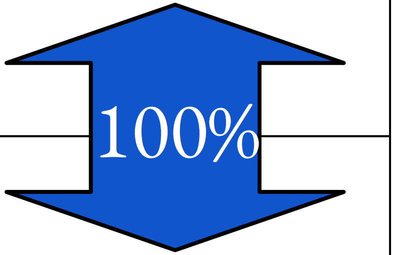
```go
data, err := readData(r)
if err != nil {
  return err
}
```

```go
data, err := readData(r)
if !(err == nil) {
  return err
}
```

Code similarity value

```go
data, err := readData(r)
if err != nil {
  return err
}
```

```go
data, err := readData(r)
if !(err == nil) {
  return err
}
```



Code similarity value: 100%

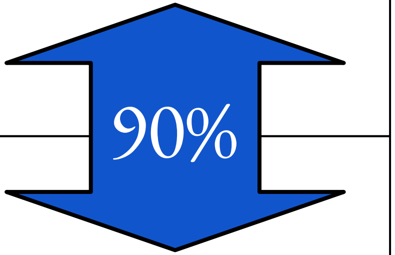
```go
data, err := readData(r)
if err != nil {
  panic(err)
}
```

```go
data, err := readData(r)
if err != nil {
  return err
}
```

Code similarity value

```
data, err := readData(r)
if err != nil {
  panic(err)
}
```

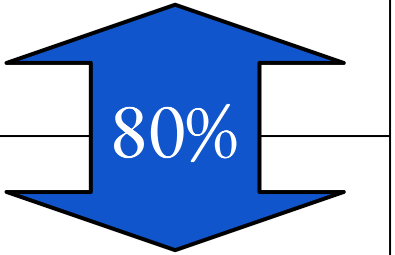
```
data, err := readData(r)
if err != nil {
  return err
}
```



Code similarity value: 90%

```go
data, err := readData(r)
if err != nil {
  panic("unexpected error")
}
```

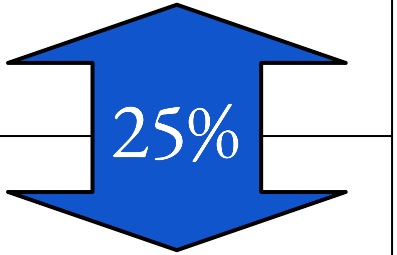
```go
data, err := readData(r)
if err != nil {
  return err
}
```



Code similarity value: 80%

```
data, _ := readData(r)
```

```
data, err := readData(r)
if err != nil {
  return err
}
```



Code similarity value: 25%

How to calculate code similarity?

The obvious idea is to calculate text distance of normalized code chunks

Text distance problems

Even word-oriented match would fail due to string literals, etc.

Solution A: AST traversal

Counting equal vs mismatching nodes; do recurse into non-matching nodes children

Solution B: text search+

Fix text issues that make text-based approach ineffective

```
if kind=="e" {
  return LevelError
} else if "w"==kind {
  return LevelWarning
}
// Default level.
return LevelInfo
```

```
if kind=="e" {
  return LevelError
} else if "w"==kind {
  return LevelWarning
}
// Default level.
return LevelInfo
```

Preprocessing

```
if kind=="e" {
  return LevelError
} else if "w"==kind {
  return LevelWarning
}
// Default level.
return LevelInfo
```

```
switch kind {
case "e":
  return 0
case "w":
  return 1
}
// Default level.
return 2
```

Preprocessing: normalize

Preprocessing: fold strings

```
if kind=="e" {
  return LevelError
} else if "w"==kind {
  return LevelWarning
}
// Default level.
return LevelInfo
```

```
switch kind {
case "#1":
  return 0
case "#2":
  return 1
}
// Default level.
return 2
```

```
if kind=="e" {
  return LevelError
} else if "w"==kind {
  return LevelWarning
}
// Default level.
return LevelInfo
```

```
switch kind {
case "#1":
  return 0
case "#2":
  return 1
}
return 2
```

Preprocessing: remove comments

```
if kind=="e" {
  return LevelError
} else if "w"==kind {
  return LevelWarning
}
// Default level.
return LevelInfo
```

```
switch v0 {
case "#1":
  return 0
case "#2":
  return 1
}
return 2
```

Preprocessing: rename local variables

Increasing text approach precision

- ❑ Normalize the code
- ❑ Fold string literals
- ❑ Remove comments
- ❑ Replace variable names

Increasing text approach precision

❑ Normalize the code

❑ Fold string literals

❑ Remove comments

❑ Replace variable names

=> Can now use Sphinx, Elasticsearch, etc.

Tools that use normalization...

Silence

Let's fix that!

Discussions and ideas are welcome

github.com/go-critic/go-critic

I’ll try to use normalization in go-critic static code analyzer

It should also be possible to write mapping-back to make normalization usable with third-party linters

Language-agnostic

All things discussed in this talk are language-agnostic and can apply to any programming language analysis

But Go is special...

But Go is special...

A lot of useful libraries to work with Go code inside the stdlib

But Go is special...

A lot of useful libraries to work with Go code inside the stdlib

language simplicity

println("Questions?")

@quasilyte

Golang Conf 2019