Code -> Linter rules

Pattern-based static analysis

Right into the action!

Step 1: find the bad code example

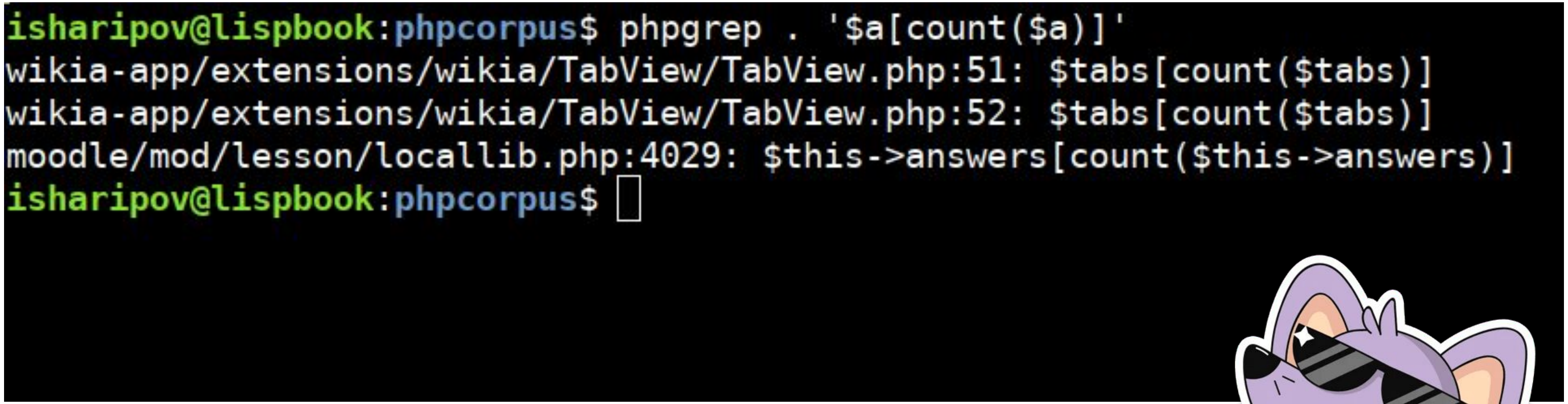
```
$last = $a[count($a)];
```

Off-by-one mistake

Step 2: extract it as a pattern

```
$last = $a[count($a)];
```

That's our pattern!



```
isharipov@lispbook:phpcorpus$ phpgrep . '$a[count($a)]'
wikia-app/extensions/wikia/TabView/TabView.php:51: $tabs[count($tabs)]
wikia-app/extensions/wikia/TabView/TabView.php:52: $tabs[count($tabs)]
moodle/mod/lesson/locallib.php:4029: $this->answers[count($this->answers)]
isharipov@lispbook:phpcorpus$
```

Step 3: apply the pattern

phpgrep by examples

```
@$_
```

Find all usages of error suppress operator

4.7s / 6kk SLOC / 56 Cores

```
in_array($x, [$y])
```

Find in_array calls that can be replaced with $x == $y

4.6s / 6kk SLOC / 56 Cores

```
$x ? true : false
```

Find all ternary expressions that could be replaced by just $x

4.7s / 6kk SLOC / 56 Cores

```
$_ == null
null == $_
```

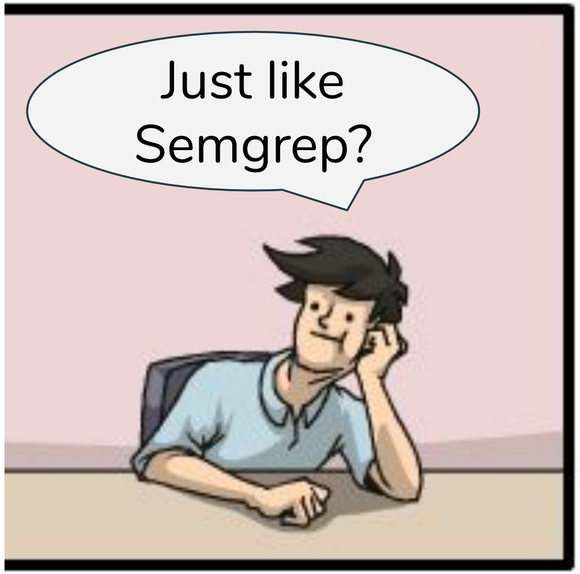
Find all non-strict comparisons with null

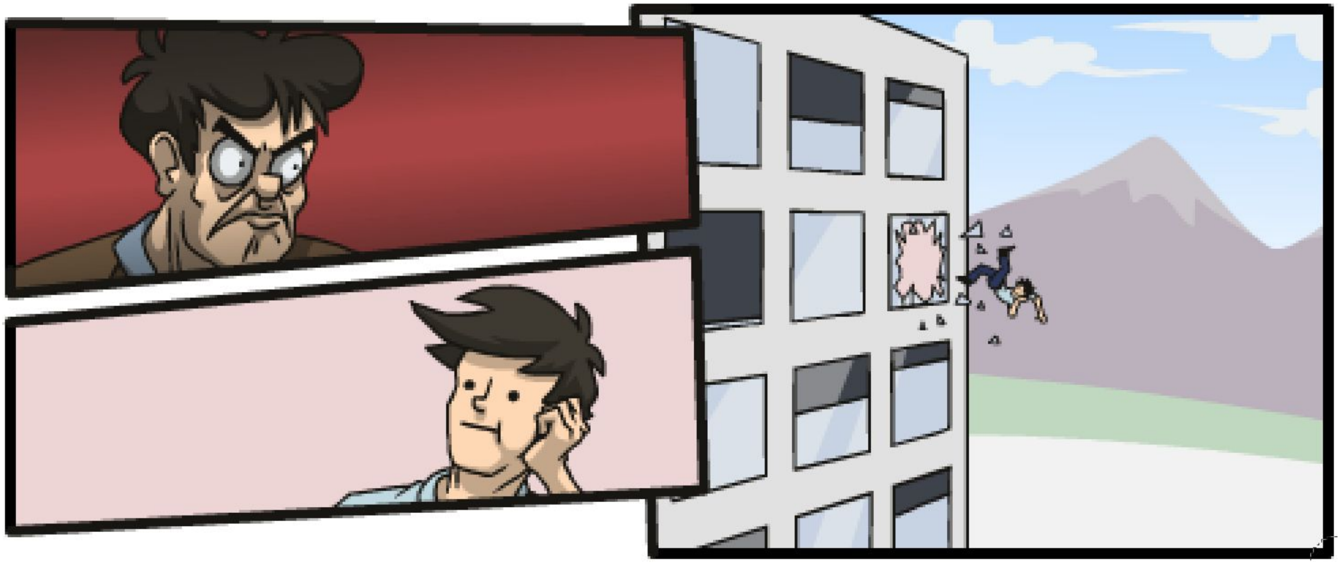
4.5s / 6kk SLOC / 56 Cores

```
for ($_ == $_; $_; $_) $_
```

Find for loops where == is used instead of = inside init clause

4.6s / 6kk SLOC / 56 Cores


Just like Semgrep?



NoVerify+phpgrep

Semgrep

Main topics for today

- A brief phpgrep history

Main topics for today

- A brief phpgrep history
- NoVerify dynamic rules

Main topics for today

- A brief phpgrep history
- NoVerify dynamic rules
- AST pattern matching

Main topics for today

- A brief phpgrep history
- NoVerify dynamic rules
- AST pattern matching
- Running rules efficiently

Main topics for today

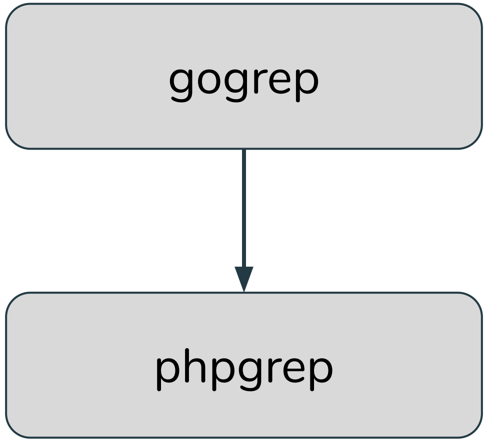
- A brief phpgrep history
- NoVerify dynamic rules
- AST pattern matching
- Running rules efficiently
- Dynamic rules pros & cons

