CS 301 Spring 2016
Lab February 5

NFA Simulator

Plan

Today we review the last couple exercises from Regular Expressions and Finite Automata, get tools configured, and build an efficient pattern matcher based on our theory of regular expressions and finite automata. **If you do not finish this during lab today, please complete it before your next meeting.**

NFA Simulator

Exercise 1. Review your answers to the final exercise on Regular Expressions and Finite Automata. Make sure everyone in your team is on the same page.

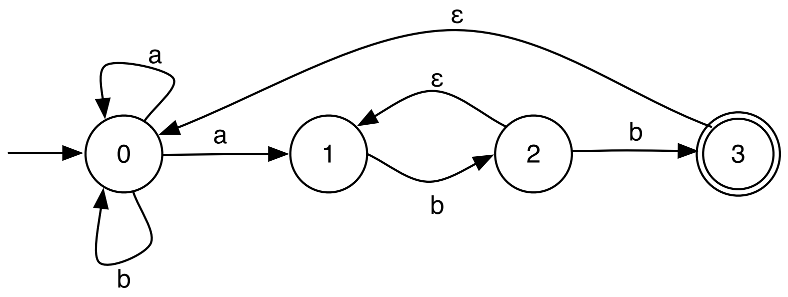
Starter Code. There is starter code for this lab. Make sure you have done the extra Mercurial setup on the Tools sheet. Then, to get starter code, run `hg pull starters` and follow `hg`'s suggestions at the end of its ouput. (Either merge and commit, or update.) In Scala IDE, import the project: File > New > Scala Project... Name it `nfasim`. Click Finish to automatically set up the project.

Exercise 2. Implement Algorithm 3.22 from Dragon. The input to your program is the transition table for an NFA over the alphabet $\{a, b\}$. The format of the input is:

- a number n indicating the number of states (labeled 0, 1, ..., $n-1$);
- a number $q \in 0..n-1$ indicating the only accepting state of the NFA. (Assume 0 is the start state).
- the transitions for each state on `a`, `b`, and ϵ.

As an example, this table and transition diagram describe the same NFA:

```
4
3
(0,1) (0)    ()
()    (2)    ()
()    (3)    (1)
()    ()     (0)
```



The Scala starter code helps parse this input format. Several sample input files are also included. Your program should print output for each string it tests, indicating whether it is accepted or rejected. The name of the file containing the input NFA and the test strings should be passed to your program as command-line arguments. In a shell working in the `nfasim` project directory, run something like:

```
scala -cp bin nfasim.NFASimulator inputs/t0.nfa aabb abab abbbababb
```

Alternatively, you may run the simulator directly from the Scala IDE. Click the Run button or menu and Run As... Scala Application. Choose the main class `nfasim.NFASimulator` and then use the Arguments tab to give the arguments `inputs/t0.nfa aabb abab abbbababb`.

Clarity and correctness are far more important than raw efficiency in this exercise. The Dragon book gives the high-level algorithm and a description of a fairly low-level, efficient implementation. Follow the high-level algorithm, but do not worry about finding the most efficient that approach. Just use standard `scala.collection` (or similar) data structures to implement a reasonable, straight-forward solution. For example, my implementation corresponds closely to the high-level algorithms shown in Dragon. It uses mutable Arrays and Stacks and immutable Maps and Sets.

Exercise 3. The `java.util.regex` package contains a `Pattern` class that uses the first algorithm. You can test a string against a regular expression using this class as follows:

```
java.util.regex.Pattern.matches("a?a?a?aaa", "aaaa")
```

Compare the performance of your program to a program that uses this method. For convenience, the data files `inputs/perf/e`n`.nfa` contain the NFAs for $(\texttt{a}?)^n\texttt{a}^n$. I have provided a basic performance test in `PerfTest.scala` that tests the style of strings from Exercise 10 on Regular Expressions and Finite Automata. Feel free to modify it and replace it or explore further.

Exercise 4. Earlier, we explored converting an NFA to a DFA before simulation. Why is this preferable to the direct NFA simulation performed here? What are the downsides to DFA conversion? Do you think the potential issues impact lexical analysis for programming languages substantially?