

IT Nights

phpgrep syntax-aware code search



Искандер @quasilyte

ВКонтакте



@quasilyte

- ❏ Go compiler (Go + x86 asm) @ Intel
- ❏ Backend infrastructure (Go) @ VK
- ❏ [Gopher Konstruktor](#)

Working on dev tools is my passion.



Talk structure

- ▣ **phpgrep vs grep**
- ▣ phpgrep features, pattern language
- ▣ Good use cases and examples
- ▣ PhpStorm structural search
- ▣ Code normalization and its applications

Talk structure

- ❏ phpgrep vs grep
- ❏ **phpgrep features, pattern language**
- ❏ Good use cases and examples
- ❏ PhpStorm structural search
- ❏ Code normalization and its applications

Talk structure

- ❏ phpgrep vs grep
- ❏ phpgrep features, pattern language
- ❏ **Good use cases and examples**
- ❏ PhpStorm structural search
- ❏ Code normalization and its applications

Talk structure

- ❏ phpgrep vs grep
- ❏ phpgrep features, pattern language
- ❏ Good use cases and examples
- ❏ **PhpStorm structural search**
- ❏ Code normalization and its applications

Talk structure

- ❏ phpgrep vs grep
- ❏ phpgrep features, pattern language
- ❏ Good use cases and examples
- ❏ PhpStorm structural search
- ❏ **Code normalization and its applications**

Today we're the code detective



First mission

Find all assignments, where assigned value is a string longer than 10 chars

Examples that should be matched

```
$s = "quite a long text";
```

```
$x = "text with \" escaped quote";
```

```
$arr[$key] = "a string key";
```

Let's try grep

Basically, regular expressions

grep (text level)

grep	\$
'\$x + \$y'	x
	_space
	+
	_space
	\$
	y

grep

```
$x = "this is a text";
```

Implication 1:
sees a line above as a sequence of characters

grep

```
$x = "this is a text";
```

Implication 2:
uses char-oriented pattern language (regexp)

grep

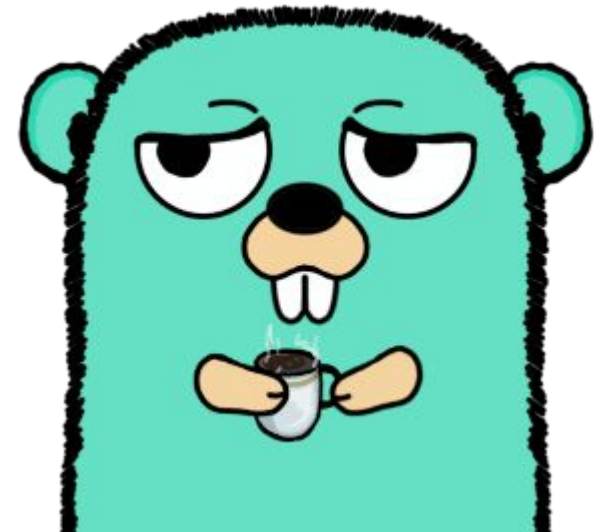
```
$x = "this is a text";
```

Implication 3:
doesn't know anything about PHP

```
$x = "this is a text";
```

```
\$\w+\s*=\s*"[^"]"{10,}\s*
```

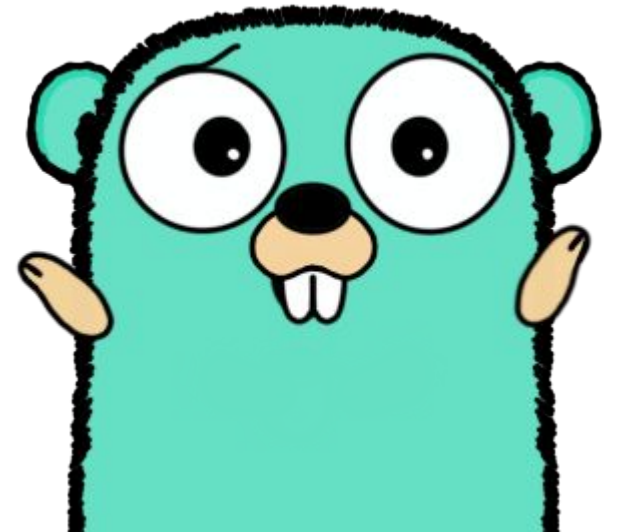
We need to deal with optional
whitespace, but that's OK




```
$x = "this is a text";
```

```
\$\w+\s*=\s*" [ ^ " ] {10,} "\s*
```

