CS420 Project

Regex in Grace

Jordan Cantrell

Architecture

I initially attempted to implement a regular expression engine similar to the one described in "Regular Expression Matching can be Simple and Fast", which takes in a pattern string, inserts a `.` character to explicitly act as the concatenation operator, converts the pattern to postfix form, and constructs an NFA from that postfix form.

But this implementation was messy. I had to have a separate method to insert explicit concatenation, so the engine would recognize `ab` as matching `a` followed by `b`. Furthermore, this implementation relies on knowledge of operator precedence to correctly rearrange into postfix form, and the conversion to postfix form would eliminate information about parentheses, which I would need to implement submatch extraction at a later stage.

So, I had to write a lexer, to read the pattern string and produce a string of tokens for operators, matching characters, and so on. In the course of writing this lexer, I realized it would be easier to have a recursive descent parser, which handles the raw input pattern string and builds the NFA directly. This approach seemed to point to an path to implement submatch extraction, as the appropriate code could just be added to the code for the `<group>` production.

The grammar I use is outlined as follows:

⟨*alternation*⟩ ::= ⟨*concatenation*⟩ | ⟨*concatenation*⟩ '`|`' ⟨*alternation*⟩

⟨*concatenation*⟩ ::= ⟨*basic*⟩ | ⟨*basic*⟩ ⟨*concatenation*⟩

⟨*basic*⟩ ::= ⟨*elem*⟩ | ⟨*elem*⟩ '`*`' | ⟨*elem*⟩ '`+`'

⟨*elem*⟩ ::= ⟨*group*⟩ | ⟨*character*⟩ | ⟨*set*⟩

⟨*group*⟩ ::= '`(`' ⟨*alternation*⟩ '`)`'

⟨*character*⟩ := ⟨*non-meta*⟩ | '\' ⟨*meta*⟩

⟨*meta*⟩ := ⟨*operator*⟩ | ⟨*namedClass*⟩

⟨*non-meta*⟩ := ⟨*digit*⟩ | ⟨*upper*⟩ | ⟨*lower*⟩ | ⟨*symbol*⟩

⟨*operator*⟩ := '`*`' | '`|`' | '`+`' | '`?`'

Where `<symbol>` is left out for brevity, and matches all the non-alphanumeric characters that aren't parentheses, brackets, or operators used in regular expressions.

To implement a parser for this grammar, I first have an overall `pattern` class, which represents a regular expression pattern. As such, its publicly available methods are `matches(text)` and `asDebugString`. Within the `pattern`, we have a `RegexParser`, which uses a `grammar` to build an `NFAFragment` corresponding to the given regular expression, and connects the fragments' dangling arrows to the matching state when finished parsing. The `grammar` is a recursive-descent parser for the grammar given above. Each rule builds an `NFAFragment` corresponding to that grammar rule.

Finally, when the `NFA` is complete, we convert it into a `DFA` which corresponds to the given pattern (we really just simulate the `DFA` by considering a `DFA` state to correspond to a whole set of `NFA` states, since this is simple to do). Thus, when we want to see if a given text matches, we can run the `DFA` over that text and ask if the `DFA` is in a matching state.

Challenges

I ran into a lot of challenges during this project. As I mentioned, I originally implemented the shunting-yard algorithm to convert the given pattern to postfix form, then built the nfa from there, but that (1) would make it difficult to implement group sub-match extraction, and required `operator` objects to know about their prececence, which in turn required some parsing, which would have been more work than just building the nfa during the parsing phase.

Unfortunately, I have found that there is a bug in the current implementation. Specifically, the `?` operator. As it stands, given a pattern of length `n`, we should end up with an nfa with roughly `n` states. This (should) make for very efficient, fast run-times, as the DFA is only ever in one DFA-state (at most `n` NFA states) at any given time, and it should be very fast to compute the next DFA state.

However, for pathological patterns such as "$(a?)^n a^n$" matched against the text "a^n", we begin to see very, very slow run-times for $n > 8$. Since there are only $2n$ states for such a pattern, I would expect this to run quickly, as the DFA is only ever in at most $2n$ nfa states. My theory is that (1) I am accidently producing many, many more states than needed by accident, hogging memory or (2) computing the next DFA state is not as efficient as I though.

Either way, it is my regret that I have run out of time to find the root of the problem.

Achievements

While I am embarrased that I didn't even manage to implement the `?` properly, and thus never even got to character classes like `[abc]` or `[^abc]`, I am still pretty proud that I even have a working product at all. I have implemented `*`, `|`, and concatenation, which is technically enough to build equivalent regular expressions for any other syntax. Even if it may be inconvenient.