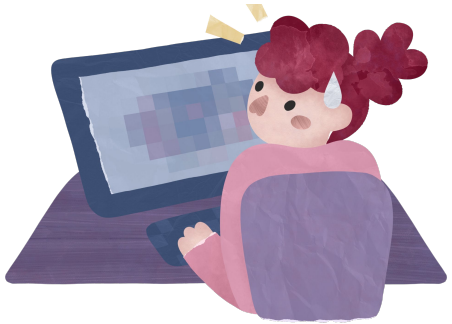
^[\s\u200c]+|[\s\u200c]+$

The Parsing and Analysis of [rR]eg(ular)* *([eE]x(pression(s)*)*)*

Will Beddow, Antonia Ritter, Shiyue Zhang,
Vicente Riquelme

The Challenges of Using Regexes

Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions



Malformed Stack Overflow Post Chokes Regex, Crashes Site

823

Blog/Article/Link Cloudflare outage caused by deploying bad regular expression that caused 100% CPU usage worldwide, dropping up to 82% of traffic (self.sysadmin)

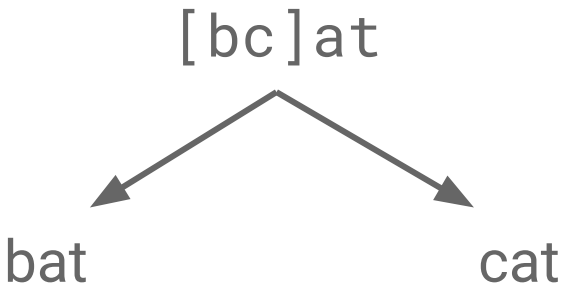
submitted 2 years ago by TyroPyro

Cloudflare Blog[1]

275 comments source share save hide give award report crosspost hide all child comments

Regular Expressions

A regular expression (**regex**) is a pattern used to match particular strings

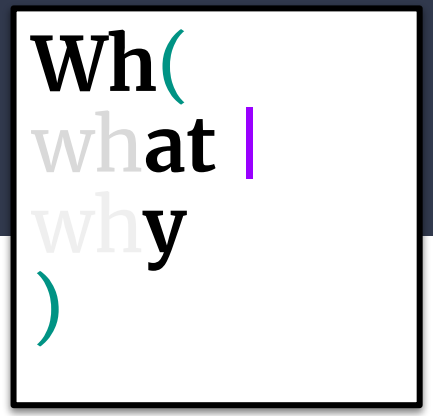


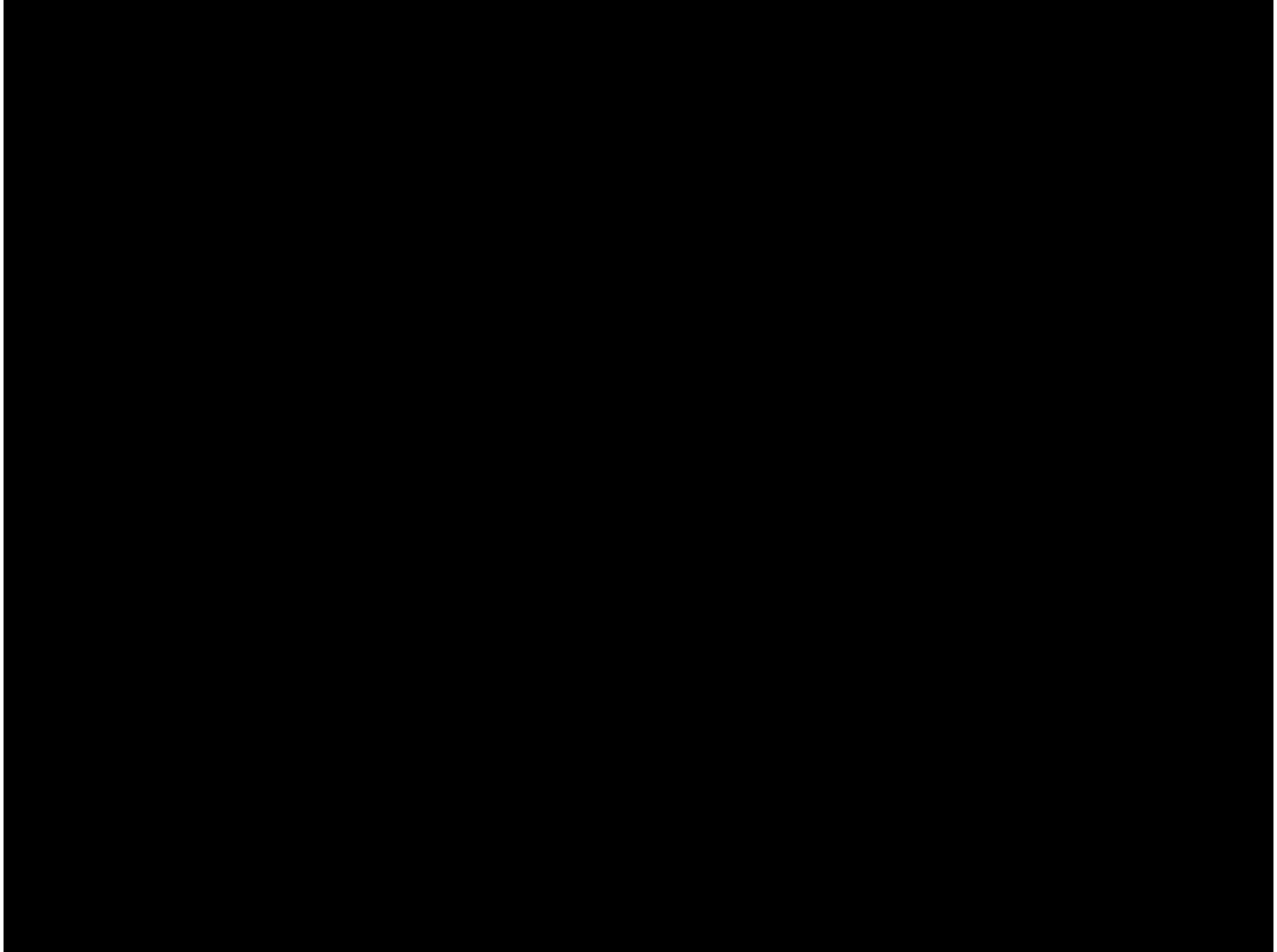

```
#?([\da-fA-F]{2})([\da-fA
-F]{2})([\da-fA-F]{2})
```

Any hex code, for example #FF5733

Project Overview

- Our tool evaluates:
  - **Security**
  - **Understandability**
  - **Generalizability**

