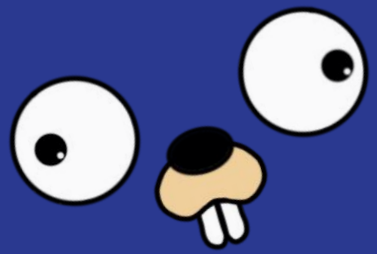
How to do regexp analysis

@quasilyte / GolangKazan 2020



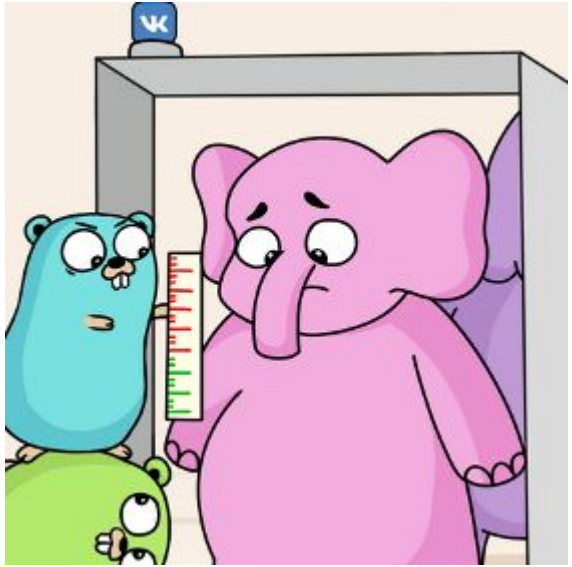
Not *why*, but *how*

Implementation advice and potential issues overview.

Open-Source analyzers

go-critic

NoVerify



Discussion plan

- Handling regexp syntax
- Analyzing regexp flow
- Finding bugs in regular expressions
- Regexp rewriting

Handling regexp syntax

Why making own parser?

Most regexp libraries use parsers that give up on the first error.

For analysis, we need rich AST (parse tree even) and error-tolerant parser.

Writing a parser

Useful resources:

- Regexp syntax docs (BNF, re2-syntax)
- Pratt parsers tutorial (RU, EN)
- Regexp corpus for tests (gist)
- Dialect-specific documentation

Composition operators

Only two:

- Concatenation: xy (“x” followed by “y”)
- Alternation: x|y (“x” or “y”)

Concatenation is **implicit**.

And we want it to be **explicit** in AST.

`0|xy[a-z]`

0 | x · y · [a-z]

Concat operation

Parsing concatenation

- Insert concat tokens
- Parse regexp like it has explicit concat

xy?

“x” “.” “y” “?”

Char classes (are hard)

- Different escaping rules
- Char-ranges can be tricky

This is char range: `[\n-\r]` 4 chars

This is not: `[\d-\r]` `\d`, "`-`" and "`\r`"

`[][]`

What is it?

Char classes syntax

`[][]`

A char class of “]” and “[“!

`[\]\[]`

Char classes syntax

`[^]*|\[[^\]]`

What is it?

Char classes syntax

`[^]*|\[[^\]]`

A single char class!

`[^\]*|\[\[^\]]`

Char classes syntax

`[+=-_]`
What will be matched?

Char classes syntax

`[+=-_]`

“F” matched

Char classes syntax

`[+=\-_]`

“F” not matched

Char classes syntax

Chars and literals

- Consecutive “chars” can be merged
- Single char should not be converted

Both forms (with and without merge) are useful. Merged chars simplify literal substring analysis.

`foox?y`

lit(foo) · ?(char(x)) · char(y)

Concat operation

AST types

There are at least two approaches:

- One type + enum tags
- Many types + shared interface/base

Both have pros and cons.

```go
type Expr struct {
  Kind  ExprKind // enum tag
  Value string   // source text
  Args  []Expr   // sub-expr list
}

type ExprKind int
```

AST types

```go
const (
  ExprNone ExprKind = iota
  ExprChar
  ExprLiteral // list of chars
  ExprConcat  // xy
  ExprAlt     // x|y
  // etc.
)
```

AST types

```go
func charExpr(val string) Expr {
  return Expr{
    Kind: ExprChar,
    Value: val,
  }
}
```

Helper for the next slide

```
Expr{
  Kind: ExprAlt, Value: "x|yz",
  Args: []Expr{
    charExpr("x"),
    {
      Kind: ExprConcat, Value: "yz",
      Args: []Expr{
        charExpr("y"), charExpr("z"),
      },
    },
  },
}
```

AST of `x|yz`

Go regexp parsing library

https://github.com/quasilyte/regex contains a `regex/syntax` package that is used in both NoVerify and go-critic.

It can parse both re2 and pcre patterns.

Analyzing regexp flow

Regexp flags

A regular expression can have an **initial set of flags**, then it can **add** or **remove** any of them inside the expression.