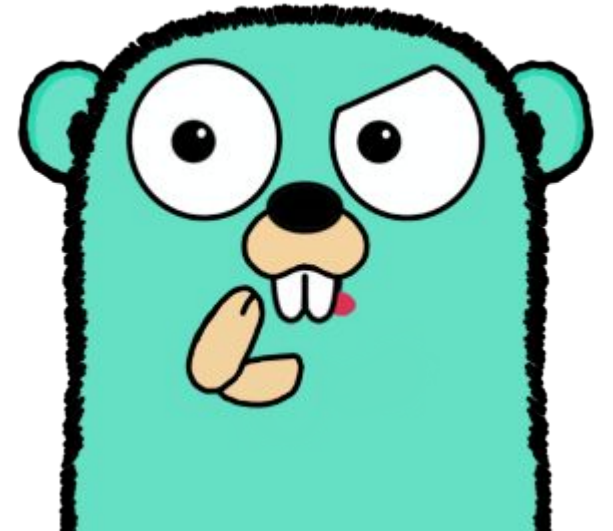
But this solutions is **wrong**.
It doesn't handle quote escaping



```
$x = "this is a text";
```

```
\$\w+\s*=\s*"?:[^\s]|\\.){10,}"\s*
```

Is it sufficient now?

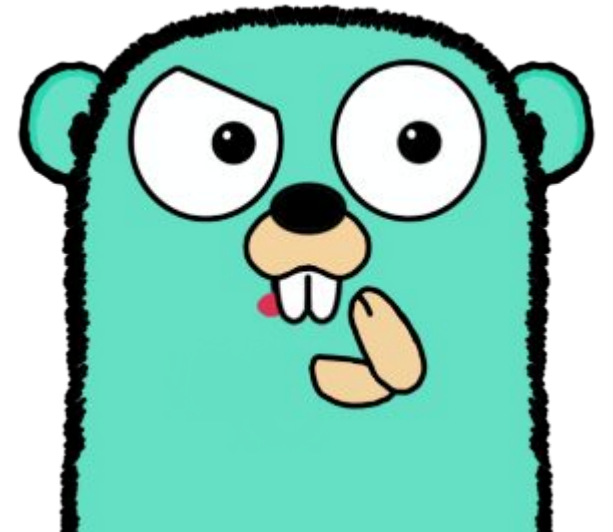


```
$x = "this is a text";
```

```
\$w+\s*=\s*"?:[^\s]|\\.){10,}\s*
```

Is it sufficient now?

Not really, we're still matching
only variable assignments



We (almost) succeeded, but...

Matching code with regexp is like trying to parse PHP using only regular expressions

phpgrep (syntax level)

phpgrep

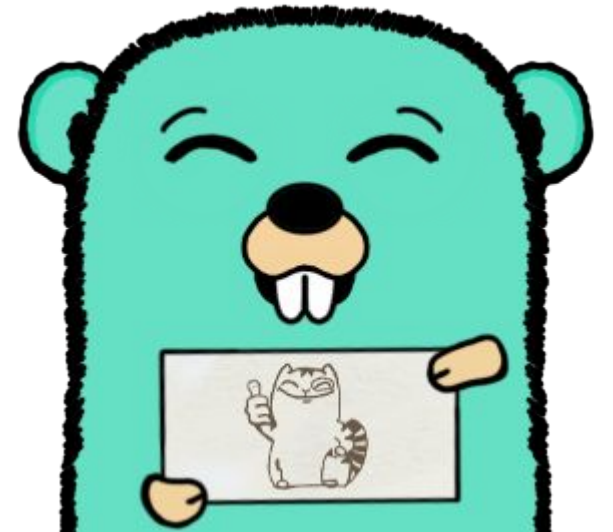
'\$x + \$y'

\$x
+
\$y

```
$x = "this is a text";
```

```
$_ = ${"s:str"}
```

Note that we don't care about
whitespace anymore



```
$x = "this is a text";
```

```
$_ = ${"s:str"} s~.{10,}
```

To apply “10 char length”
restrictions, we use result filtering
(more on that later)