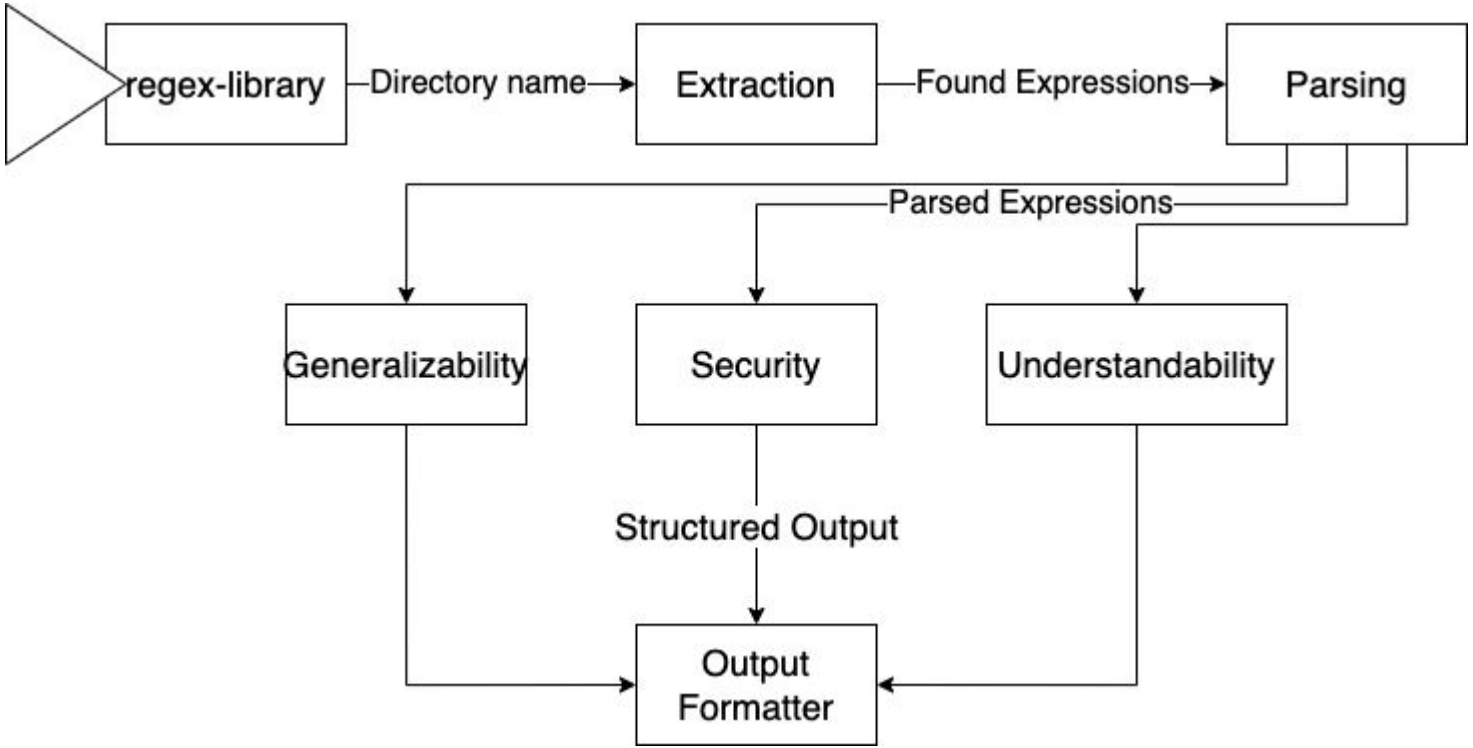




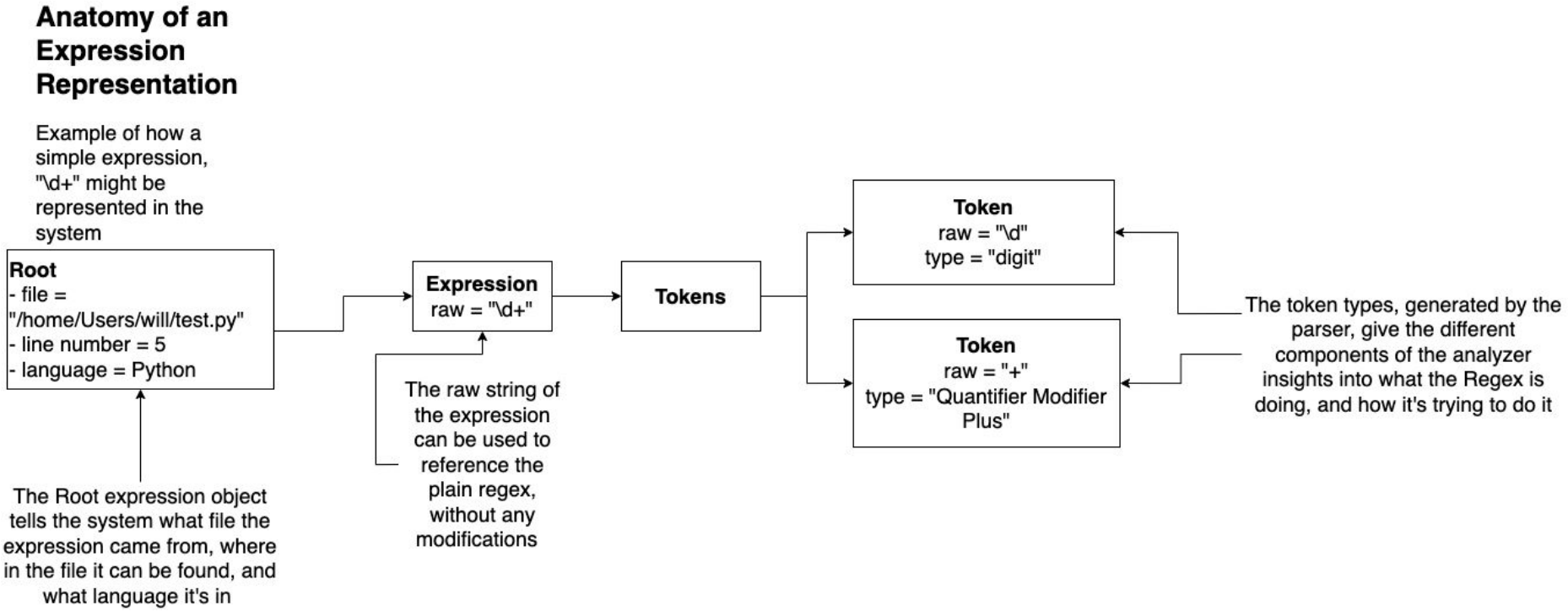

antonia@ARXPS:~/Dropbox (Carleton College)/21_CS_Comps/regex-library$ python3 main.py ../demo-input/ --security
Expression ^<\!\-\-(.*)+(\/){0,1}\-\->a(?=bc|b)c\141S{3}$ from ../demo-input/example_1.py line 4
|0import re
|1
|2def main():
>3   my_patt = re.compile(r"^<\!\-\-(.*)+(\/){0,1}\-\->a(?=bc|b)c\141S{3}$")
|4   my_patt.match(" test ")
|5
|6if __name__ == "__main__":
|7   main()
^Understandability suggestion: a
[aaa]
^Understandability suggestion: repeated characters in []
S{3}
^Understandability suggestion: SSS
?=
^Java, JavaScript, and Golang may not support conditionals
<!---->abcbcaaa<!---->abcbcaaa<!------->abcbc---->abcbcaaaS-->abcbc>abcb>abcbcaaaS"aS---->abcbcaa!--->abcbcaaaS
^This string gave the expression a run time longer than 1 second
<!-->abcbcaaa---->abcbcaaaS>abcbc+rcae>abcbcs<!---->abcbP->abcaafu&<
8fi>abcbcaaaS-caa<!---->abcbcaaaSbcaaa<!---->abcbcaaaS`-->abc>abcbca
^This string gave the expression a run time longer than 1 second
Expression Hello from ../demo-input/folder_1/example_2.js line 4
|0
|1function test_function() {
|2   let text = "Hello";
>3   let n = text.search(/Hello/i);
|4}
|5
The raw expression, and where it was found
Excerpt of file where expression was found
Targeted tips and analysis