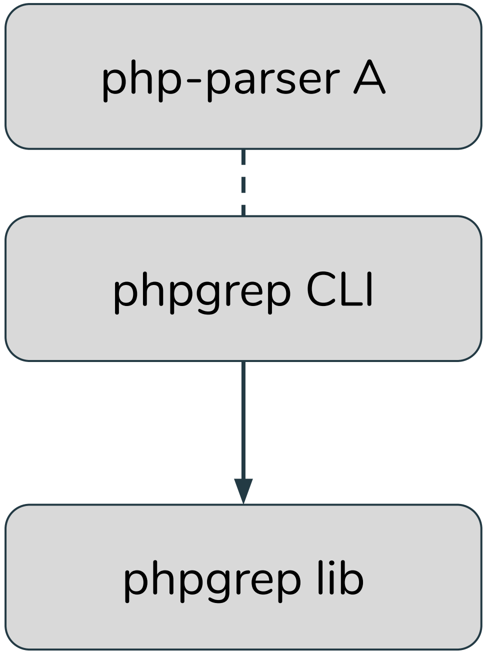
phpgrep history

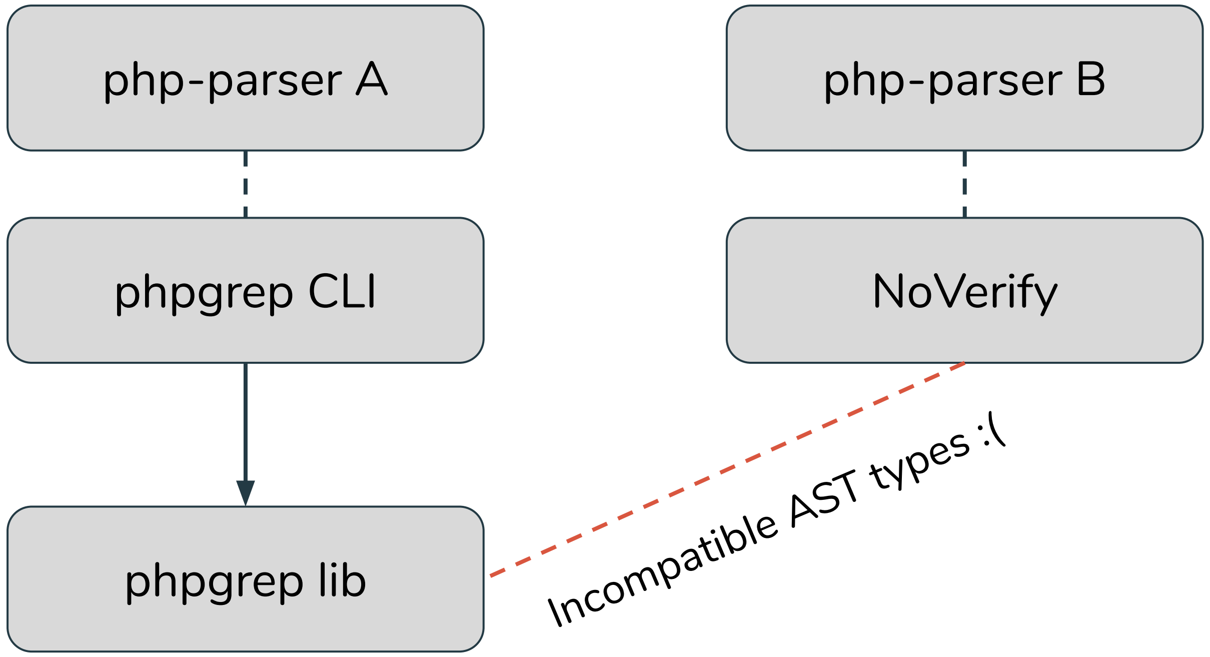
gogrep

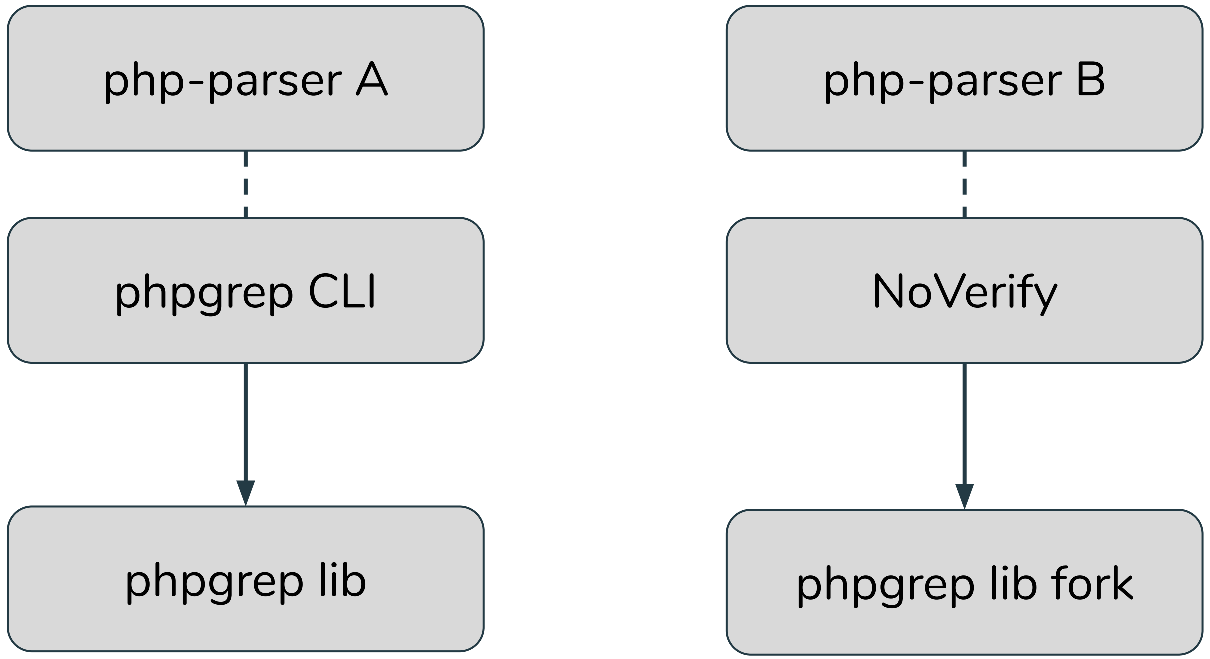
gogrep

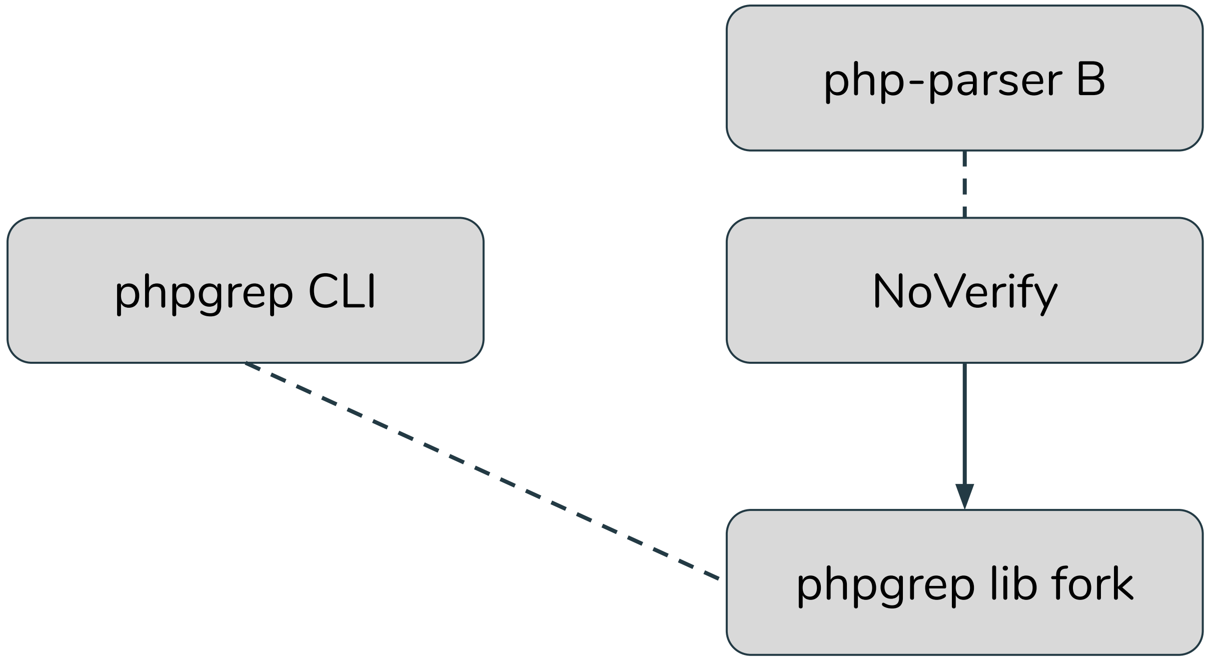
gogrep is cool!


gogrep
phpgrep


php-parser A
phpgrep CLI
phpgrep lib


php-parser A
php-parser B
phpgrep CLI
NoVerify
phpgrep lib
Incompatible AST types :(


php-parser A
phpgrep CLI
phpgrep lib
php-parser B
NoVerify
phpgrep lib fork


php-parser B
phpgrep CLI
NoVerify
phpgrep lib fork

NoVerify dynamic rules

Concepts overview

phpgrep

noverify

dynamic rules

Structural PHP search using AST patterns

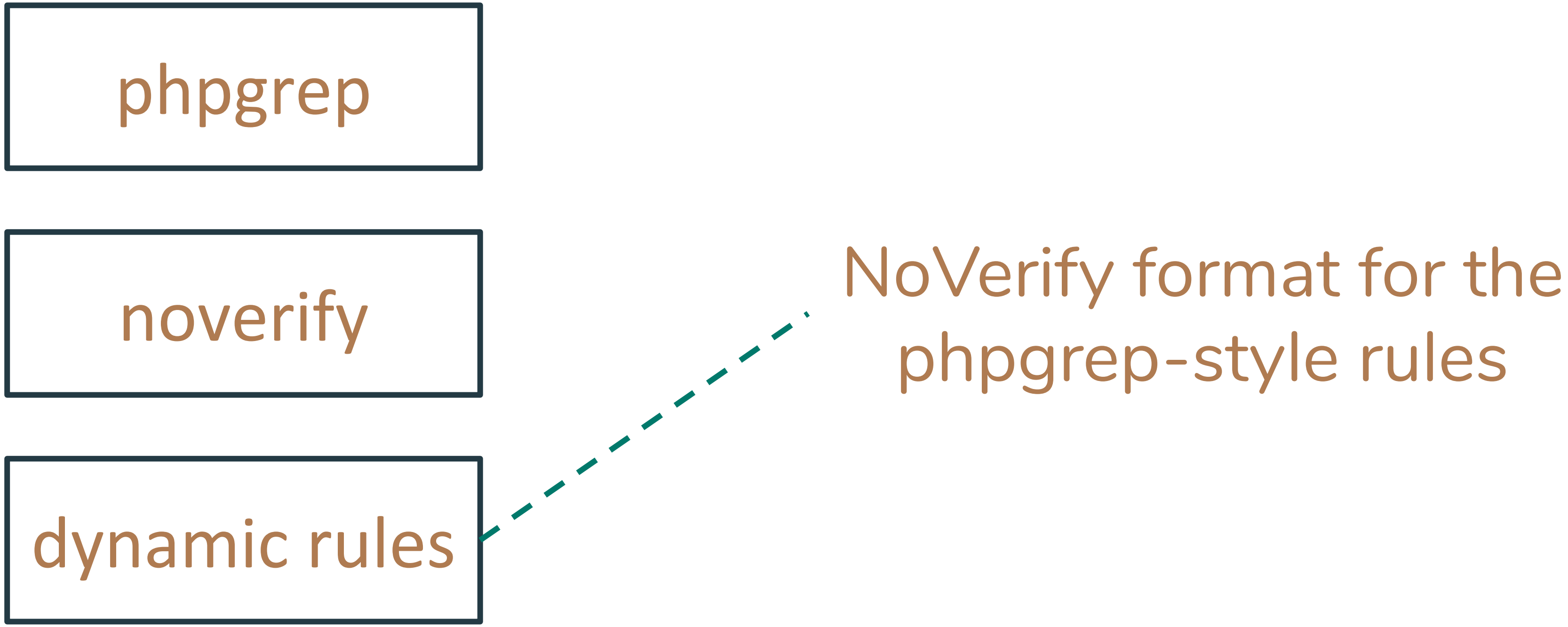
Concepts overview

phpgrep

noverify

dynamic rules

PHP linter capable of running dynamic rules



Concepts overview

phpgrep

noverify

dynamic rules

Written in GO

Concepts overview

Dynamic rules vs phpgrep

- Types info (NoVerify type inference)

Dynamic rules vs phpgrep

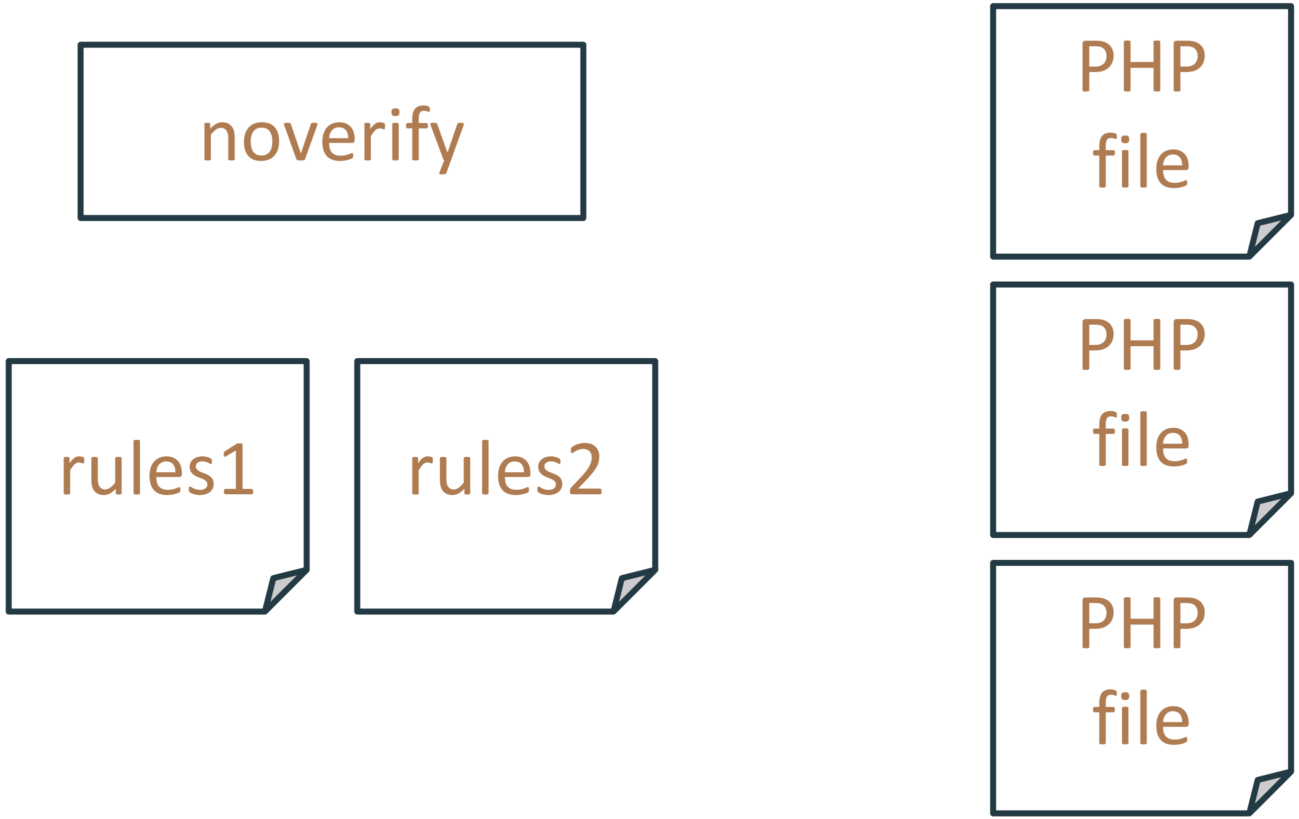
- Types info (NoVerify type inference)
- Efficient multi-pattern execution