Second mission

Find fstat function call with 2 arguments

Examples that should be matched

`fstat($f, $flags)`

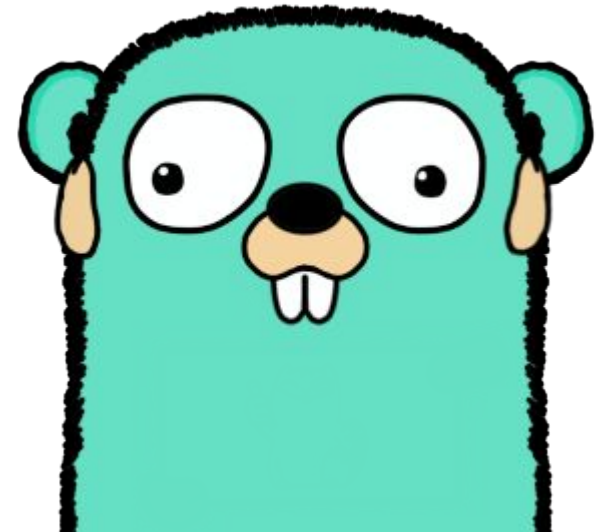
`fstat(getFile($ctx, $name), 0)`

`fstat($f->name, "b")`

```
fstat(f($x, $y), $flags);
```

```
fstat\(. *?, [^,]*\)
```

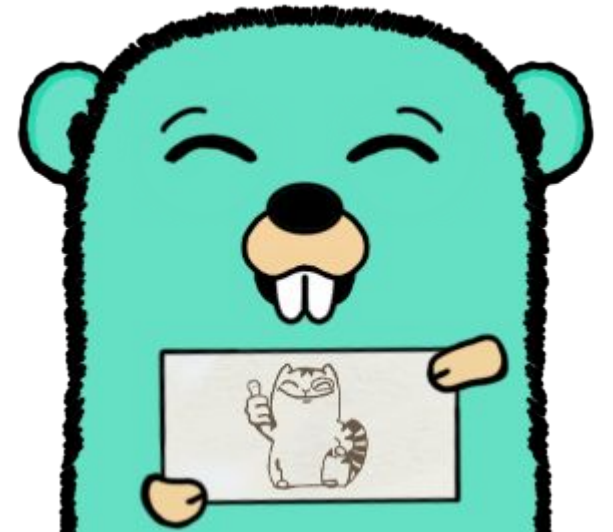
It's hard to match with regexp
because arguments may be
complex and contain commas (,)



```
fstat(f($x, $y), $flags);
```

```
fstat($_, $_)
```

With phpgrep it's super simple to
match arbitrary expressions



Third mission

Find array literals with duplicated keys
(No regexp example this time!)

Examples that should be matched

[7=>"1", 7=>"2"]

[\$x, 7=>"1", 7=>"2"]

[7=>"1", \$x, 7=>"2", \$y]

$[1 \Rightarrow \$x, 2 \Rightarrow \$y, 1 \Rightarrow \$z]$

$[\${"*"}, \$k \Rightarrow \$_, \${"*"}, \$k \Rightarrow \$_, \${"*"}]$

$\${"*"} - \text{capturing of 0-N exprs}$



[**1** => \$x, 2 => \$y, **1** => \$z]

[\$ { " * " } , **\$k** => \$ _ , \$ { " * " } , **\$k** => \$ _ , \$ { " * " }]

\$k - named non-empty expr
capture



$[1 \Rightarrow \$x, 2 \Rightarrow \$y, 1 \Rightarrow \$z]$

$[\$ \{ " * " \}, \$k \Rightarrow \$_ , \$ \{ " * " \}, \$k \Rightarrow \$_ , \$ \{ " * " \}]$

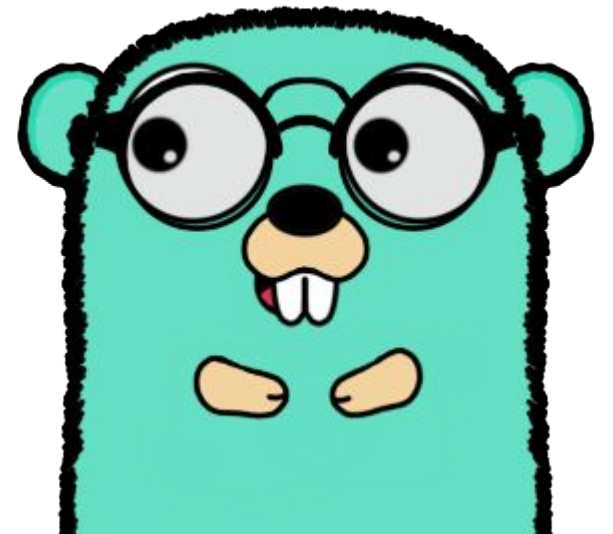
$\$_{}$ - any non-empty expr capture



[1=>\$x, 2=>\$y, 1=>\$z]

[\${" * "}, \$k=>\$_, \${" * "}, \$k=>\$_, \${" * "}]

An array with at least 2 identical
key exprs. They can be located at
any position



The pattern language

Features and syntax overview

Running phpgrep

```
phpgrep . '${"x:var"}++' 'x=i,j'
```

File or directory name to search.