Output

Structure of the library



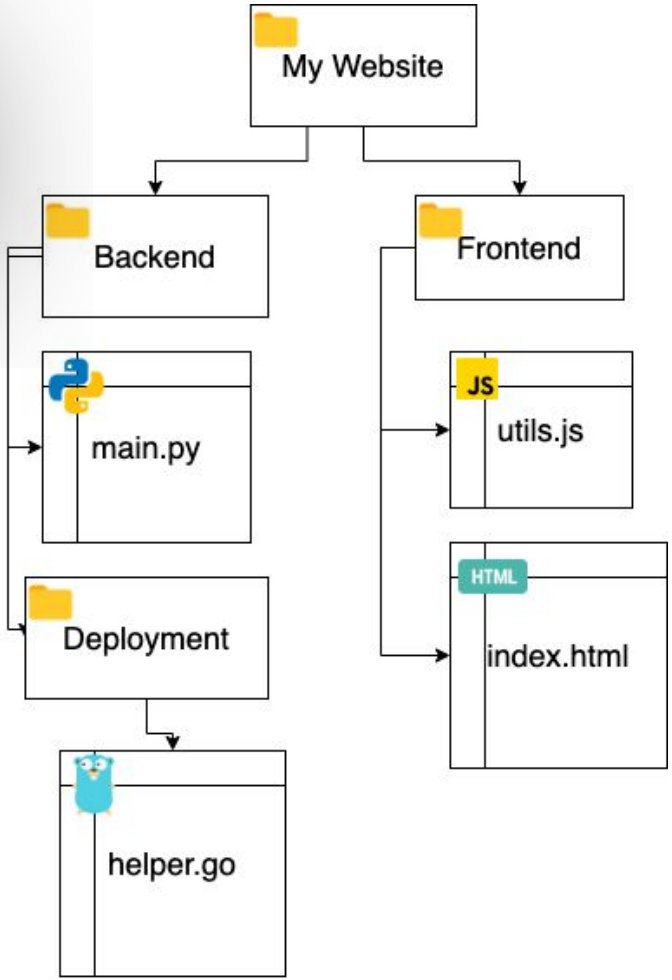
Data Format


Anatomy of an Expression Representation
Example of how a simple expression, "\d+" might be represented in the system
Root
- file = "/home/Users/will/test.py"
- line number = 5
- language = Python
The Root expression object tells the system what file the expression came from, where in the file it can be found, and what language it's in
Expression
raw = "\d+"
The raw string of the expression can be used to reference the plain regex, without any modifications
Tokens
Token
raw = "\d"
type = "digit"
Token
raw = "+"
type = "Quantifier Modifier Plus"
The token types, generated by the parser, give the different components of the analyzer insights into what the Regex is doing, and how it's trying to do it

Brew

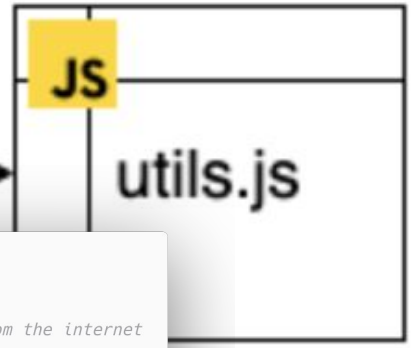
```
brew install zhangshyue/regexanalyzer/regexAnalyzer
regex-library [directory_path]
```

```
python3 regex-library/main.py "My Website"
```



Finding Regular Expressions in Files

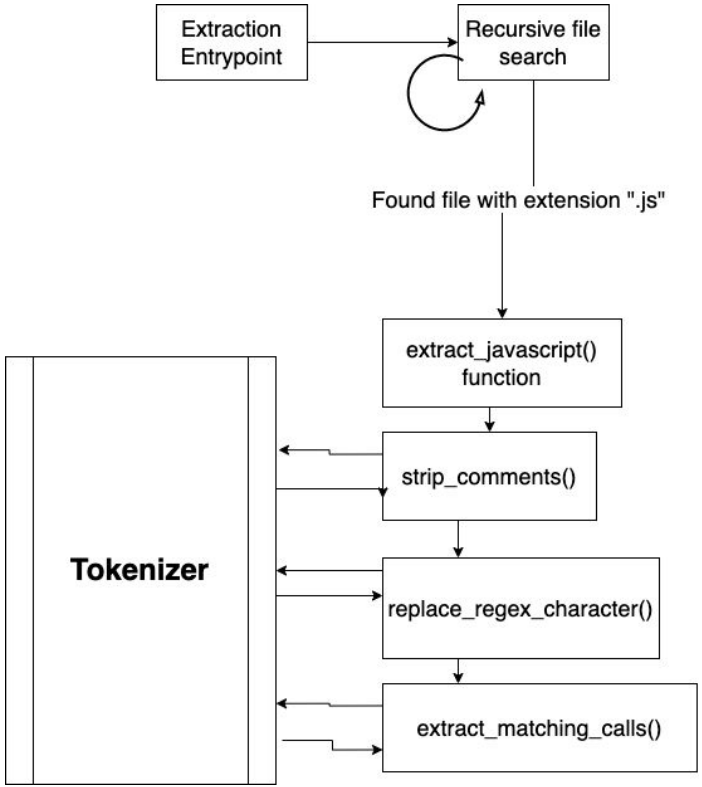
```
match language {
    SupportedLanguage::Python ⇒ res = extract_python( contents_raw: file_contents),
    SupportedLanguage::JavaScript ⇒ res = extract_js( contents_raw: file_contents),
    SupportedLanguage::Golang ⇒ res = extract_go( contents_raw: file_contents),
    SupportedLanguage::Java =
    SupportedLanguage::Rust =
    SupportedLanguage::PHP =
    SupportedLanguage::Ruby =
}
```




```
async function get_page() {
    let internet_res = await fetch("https://example.com"); // Download the result from the internet
    return await internet_res.text();
}

/**
 * The main function
 */
function main() {
    let my_regex = /(.*)\W+/;
    get_page.then((page_text) => {
        my_regex.exec(get_page());
        do_other_stuff();
    })
}
```

Extraction Process Overview

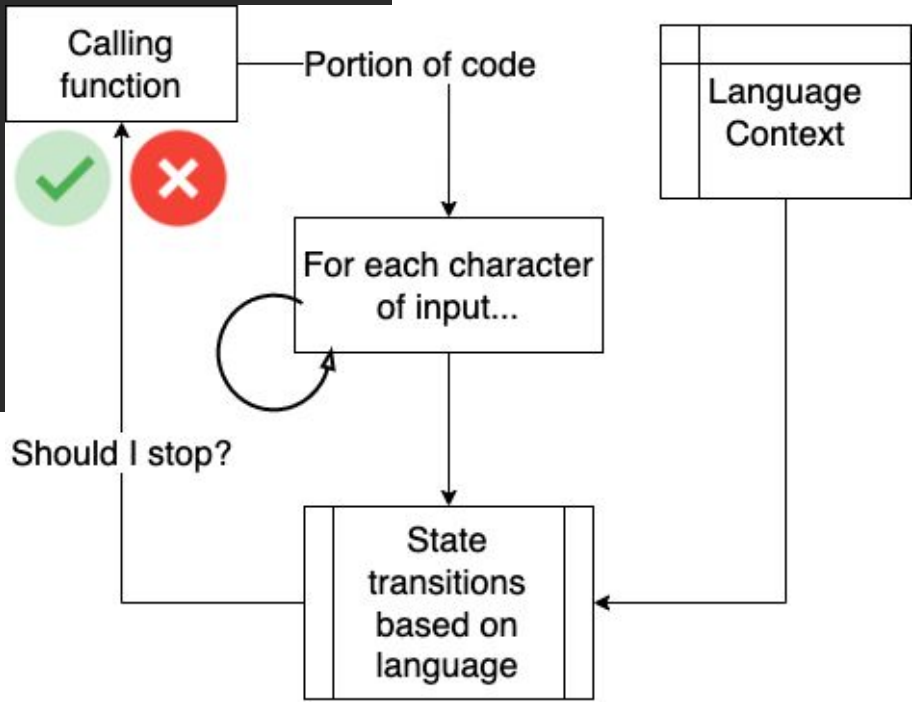



```
async function get_page() {
    let internet_res = await fetch("https://example.com");
    return await internet_res.text();
}


function main() {
    let my_regex = new RegExp("(.*)\W+");
    get_page.then((page_text) => {
        my_regex.exec(get_page());
        do_other_stuff();
    })
}
```