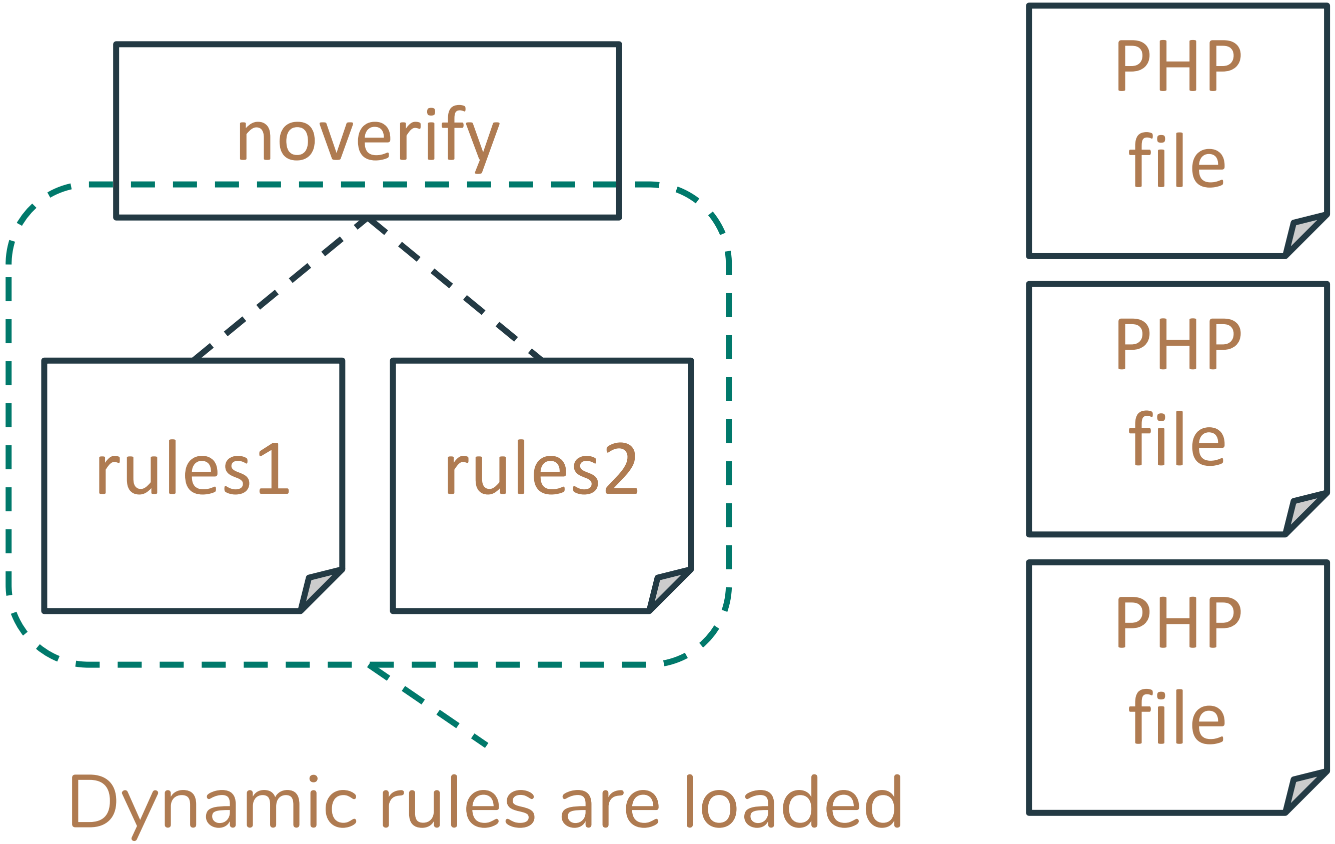
Dynamic rules vs phpgrep

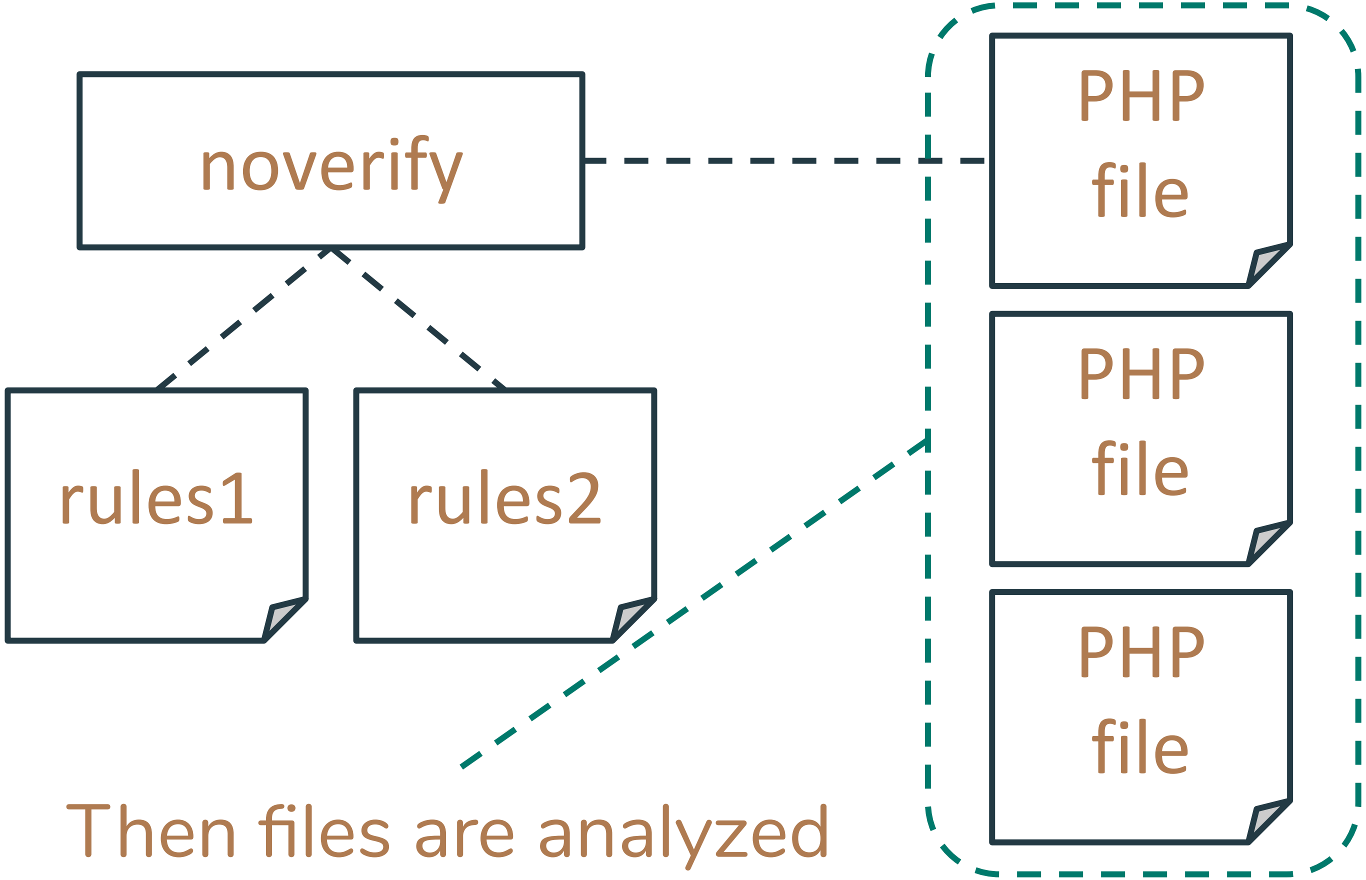
- Types info (NoVerify type inference)
- Efficient multi-pattern execution
- Logical pattern grouping

Dynamic rules vs phpgrep

- Types info (NoVerify type inference)
- Efficient multi-pattern execution
- Logical pattern grouping
- Documentation mechanisms


noverify
rules1
rules2
PHP file
PHP file
PHP file


noverify
rules1
rules2
Dynamic rules are loaded
PHP file
PHP file
PHP file



Then files are analyzed

Dynamic rule example

```
function ternarySimplify() {
  /** @warning rewrite as $x ?: $y */
  $x ? $x : $y;
}
```

Dynamic rule example

```
function ternarySimplify() {
  /** @warning rewrite as $x ?: $y */
  $x ? $x : $y;
}
```

Dynamic rules group name

Dynamic rule example

```
function ternarySimplify() {
  /** @warning rewrite as $x ?: $y */
  $x ? $x : $y;
}
```

Warning message

Dynamic rule example

```
function ternarySimplify() {
  /** @warning rewrite as $x ?: $y */
  $x ? $x : $y;
}
```

phpgrep pattern

Is this transformation safe?

```
f() ? f() : 0
      =>
  f() ?: 0
```

Is this transformation safe?

```
$$\boxed{f()}$$ ? f()  : 0
          =>
     f() ?: 0
```

Only if f() is free of side effects

Dynamic rule example (extended)

```
function ternarySimplify() {
  /**
   * @warning rewrite as $x ?: $y
   * @pure $x
   */
  $x ? $x : $y;
}
```

Dynamic rule example (extended)

```
function ternarySimplify() {
  /**
   * @warning rewrite as $x ?: $y
   * @pure $x
   */
  $x ? $x : $y;
}
```

$x should be side effect free

Dynamic rule example (extended)

```
function ternarySimplify() {
  /**
   * @warning rewrite as $x ?: $y
   * @pure $x
   * @fix $x ?: $y
   */
  $x ? $x : $y;
}
```

auto fix action for NoVerify

Dynamic rule example (@comment)

```
/**
 * @comment Find ternary expr that can be simplified
 * @before  $x ? $x : $y
 * @after   $x ?: $y
 */
function ternarySimplify() {
  // ...as before
}
```

Dynamic rule documentation

```
function argsOrder() {
  /** @warning suspicious args order */
  any: {
    str_replace($_, $_, ${"char"}, ${"*"});
    str_replace($_, $_, "", ${"*"});
  }
}
```

```
function argsOrder() {
  /** @warning suspicious args order */
  any: {
    str_replace($_, $_, ${"char"}, ${"*"});
    str_replace($_, $_, "", ${"*"});
  }
}
```

“any” pattern grouping

```
function bitwiseOps() {
  /**
   * @warning maybe && is intended?
   * @fix $x && $y
   * @type bool $x
   * @type bool $y
   */
  $x & $y;
}
```

```
function bitwiseOps() {
  /**
   * @warning maybe && is intended?
   * @fix $x && $y
   * @type bool $x
   * @type bool $y
   */
  $x & $y;
}
```

Type filters

Type matching examples

`T`	T typed expression
`object`	Arbitrary object type
`T[]`	Array of T-typed elements
`!T`	Any type except T
`!(A|B)`	Any type except A and B
`?T`	Same as (T|null)

```
function stringCmp() {
  /**
   * @warning compare strings with ===
   * @fix $x === $y
   * @type string $x
   * @or
   * @type string $y
   */
  $x == $y;
}
```

```
function stringCmp() {
  /**
   * @warning compare strings with ===
   * @fix $x === $y
   * @type string $x
   * @or
   * @type string $y
   */
  $x == $y;
}
```

Or-connected constraints

How to run custom rules

1. Create a rules file
2. Run NoVerify with -rules flag

```
$ noverify -rules rules.php target
```

AST pattern matching

“$x = $x” pattern string

“$x = $x” pattern string

⇩

Parsed AST

“$x = $x” pattern string

⇩

Parsed AST

⇩

Modified AST (with meta nodes)

Matching AST

```
function match(Node $pat, Node $n)
```

```
$pat is a compiled pattern
$n is a node being matched
```

Algorithm

- Both $pat and $n are traversed
- Non-meta nodes are compared normally
- $pat meta nodes are separate cases
- Named matches are collected (capture)

Meta node examples

- $x is a simple “match any” named match
- $_ is a “match any” unnamed match
- ${"str"} matches string literals
- ${"str:x"} is a capturing form of ${"str"}
- ${"*"} matches zero or more nodes

Valid PHP Syntax!