By default, phpgrep recurses into nested directories

Running phpgrep

```
phpgrep . '${"x:var"}++' 'x=i,j'
```

Pattern to search, written in phpgrep pattern language (PPL)

Running phpgrep

```
phpgrep . '${"x:var"}++' 'x=i,j'
```

Additional filter (can have many) that excludes results if they don't match given criteria.

Every filter is a separate command-line arg

PPL (phpgrep pattern language)

It's almost normal PHP code, but with 2 differences to keep in mind.

1. `$<name>` is used for “any expr” matching
2. `${"<expr>"}` is a special matcher expression

PPL (phpgrep pattern language)

It's almost normal PHP code, but with 2 differences to keep in mind.

=> Can be parsed by any PHP parser.

PPL (phpgrep pattern language)

Matcher expressions can specify the kind of nodes to match.

PPL (phpgrep pattern language)

Matcher expressions can specify the kind of nodes to match.

Filters are used to add additional conditions to the matcher variables.

Pattern language

Matcher variables

$$\text{\$}x = \text{\$}y$$

All assignments

$$\text{\$}x = \text{\$}x$$

Self-assignments

Pattern language

Matching variables literally

```
$x 'x=foo'  
$foo variable
```

```
$x 'x~_id$'  
Variable with “_id” suffix
```

Pattern language

Matching strings

"abc"

"abc" string

`${"x:str"} 'x~abc'`

String that contains "abc"

Pattern language

Matching numbers

15

Int of 15

`${"s:int"} 'x=10,15'`

Int of 10 or 15

Use cases and (more) examples

Let's use phpgrep for something cool

Use case 1

Finding a bug over entire code base

Weird operations precedence

Easy to make mistake, tough consequences

```
const MASK = 0xff00;  
$x = 0x00ff;
```

```
$x & MASK > 128;  
// => false (?)
```


Weird operations precedence

Easy to make mistake, tough consequences

```
const MASK = 0xff00;  
$x = 0x00ff;
```

```
$x & (MASK > 128);
```

No, it returns 1 (which is true)!

Weird operations precedence

Easy to make mistake, tough consequences

```
const MASK = 0xff00;  
$x = 0x00ff;
```

```
( $x & MASK ) > 128;  
// => false
```

$$\$x \ \& \ \$mask \ > \ \$y$$

Finds all similar defects in the
code base



Pattern alternations

Currently, there is no way to express any kind of pattern alternation.

Pattern alternations

Currently, there is no way to express any kind of pattern alternation.

Can't say ``$x <op> $y`` to match any kind of binary operator `<op>`.

Patterns alternation

Workaround: running phpgrep several times

```
$x & $mask < $y  
$x & $mask == $y  
$x & $mask != $y
```

(And so on)

Use case 2

Running a set of phpgrep patterns as checks as a part of CI pipeline

Project-specific CI checks

Imagine that there are some project conventions you want to enforce.

Project-specific CI checks

Imagine that there are some project conventions you want to enforce.

You can write a set of patterns that catch them and make CI reject the revision.

Project-specific CI checks

1. Prepare a list of patterns.
2. For every pattern, write associated message.
3. Run phpgrep for every pattern inside pipeline.
4. If any of phpgrep runs matches, stop build.

For every match, print associated message

Project-specific CI checks

1. Prepare a list of patterns.
2. For every pattern, write associated message.
3. Run phpgrep for every pattern inside pipeline.
4. If any of phpgrep runs matches, stop build.

For every match, print associated message

Project-specific CI checks

1. Prepare a list of patterns.
2. For every pattern, write associated message.
3. Run `phpgrep` for every pattern inside pipeline.
4. If any of `phpgrep` runs matches, stop build.

For every match, print associated message

Project-specific CI checks

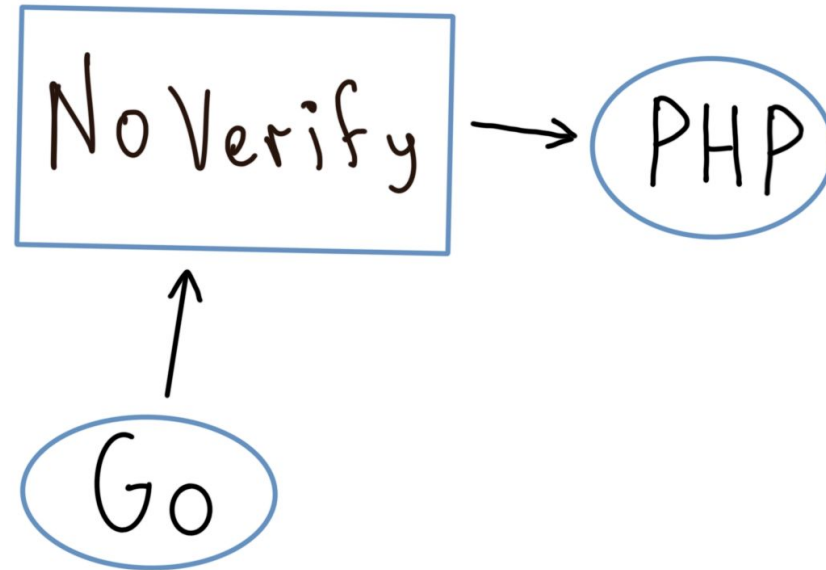
1. Prepare a list of patterns.
2. For every pattern, write associated message.
3. Run phpgrep for every pattern inside pipeline.
4. If any of phpgrep runs matches, stop build.

