NFA Construction and Simulation in `thegrep`

PS06 - COMP590 - Spring 2019

Overview

In the second part of `thegrep` you will implement Thompson's Construction for converting a Regular Expression to a Nondeterministic Finite Automata (NFA). Then, once constructed, you will simulate the NFA to determine whether lines of input are accepted by it or not. When you've completed this problem set you will have a working, mini implementation of `egrep`.

Getting Started

Skeleton code for an NFA representation is provided and based heavily on the code explored in lecture 21. It uses arena-style allocation as discussed in lecture to safely construct graphs with cycles. You can find the starter skeleton code here and should go ahead and add this file to your `src` directory in a file named `nfa.rs`:

https://raw.githubusercontent.com/comp590-19s/starter-ps06-thegrep-nfa/master/nfa.rs

Additionally, for diagnostic purposes, we are providing you with some helper functions to generate dumps of your NFA for debugging. As shown in lecture, the `nfa_dot` function produces a DOT graph representation that can be used as an input for the `dot` utility program to produce a visualization of your NFA. You should create a directory in your `src` directory named `nfa` and place the following `helpers.rs` file in it:

https://raw.githubusercontent.com/comp590-19s/starter-ps06-thegrep-nfa/master/nfa/helpers.rs

You will want to include these files in your `main.rs` with the following snippet:

```rust
pub mod nfa;
use self::nfa::NFA;
use self::nfa::helpers::nfa_dot;
```

Depending on how you designed your `Parser` and `Tokenizer`, you may need to update the starter code in `nfa.rs` to correctly make use of *your* implementation from the previous problem set. The `NFA`'s `from` method, specifically, expects an `AST` representing your regular expression's parsed structure.

Part 1 - Wiring up a DOT Representation

Your first challenge is to "wire in" the provided support code and reach a point of being able to produce a visualization of your generated NFA. We're providing enough code to produce an NFA, and DOT representation, of the simplest regular expression possible: '.' - a pattern that matches any single character.

To begin, add an additional command-line flag to `thegrep`'s options that will cause it to produce a DOT representation of the NFA and exit. Its short flag is `d` and long form is `dot`. If `thegrep` is run with this flag, it should do roughly the following:

```
let nfa = NFA::from(&pattern).unwrap();
println!("{}", nfa_dot(&nfa));
std::process::exit(0);
```

Notice an `NFA` is being constructed `from` a string `pattern`. You will not have a variable named `pattern`, but will rather need to reference the pattern provided as a command-line option. Then, the helper function `nfa_dot` is being called (you imported it in the Getting Starting steps above) to produce the DOT representation of your regular expression. Finally, the process exits forcefully such that no further code will run.

To know that you're ready to begin work on the NFA construction, try running with the following command-line options which produce a DOT representation of a pattern that matches *any* single character:

```
$ cargo run -- -d '.'
digraph nfa {
 node [shape = circle];
 start [shape="none"]
 start -> 1
 1 -> 2 [label="ANY"]
 2 [shape="doublecircle"]
}
```

Once you are successfully producing the output above you are ready to continue forward. Do not attempt further implementation until the above is working.

To produce a visualization of your `NFA`, pipe the command to `dot` like so:

```
$ cargo run -- -d '.' | dot -Tsvg -o /vagrant/nfa.svg
```

The command above produces a scalable vector graphics file in your virtual machine's folder on your host operating system. You should open the file `nfa.svg` in a browser and see the start state is 1 and it transitions to an accepting state 2 with a transition edge labelled `ANY`.

Part 2 - Constructing the NFA

Your next goal is to construct an NFA from a parsed abstract syntax tree (AST) using an adaptation of Thompson's Algorithm (click this link to view). You will do so by adding and joining together `Match` and `Split` states as you visit the AST recursively. You should review lecture 23 to conceptually understand the construction of an AST from an NFA.

Before you begin writing code you should refamiliarize yourself with the `NFA` starter code. Specifically, you should read and think through the methods `from`, `gen_fragment`, `add`, `join`, and `join_fragment`.

Your work will be in `gen_fragment` and any helper functions you write to support it (which are recommended). A working implementation to generate a `Fragment` for an `AST::AnyChar` is provided as an example. You will need to ensure `gen_fragment` works correctly for all other variants of your `Parser`'s `AST` enum. We suggest you add support in this order:

1. `AST::Char`
2. `AST::Catenation`
3. `AST::Alternation`
4. `AST::Closure`

As you add support for each type of AST node, you should test by generating an SVG of your resulting NFA. For example, after adding `AST::Char` support, try:

```
$ cargo run -- -d 'a' | dot -Tsvg -o /vagrant/nfa.svg
```

And after adding `AST::Catenation` support, try:

```
$ cargo run -- -d 'ab' | dot -Tsvg -o /vagrant/nfa.svg
$ cargo run -- -d 'a.b.c' | dot -Tsvg -o /vagrant/nfa.svg
```

And so on. You do not need to write unit tests for this part of the problem set and will not be penalized for failing to do so. You are encouraged to try, however. Naively unit testing this stage of the problem set will be cumbersome. *Why?* If you put in the work to tidily unit test this part of the problem set please show off your code to Kris for 5% extra credit.

After properly constructing an NFA made of any combination of AST nodes, deeply consider the following questions which you could imagine being on an exam:

In Thompson's paper a "pushdown stack" is employed to construct the NFA from a postfix representation of a regular expression. You did not need to explicitly use a stack (and should not have) to construct your NFA. Why not? What is the relationship between postfix notation and an AST?

Simulating the AST

Instructions for simulating your AST for testing inputs are forthcoming.