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

matches

```
$foo->x = "abc";
     $x = '';
```

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

rejects

```
$foo->x = f();
     $x = $y;
```

```
f()
```

```
matches
```

```
f()
F()
```

Unless explicitly marked as case-sensitive

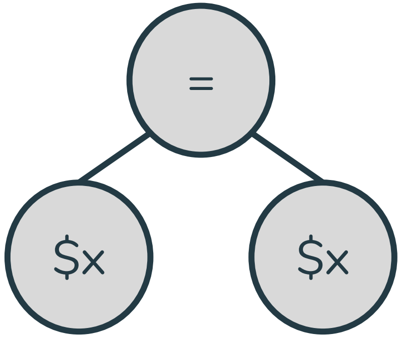
```
new T()
```

matches

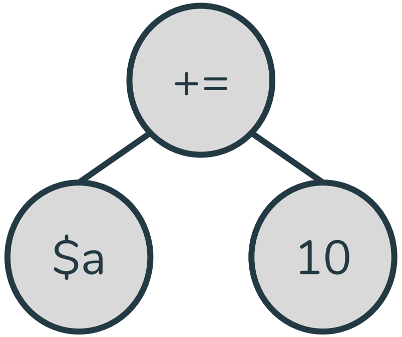
```
new T()
new t()
```

Unless explicitly marked as case-sensitive

Pattern $x=$x

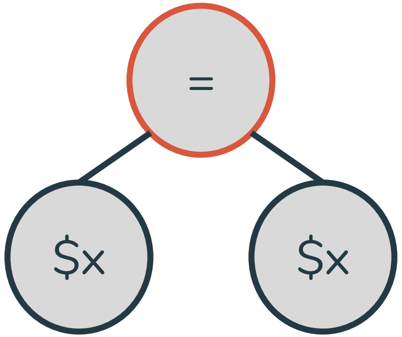


Target $a+=10

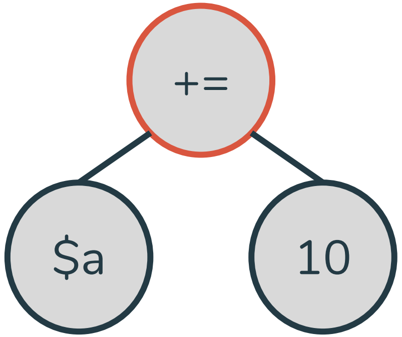


Pattern matching

Pattern $x=$x

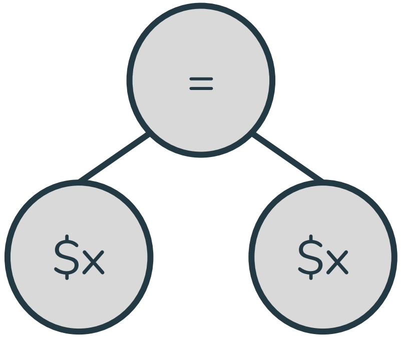


Target $a+=10

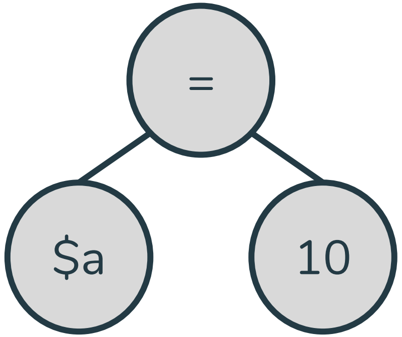


Pattern matching

Pattern $x=$x

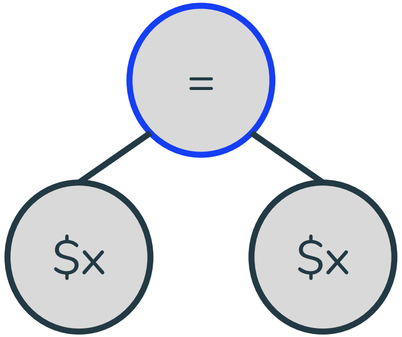


Target $a=10

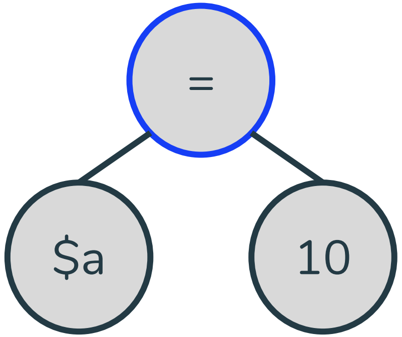


Pattern matching

Pattern $x=$x

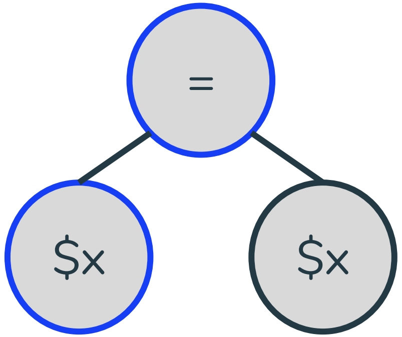


Target $a=10

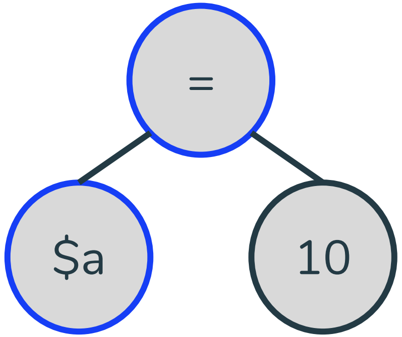


Pattern matching

Pattern $x=$x



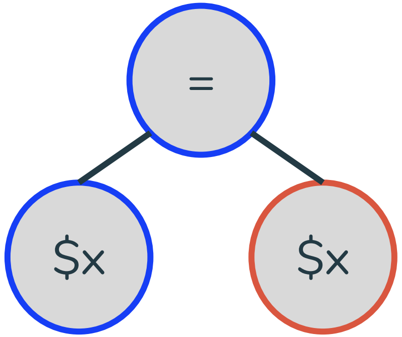
Target $a=10



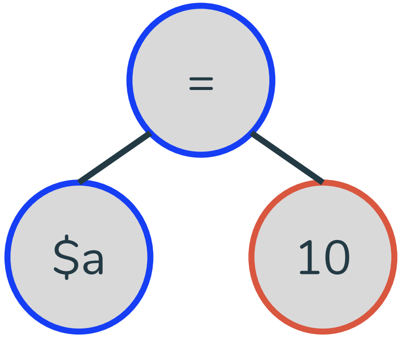
$x is bound to $a

Pattern matching

Pattern $x=$x



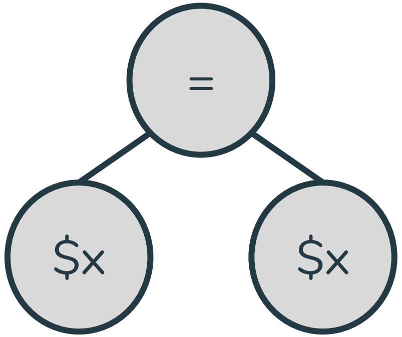
Target $a=10



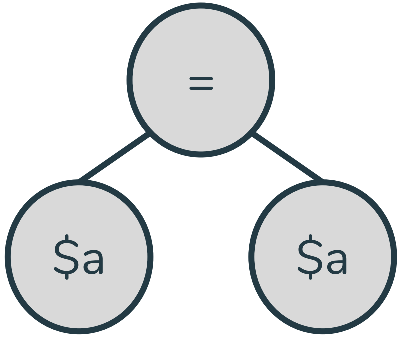
$a != 10

Pattern matching

Pattern \$x=\$x

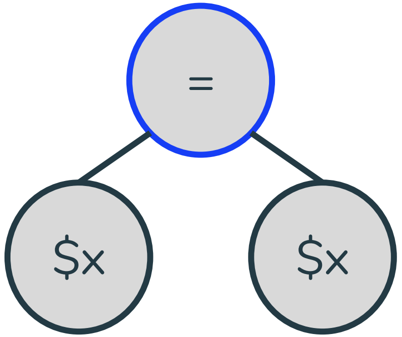


Target \$a=\$a

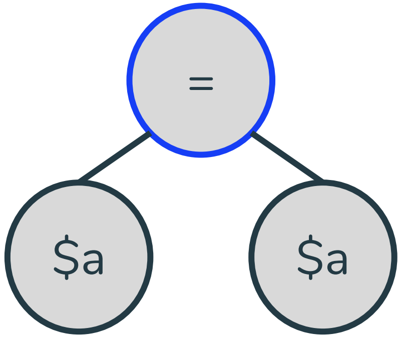


Pattern matching

Pattern $x=$x

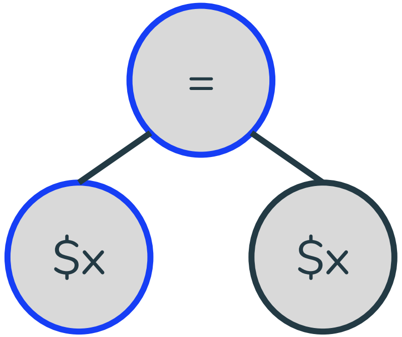


Target $a=$a

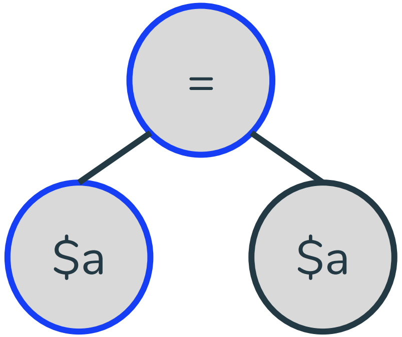


Pattern matching

Pattern $x=$x



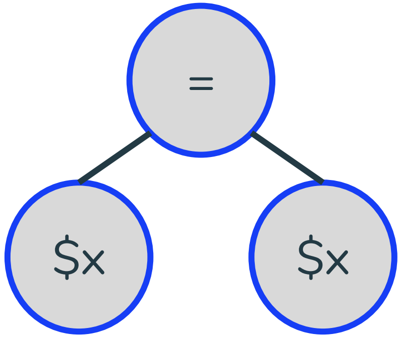
Target $a=$a



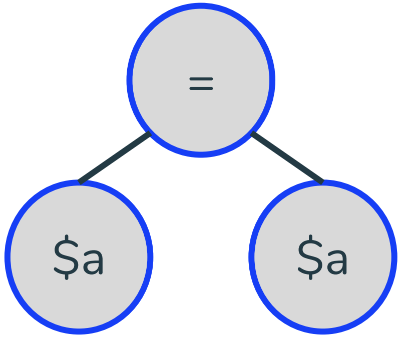
$x is bound to $a

Pattern matching

Pattern $x=$x



Target $a=$a



$a = $a, pattern matched

Pattern matching

Trying to make pattern matching work faster...

