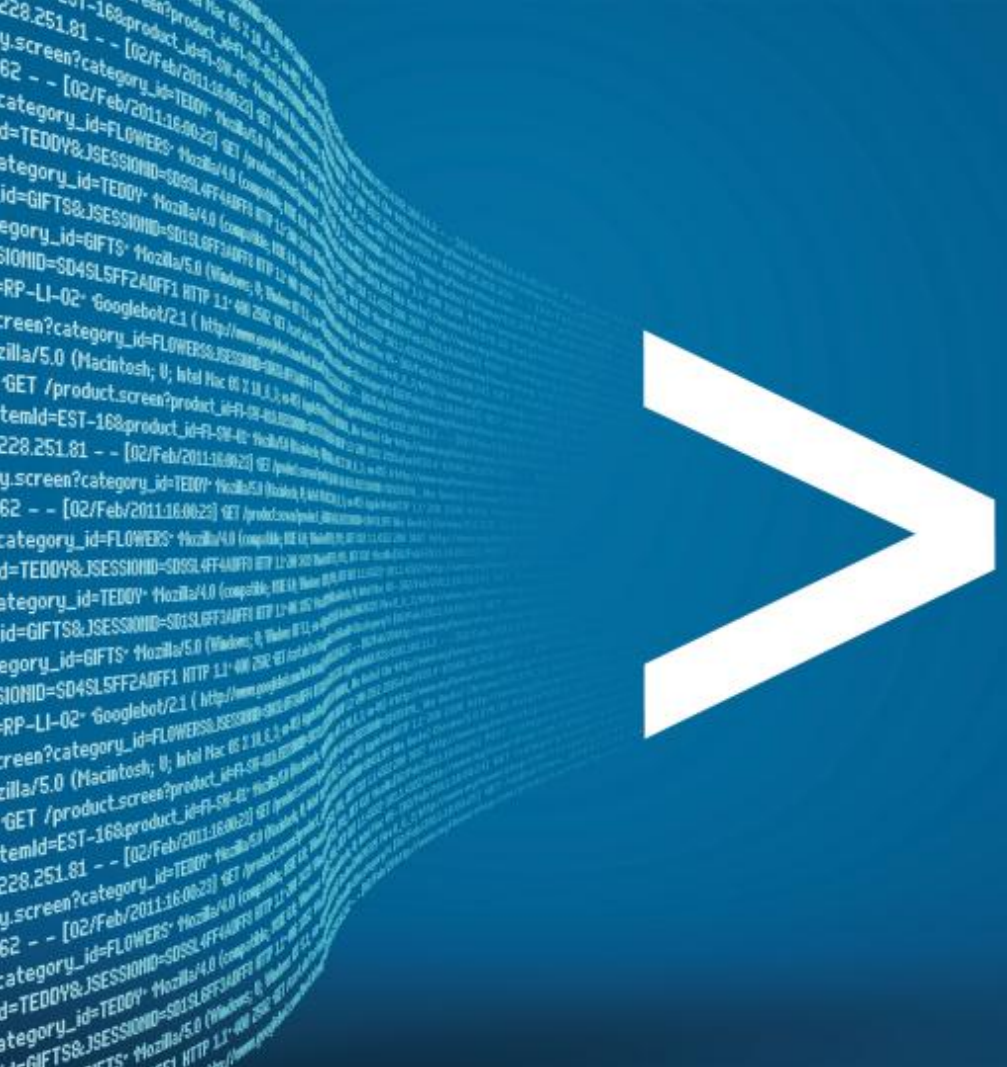


Regular Expressions

David Foster

Software Engineer

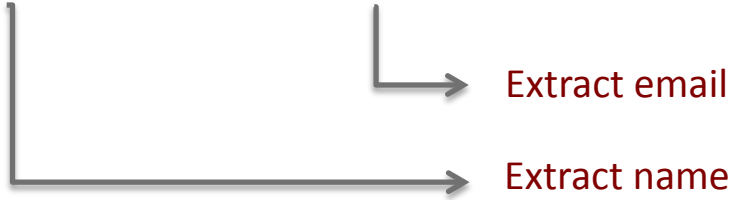
Aug 2013

splunk>

What is a regular expression?