General Purpose Tokenizer: A Finite State Machine

```
pub static ref JS_CONTEXT: LanguageContext = LanguageContext {
    single_line_comment_starts: vec!["//"],
    multi_line_comment_starts: vec![("/*", "*/")],
    single_line_string_starts: vec!["'", "\""],
    multi_line_string_starts: vec!["`"],
    call_open_sym: b'(',
    call_close_sym: b')',
    special_regex_char: Some(b'/')
};
```



Parsing

```
message Token {
    string token = 8;
    TokenType type = 9;

    oneof sub_type {
        FlagType flag = 10;
        SubstitutionType substitution = 11;
        QuantifierModifierType quantifiermodifier = 12;
        AnchorType anchor = 13;
        string character = 14;
        LookaroundType lookaround = 15;
        EscapeType escape = 16;
        GroupReferenceType groupref = 17;
        CharacterClassType characterclass = 18;
    }
}
```

{

token: “g“
type: Flag
flag: Global

}

{

token: “\t“
type: Escape
escape: Tab

}

Parsing


```
expression {
  raw: "(.*)\\W+"
  tokens {
    token: "("
    type: GroupReference
    groupref: OpenCapture
  }
  tokens {
    token: "."
    type: CharacterClass
    characterclass: Dot
  }
  tokens {
    token: "*"
    type: QuantifierModifier
    quantifiermodifier: Star
  }
  tokens {
    token: ")"
    type: GroupReference
    groupref: CloseCapture
  }
  tokens {
    token: "\\W"
    type: CharacterClass
    characterclass: NotWord
  }
  tokens {
    token: "+"
    type: QuantifierModifier
    quantifiermodifier: Plus
  }
}
```

4. token: “)“
type: GroupReference
groupref: CloseCapture

5. token: “\W“
type: CharacterClass
characterclass: NotWord

6. token: “+“
type: QuantifierModifier
quantifiermodifier: Plus

Understandability Component

- Single bounded class:
  S{3} = SSS = S{3,3}
- Lower bounded class:
  A{2,} = AAA* = AA+
- Custom Character class:
  [0-9a] = [\da] = [0123456789a]
- Literal class:
  \a\$ = \a[$] = \x61\x24 = \141\044

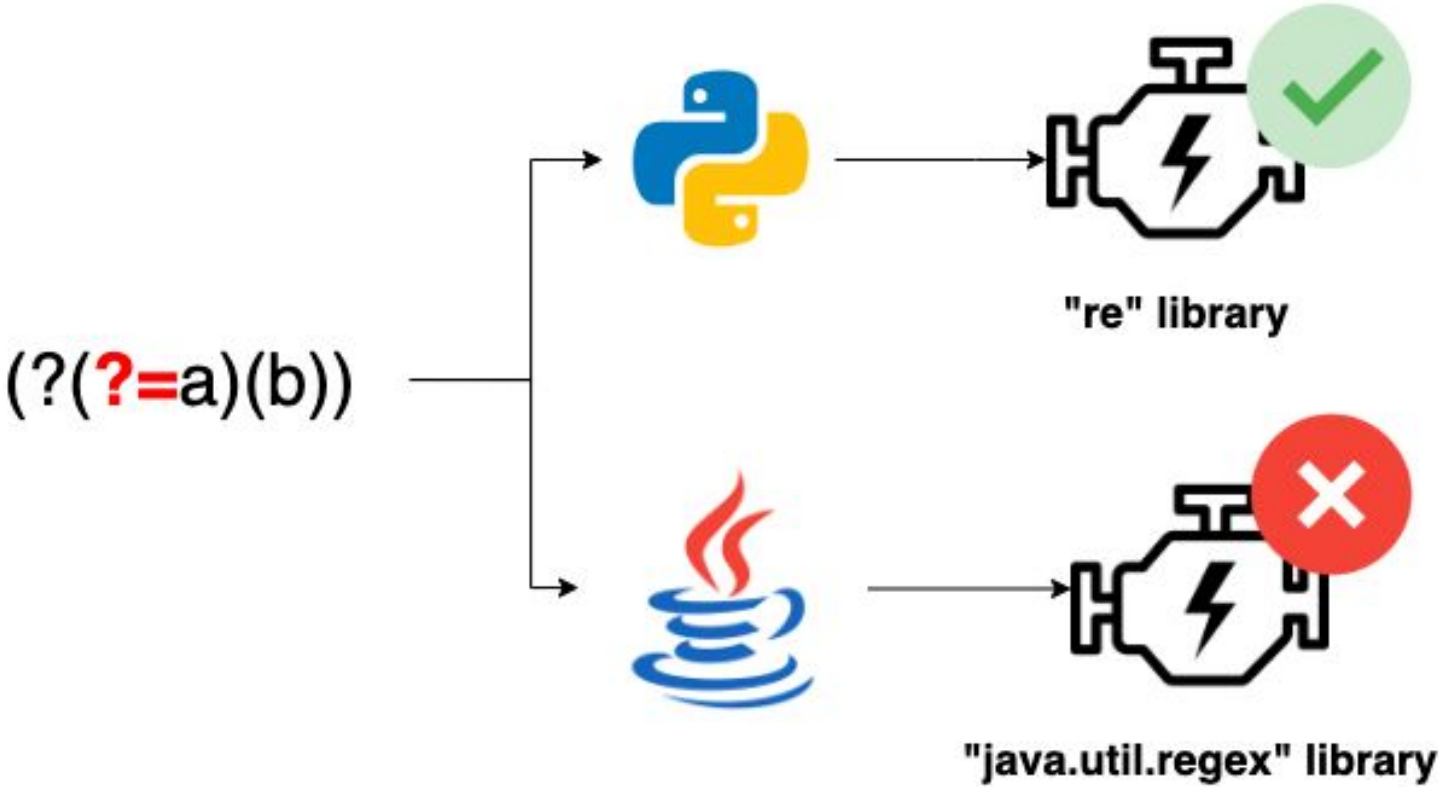
Carl Chapman, Peipei Wang, and Kathryn T. Stolee. 2017. Exploring regular expression comprehension.

Understandability Component

~~\x61[0-9]S{2}~~

a\dSS

Generalizability Component: Regex Engines



Generalizability Component: Functionality

	"+" quantifier	Negated character classes
Boost.Regex	Yes	Yes

```
pub enum Feature {
    Flags,
    Backreferences,
    Conditionals,
    AtomicGroups,
    NamedCapture,
}
```

Language feature comparison (part 1)

	"+" quantifier	Negated character classes	Non-greedy quantifiers [Note 1]	Shy groups [Note 2]	Recursion	Look-ahead	Look-behind	Backreferences [Note 3]	>9 indexable captures
Boost.Regex	Yes	Yes	Yes	Yes	Yes[Note 4]	Yes	Yes	Yes	Yes
Boost.Xpressive	Yes	Yes	Yes	Yes	Yes[Note 5]	Yes	Yes	Yes	Yes
CL-PPCRE	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
EmEditor	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No
FREJ	No[Note 6]	No	Some[Note 6]	Yes	No	No	No	Yes	Yes
GLib/GRegex	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GNU grep	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	?
Haskell	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
RXP	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes
ICU Regex	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Java	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
JavaScript (ECMAScript)	Yes	Yes	Yes	Yes	No	Yes	Yes[Note 7]	Yes	Yes
JGsoft	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Lua	Yes	Yes	Some[Note 8]	No	No	No	No	Yes	No
.NET	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
OCaml	Yes	Yes	No	No	No	No	No	Yes	No
PCRE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Perl	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PHP	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Python	Yes	Yes	Yes	Yes	Yes[Note 9]	Yes	Yes	Yes	Yes
Qt/QRegExp	Yes	Yes	Yes	Yes	No	Yes	No	Yes	Yes
RE2	Yes	Yes	Yes	Yes	No	No	No	No	Yes
Ruby / Onigmo	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
TRE	Yes	Yes	Yes	Yes	No	No	No	Yes	No
Vim	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No
RGX	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Tcl	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
TRegExpr	Yes	?	Yes	?	?	?	?	?	?
XML Schema	Yes	Yes	No	N/A	No	No	No	No	N/A
XPath 3/XQuery	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes
XRegExp	Yes	Yes	Yes	Yes	No	Yes	No	Yes	Yes