“$x = $x” pattern string

$$\Downarrow$$

Parsed AST

$$\Downarrow$$

~~Modified AST~~

“$x = $x” pattern string

⇩

Parsed AST

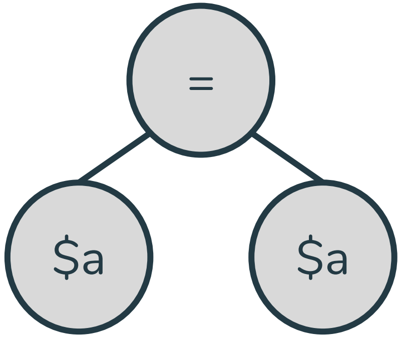
⇩

Polish notation + stack

Pattern $x=$x

Instructions	Stack
<Assign>	=
<NamedAny x>	
<NamedAny x>	

Target $a=$a

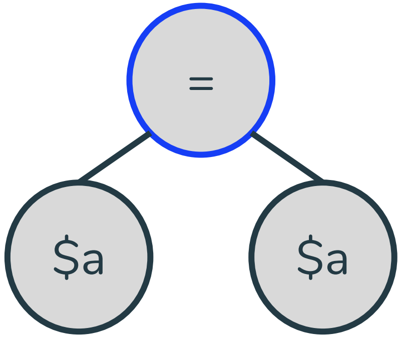


Stack-based matching

Pattern $x=$x

Target $a=$a

Instructions	Stack
<Assign>	$a
<NamedAny x>	$a
<NamedAny x>	

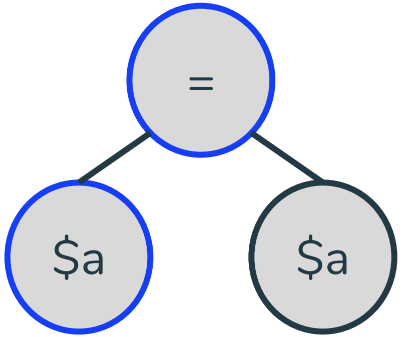


Stack-based matching

Pattern $x=$x

Instructions	Stack
<Assign>	$a
<NamedAny x>	
<NamedAny x>	

Target $a=$a

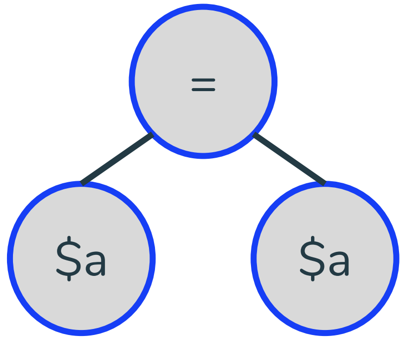


Stack-based matching

Pattern $x=$x

Instructions	Stack
<Assign>	
<NamedAny x>	
<NamedAny x>	

Target $a=$a

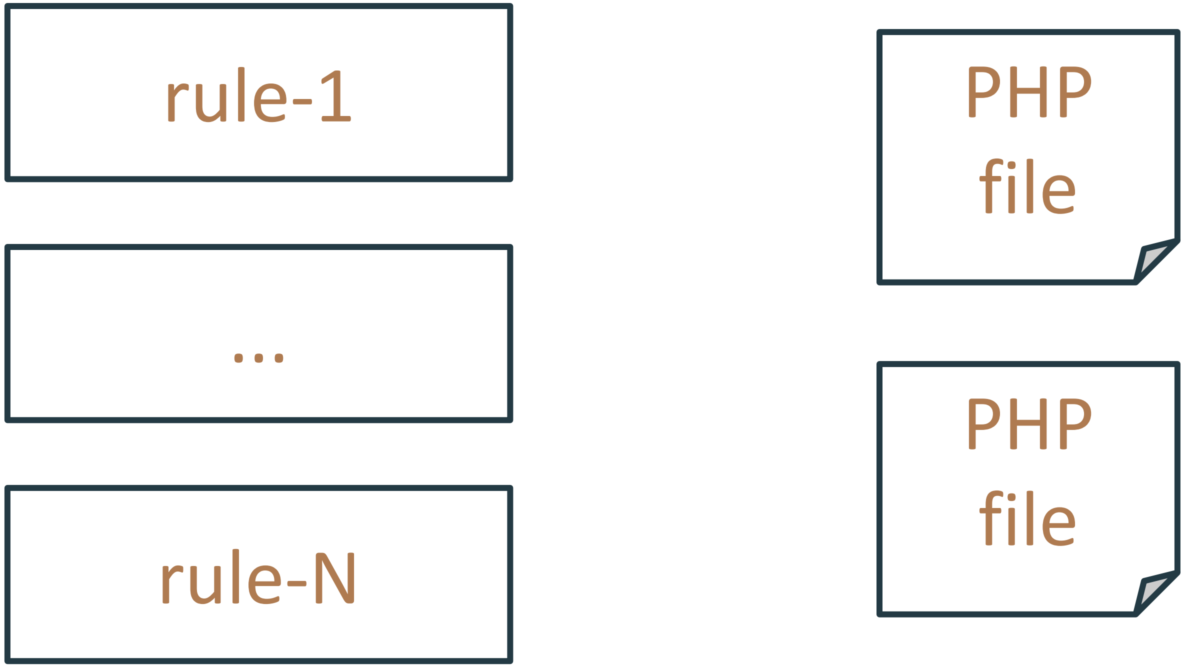


Stack-based matching

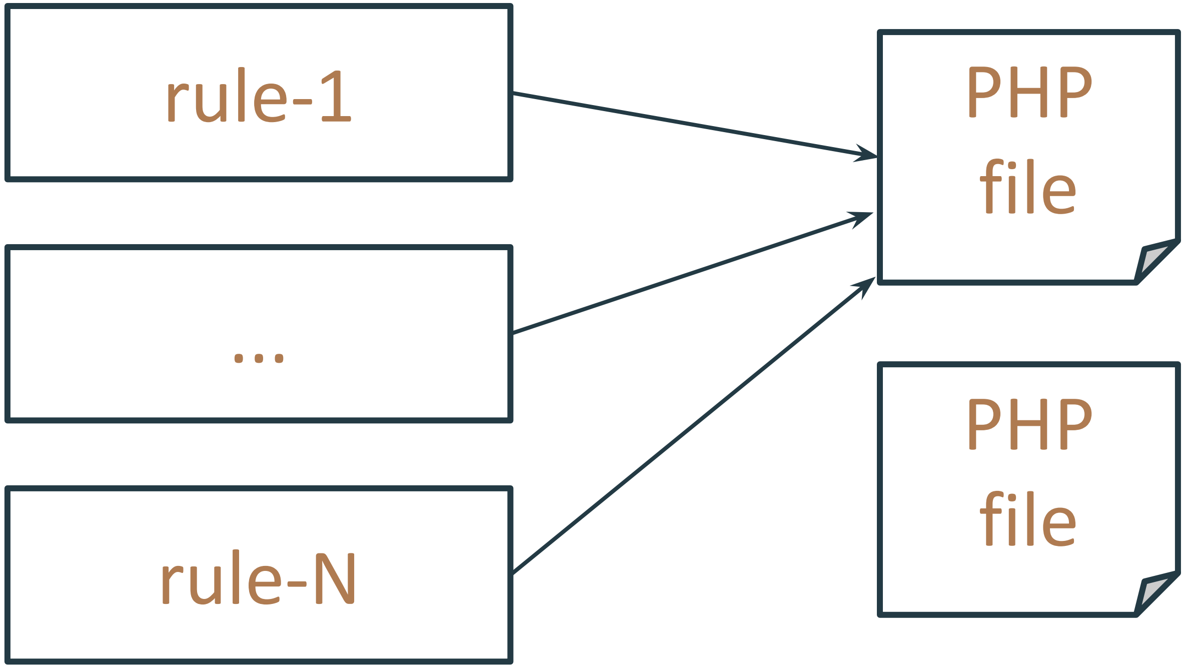
Stack-based matching

- 2-4 times faster matching
- No AST types dependency
- More optimization opportunities

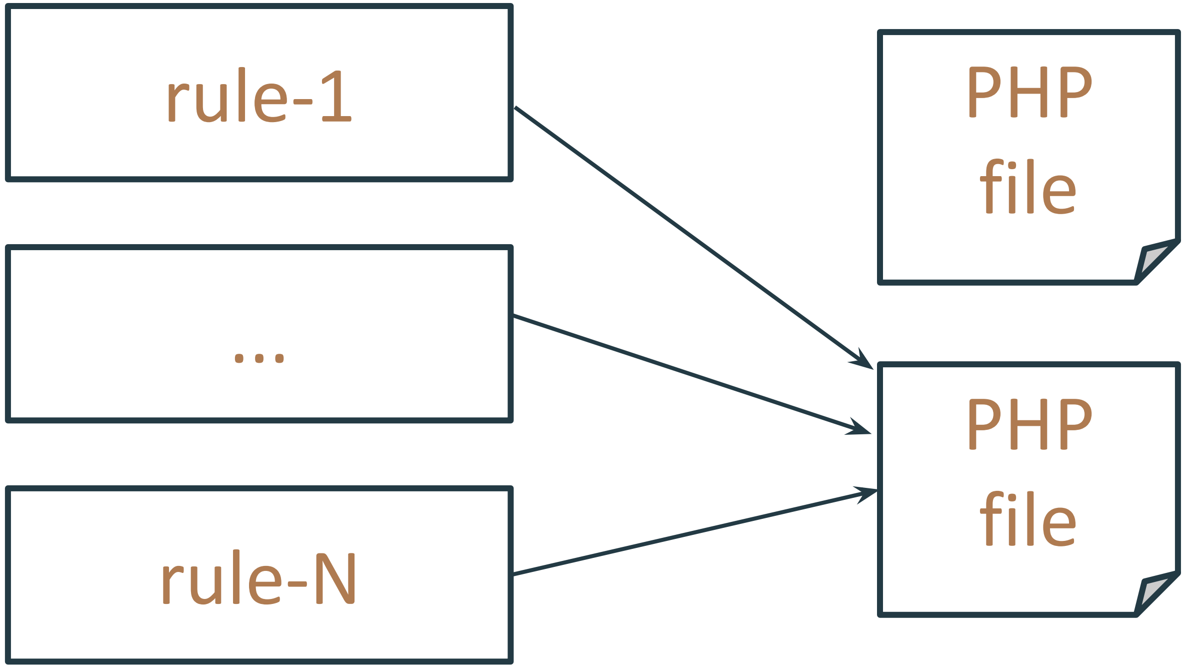
Running rules efficiently



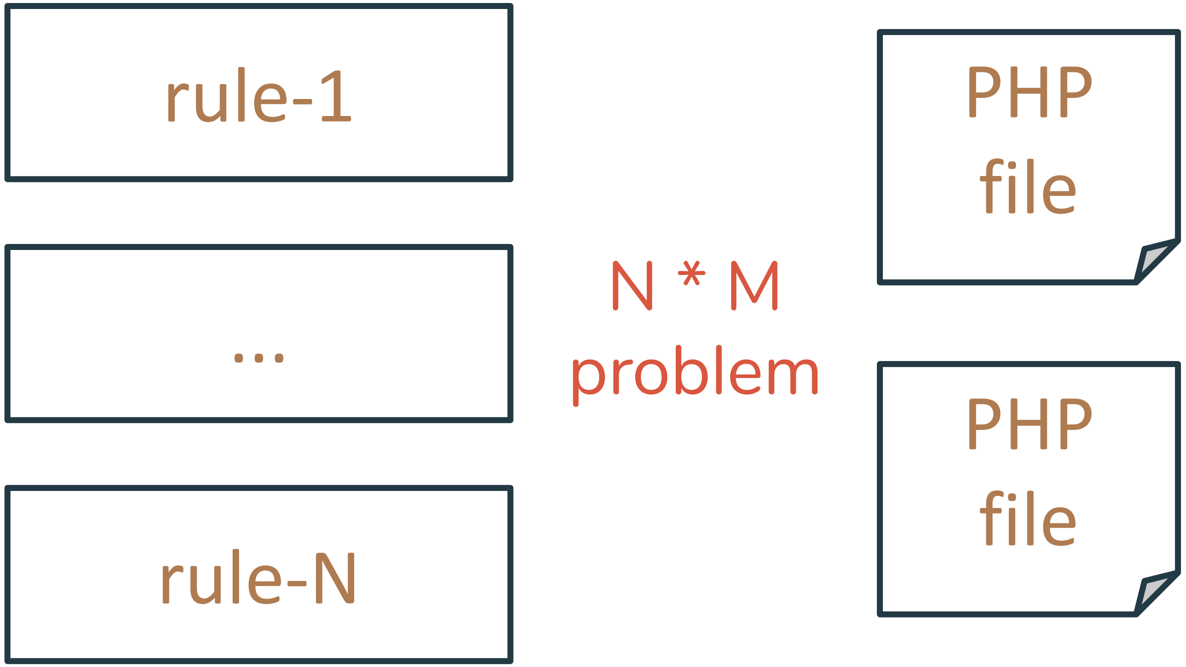
Imagine that we have a lot of rules...



