Credit: Randall Munroe xkcd.com

Lab 7: Regular Expressions

CS 2112 Fall 2024

October 21 / 23, 2024

Announcements

- Congratulations on finishing the prelim!
- A4 design doc feedback released
- A4 due Tuesday, Oct 29
- TA evaluations due Friday, Oct 25

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.
- Example: the language of `(a|b)c?` is {`a`, `b`, `ac`, `bc`}.

The simplest regex

- The simplest regular expression is just a string
- The regex `CS2112` matches only the string "`CS2112`" (that is, its language is the singleton set {`CS2112`}).

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

- `a*` matches any number of a's, including the empty string: its language is $\{\varepsilon, \texttt{a}, \texttt{aa}, \ldots\}$ where ε denotes the empty string.
- `(ab)*` matches any number of `ab`'s, including the empty string: its language is $\{\varepsilon, \texttt{ab}, \texttt{abab}, \ldots\}$
  - 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}` matches 000
- `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 01. This is the same as 0|1.
- [01]{2} matches 00, 11, 01, or 10

Character classes

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.
- `[A-Za-z]` matches any lowercase or uppercase letter
- Note that there are ASCII characters between `Z` and `a`, so `[A-z]` will also match characters like `[`, `^`, etc.

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` = `[0-9]`, `\D` = `[^0-9]`
- `\s` matches white space. Same as `[ \n\r\t\f]`. (`\f`: form feed, page break)
- `\w` matches "word" characters, basically not whitespace and punctuation. Same as `[a-zA-Z0-9_]`.
- `.` matches anything but a newline. This is super useful.
- There are a lot of these, fortunately the internet knows all of them!

Combinations

- Character classes and Quantifiers mix to give useful expressions
- `[a-z]*` matches any number of consecutive lowercase characters, or the empty string
- `[0-9]+` matches all numbers (that may or may not have leading 0s)
- `\d{3}` matches all three digit numbers (that may or may not have leading 0s)
- `.*` matches all lines

Groups

- Groups allow a section of the expression to be remembered for later
- Use parentheses for grouping
- \1 matches the substring captured by the first capture group, \2 matches the substring captured by the second capture group, etc.
- `(0|1)` matches `0` or `1`
- `(0|1):\1` matches `1:1` or `0:0` but not `0:1`
- `(\d):\1` matches `1:1` or `7:7` but not `2:3`
- We'll see later that groups can be captured and extracted to do something useful after matching.

Anchoring

- ^ (when not used in a character class) matches the beginning of a string
- $ matches the end of a string
- Anchors are used to constrain or "anchor" a regex to the beginning or end of a string
  - `^[A-Z]*$` matches entire strings that consist only of capital letters

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.
- For example, `A+` matches one or more `A`s, but `A\+` matches `A+`.

Examples

- Multiple combinations start to get at the real power of regex
- `[a-h][1-8]` matches all squares on a chess board
- `[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. (A netID has 2 or 3 lowercase letters followed by a string of 1 or more digits.)
- Write a regex to match all even numbers (positive, negative, or 0). Only numbers that do not start with a zero (unless the number is 0) should be matched.
- **Challenge:** Write a regex to match all strings of the form $\{0^n1^n : n \in \mathbb{N}\}$ (n zeros followed by n ones for all natural numbers n).
- Note: You can use https://regexr.com/ or https://regex101.com/ to test your regex on various strings.

Answers

- Write a regex to match Cornell netIDs. (A netID has 2 or 3 lowercase letters followed by a string of 1 or more digits.)
  - Answer: `[a-z]{2,3}\d+`

Answers

- Write a regex to match Cornell netIDs. (A netID has 2 or 3 lowercase letters followed by a string of 1 or more digits.)
  - Answer: `[a-z]{2,3}\d+`
- Write a regex to match all even numbers (positive, negative, or 0). Only numbers that do not start with a zero (unless the number is 0) should be matched.
  - Answer: `-?([1-9]\d*)?[02468]`
  - Don't want -0? Try `([1-9]\d*)?[02468]|-[2468]|-([1-9]\d*)[02468]`

Answers

- Write a regex to match Cornell netIDs. (A netID has 2 or 3 lowercase letters followed by a string of 1 or more digits.)
  - Answer: `[a-z]{2,3}\d+`
- Write a regex to match all even numbers (positive, negative, or 0). Only numbers that do not start with a zero (unless the number is 0) should be matched.
  - Answer: `-?([1-9]\d*)?[02468]`
  - Don't want -0? Try `([1-9]\d*)?[02468]|-[2468]|-([1-9]\d*)[02468]`
- Write a regex to match all strings of the form $\{0^n 1^n : n \in \mathbb{N}\}$.
  - **This is impossible due to the Pumping Lemma!!** Why? Take CS 2800 to find out!

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)".

```
String alumni = "Ted&Ashneel&Sam&Michael&Sam";

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

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

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, and they 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.
- Note that you must escape your backslashes when coding literals

```
Pattern p1 = Pattern.compile("[a-z]{2,3}\\d+");
Pattern p2 = Pattern.compile("\\\\");
```

Java.util.regex.Matcher

- The `matcher` method inside `Pattern` allows you to get a `Matcher` object set to match on a specific string.

```
Pattern p1 = Pattern.compile("[a-z]{2,3}\\d+");
Matcher m1 = p1.matcher("acm22");
```

Java.util.regex.Matcher

- 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
  - Anchors are useful: We can `find` `abc` in `abcd`, but we cannot `find` `abc$` in `abcd`.
- `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:

```
Pattern p1 = Pattern.compile("([a-z]{2,3}\\d+)@.+");
Matcher m = p1.matcher("acm22@cornell.edu");
m.matches();
System.out.println("NetID: " + 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.

An example in my own project!

```
String regex = "^((?<pawnQuiet>[a-h][1-8])|" +
        "(?<pawnCapture>[a-h]x[a-h][1-8])|" +
        "(?<regular>[KQRBN](?<startFile>[a-h]?)(?<startRank>[1-8]?)(?<capture>x?)[a-h][1-8])|" +
        "(?<prom>[a-h][18]=[QRBN])|" +
        "(?<promCapture>[a-h]x[a-h][18]=[QRBN])|" +
        "(?<castleK>O-O|0-0)|" +
        "(?<castleQ>O-O-O|0-0-0))(?<check>\\+?)(?<mate>#?)$";
```

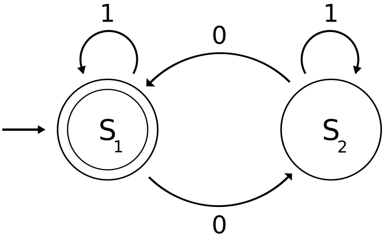
Credit: James's Chess Project

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 op num` **or** `op num num`
  where `num` represents a positive decimal number (with or without a decimal point) and `op` is the operation, one of `+`, `-`, `*`, `/` or %.
- Parse the input and then print the result of the math. Implement it as a console application, because command line parses whitespace.

Kleene's Theorem

- **Kleene's Theorem**: A language is regular if and only if it can be recognized by a finite automaton.
- What are finite automata? Why is this true? Take CS 2800 to find out!



Credit: Wikipedia

Exercise: Regex Crossword

https://regexcrossword.com/