Regular Expressions

Regular Expressions are so cool. Knowledge of regexes will allow you to save the day.

CONTENTS

Definitions

In formal language theory, a **regular expression** (a.k.a. regex, regexp, or r.e.), is a string that represents a regular (type-3) language.

Huh??

Okay, in many programming languages, a **regular expression** is a **pattern** that **matches** strings or pieces of strings. *The set of strings they are capable of matching goes way beyond what regular expressions from language theory can describe.*

Basic Examples

Rather than start with technical details, we'll start with a bunch of examples.

Regex	Matches any string that
`hello`	contains {hello}
`gray|grey`	contains {gray, grey}
`gr(a|e)y`	contains {gray, grey}
`gr[ae]y`	contains {gray, grey}
`b[aeiou]bble`	contains {babble, bebble, bibble, bobble, bubble}
`[b-chm-pP]at|ot`	contains {bat, cat, hat, mat, nat, oat, pat, Pat, ot}
`colou?r`	contains {color, colour}
`rege(x(es)?|xps?)`	contains {regex, regexes, regexp, regexps}
`go*gle`	contains {ggle, gogle, google, gooogle, goooogle, ...}
`go+gle`	contains {gogle, google, gooogle, goooogle, ...}
`g(oog)+le`	contains {google, googoogle, googoogoogle, googoogoogoogle, ...}
`z{3}`	contains {zzz}
`z{3,6}`	contains {zzz, zzzz, zzzzz, zzzzzz}
`z{3,}`	contains {zzz, zzzz, zzzzz, ...}
`[Bb]rainf\*\*k`	contains {Brainf**k, brainf**k}
`\d`	contains {0,1,2,3,4,5,6,7,8,9}
`\d{5}(-\d{4})?`	contains a United States zip code
`1\d{10}`	contains an 11-digit string starting with a 1
`[2-9]|[12]\d|3[0-6]`	contains an integer in the range 2..36 inclusive
`Hello\nworld`	contains Hello followed by a newline followed by world
`mi.....ft`	contains a nine-character (sub)string beginning with mi and ending with ft (Note: depending on context, the dot stands either for "any character at all" or "any character except a newline".) Each dot is allowed to match a different

	character, so both microsoft and minecraft will match.
`\d+(\.\d\d)?`	contains a positive integer or a floating point number with exactly two characters after the decimal point.
`[^i*&2@]`	contains any character other than an i, asterisk, ampersand, 2, or at-sign.
`//[^\r\n]*[\r\n]`	contains a Java or C# slash-slash comment
`^dog`	begins with "dog"
`dog$`	ends with "dog"
`^dog$`	is exactly "dog"

Notation

There are many different syntaxes for regular expressions, but in general you will see that:

- Most characters stand for themselves
- Certain characters, called metacharacters, have special meaning and **must be escaped** (usually with `\` if you want to use them as characters. In most syntaxes the metacharacters are:

  ( ) [ ] { } ^ $. \ ? * + |

- Within square brackets, you only have to escape (1) an initial `^`, (2) a non-initial or non-final `-`, (3) a non-initial `]`, and (4) a `\`.

Using Regular Expressions

Many languages allow programmers to define regexes and then use them to:

- **Validate** that a piece of text (or a portion of that text) matches some

pattern
- **Find** fragments of some text that match some pattern
- **Extract** fragments of some text
- **Replace** fragments of text with other text

Generally a regex is first **compiled** into some internal form that can be used for super fast validation, extraction, and replacing. Sometimes there is an explicit `compile` function or method, and sometimes special syntax is used to compile, such as the very common form `/.../`.

