grep

Project Details

In this project, you will implement grep with NFAs and Boyer-Moore string matching.

Project Restrictions

You may implement this project in your choice of Kotlin or Rust.

Deliverables

- NFA-based `grep`, with Boyer-Moore literal search optimization

Task 1: Writing a CFG For Regular Expressions

We have provided an Earley parsing implemention in both Rust and Java for you to use in your project. Understanding Earley parsing is not an explicit goal of this project; you are not required to read and understand the implementation. Your task is to create an instance of the CFG class (Kotlin) or struct (Rust) that specifies a context-free grammar for (our flavor of) regular expressions:

- If `A` and `B` are regular expressions, then `A|B` is a regular expression (union).
- If `A` and `B` are regular expressions, then `AB` is a regular expression (concatenation).
- If `A` is a regular expressions, then `A*` is a regular expression (Kleene star).
- If `A` is a regular expression, then `(A)` is a regular expression.
- If `A` is a regular expression, then `A+` is a regular expression.
- If `A` is a regular expression, then `A?` is a regular expression.
- Tab (0x09) and all characters between space (0x20) and tilde (0x7E), except `{ |, *, (, ), ., +, ?, \}` are regular expressions (literals).
- The special characters `{ |, *, (, ), ., +, ?, \}` when escaped with `\`, e.g. `\+` are regular expressions (literals).
- `.` represents any literal.
- `\s`, `\d`, `\w`, are regular expressions that represent, respectively, any letter literal, any digit literal, and any whitespace literal.
- `\S`, `\W`, `\D` are regular expressions that are negations of the above.

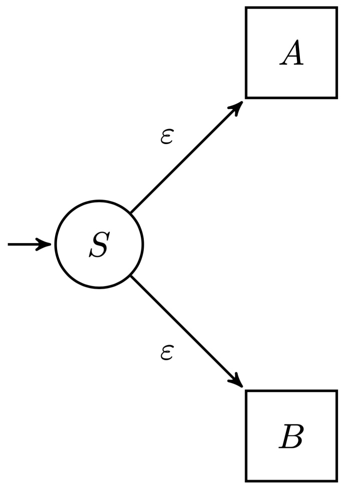
Make sure your grammar is not ambiguous. The precedence order, from highest to lowest is: parentheses, star / plus / question mark, concatenation, union. See here for notes on designing a grammar.

Task 2: NFA

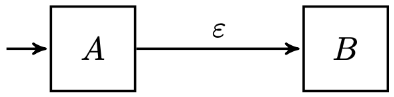
Our grep implementations will be backed by an NFA to test whether a regular expression accepts an input string. Your task is to design an NFA class that supports converting a parsed regular expression to an equivalent NFA.

We can construct an NFA with each fundamental operator by building on subexpression NFAs. If A and B are regular expressions, then

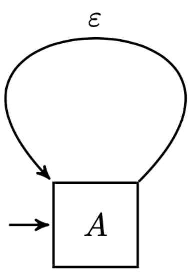
`A|B` (union)



`AB` (concatenation)



`A*` (Kleene star)



Note that we can compose more complex operations (e.g. +) from these fundamental ones. This construction is known as Thompsons construction, described further in this article.

However, we will require one key difference in our NFAs: the construction adds ϵ-transitions, which complicate checking whether a string is accepted by an NFA. Instead of storing ϵ-transitions, you should compute an equivalent NFA with no ϵ-transitions, or its "ϵ-closure".

At a high level, we can compute the "ϵ-closure" by replacing the ϵ-transitions in the NFA with "everywhere they go".

You will likely find the ϵ-closure notes to be helpful.

Task 3: Grep 1.0

Your grep program will take two arguments: the regular expression and the name of a file to search, and output all non-overlapping matches. You should match *greedily* - that is, make as long of a match as possible before moving on. Dont consider matches that continue over multiple lines.

When you print a match, use the format `LINE:MATCH`, where

- `LINE` is the line number that the matched occured on,
- `MATCH` is the actual matched text.

This is `grep`s output format when the `only-matching` and `line-number` flags are enabled.

Task 4: Prefix Extraction

We have a working version of `grep` now... but lets try to optimize it! Regular expressions often have *literal prefixes*. For example...

- `foo(d|l)`
- `Caltech|California`
- `(na)+ batman`

Each of these regular expressions starts with a prefix that is *fixed sequence of literals*. We can feed these prefixes to string-searching algorithms to find the starts of possible matches, reducing the number of places we need to start the NFA.

Implement prefix extraction for your NFA by extracting the longest possible prefix for all accepted strings and creating a modified NFA that matches strings without the prefix.

Task 5: Boyer-Moore and Grep 2.0

Boyer-Moore is an efficient string searching algorithm for a single pattern in a text. Because we want to find all matches, we will also implement the Galil rule to improve our efficiency when searching for multiple matches in a text.

Finally, integrate prefix extraction and Boyer-Moore into your `grep` implementation.