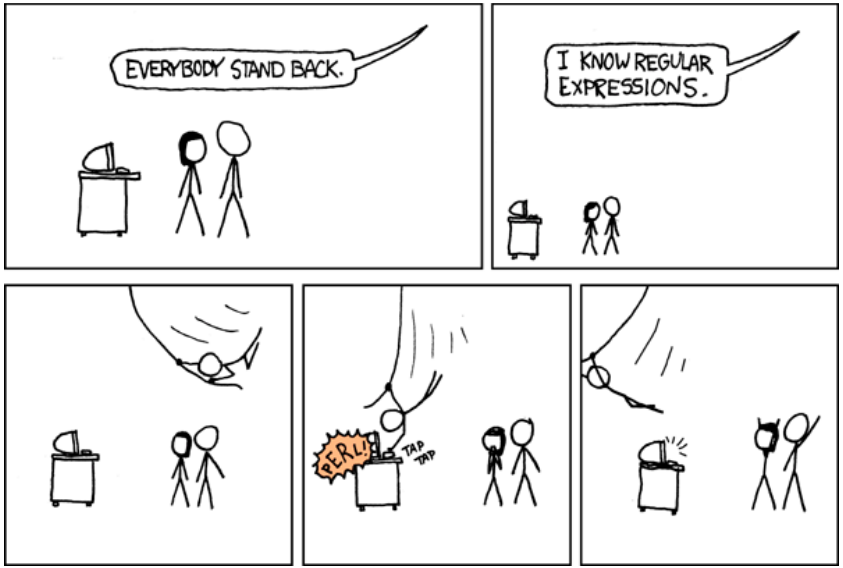
- A compact language for matching strings

- Boris Mann <bmann@example.com>



When to use them

- Regexes are good for:
  - Mechanical text transformations
  - Fuzzy searching
  - Optimized text manipulation



http://xkcd.com/208/

When NOT to use them

- Regexes are poor for:
  - Generalized parsing
    - HTML, XHTML, SGML
    - XML
    - JSON
  - Matching binary strings

> “Some people, when confronted with a problem, think “I know, I'll use regular expressions.” Now they have two problems.”

Don’t do this

- Matches every word in the English language:

```
(?:s(?:(?:u(?:b(?:(?:s(?:t(?:a(?:n(?:t(?:i(?:a(?:l(?:(?:i(?:s(?:m|
t)|a|ty|ze)|ly|ness))?|t(?:i(?:on|ve)|e|or)|bility)|v(?:e(?:(?:ly|
ness))?|al(?:ly)?|i(?:ty|ze))|fy|ous|ze))?|c(?:e(?:less)?|h)|
dard(?:ize)?)|lagmit(?:e|ic)|ge|tion)|r(?:a(?:t(?:o(?:s(?:pher(?:e|
ic)|e)|r)|i(?:ve)?|al|e|um)|ct(?:ion)?)|uct(?:(?:ion(?:al)?|
ur(?:al|e)))?|iate)|itu(?:t(?:i(?:on(?:a(?:l(?:ly)?|ry))?|
ng(?:ly)?|ve(?:ly)?)|e(?:(?:d|r))?|able)|ent)|o(?:r(?:eroom|y)|ck)|
yl(?:ar|e)|ernal)|e(?:r(?:v(?:i(?:en(?:t(?:(?:ly|ness))?|c(?:e|y))|
ate)|e)|o(?:sa|us)|ies|rate)|c(?:u(?:t(?:e|ive)|rity)|retar(?:ial|
y)|t(?:ion)?|ive)|quen(?:t(?:(?:ial(?:ly)?|ly|ness))?|c(?:e|y))|
ns(?:u(?:al|ous)|ation|ible)|pt(?:uple)?|mi(?:fusa|tone)|xtuple|...
```

https://gist.github.com/noprompt/6106573/raw/fcb683834bb2e171618ca91bf0b234014b5b957d/word-re.clj

Simple Expressions

Simple Expressions (1/4)

- Goal: Match “hiss!” or anything similar with two or more s letters

- hiss+!

♦+	Repetition (1..∞)
♦	Literal Character

Simple Expressions (2/4)

- Goal: Match one or more “buffalo” words, separated by spaces

- buffalo(buffalo)*

( )	Group
◆*	Repetition (0..∞)
◆	Literal Character

Simple Expressions (3/4)

- Goal: Match any “word”

[a-zA-Z]+

- But what about words like “can't”?