For every match, print associated message

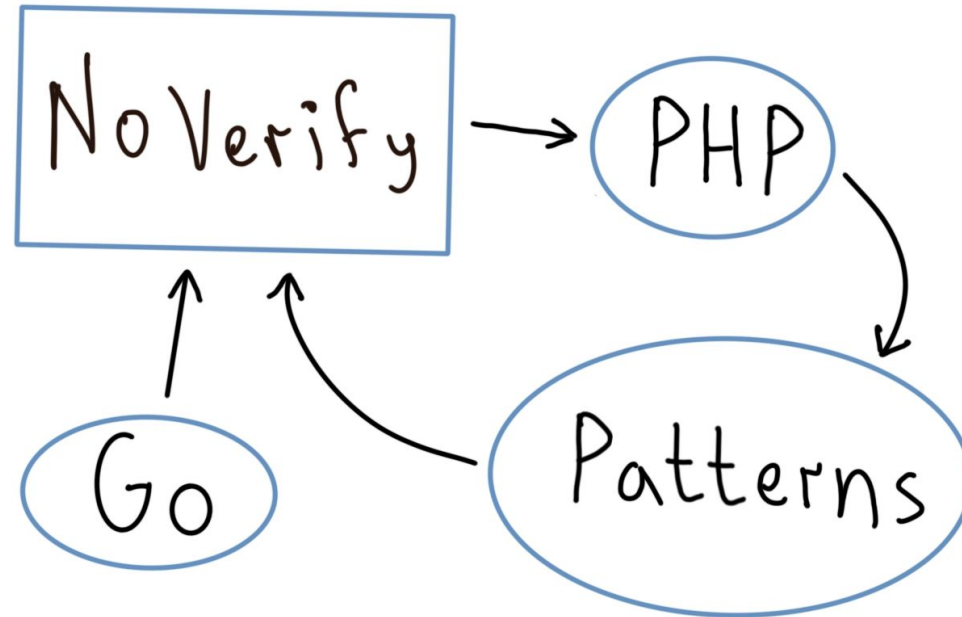
Pluggable linter rules

How phpgrep is planned to be used inside
NoVerify linter

NoVerify



NoVerify



Use case 3

Refactoring (search and replace)

Refactoring

Modernizing the code

```
array("${*}") => ["${*}"]
```

Replace old array syntax with new

```
isset($x) ? $x : $y => $x ?? $y
```

Use null coalescing operator

Refactoring

Project-specific evolution

```
// Don't use $conn default value!  
function derp($query, $conn = null)
```

```
    derp($x) <and> derp($x, null)
```

```
Find derp unwanted derp calls
```

phpgrep performance

Running a list of patterns:

- $O(N)$ complexity
- Becomes slow with high N

=> Optimizations are required

phpgrep performance

Can still be many times faster than grep with intricate regular expression.

It's a question of "a few seconds" vs "a several tens of minutes".

PhpStorm structural search

The closest functional equivalent to phpgrep

Structural search and replace (SSR)

There are some differences between the pattern languages used by PhpStorm and phpgrep.

- ❏ `$<name>$` used for all search “variables”
- ❏ All filters & options are external to the pattern

Structural search and replace (SSR)

There are some differences between the pattern languages used by PhpStorm and phpgrep.