1. ^ *Non-greedy* quantifiers match as few characters as possible, instead of the default as many. Note that many older, pre-POSIX engines were non-greedy and didn't have greedy quantifiers at all.
2. ^ *Shy groups*, also called *non-capturing* groups cannot be referred to with backreferences; non-capturing groups are used to speed up matching where the group's content does not need to be accessed later.
3. ^ *Backreferences* enable referring to previously matched groups in later parts of the regex and/or replacement string (where applicable). For instance, *([ab]+)\1* matches "abab" but not "abaab".
4. ^ "Perl Regular Expression Syntax - 1.47.0".
5. ^ "User's Guide - 1.47.0".
6. ^ a b FREJ have no repetitive quantifiers, but have "optional" element which behaves similar to simple "?" quantifier.
7. ^ As of ES2018
8. ^ Lua's only non-greedy quantifier is `-`, which is a non-greedy version of `*`. It does not have non-greedy versions of `+` or `?`; in the former case, the non-greedy effect can be achieved by repeating the token followed by `-`, but in the latter case, there is no equivalent.
9. ^ Supported by the optional regex library only.

Part 2 [edit]

Language feature comparison (part 2)

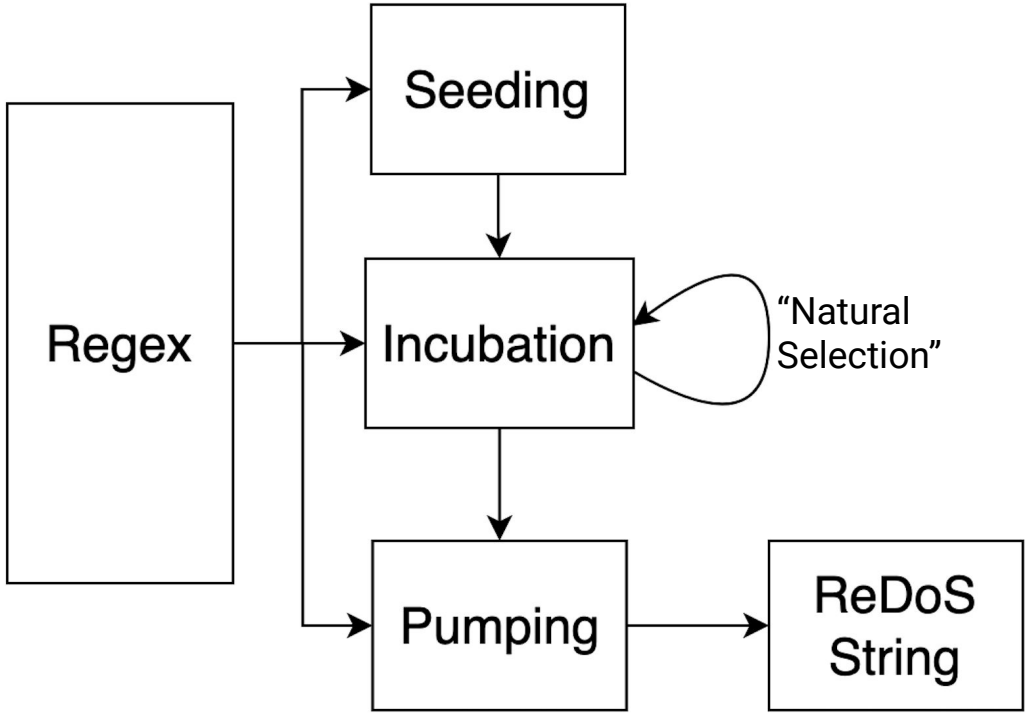
	Directives [Note 1]	Conditionals	Atomic groups [Note 2]	Named capture [Note 3]	Comments	Embedded code	Unicode property support [3]	Balancing groups [Note 4]	Variable-length look-behinds [Note 5]
Boost.Regex	Yes	Yes	Yes	Yes	Yes	No	Some[Note 6]	No	No
Boost.Xpressive	Yes	No	Yes	Yes	Yes	No	No	No	No
CL-PPCRE	Yes	Yes	Yes	Yes	Yes	Yes	Some[Note 6]	No	No
EmEditor	Yes	Yes	?	?	Yes	No	?	No	No
FREJ	No	No	Yes	Yes	Yes	No	?	No	No
GLib/GRegex	Yes	Yes	Yes	Yes	Yes	No	Some[Note 6]	No	No
GNU grep	Yes	Yes	?	Yes	Yes	No	No	No	No
Haskell	?	?	?	?	?	No	No	No	No
RXP	Yes	Yes	No	Yes	Yes	No	No	No	No
ICU Regex	Yes	No	Yes	Yes[Note 7]	Yes	No	Yes	No	No
Java	Yes	No	Yes	Yes[Note 8]	Yes	No	Some[Note 6]	No	No
JavaScript (ECMAScript)	No	No	No	No	No	No	Some[Note 6][Note 9][4]	No	No
JGsoft	Yes	Yes	Yes	Yes	Yes	No	Some[Note 6]	No	Yes
Lua	No	No	No	No	No	No	No	No	No
.NET	Yes	Yes	Yes	Yes	Yes	No	Some[Note 6]	Yes	Yes
OCaml	No	No	No	No	No	No	No	No	No
PCRE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Perl	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No[Note 10]
PHP	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Python	Yes	Yes	Yes[Note 11]	Yes	Yes	No	Yes[Note 12]	No	Yes[Note 11]
Qt/QRegExp	No	No	No	No	No	No	No	No	No
RE2	Yes	No	?	Yes	No	No	Some[Note 6]	No	No
Ruby / Onigmo	Yes	Yes	Yes	Yes	Yes	Yes	Some[Note 6]	No	No
Tcl	Yes	No	Yes	No	Yes	No	Yes	No	No
TRE	Yes	No	No	No	Yes	No	?	No	No
Vim	Yes	No	Yes	No	No	No	No	No	Yes
RGX	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No
XML Schema	No	No	No	No	No	No	Yes	No	No
XPath 3/XQuery	No	No	No	No	No	No	Yes	No	No
XRegExp	Leading only	No	No	Yes	Yes	No	Yes	No	No

1. ^ Also known as *flags modifiers*, *modes modifiers* or *option letters*. Example pattern: "(?i:test)".
2. ^ Also called *independent sub-expressions*.
3. ^ Similar to back references, but with names instead of indices.
4. ^ Special feature allowing to match balanced constructs without recursion.
5. ^ Refers to the possibility of including quantifiers in look-behinds, thus making their length unpredictable.
6. ^ a b c d e f g h i Unicode property support may be incomplete (products are continuously updated!). *All will be incomplete* when a new Unicode revision is released *until* they are updated to comply.
7. ^ Available
8. ^ Available

Security Component

- What is a ReDoS attack?
  - “Regex Denial of Service”
  - Error when a certain input string takes a regex exponential time to match
- Example:
  - `regex = “^(([a-z])+.)+[A-Z]([a-z])+$”`
  - `input = “aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!”`

ReScue Algorithm



Input: Regex
Output: ReDoS String

- Genetic algorithm for incubation.
- Most desirable strings kept each generation