[a-zA-Z']+

Need to accept apostrophes too.
(The computer can’t guess this magically.)

[◇]	Char in ◇
◆+	Repetition (1..∞)
◆	Literal Character

Simple Expressions (4/4)

- Goal: Match the same word repeated 1 or more times

$$\underbrace{\text{([a-zA-Z]+)}}_{\backslash 1}\underbrace{\text{(\backslash 1)}}_{\backslash 2}\text{*}$$

- Now things are getting interesting…

( )	Group
[◇]	Char in ◇
◆*	Repetition (0..∞)
◆+	Repetition (1..∞)
\1	Backreference
◆	Literal Character

Real World Examples

Example: Email Extraction

- Boris Mann <bmann@example.com> → bmann@example.com
- John Doe <jdoe@example.com> → jdoe@example.com
- Bob Waters <bwaters@example.com> → bwaters@example.com

[^<]* <([^>]+)> → \1

( )	Group
[^◇]	Char NOT in ◇
◆*	Repetition (0..∞)
◆+	Repetition (1..∞)
\1	Backreference
◆	Literal Character

Example: Fuzzy Matching

- Getting Started with the new App Framework
- …
- …

- App(|%20|\+)Framework

or even better…

- App.{0,5}Framework

( )	Group
|	Choice (OR)
\◆	Escaped Char
◆	Literal Character
.	Any Character
◆{n,k}	Repetition (n..k)

Example: Change File Extension

- README.markdown → README.md
- Buttercup.JPG → Buttercup.jpg
- com.splunk.Input.htm → com.splunk.Input.html

^(.+)\.([a-z]+)$ → \1.md

Only \1 is special for replacements.
Dot is not special here.

( )	Group
[◇]	Char in ◇
^◇	Anchor to start
◇$	Anchor to end

- https://github.com/davidfstr/renameregex
  - Note: Java replacement expressions use $1 instead of \1.

Memory Tip: ^ vs. $

- These match the beginning and end of input.
- I sometimes forget which is which.

^ = "Wake up at the start of the day…"

$ = "…and make money by the end of it."

Example: Find Identifiers

- ChartElement → ChartView
- SingleElement → SingleView
- TableElement → TableView

\b([a-zA-Z]+)Element\b → \1View

\b	Anchor to word boundary

- When matching word boundaries on both ends, many editors have a "Match Entire Word" option that does the same thing as adding \b to each side.

Advanced Expressions

Advanced: Reluctant Quantifiers (1/3)

- Goal: Delete the first item in a comma-separated list

^(.+),(.+)$ → \2

1,2,3,4,5 → 5 Oops

Advanced: Reluctant Quantifiers (2/3)

- What happened?
  - The first .+ ate everything and matched the *last* comma in the list instead of the first one.

^(.+),(.+)$ → \2

Very hungry. Om nom nom.

Advanced: Reluctant Quantifiers (3/3)

- We want to make the + less hungry.
  - Every quantifier (+, *, {n,k}) has a *reluctant* version that eats as *little* as possible. Just add a ? after the greedy version.

^(.+?),(.+)$ → \2

"Do I really want to eat that character? I'm on a diet."

1,2,3,4,5 → 2,3,4,5 Yay!

Tip: Avoid the dot

- If we had used a more specific regex, it wouldn't even be necessary to use a reluctant quantifier:

^([^,]+),(.+)$ → \2

No dot? No ambiguity.

1,2,3,4,5 → 2,3,4,5 Still good

Advanced: Abbreviated Character Classes

- Not recommended since they're hard to remember.

Character Set	Abbreviation
[A-Za-z0-9_]	\w (*word*, NOT whitespace)
[^A-Za-z0-9_]	\W (non-word)
[\t\r\n\v\f]	\s (white*space*)
[0-9]	\d (digit)

- Prefer writing out character sets explicitly.