Validation

Example: find `"color"` or `"colour"` in a given string.

```
// Java
Pattern p = Pattern.compile("colou?r");
Matcher m = p.matcher("The color green");
m.find();                           // returns true
m.start();                          // returns 4
m.end();                            // returns 9
m = p.matcher("abc");
m.find();                           // returns false


# Perl
$p = /colou?r/;
"The color green" =~ $p;            # returns 1 (cuz no Perl true)
"abc" =~ $p;                        # returns 0 (cuz no Perl false)


# Ruby
p = /colou?r/
"The color green" =~ p              # returns 4
"abc" =~ p                          # returns nil


# Python
p = re.compile("colou?r")
m = p.search("The color green")
m.start()                           # returns 4
m = p.search("abc")                 # returns None
```

```
// JavaScript
const p = /colou?r/;
"The color green".search(p);        // returns 4
"abc".search(p);                    // returns -1
```

If you want to know if an entire string matches a pattern, define the pattern with `^` and `$`, or with `\A` and `\Z`. In Java, you can call `matches()` instead of `find()`.

Extraction

After doing a match against a pattern, most regex engines will return you a bundle of information, including such things as:

- the part of the text that matched the pattern
- the index within the string where the match begins
- each part of the text matching the parenthesized portions within the pattern
- (sometimes) the text before the matched text
- (sometimes) the text after the matched text

Example in Ruby:

```
>> pattern = /weighs (\d+(\.\d+)?) (\w+)/
=> /weighs (\d+(\.\d+)?) (\w+)/
>> pattern =~ 'The thing weighs 2.5 kilograms or so.'
=> 10
>> $&
=> "weighs 2.5 kilograms"
>> $1
=> "2.5"
>> $2
=> ".5"
>> $3
=> "kilograms"
>> $`
=> "The thing "
>> $'
=> " or so."
```

The same thing in JavaScript:

```
> const pattern = /weighs (\d+(\.\d+)?) (\w+)/
> pattern.exec('The thing weighs 2.5 kilograms or so.')
[ 'weighs 2.5 kilograms',
  '2.5',
  '.5',
  'kilograms',
  index: 10,
```

```
  input: 'The thing weighs 2.5 kilograms or so.' ]
```

Note how in JavaScript, the match result object looks like an array and an object.

The so-called *group numbers* are found by counting the left-parentheses in the pattern:

TODO PICTURE GOES HERE

Sometimes you need parentheses only for precedence purposes and you don't want to incur the cost of extracting a group. We have **non-capturing groups** for this purpose.

Ruby:

```
>> phone = /((\d{3})(?:\.|-))?(\d{3})(?:\.|-)(\d{4})/
=> /((\d{3})(?:\.|-))?(\d{3})(?:\.|-)(\d{4})/
>> phone =~ 'Call 555-1212 for info'
=> 5
>> [$`, $&, $', $1, $2, $3, $4, $5]
=> ["Call ", "555-1212", " for info", nil, nil, "555", "1212", nil]
>> phone =~ '800.221.9989'
=> 0
>> [$`, $&, $', $1, $2, $3, $4, $5]
=> ["", "800.221.9989", "", "800.", "800", "221", "9989", nil]
>> phone =~ '1800.221.9989'
=> 1
>> [$`, $&, $', $1, $2, $3, $4, $5]
=> ["1", "800.221.9989", "", "800.", "800", "221", "9989", nil]
```

JavaScript:

```
> const r = /((\d{3})(?:\.|-))?(\d{3})(?:\.|-)(\d{4})/g;
> const m = r.exec("Call 1.800.555-1212 for info");
> m.index
7
> JSON.stringify(m);
["800.555-1212","800.","800","555","1212"]
```

Java

```
Pattern phone = Pattern.compile("((\\d{3})(?:\\.|-))?(\\d{3})(?:\\.|-)(\\d{4})
String[] tests = {"Call 555-1212 for info", "800.221.9989", "1800.221.9989"};
for (String s : tests) {
    Matcher m = phone.matcher(s);
    m.find();
    System.out.println("groupCount = " + m.groupCount());
    System.out.println("group(0) = " + m.group(0));
    System.out.println("group(1) = " + m.group(1));
```

```
        System.out.println("group(2) = " + m.group(2));
        System.out.println("group(3) = " + m.group(3));
        System.out.println("group(4) = " + m.group(4));
    }
```

```
groupCount = 4
group(0) = 555-1212
group(1) = null
group(2) = null
group(3) = 555
group(4) = 1212
groupCount = 4
group(0) = 800.221.9989
group(1) = 800.
group(2) = 800
group(3) = 221
group(4) = 9989
groupCount = 4
group(0) = 800.221.9989
group(1) = 800.
group(2) = 800
group(3) = 221
group(4) = 9989
```