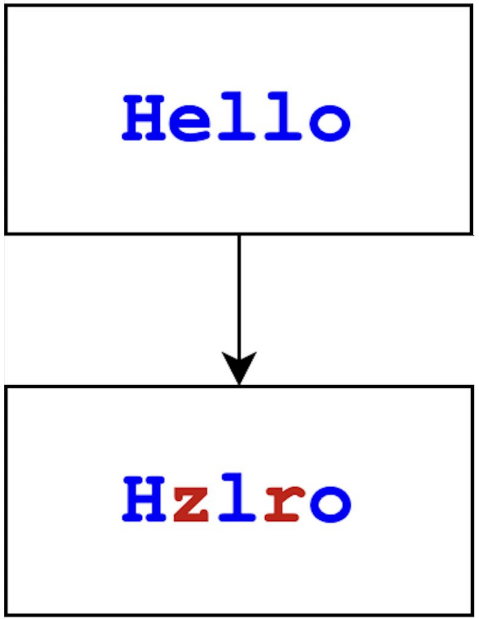
"ReScue: Crafting Regular Expression DoS Attacks"
Yuju Shen et al., Nanjing University

Seeding (Phase 1)

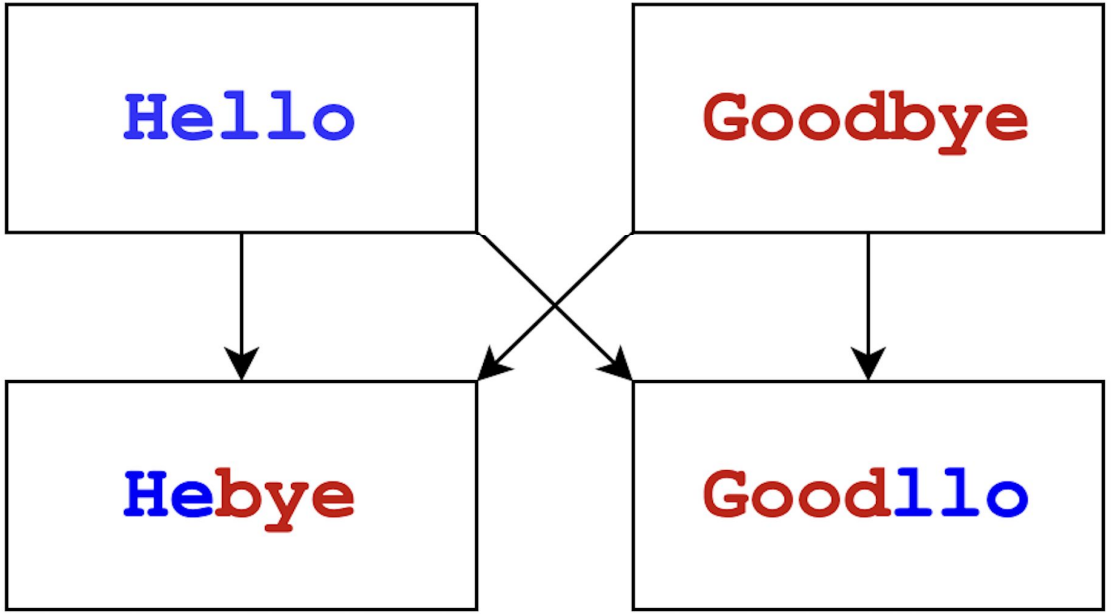
- Regex = “`hello.{1,10}goodbye`”
- Extract tokens from expression:
  - “`hello`”
  - “`goodbye`”
- Make seeds from the tokens, mixing in random characters:
  - “`goodbyeYY7goodbye6qL<GWhellogoodbye2>hello`”
  - “`hellogoodbyehellogoodbyeQhellogoodbye#K(goodbye \\\\fXhellohello4hellogoodbye`”

Genetic Algorithm Mutations (Phase 2)

Mutation:

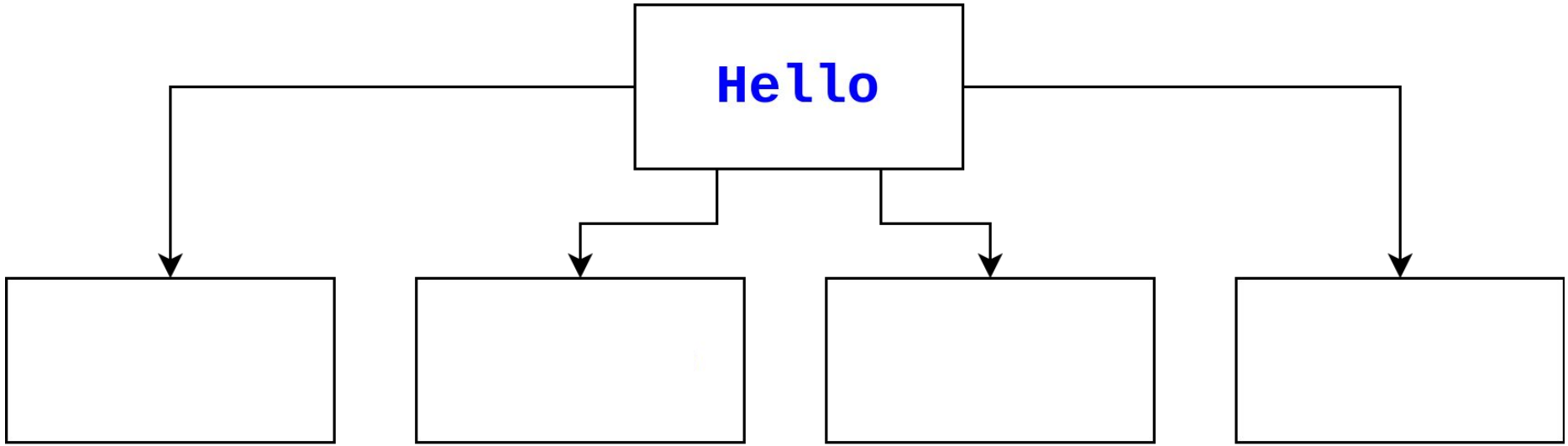


Crossover:

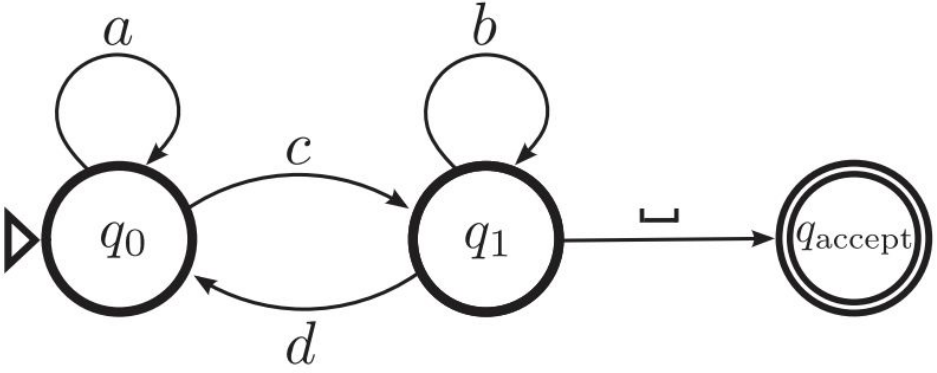


Genetic Algorithm Mutations (Phase 2)

Randomized Mutation:



Regex as Finite State Machines



State diagram for regex:
(a*cb*d)* a*cb*

- Any regex can be described as a Finite State Machine
- Match steps are graph dependent
- Implications for fitness functions and timing

Timing as a Fitness Function

- Slower strings = fitter
- Kept fittest strings in each generation
- RESCUE finds fittest strings by calculating matching steps per character
    - Essentially **O(**match(regex, input)**)**