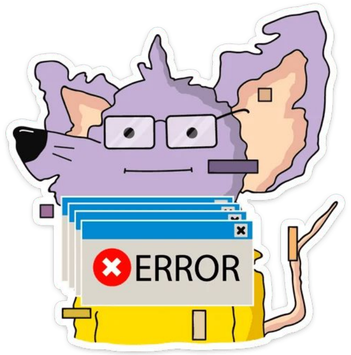
Imagine that we have a lot of rules...



Imagine that we have a lot of rules...



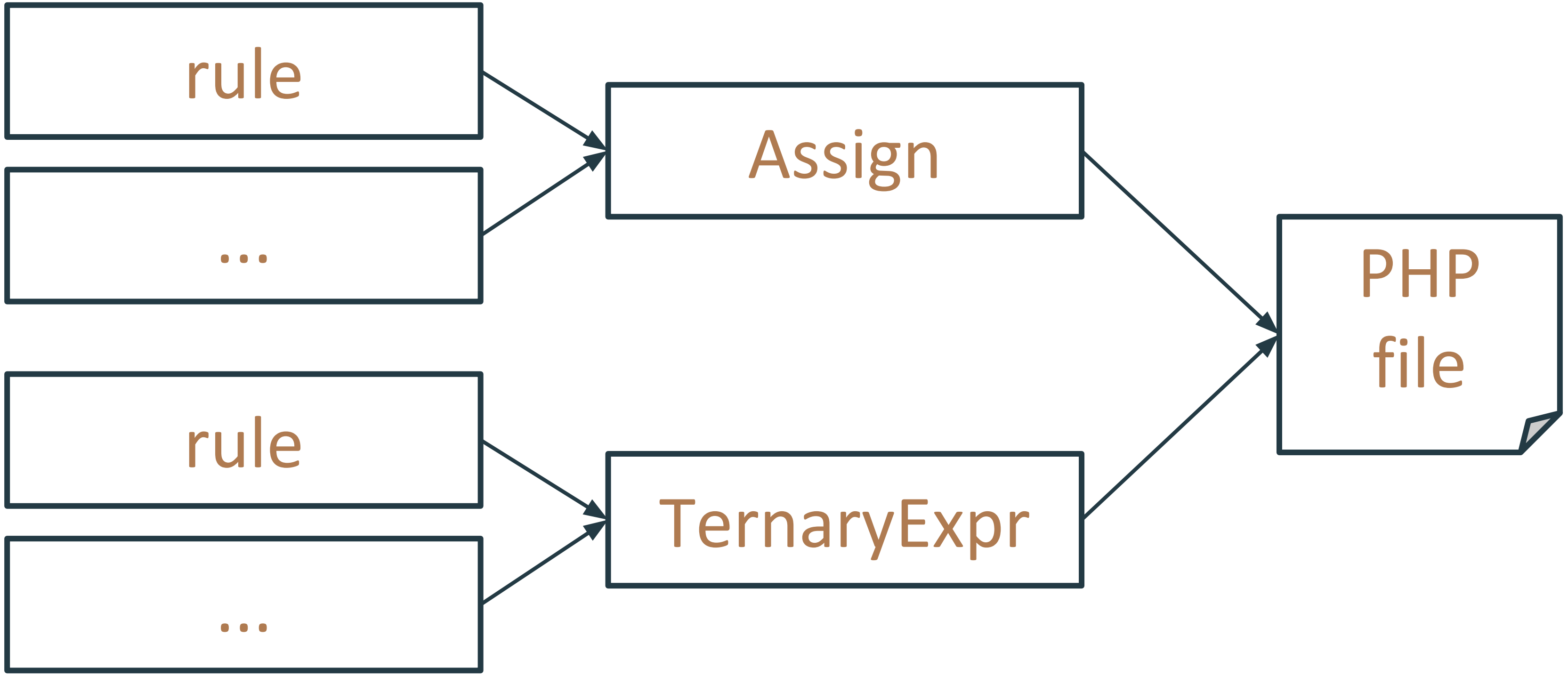
Imagine that we have a lot of rules...