Substitution

Many languages have `replace` or `replaceAll` methods that replace the parts of a string that match a regex. Sometimes you will see a `g` flag on a regex instead of a `replaceAll` function.

```
alert("Rascally Rabbit".replace(/[RrLl]/g, "w"));
```

Components of Regexes

Character Classes

- Square brackets `[ ]` — means exactly one character
- A leading `^` negates, a non-leading, non-terminal `-` defines a range:

```
[abc]          a or b or c
[^abc]         any character _except_ a, b, or c (negatior
```

```
[a-zA-Z]       a through z or A through Z, inclusive (rang
```

- If you have a `]` in your set, put it first. Use `\` to escape.
- Java allows crazy extensions:

```
[a-d[m-p]]     [a-dm-p] (union, Java only, I think)
[a-e&&[def]]   [de] (intersection, Java only, I think)
[a-r&&[^bq-z]] [ac-p] (subtraction, Java only, I think)
```

- Other ways to say exactly one character from a set are:

```
\d             [0-9]
\D             [^\d]
\s             [ \t\n\x0B\f\r]
\S             [^\s]
\w             [a-zA-Z0-9_]
\W             [^\w]
.              any character at all, except maybe not a li
```

Groups

Defined above, in the section on extraction.

Quantifiers

Generally, 18 types:

	Eager	Reluctant	Possessive
Zero or one	`?`	`??`	`?+`
Zero or more	`*`	`*?`	`*+`
One or more	`+`	`+?`	`++`
m times	`{m}`	`{m}?`	`{m}+`
At least m times	`{m,}`	`{m,}?`	`{m,}+`
At least m, at most n times	`{m,n}`	`{m,n}?`	`{m,n}+`

Eager (Greedy and Generous) — match as much as possible, but give back

```
\w+\d\d\w+  // matches abcdef42skjhfskjfhsjdfs
            // but inefficiently
```

Possessive — match as much as possible, but do NOT give back

```
\w++\d\d\w+  // does not match abcdef42skjhfskjfhsjdfs
             // but is efficient
```

Reluctant — match as little as possible

```
\w+?\d\d\w+  // matches abcdef42skjhfskjfhsjdfs
             // efficiently, yay!
```

Backreferences

Things captured can be used later:

```
<(\w+)>[^<]*</\1>
```

Anchors, Boundaries, Delimiters

Some regex tokens do not consume characters! They just assert the matching engine is at a particular place, so to speak.

- `^`: Beginning of string (or line, depending on the mode)
- `$`: End of string (or line, depending on the mode)
- `\A`: Beginning of string
- `\z`: End of string
- `\Z`: Varies a lot depending on the engine, so be careful with it
- `\b`: Word boundary
- `\B`: Not a word boundary

Read more about these at Rexegg.

Also, the lookarounds (up next!) don't consume any characters either!

Lookarounds

- Lookarounds do not consume anything
- Even though they have parens, they do not capture
- **Positive Lookahead**: Matches only if followed by something

```
Hillary(?=\s+Clinton)
```

matches the Hillary in Hillary Clinton but not the Hillary in Hillary Makasa.

