The Regex Warm-Up

Prepared by Mark on February 14, 2025

Last time, we discussed Deterministic Finite Automata. One interesting application of these mathematical objects is found in computer science: Regular Expressions.
This is often abbreviated "regex," which is pronounced like "gif."

Regex is a language used to specify patterns in a string. You can think of it as a concise way to define a DFA, using text instead of a huge graph.
Often enough, a clever regex pattern can do the work of a few hundred lines of code.

Like the DFAs we've studied, a regex pattern *accepts* or *rejects* a string. However, we don't usually use this terminology with regex, and instead say that a string *matches* or *doesn't match* a pattern.

Regex strings consist of characters, quantifiers, sets, and groups.

Quantifiers
Quantifiers specify how many of a character to match.
There are four of these: `+`, `*`, `?`, and `{ }`.

`+` means "match one or more of the preceding token"
`*` means "match zero or more of the preceding token"
For example, the pattern `ca+t` will match the following strings:

- `cat`
- `caat`
- `caaaaaaaat`

`ca+t` will **not** match the string `ct`.
The pattern `ca*t` will match all the strings above, including `ct`.

`?` means "match one or none of the preceding token"
The pattern `linea?r` will match only `linear` and `liner`.

Brackets `{min, max}` are the most flexible quantifier.
They specify exactly how many tokens to match:
`ab{2}a` will match only `abba`.
`ab{1,3}a` will match only `aba`, `abba`, and `abbba`.
`ab{2,}a` will match any `ab...ba` with at least two `b`s.

Problem 1:
Write the patterns `a*` and `a+` using only `{ }`.

Problem 2:
Draw a DFA equivalent to the regex pattern `01*0`.

Characters, Sets, and Groups
In the previous section, we saw how we can specify characters literally:
`a+` means "one or more `a` characters"
There are, of course, other ways we can specify characters.

The first such way is the *set*, denoted `[ ]`. A set can pretend to be any character inside it.
For example, `m[aoy]th` will match `math`, `moth`, or `myth`.
`a[01]+b` will match `a0b`, `a111b`, `a1100110b`, and any other similar string.

We can negate a set with a `^`.
`[^abc]` will match any single character except `a`, `b`, or `c`, including symbols and spaces.

If we want to keep characters together, we can use the *group*, denoted `( )`.
Groups work exactly as you'd expect, representing an atomic[1] group of characters.
`a(01)+b` will match `a01b` and `a010101b`, but will **not** match `a0b`, `a1b`, or `a1100110b`.

Problem 3:
You are now familiar with most of the tools regex has to offer.
Write patterns that match the following strings:

- An ISO-8601 date, like `2022-10-29`.
  Hint: Invalid dates like `2022-13-29` should also be matched.
- An email address.
  Hint: Don't forget about subdomains, like `math.ucla.edu`.
- A UCLA room number, like `MS 5118` or `Kinsey 1220B`.
- Any ISBN-10 of the form `0-316-00395-7`.
  Hint: Remember that the check digit may be an `X`. Dashes are optional.
- A word of even length.
  Hint: The set `[A-z]` contains every english letter, capitalized and lowercase.
  `[a-z]` will only match lowercase letters.
- A word with exactly 3 vowels.
  Hint: The special token `\w` will match any word character.
  It is equivalent to `[A-z0-9_]`. `_` represents a literal underscore.
- A word that has even length and exactly 3 vowels.
- A sentence that does not start with a capital letter.

Problem 4:
If you'd like to know more, check out `https://regexr.com`. It offers an interactive regex prompt, as well as a cheatsheet that explains every other regex token there is.
You can find a nice set of challenges at `https://alf.nu/RegexGolf`.

[1] In other words, "unbreakable"