REGEX GOLF:
YOU TRY TO MATCH ONE GROUP BUT NOT THE OTHER.
/M | [TN]|B/ MATCHES STAR WARS SUBTITLES BUT NOT STAR TREK.
COOL.
META-REGEX GOLF:
SO I WROTE A PROGRAM THAT PLAYS REGEX GOLF WITH ARBITRARY LISTS...
UH OH...
META-META-REGEX GOLF:
...BUT I LOST MY CODE, SO I'M GREPPING FOR FILES THAT LOOK LIKE REGEX GOLF SOLVERS.
...AND BEYOND:
REALLY, THIS IS ALL /(META-)*REGEX GOLF/.
NOW YOU HAVE INFINITE PROBLEMS.
NO, I HAD THOSE ALREADY.

Credit: Randall Munroe xkcd.com

CS 2112 Lab 7: Regular Expressions

October 21 / 23, 2019

Regex Overview

- Regular Expressions, also known as ‘regex’ or ‘regexps’ are a common scheme for pattern matching in strings
- A regular expression is represented as a single string and defines a set of matching strings
- The set of strings matched by a regex is the *language* of the regular expression.

Regex implementations

- Java supports Perl-style regular expressions through `java.util.regex`
- The `easyIO` package provided with the course also supports regular expressions.
- Regex terminology is incredibly variable from source to source, almost everything presented here has other names in certain contexts.

The simplest regex

- The simplest regular expression is just a string
- The regex `CS2112` matches only the string "CS2112"
- We can add special characters to add more power.

Concatenation and Alternation

- The concatenation AB of two regular expressions A and B matches all strings with a first part matched by A followed by a second part matched by B.
  - Regex `ab` is really just the concatenation of `a` and `b`.
- The alternation $A|B$ of regexes A and B matches any string that is matched by *either* A or B.
  - Regex `hello|goodbye` matches both `hello` and `goodbye`.
  - Regex `d(aa|bb)c` matches both `daac` and `dbbc`.

Quantifiers

- ab matches only the string ab
- (ab) matches only the string ab (parentheses just do grouping)
- (ab)* matches any number of ab's, including the empty string: "", "ab", "abab", etc.
    - Precedence: ab* matches an a followed by any number of b's: "a", "ab", "abb", etc.
- (ab)+ matches one or more ab's. (Same as ab(ab)*)
- (ab)? matches "ab" or the empty string. (Same as ab|)
- 0{3,5} matches 000, 0000, or 00000

Character classes

- Character classes specify a set of characters to match against: syntactic sugar for alternation.
- [1] is a trivial class that behaves just like “1”.
- [01] matches 0 or 1 (but not both: same as 0|1)
- [01]{2} matches 00, 11, 01, or 10
- Ranges let you match sets of consecutive characters without typing them all out:
  - [a-z] matches any lowercase letter, [a-z]+ any lowercase word.
  - [0-9] matches any digit.

Combinations

- Character classes and Quantifiers mix to give useful expressions
- `[a-z]*` matches any number of consecutive lowercase characters
- `[0-9]+` matches all numbers
- `[0-9]{3}` matches all three digit numbers
- `[A-z]{4}` matches all four letter words

Negation

- The `^` character beginning a character class is the logical negation operator
- `[^0]` matches any character but `0`
- `[^abc]` matches any character but `abc`
- `[^a-z]` matches any character but lowercase letters

Predefined Character classes

- Predefined character classes are shorthand for commonly used character classes
- In most cases the capital letter is the negation of the lowercase
- `\d = [0123456789]`, `\D = [^0123456789]`
- `\s` matches white space (`\t`, `\n`, `\r`, etc.)
- `\w` matches "word" characters, basically not whitespace and punctuation.
- `.` matches anything but a newline. This is super useful.
- There are a lot of these, fortunately the internet knows all of them!

Groups

- Groups allow a section of the expression to be remembered for later
- \n matches the substring captured by the n^{th} capture group.
- `(\d):\1` matches `1:1` or `7:7` but not `2:3`
- `(0|1)` matches `0` or `1`
- `(0|1):\1` matches `1:1` or `0:0` but not `0:1`
- `(10)` matches the string `10` but not `1` or `0` alone
- We'll see later that groups can be captured and extracted to do something useful after matching.

Escapes

- regex uses the standard escape sequences like `\n, \t, \\`
- Characters normally used in quantifiers and groups must also be escaped
- This includes `\+ \( \. \^` among others.

Examples

- Multiple combinations start to get at the real power of regex
- `[A-z][0-9]` matches things like `A1`, `B6`, `q0`, etc.
- `[A-Z][a-z]* [A-z][a-z]*` matches a properly capitalized first and last name (unless you have a name like O'Brian or McNeil)
- `java\.util\.[^(Scanner)].*` matches things disallowed on A3.

Exercise

Write a regex to match Cornell netIDs.

Java.lang.String

The easiest way to start using regular expressions in Java is through methods provided by the String class. Two examples are ”String.split(String)” and ”String.replaceAll(String,String)”.

```java
String TAs = "Reese&Matt&Clara&Chin";

String[] arr = TAs.split("&");
for(String s : arr){System.out.println(s);}

System.out.println(TAs.replaceAll("&[^&]+", "&Reese"));
```

Output: Reese&Reese&Reese&Reese

Java.util.regex

- More powerful operations are unlocked by the `Java.util.regex` package.
- There are two main classes in this package `Pattern` and `Matcher`
- `Pattern` objects represent regex patterns have a method to return a `Matcher` that allows the pattern to be used.

Java.util.regex.Pattern

- The Pattern object has no public constructor and instead has a compile method that returns a Pattern object.
- The Java specific version of regular expressions is documented on the Pattern api page, and is well worth reading.
- Note that you must escape your backslashes when coding literals

```
Pattern p1 = Pattern.compile("[a-z]{2,3}\\d+");
```

Java.util.regex.Matcher

- Matcher does the actual matching work, as the name suggests. Again there is no constructor, but instead a method inside Pattern that allows you to get a Matcher object set to match on a specific string.
- The principal operations of the Matcher are matches and find. matches returns true if the entire string matches the pattern, find returns true if any part of the string matches the pattern
- Matcher also has methods for operations such as replacement or group capturing.

Input checking

```java
public boolean isUpperLevelCS(String course){
    Pattern p = Pattern.compile("CS[456]\\d{3}");
    Matcher m = p.matcher(course);
    return m.matches();
}
```

This example isn't very powerful, what else can we do?

Capture example

Here is another example this time used to capture a match:

```java
Pattern p1 = Pattern.compile("([a-z]{2,3}\\d+)@.+");
Matcher m = p1.matcher("rpg55@cornell.edu");
m.matches();
System.out.println("First group: " + m.group(1));
```

This starts to get at the real utility of regex, but this rabbit hole goes much deeper than we have time for.

Exercise: Regex Crossword

https://regexcrossword.com/

Challenge: Command line parsing

- Regex can be used to parse command line or console inputs, capturing can be used to grab the different tags and access them
- Write a calculator using regex that takes commands of the form:
  `num num -f` *or* `num -f num` *or* `-f num num`
  Where `num` represents a positive decimal number (with or without a decimal point) and `-f` is the operation flag, one of `-+` `--` `-*` `-/` or `-%`.
- Parse the input and then print the result of the math. Implement it as a console (or GUI) application, because command line parses whitespace.