- ❏ `$<name>$` used for all search “variables”
- ❏ All filters & options are external to the pattern

Structural search and replace (SSR)

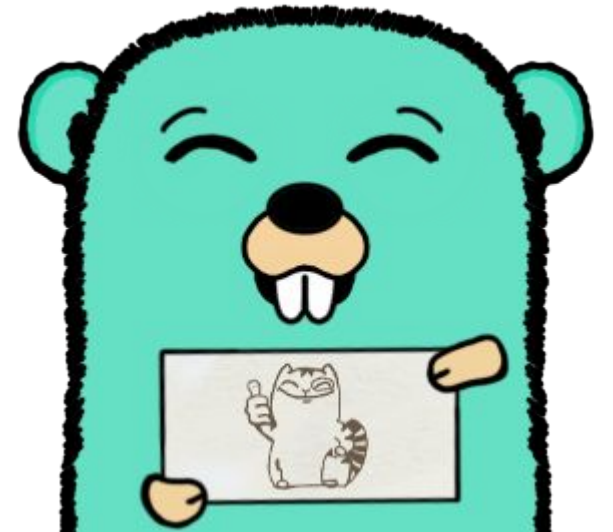
Filter examples.

- ❏ Regular expressions
- ❏ Type constraints
- ❏ Count (range)
- ❏ PSI-tree Groovy scripts

```
fstat(f($x, $y), $flags);
```

```
fstat($x$, $y$)
```

You can solve same tasks with
SSR in almost the same way



So, why making phpgrep?

We know that PhpStorm is cool, but...

- ❏ Not everyone is using PhpStorm
- ❏ phpgrep is a standalone tool
- ❏ phpgrep is a Go library, not just an utility

Why making phpgrep?

We know that PhpStorm is cool, but...

- ❏ Not everyone is using PhpStorm
- ❏ **phpgrep is a standalone tool**
- ❏ phpgrep is a Go library, not just an utility

Why making phpgrep?

We know that PhpStorm is cool, but...

- ❏ Not everyone is using PhpStorm
- ❏ phpgrep is a standalone tool
- phpgrep is a Go library, not just an utility

Why making phpgrep?

- ~~We know that PhpStorm is cool, but...~~
- ~~❑ Not everyone is using PhpStorm~~
 - ~~❑ phpgrep is a standalone tool without deps~~
 - ~~❑ phpgrep is a Go library, not just an utility~~



Everything becomes better when re-written in Go!

Code normalization

How we can do it and what it enables

What is code normalization?

It's a way to turn input source code X into a normal (canonical) form.

What is code normalization?

It's a way to turn input source code X into a normal (canonical) form.

Different input sources X and Y may end up in a same output after normalization.

What is code normalization?

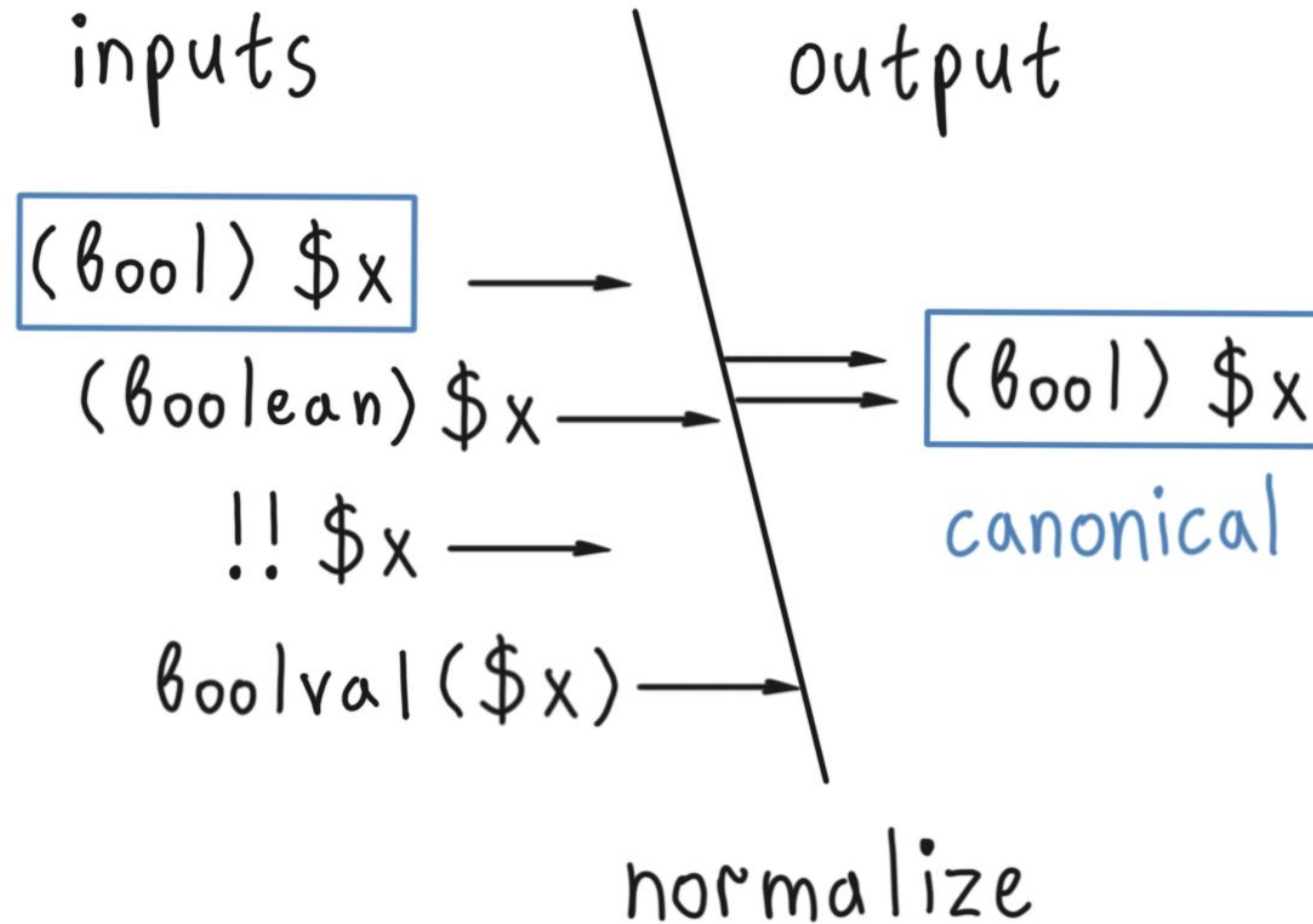
The exact rules of what is “normalized” are not that much relevant.