- This requires a regex engine

Timing as a Fitness Function

- Instead we use **time(**match(regex, input)**)**

$$f_{fitness}(s) = \frac{|\tau(s)|}{|s|} \rightarrow f_{fitness}(s) = t(s)$$

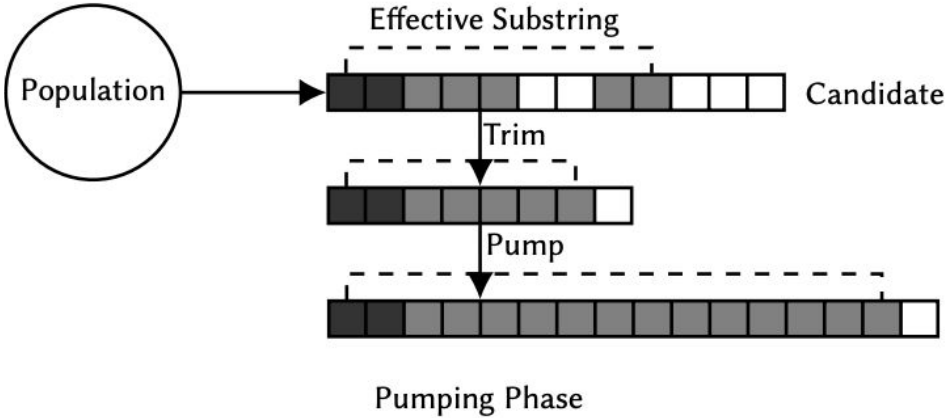
- Matching steps *τ(s)* ➤ process time *t(s)*
- Multiprocessing to allow for timeouts

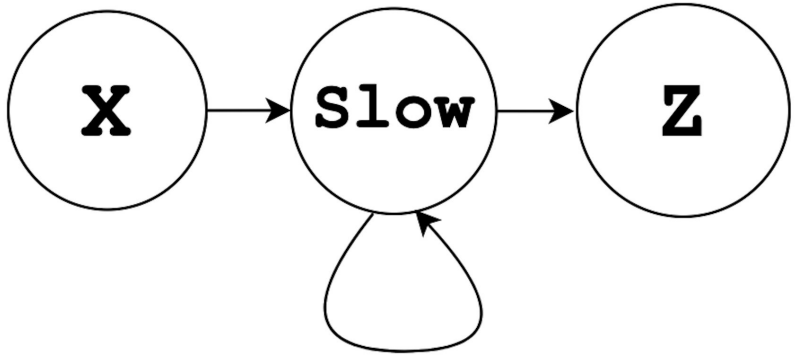
s = input string

τ(s) = matching steps of s

t(s) = time to match s

Pumping (Phase 3)

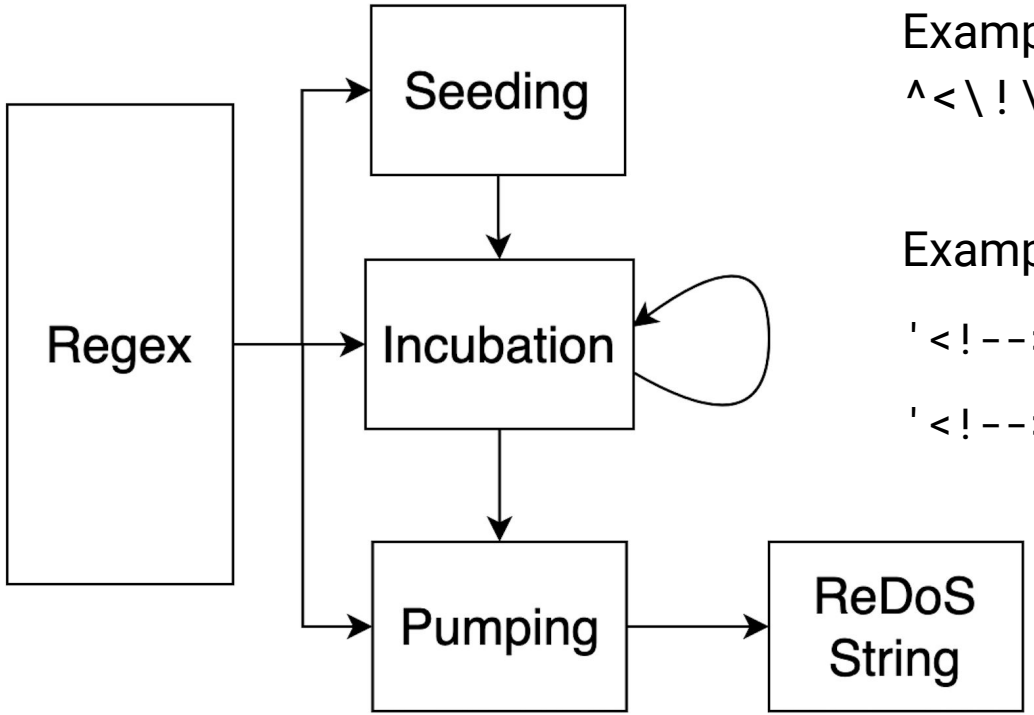





```
XslowZ -> XslowslowslowZ
```

```
Pumped output causes ReDoS
attack if used as input
```

ReScue Algorithm



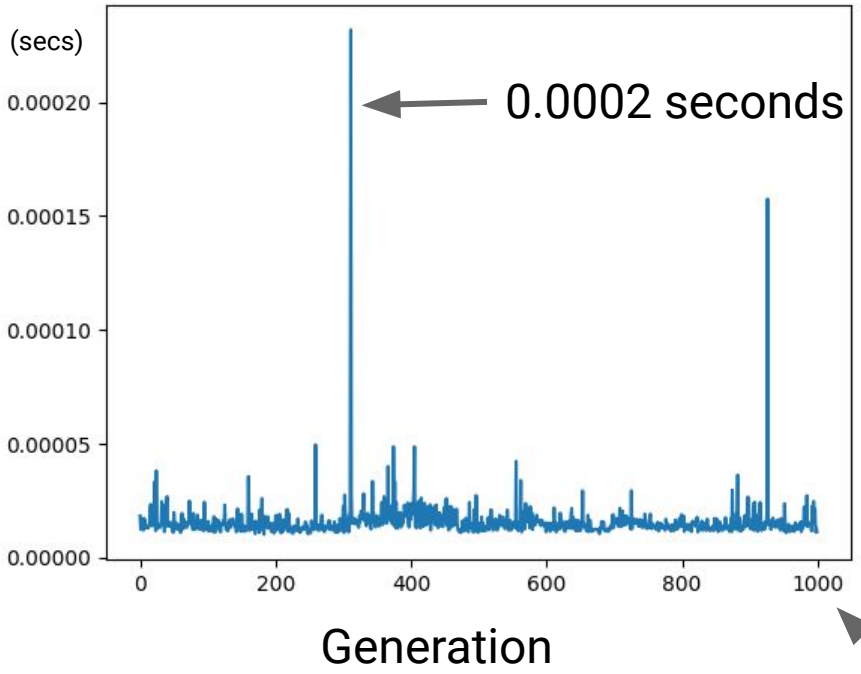
Example Regex:

```
^<\!\-\-(.*)+(\/){0,1}\-\->$
```

Example Output:

