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:

```
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 string in the form of a regular expression.

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 NFA from an AST.

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?

Part 3 - Simulating the NFA

The purpose of `thegrep` is to filter streaming lines of text input to only lines that match a regular expression pattern. In this part of the problem set you will:

1. Implement the NFA's `accepts` method
2. Add additional options to your command-line interface for file(s)
3. Read those files, or read from `stdin`, line-by-line and test whether the NFA `accepts` the line. If it does, print the line out, otherwise continue to the next line.

Implementing the `accepts` method

Once you are convinced of the correctness of your NFAs constructed in Part 2, based on the generated transition diagrams, you can begin work implementing the `NFA` struct's `accepts` method. When an `NFA` is given an input string, its `accepts` method will return `true` if the input string is *accepted* by the NFA, false otherwise.

To determine whether a string is accepted by the NFA you must implement an algorithm to simulate the NFA's state transitions when processing the input string character-by-character. You are encouraged to solve this problem among yourselves based on your understanding of how to test acceptability given an epsilon transition diagram and an input string (discussed in lecture 20).

You *must* implement test cases for the `accepts` method. Starting with a simple test case before beginning will help you with its implementation. Starting with the simplest cases and adding more complex cases as you progress is encouraged.

If you're feeling particularly stumped, here are two helpful links with more details on NFAs and NFA simulations:

1. Wikipedia's page on non-deterministic finite automata
2. Russ Cox's page on implementing regular expressions in C

Once you have a working algorithm, you should ensure it is cleanly implemented and well commented. One benefit of having test cases here is you can commit a working version of your code, refactor its implementation a bit, pass all tests, and repeat. Should you find yourself in a place where a refactoring fails to work out as you hoped, you can easily revert to the working commit.

Filtering Streaming Input

Once your `accepts` method is operational, add a list of files to your command-line options (as shown in lecture 18). Just like the real `egrep`, if one or more files is passed as the final arguments to the command their contents will be streamed and matched against. Otherwise, standard input will be read from. Refer to the implementation of `cat` in lecture 18, or Chapter 18: Input and Output in the *Programming Rust* textbook for additional information.

Notice in `egrep` a pattern like `toma` accepts the string `automata`. You will need to consider how to achieve the same in your program. Perhaps there's a pattern you could automatically

insert before and after the user's pattern, before constructing the NFA, that will accept anything?

As each line is read from either source, it should be tested against the command's pattern argument's NFA's accepts method. If accepted, print the line, otherwise ignore it. You can test your implementation against the american english dictionary file `~/dict` file we established in lecture 5's first demo of `egrep`. Try testing with various patterns, such as:

```
$ cargo run -- 'aut...a' ~/dict
Chautauqua
Chautauqua's
automata
beautification
beautification's
```

You will notice a performance difference between your program and `egrep`. To dramatically improve the performance of your implementation, try running your code for compiled with release optimizations:

```
$ cargo run --release -- 'auto...a' ~/dict
```

If you want to see how much time it takes for your program to run in debug mode (default), versus release mode, versus `egrep`, try prefixing each command with the `time` command:

```
$ cargo ./target/debug/thegrep 'auto...a' ~/dict
$ cargo ./target/release/thegrep 'auto...a' ~/dict
$ egrep 'auto...a' ~/dict
```

Don't be dismayed if your release is an order of magnitude or two slower than `egrep`. Remember: it's a program that's been written, rewritten, and optimized for nearly 50 years. Time permitting, you are encouraged to try optimizing your implementation to improve its performance.

Grading Rubric Breakdown

Autograding will only test Part 3's implementation.

1. 10pts - Filters file(s) argument(s)
2. 10pts - Filters stdin in absence of file(s)
3. 20pts - Successfully accepts concatenations
4. 20pts - Successfully accepts alternations
5. 20pts - Successfully accepts closures

Hand-graded Points

1. 10 points - Clean implementation of `accepts` method with clear documentation
2. 10 points - Unit tests for `accepts` method