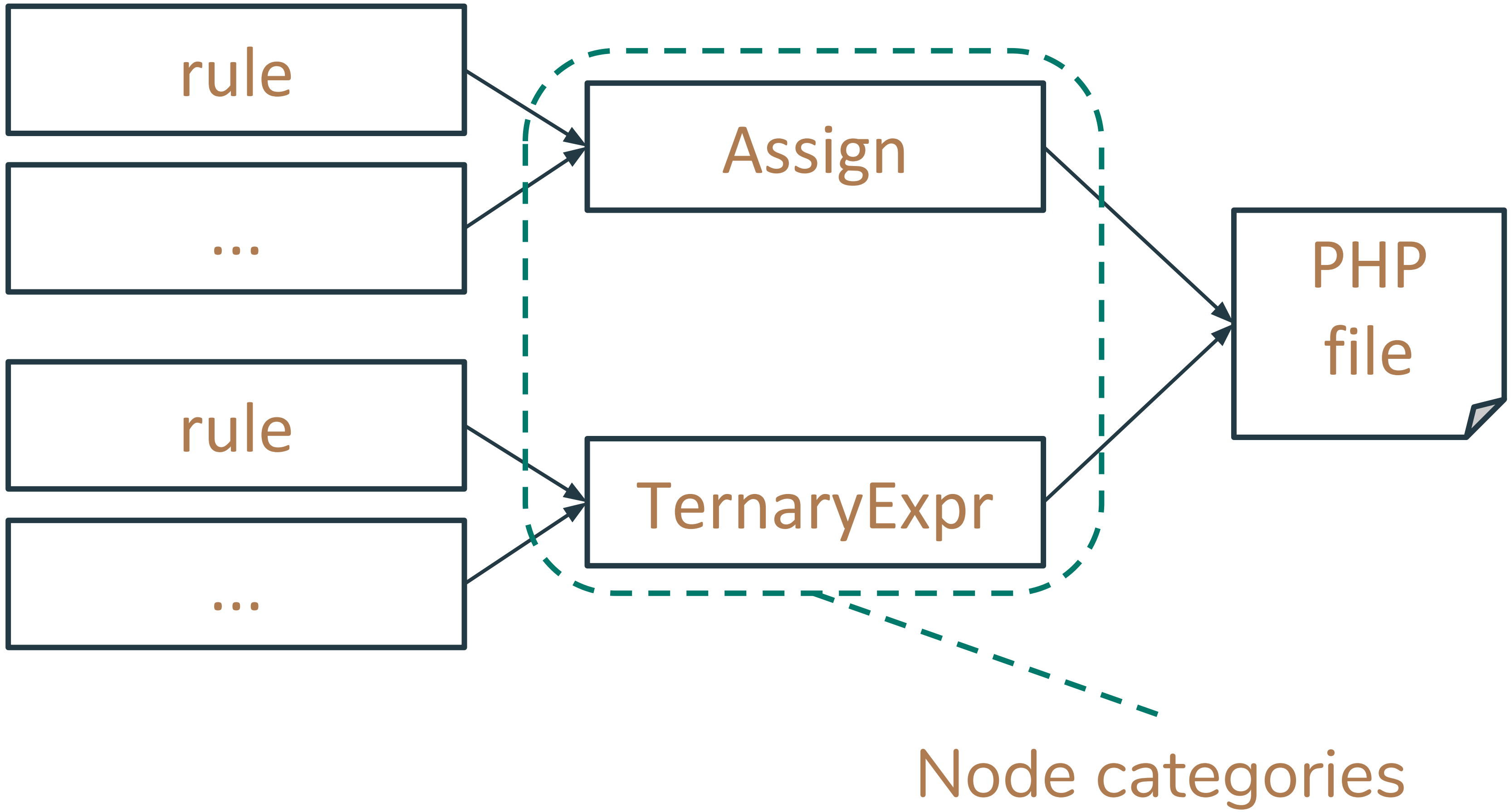


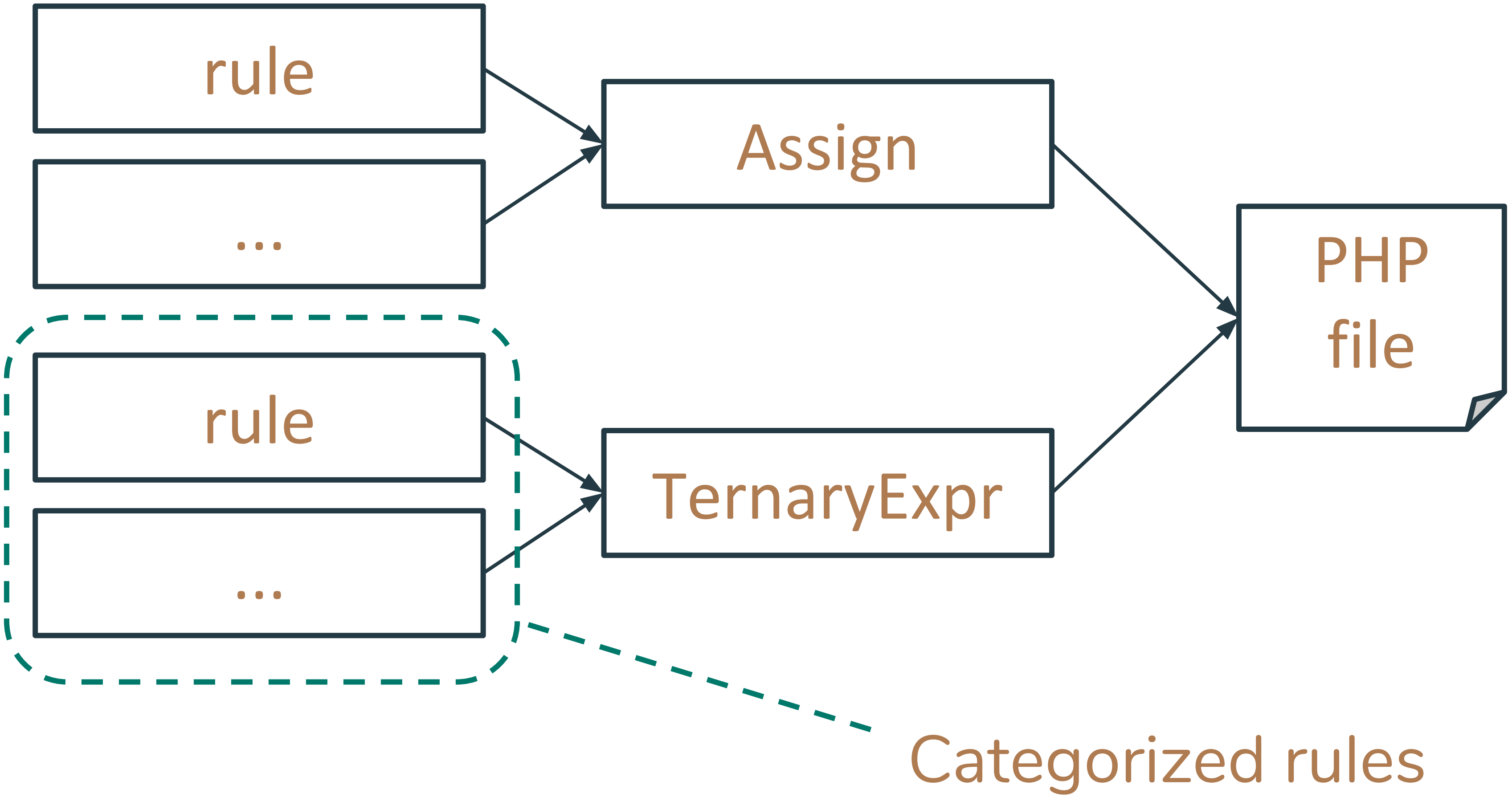
N*M cure: categorized rules

- AST is traversed only once
- For every node, run only relevant rules

We can tune the matching engine to work very fast

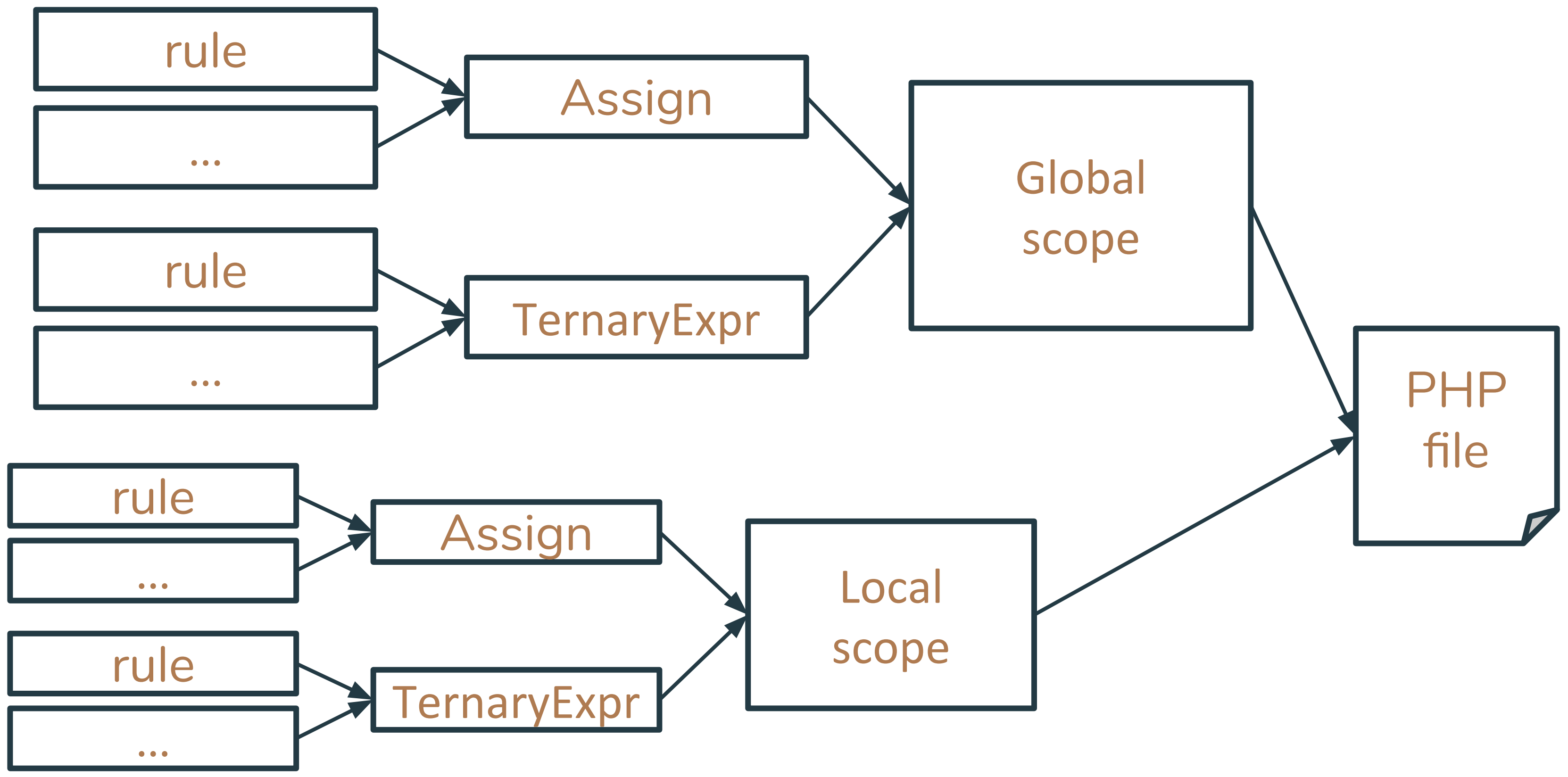

rule
...
rule
...
Assign
TernaryExpr
PHP file

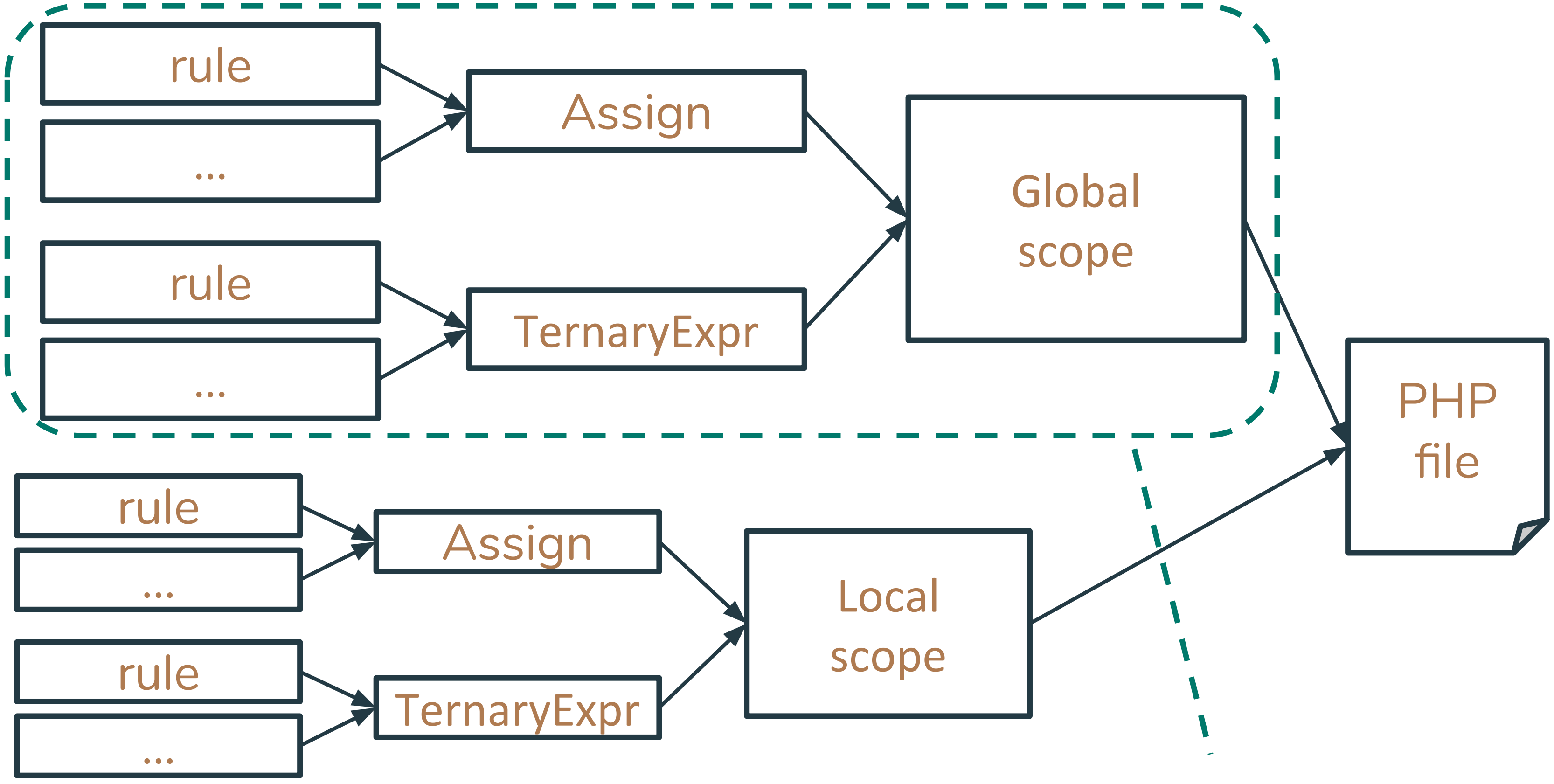

rule
...
rule
...
Assign
TernaryExpr
PHP file
Node categories


rule
...
rule
...
Assign
TernaryExpr
PHP file
Categorized rules

Extra registry layer: scopes

- Local: run rules only inside functions
- Root: run rules only inside global scope
- Universal: run rules everywhere


rule
...
Assign
rule
...
TernaryExpr
Global scope
PHP file
rule
...
Assign
rule
...
TernaryExpr
Local scope



Scoped group

Extra registry layer: expr vs stmt

- Expression can’t contain a statement
- Some statements are top-level only

We don’t use this knowledge right now.

Group cutoff

If any rule from a group matched, all other rules inside the group are skipped for the current node.

- Helps to avoid matching conflicts
- Improves performance

```
// input: $a[0] = $a[0] + 1

function assignOp() {
  /** @fix ++$x */
  $x = $x + 1;

  /** @fix $x += $y */
  $x = $x + $y;
}
```

```
// input: $a[0] = $a[0] + 1

function assignOp() {
  /** @fix ++$x */
  $x = $x + 1;

  /** @fix $x += $y */
  $x = $x + $y;
}
```

Matched,
++$a[0]
suggested

```
// input: $a[0] = $a[0] + 1

function assignOp() {
  /** @fix ++$x */
  $x = $x + 1;

  /** @fix $x += $y */
  $x = $x + $y;
}
```

Skipped

Dynamic rules pros & cons

Dynamic rules advantages

- No need to re-compile NoVerify

Dynamic rules advantages

- No need to re-compile NoVerify
- Simple things are simple

Dynamic rules advantages

- No need to re-compile NoVerify
- Simple things are simple
- No Go coding required

Dynamic rules advantages

- No need to re-compile NoVerify
- Simple things are simple
- No Go coding required
- Rules are declarative

Dynamic rules advantages

- No need to re-compile NoVerify
- Simple things are simple
- No Go coding required
- Rules are declarative
- No need to know linter internals