Advanced: Noncapturing Groups

- A special kind of ( ) that cannot be referenced by \1, \2, …, \n
  - Useful when the ( ) is only there for a | or a quantifier: (◇)?, (◇)+, (◇)*

- Goal: Recognize an integer (5) or decimal (5.37)
  - *but not .37 (to keep this demo simple)*

(?:◇) Non-C Group

\1 \2

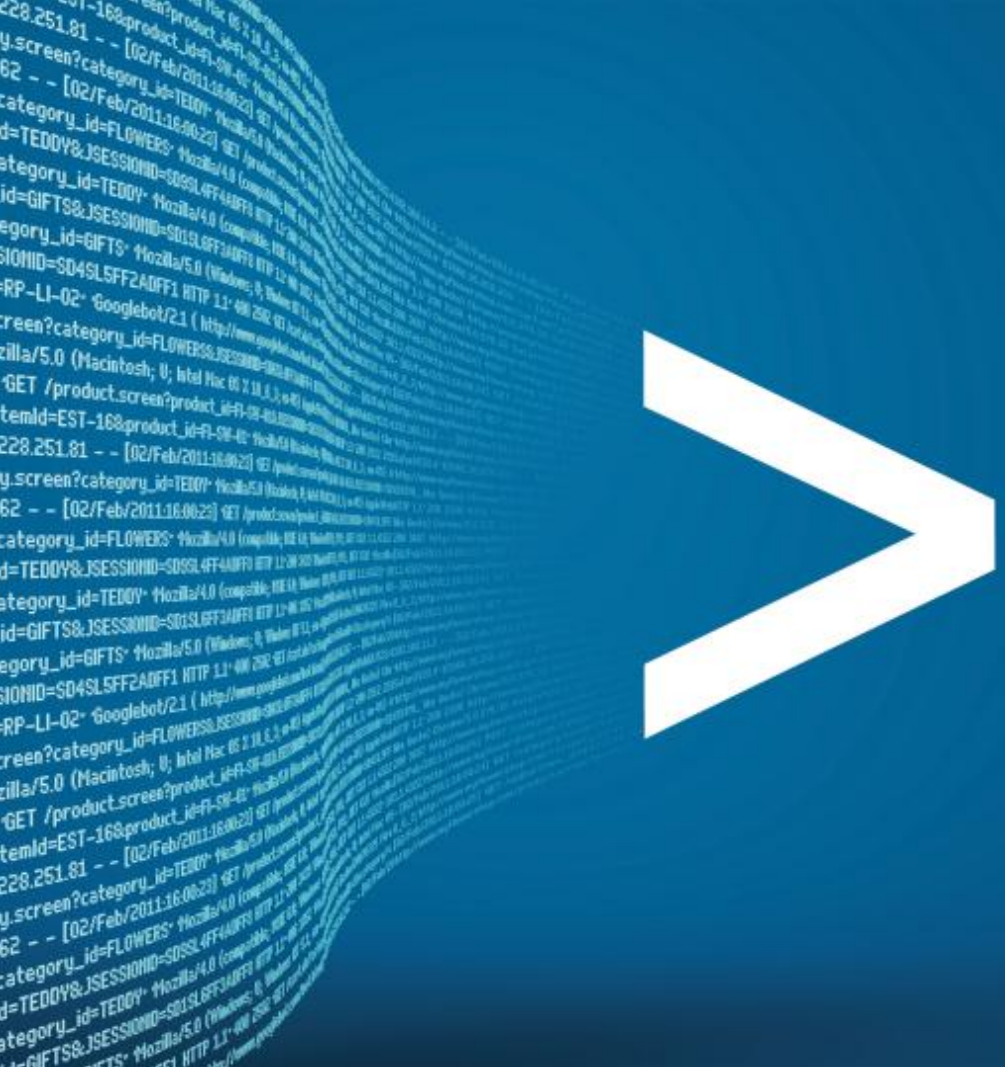
([0-9]+)(?:\.([0-9]+))?

Noncapturing Group

Syntax Summary

( )	Group
[◇]	Char in ◇
[^◇]	Char NOT in ◇
|	Choice (OR)
.	Any Character
◆?	Repetition (0..1)
◆*	Repetition (0..∞)
◆+	Repetition (1..∞)
\1	Backreference
\◆	Escaped Char
◆	Literal Character

(?:◇)	Noncapturing Group
\w	Word Character
\s	Whitespace Character
\d	Digit Character
◆??	Reluctant Repetition (0..1)
◆*?	Reluctant Repetition (0..∞)
◆+?	Reluctant Repetition (1..∞)
^◇	Anchor to start
◇$	Anchor to end
\b	Anchor to word boundary



Thank You

splunk>