The effect is localized to the current (potentially capturing) group.

```
`/((?i)a(?m)b(?-m)c)d/s`
        ^----------
```

flags: si

Entered a group with “i” flag

Concat operation

`/((?i)a(?m)b(?-m)c)d/s`

_^

flags: sim

Mid-group flags: add “m”

Concat operation

```
`/((?i)a(?m)b(?-m)c)d/s`
              ---------------^
```

flags: si

Mid-group flags: clear “m”

Concat operation

```
`/((?i)a(?m)b(?-m)c)d/s`
  -----------------^
```

flags: s

Left a group with “i” flag

Concat operation

Flags flow

- Flags are lexically scoped
- Groups are a scoping unit
- Leaving a group drops a scope
- Entering a group adds a scope

Back references

- Rules vary among engines/dialects
- Syntax may clash with octal literals
- Can also be relative/named: \g{-1}, etc

We'll use PHP rules as an example.

`\0`	???
`\1 … \9`	???
`\10 … \77`	???

Back reference QUIZ! (PHP)

\0	Octal literal
\1 … \9	???
\10 … \77	???

Back reference QUIZ! (PHP)

\0	Octal literal
\1 … \9	Back reference
\10 … \77	???

Back reference QUIZ! (PHP)

\0	Octal literal
\1 … \9	Back reference
\10 … \77	It depends!

Back reference QUIZ! (PHP)

Groups flow

- Capturing groups are numbered from left to right.
- Non-capturing groups are ignored.
- Groups can have a name.

Finding bugs in regular expressions

“^” anchor diagnostic

Let’s check that “^” is used only in the beginning position of the pattern.

Because if it follows a non-empty match, it’ll never succeed.

`^foo`
`^a|^b`
`a|(b|^c)`

Correct “^” usages

`foo^`

`a^b`

`(a|b)^c`

Incorrect “^” usages

Algorithm

- Traverse all starting branches
- Mark all reached “^” as “good”

Then traverse a pattern AST normally and report any “^” that was not marked.

The starting branches?

- For every “concat” met, it’s the first element (applied recursively).
- If root regexp element is not “concat”, consider it to be a concat of 1 element.

`google.com`

URL matching

`google.com`

http://googleocom.ru

URL matching

`google.com`

http://googleocom.ru

http://a.github.io/google.com

URL matching

`google\.com`

http://googleocom.ru

http://a.github.io/google.com

URL matching

`^https?://google\.com/`

http://googleocom.ru

http://a.github.io/google.com

URL matching

URL matching

When "." is used before common domain name like "com", it's probably a mistake.

If we have char sequences represented as a single AST node, this analysis is trivial.

`google.com`

lit(google) · . · **lit**(com)

Warn if “.” is followed by a **lit** with domain name value.

Handling unescaped dot

Regexp rewriting

Regexp input generation

It's quite simple to generate a string that will be matched by a regular expression if you have that regexp AST.

```
`\w*[0-9]?$`
*(\w) · ?([0-9]) · $
```

Generating matching string (N=2)

`\w*[0-9]?$`

***(\w)** · ?([0-9]) · $

aa

N matches of \w

Generating matching string (N=2)

`\w*[0-9]?$`

*(\w) · **?([0-9])** · $

aa**7**

1 match of [0-9]

Generating matching string (N=2)

`\w*[0-9]?$`

*(\w) · ?([0-9]) · **$**

aa7

May do nothing for $

Generating matching string (N=2)

Regexp input generation

Generating a non-matching strings can be useful for catastrophic backtracking evaluation.

Regexp simplification

Instead of writing a matching characters we can write the pattern syntax itself.

By replacing recognized AST node sequences with something simpler, we can perform a regexp simplification.

`\dxx*`

\d · x · *(x)

Regexp simplification

`\dxx*`

\d · x · *(x)

\d

Can’t simplify \d, write as is

Regexp simplification

`\dxx*`

\d · **x** · ***(x)**

\d**x+**

xx* -> x+

Regexp simplification

```
x{1,} -> x+
```

```
[a-z\d][a-z\d] -> [a-z\d]{2}
```

```
[^\d] -> \D
```

```
a|b|c -> [abc]
```

Oh, the possibilities!

https://quasilyte.dev/regexp-lint/

RegExp Lint Share Leave feedback GitHub

```
(?:https|http)://google.com/search\?q\=golang
```

Go

PHP

```
warning: '.com/' should be '\.com/'
suggestion: `https|http` -> `https?`
suggestion: `\=` -> `=`
```

- [x] Validation errors
- [x] Potential issues
- [x] Style suggestions

Online Demo

Submit your ideas! :)

If you have a particular regexp simplification or bug pattern that is not detected by regexp-lint, let me know.

Thank you.