What is code normalization?

The exact rules of what is “normalized” are not that much relevant.




What is relevant is that among N alternatives we call only one of them as canonical.

Code normalization



Why we need normalization?

So your pattern can match more identical code.

-  Fuzzy code search
-  Code duplication/similarity analysis
-  Code simplifications, easier static analysis

Why we need normalization?

So your pattern can match more identical code.

- ❏ Fuzzy code search
- ❏ Code duplication/similarity analysis
- ❏ Code simplifications, easier static analysis

Why we need normalization?

So your pattern can match more identical code.

- ❏ Fuzzy code search
- ❏ Code duplication/similarity analysis
- ❏ Code simplifications, easier static analysis

But what about subtle details?

Some forms are **almost** identical,
but we still might want to consider them as
100% interchangeable.

But what about subtle details?

Some forms are **almost** identical,
but we still might want to consider them as
100% interchangeable.

We use “normalization levels” to control that.

Normalization levels

The best rule set depends on the goals.

Next statements apply:

Normalization levels

The best rule set depends on the goals.

Next statements apply:

- ❑ More strict => less normalization
- ❑ Less strict => more normalization

Matching more with less

Operation equivalence

Are expressions below identical?

```
intval($x)  
(int)$x
```

Matching more with less

Operation equivalence

Are expressions below identical?

`intval($x)`

`(int)$x`

Yes!

Matching more with less

Operation equivalence

Are expressions below identical?

$+\$x$
 $(\text{int})\$x$

Matching more with less

Operation equivalence

Are expressions below identical?

`+$x`

`(int)$x`

Not always!

Matching more with less

Operation equivalence

Are expressions below identical?

`+$x`

`(int)$x`

But sometimes we don't care

Matching more with less

Operation reordering

Are expressions below identical?

```
$x++; $y--;  
$y--; $x++;
```

Matching more with less

Operation reordering

Are expressions below identical?