- **Negative Lookahead**: Matches only if not followed by something

```
q(?!u)
\b(?:[a-eg-z]|f(?!oo))\w*\b      // Word not starting with
\b(?:[a-eg-z]|f(?!oo\b))\w*\b    // Any word except foo
((?!foo).*)
```

- **Positive Lookbehind**: Matches only if preceded by something

```
(?<=-)\p{L}+              // a word following a hyphen
(?<=http://)\S+           // URL not including the http://
```

- **Negative Lookbehind**: Matches only if not preceded by something

```
(?<![-+\d])(\d+)      // Digits not preceded by a digit, +
```

- Lookarounds show up in search and replace applications

```
Pattern p = Pattern.compile("Hillary(?=\\s+Clinton)");
String text = "Once Hillary Clinton was talking about Sir\n" +
    "Edmund Hillary to Hillary Makasa and then Hillary\n" +
    "Clinton had to run off on important business.";
Matcher m = p.matcher(text);
System.out.println(m.replaceAll("Secretary"));
```

Note: Read this awesome article on lookarounds.

Modifiers

A modifier affects the way the rest of the regex is interpreted. Not every language supports all of the modifiers below. For example, JavaScript (officially) supports only i, g, and m.

Modifier	Meaning
`g`	global
`i`	ignore case
`m`	multiple line
`s`	single line (DOTALL): Means that the dot matches any character at all. Without this modifier, the dot matches any character except a newline.
`x`	ignore whitespace in the pattern
`d`	Unix line mode: Considers only U+000A as a line separator, rather than U+000D or the U+000D/U+000A combo or even U+2028.
`u`	Unicode case: in this mode the case-insensitive modifier respects Unicode cases; outside of this mode that modifier only consolidates cases of US-ASCII characters.

Performance Pitfalls

You should know some things about how your *regex engine* works since two "equivalent" regexes can have drastic differences in processing speed.

- It is possible to write regexes that take exponential time to match, but you pretty much have to TRY to make one (they’re pathological)
- It is more common to accidentally create regexes that run in quadratic time
- Main types of problems
  - Recompilation (from forgetting to compile regexes used multiple times)

```
// Java shortcut, should not be used in most cirumstances
s.matches("colou?r");
```

- Dot-star in the Middle (which causes backtracking)
  - Solution 1: Use negated character class
  - Solution 2: Use reluctant quantifiers
- Nested Repetition

Performance Tips

Always do the following:

- Use non-capturing groups when you need parentheses but not capture.
- If the regex is very complex, do a quick spot-check before attempting a match, e.g.
  - Does an email address contain '`@`'?
- Present the most likely alternative(s) first, e.g.
  - `black|white|blue|red|green|metallic seaweed`
- Reduce the amount of looping the engine has to do
  - `\d\d\d\d\d` is faster than `\d{5}`
  - `aaaa+` is faster than `a{4,}`
- Avoid obvious backtracking, e.g.
  - `Mr|Ms|Mrs` should be `M(?:rs?|s)`
  - `Good morning|Good evening` should be `Good (?:morning|evening)`

Miscellaneous Language-Specific Notes

A few things that are good to know:

- Java's built-in support for regexes exceeds that of many languages
- Especially good for Unicode and for character classes (has more than

Perl)
- Syntax is more cumbersome (string literal support weak, no operator for matching...) — live with it!
- Perl has nice regex, too, even allows you can even embed code inside them.
- Great Perl documentation at the perldoc pages perlrequick, perlretut, perlre. and perlreref.
- JavaScript seems to less extensive support than other languages, but I think this is changing.
- Python puts regex functions in a module.
- Python docs are here.

Study and Practice

Here are some good sources:

- The Premier Site for Regexes
- Maybe this is a better premier site
- Regex 101—Develop regexes online
- RegExr (Awesome online tool for Java regexes)
- Rubular (Ruby Online Regex Tester)
- Perl Regular Expressions Tutorial
- Perl Regular Expressions (Manual)
- Perl Regular Expressions Quick Reference
- Ruby Regexp class documentation
- Java Regex Tutorial
- Java Regex Optimization Article
- Java Pattern class API docs
- JavaScript Regular Expressions (at Mozilla Developer Center)
- Regexpal (online JavaScript regex tester)
- Python Regular Expressions