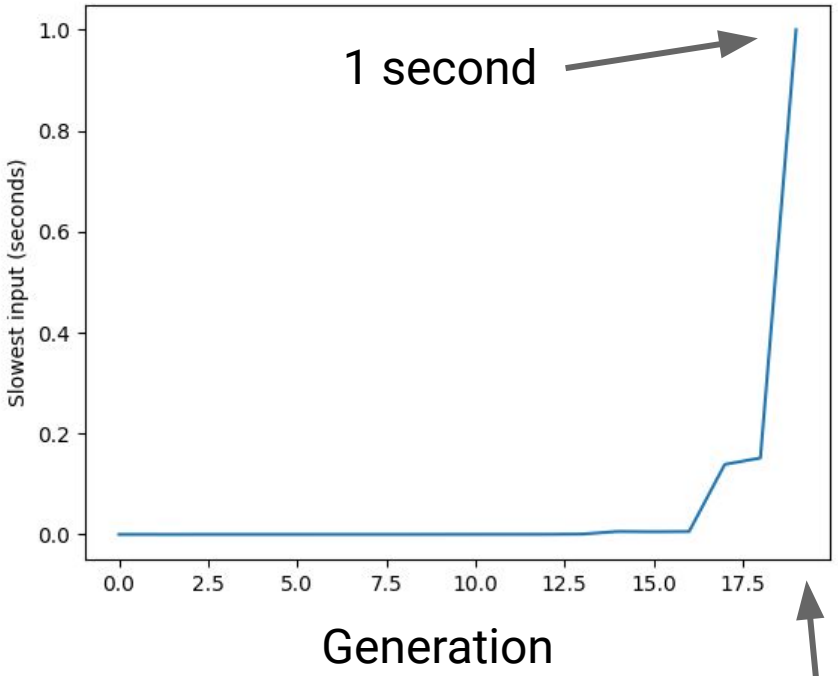
```
'<!-->`!--VvFcwh\hh^b!--^b!--' (6s)
'<!-->`!--VvFcwh\h^b!--h^b!--^b!--'
(4m)
```

`^(?=hello)[a-z]{5}`



1000

`^(([a-z])+.)+[A-Z]([a-z])+$`



18

Evolutionary Tracks

Output

```
?=
^Java, JavaScript, and Golang may not support conditionals
```

```
antonia@ARXPS:~/Dropbox (Carleton ege)/21_CS_Comps/regex-library$ python3 main.py ../demo-input/ --security
Expression ^<\!\-\-(.*)+(\/){0,1}\-\->a(?=bc|b)c\141S{3}$ from ../demo-input/example_1.py line 4
|0import re
|1
|2def main():
>3    my_patt = re.compile(r"^<\!\-\-(.*)+(\/){0,1}\-\->a(?=bc|b)c\141S{3}$")
|4    my_patt.match("<div>test</div>")
|5
|6if __name__ == "__main__":
|7    main()
\141
^Understandability suggestion: a
[aaa]
^Understandability suggestion: repeated characters in []
S{3}
^Understandability suggestion: SSS
?=
^Java, JavaScript, and Golang may not support conditionals
<!---->abcbcaaa<!---->abcbcaaa<!------->abcbc---->abcbcaaaS-->abcbc>abcb>abcbcaaaS"aS---->abcbcaa!--->abcbcaaaS
^This string gave the expression a run time longer than 1 second
<!-->abcbcaaa---->abcbcaaaS>abcbc+rcae>abcbcs<!---->abcbP->abcaafu&<!---->abcbcaaaS---->abcbcaaaS@--
8fi>abcbcaaaS-caa<!---->abcbcaaaSbcaaa<!---->abcbcaaaS`-->abc>abcbca---->abcbca
^This string gave the expression a run time longer than 1 second
Expression Hello from ../demo-input/folder_1/example_2.js line 4
|0
|1function test_function() {
|2    let text = "Hello";
>3    let n = text.search(/Hello/i);
|4}
|5
```

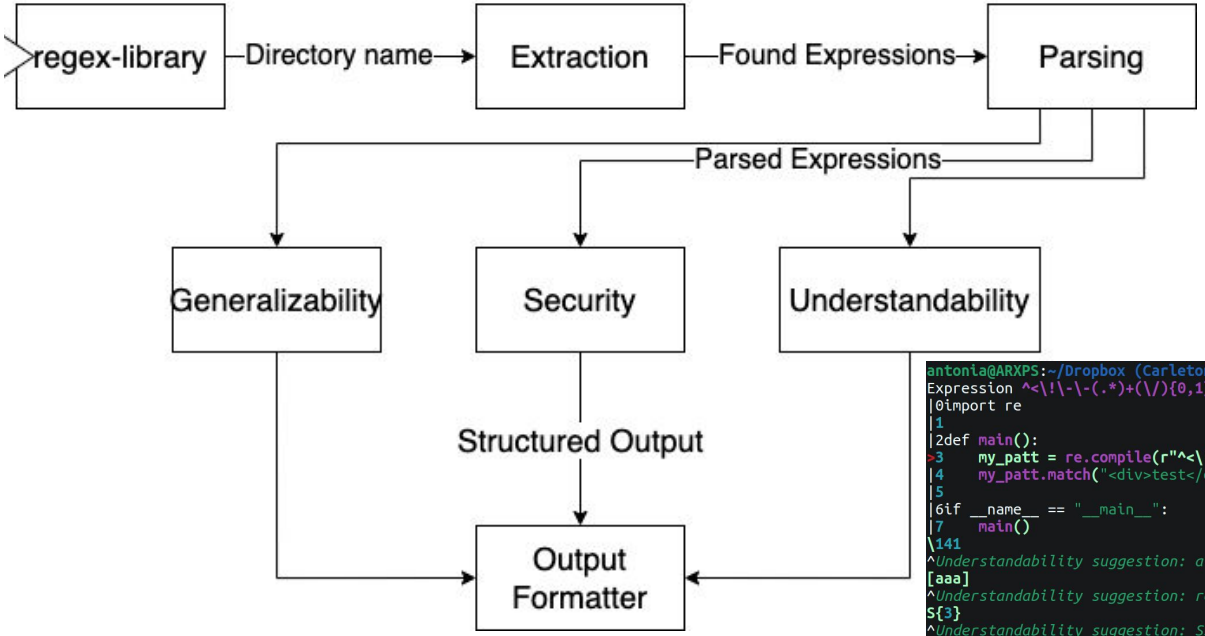
Annotation Fields

- Note
- One of
  - Token
  - Expression
  - Entity

rich_

References

- Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, 225–235. DOI:https://doi.org/10.1145/3238147.3238159
- Carl Chapman, Peipei Wang, and Kathryn T. Stolee. 2017. Exploring regular expression comprehension. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, 405–416.




```
antonia@ARXPS:~/Dropbox (Carleton College)/21_CS_Comps/regex-library$ python3 main.py ../demo-input/ --security
Expression ^<\!\-\-(.*)+(\/){0,1}\-\->a(?=bc|b)c\141${3}$ from ../demo-input/example_1.py line 4
|0import re
|1
|2def main():
>3    my_patt = re.compile(r"^<\!\-\-(.*)+(\/){0,1}\-\->a(?=bc|b)c\141${3}$")
|4    my_patt.match("<div>test</div>")
|5
|6if __name__ == "__main__":
|7    main()
\141
^Understandability suggestion: a
[aaa]
^Understandability suggestion: repeated characters in []
S{3}
^Understandability suggestion: SSS
?=
^Java, JavaScript, and Golang may not support conditionals
<!---->abcbcaaa<!---->abcbcaaa<!------->abcbc---->abcbcaaaS-->abcbc>abcb>abcbcaaaS"aS---->abcbcaa!--->abcbcaaaS
^This string gave the expression a run time longer than 1 second
<!-->abcbcaaa---->abcbcaaaS>abcbc+rcae>abcbcs<!---->abcbP->abcaafu&<!---->abcbcaaaS---->abcbcaaaS@--
8fi>abcbcaaaS-caa<!---->abcbcaaaSbcaaa<!---->abcbcaaaS`-->abc>abcbca---->abcbca
^This string gave the expression a run time longer than 1 second
Expression Hello from ../demo-input/folder_1/example_2.js line 4
|0
|1function test_function() {
|2    let text = "Hello";
>3    let n = text.search(/Hello/i);
|4}
|5
```

[qQ](uestions| *\& *[aA])\?*