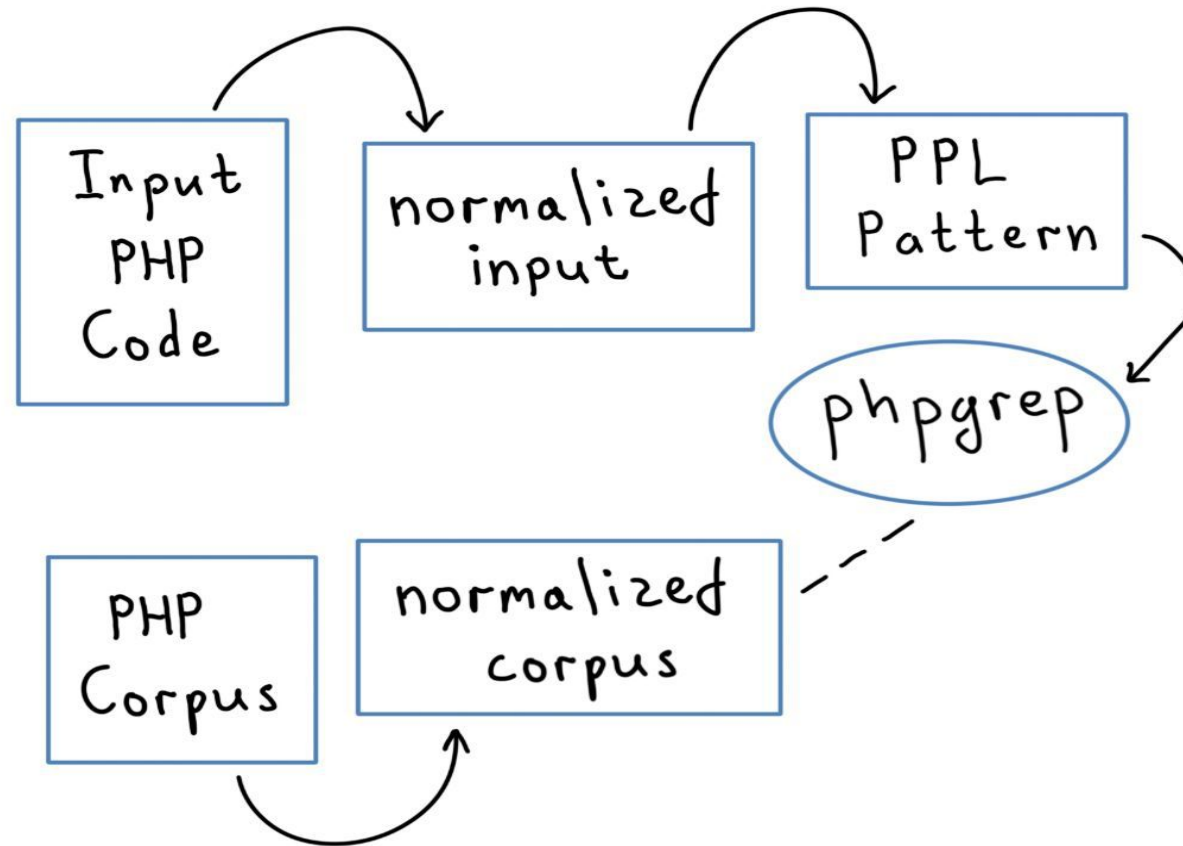
`$x++; $y--;`

`$y--; $x++;`

Independent ops can be reordered

Normalization level is a phpgrep parameter that
can improve the search results

Code search v2



Smooth transition slide...



The next time you're going to do code search,
make sure you're using the proper tools.

Like phpgrep and code normalization.

Closing words...



#golang
user group in Kazan
#GolangKazan



The end



Need more?

Slides that are optional.

Go performance

Answering why using Go is a viable option
for PHP tool.

What phpgrep does?

1. File I/O
2. PHP files parsing
3. The matching itself (AST against pattern)

What phpgrep does?

1. File I/O
2. PHP files parsing
3. The matching itself (AST against pattern)

With a careful use of goroutines, it's possible to make I/O faster.

What phpgrep does?

1. File I/O
2. PHP files parsing
3. The matching itself (AST against pattern)

(2) and (3) get a lot of benefits from the performance of compiled language.

Go memory management story

- ❏ Garbage collection
- ❏ Slices are the main “memory resource”
- ❏ Pointers should be local and short-lived

I'll explain why it matters.

Garbage collector

Needs to visit every reachable pointer.

Garbage collector

Needs to visit every reachable pointer.

More pointers => more work.

Garbage collector

Needs to visit every reachable pointer.

More pointers => more work.

Can take a **lot** of execution time.

Slice of strings, “hidden” pointers

```
pool := []string{s1, s2, s3}
```

```
slice := make([]string, N)  
for i := range slice {  
    slice[i] = pool[0]  
}
```

Slice of pool indexes, pointer-free

```
pool := []string{s1, s2, s3}
```

```
slice := make([]int, N)  
for i := range slice {  
    slice[i] = 0  
}
```

Slice of pool indexes, pointer-free

```
- slice := make([]string, N)
+ slice := make([]int, N)

- slice[i] = pool[0]
+ slice[i] = 0
```

Performance comparison

old time/op	new time/op	delta
2.53ms \pm 1%	0.42ms \pm 1%	-83.25%

Pointer-free code spends far less time in “runtime” (GC).

Go memory management story

- ❏ Garbage collection
- ❏ Slices are the main “memory resource”
- ❏ Pointers should be local and short-lived

Your memory pools should be slices of value types (i.e. `[]T` instead of `[]*T`).

Go memory management story

- ❏ Garbage collection
- ❏ Slices are the main “memory resource”
- ❏ Pointers should be local and short-lived

You return a pointer to a pool slice element.
That pointer should be as local as possible.