PHPDoc-based attributes

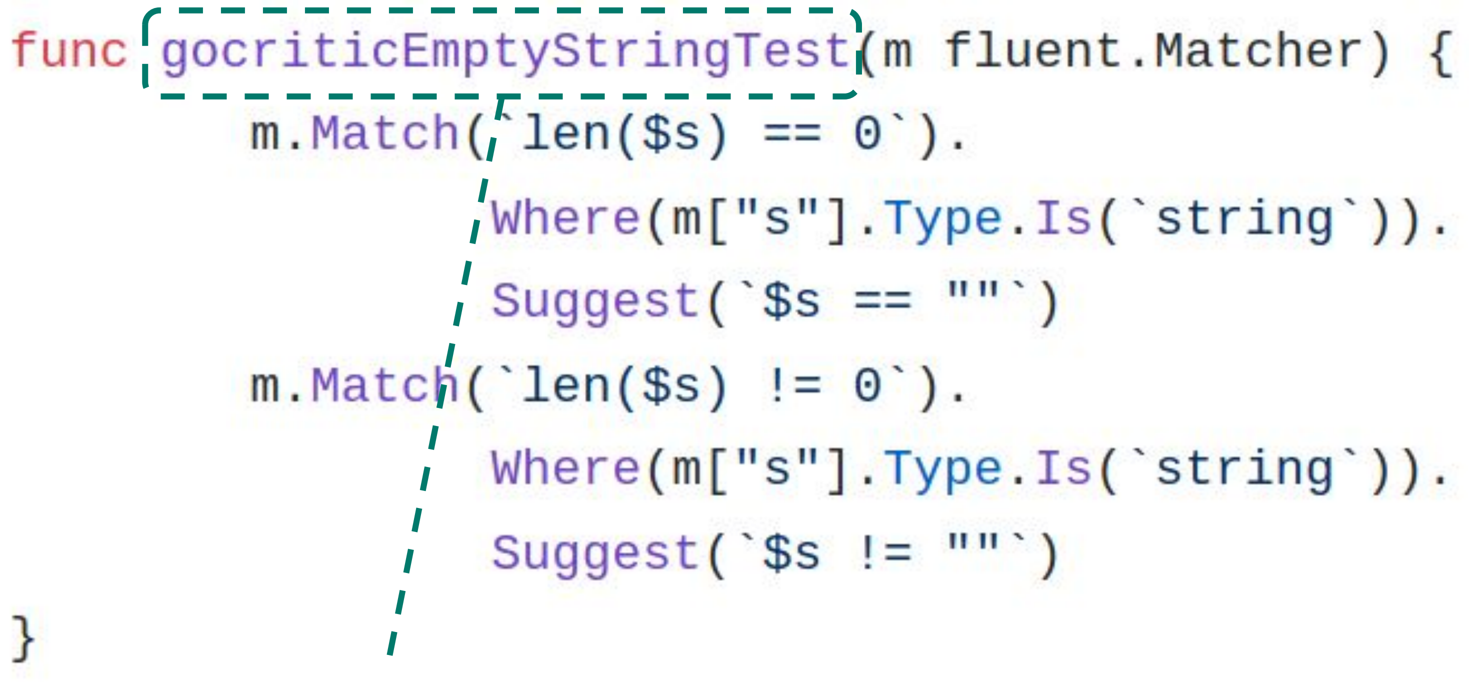
- Not very composable
- Too verbose for non-trivial cases
- Hard to get the autocompletion working

AST pattern limitations

- Hard to express flow-based rules
- PHP syntax limitations
- Recursive block search is problematic

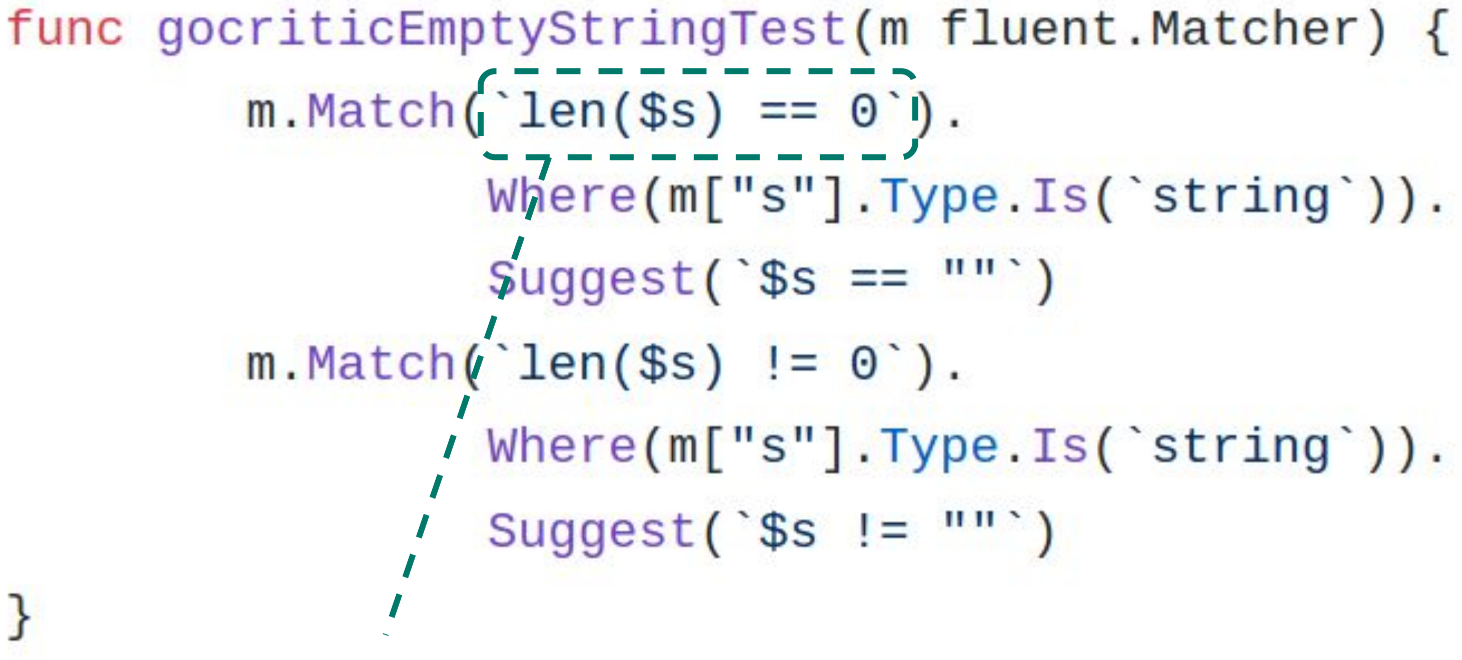
Comparison with Ruleguard

```go
func gocriticEmptyStringTest(m fluent.Matcher) {
	m.Match(`len($s) == 0`).
		Where(m["s"].Type.Is(`string`)).
		Suggest(`$s == ""`)
	m.Match(`len($s) != 0`).
		Where(m["s"].Type.Is(`string`)).
		Suggest(`$s != ""`)
}
```



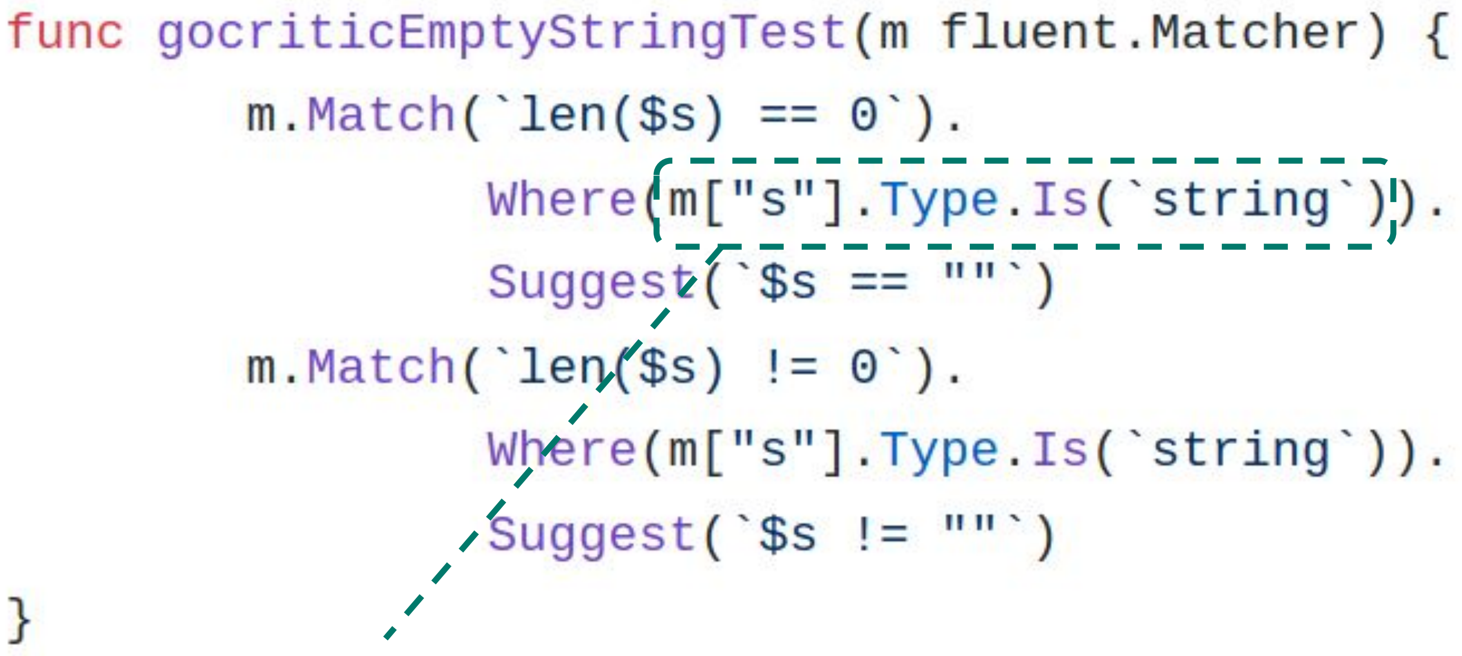
```
func gocriticEmptyStringTest(m fluent.Matcher) {
        m.Match(`len($s) == 0`).
                Where(m["s"].Type.Is(`string`)).
                Suggest(`$s == ""`)
        m.Match(`len($s) != 0`).
                Where(m["s"].Type.Is(`string`)).
                Suggest(`$s != ""`)
}
```

Rule group name



```go
func gocriticEmptyStringTest(m fluent.Matcher) {
	m.Match(`len($s) == 0`).
		Where(m["s"].Type.Is(`string`)).
		Suggest(`$s == ""`)
	m.Match(`len($s) != 0`).
		Where(m["s"].Type.Is(`string`)).
		Suggest(`$s != ""`)
}
```

gogrep pattern



```
func gocriticEmptyStringTest(m fluent.Matcher) {
        m.Match(`len($s) == 0`).
                Where(m["s"].Type.Is(`string`)).
                Suggest(`$s == ""`)
        m.Match(`len($s) != 0`).
                Where(m["s"].Type.Is(`string`)).
                Suggest(`$s != ""`)
}
```

Type filter

```
func gocriticEmptyStringTest(m fluent.Matcher) {
        m.Match(`len($s) == 0`).
                Where(m["s"].Type.Is(`string`)).
                Suggest(`$s == ""`)
        m.Match(`len($s) != 0`).
                Where(m["s"].Type.Is(`string`)).
                Suggest(`$s != ""`)
}
```

Auto fix action

```go
func gocriticBoolExprSimplify(m fluent.Matcher) {
        m.Match(`!!$x`).Suggest(`$x`)
        m.Match(`!($x != $y)`).Suggest(`$x == $y`)
        m.Match(`!($x == $y)`).Suggest(`$x != $y`)
}
```

Target language

go-ruleguard	Go
NoVerify rules	PHP

NoVerify vs Ruleguard

DSL core

go-ruleguard	Fluent API DSL
NoVerify rules	Top-level patterns + PHPDoc

NoVerify vs Ruleguard

Filtering mechanism

go-ruleguard	Go expressions
NoVerify rules	PHPDoc annotations

NoVerify vs Ruleguard

Type filters

go-ruleguard	Type matching patterns
NoVerify rules	Simple type expressions

NoVerify vs Ruleguard

Links

- NoVerify - static analyzer (linter)
- phpgrep - structural PHP search
- phpgrep VS Code extension
- Dynamic rules example
- Dynamic rules for static analysis article
- Ruleguard - dynamic rules for Go

Code -> Linter rules

Pattern-based static analysis