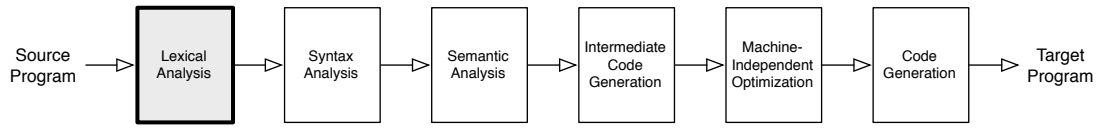
HW 1: Regular Expressions and Automata

CSCI 434T
Fall, 2011

Overview



The goal of the first week is to cover two topics:

- *A general introduction to compilation.* As part of this introduction, you will read about the overall architecture of a compiler, the separate phases of compilation, and how programming language design and computer architecture affects compiler design. In addition, you will read a brief review of key programming language ideas.
- *Lexical Analysis.* The first compilation stage is lexical analysis — the task of breaking source code text into a language's most basic lexemes. You will read about the basic theory behind lexical analysis (regular expressions, transition diagrams, finite automata) and gain an appreciation for how theory guides the design and implementation of a lexical analyzer.

Readings

- Dragon, Chapter 1. *Fairly light reading — mostly background.*
- Dragon, Chapter 3.3, 3.5–3.7
- "Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)", Russ Cox, January 2007. *Don't worry about the C code implementations — pay more attention to the basic ideas and commentary.*

Exercises

1. Dragon 1.1.2
2. Dragon 1.1.4
3. Dragon 1.6.1
4. Dragon 3.3.2
5. Dragon 3.3.9
6. Write a regular expression for http and ftp URLs. An URL consists of four parts: the protocol (`http://` or `ftp://`), the DNS name or the IP address of a host, an optional port number, and an optional pathname for a file. For simplicity, we assume that:
   - A DNS name is a list of non-empty alphabetical strings separated by periods.
   - An IP address consists of four non-negative integers of at most three digits each, separated by periods.
   - A port number is a positive integer following a colon. (e.g. :8080)

- The pathname part is a unix-style absolute pathname. The allowed symbols are letters, digits, period and slash. A sequence of two consecutive slashes `//` is forbidden, i.e. no empty directory name.
- A URL may end with a slash as long as it does not create the sequence `//`.

7. Write the DFAs for each of the following:

   (a) Binary numbers that contain the substring 011.

   (b) Binary numbers that are multiples of 3 and have no consecutive 1's. (You solution can accept or reject the empty string – either is fine.)

8. A comment in the C language begins with the two-character sequence `/*`, followed by the body of the comment, and then the sequence `*/`. The body of the comment may not contain the sequence `*/`, although it may contain the sequence `/*`, or the characters `*` and `/`. We use the notation `(E)*` for the Kleene closure of `E`; all other occurrences of `*` refer to the character itself, not to the Kleene operator.

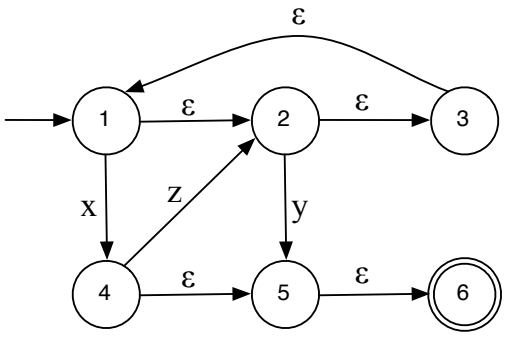
   (a) Show that the following regular expression does not correctly describe C comments:

   ```
   /* (/)* ([^*/] | [^*]/ | *[^/])* (*)* */
   ```

   (b) Draw the DFA that accepts C comments and then use it to write the regular expression that correctly describes C comments.

   (c) Write the DFA and the regular expression for C comments extended such that the sequence `*/` is permitted, as long as it appears inside quotes. For example, `/* ab "cd */" ef */` would be a valid comment.

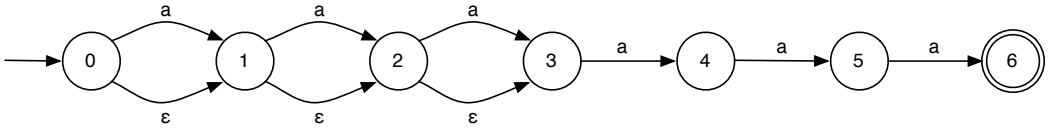
9. Convert the following NFA to a DFA. For each DFA state, indicate the set of NFA states to which it corresponds. Make sure you show the initial state and the final states in the constructed DFA.



10. Russ Cox describes how inefficient some regular expression pattern matchers can be. You will now build a pattern matcher that can outperform Java's regular expression package. More specifically, you will write a program that, given a description of an NFA, simulates its behavior on an input string.

    (a) To motivate the design and to see how a little bit of theory can dramatically improve software design, we start by comparing the two simulation algorithms in section "Regular Expression Search Algorithms" section of Cox's article.

    Here is the NFA corresponding to "`a?a?a?aaa`":



Enumerate all paths taken through the NFA using the backtracking search algorithm on input `aaab`. How many are there? Given the NFA for $(\texttt{a?})^n\texttt{a}^n$, how many paths may need to be explored to test an input string of length $n+1$? (A Big-O bound is fine, and a one or two sentence explanation is sufficient.)

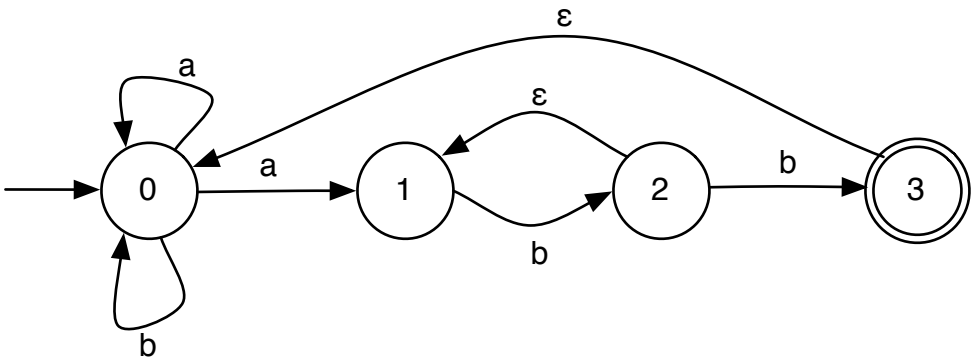
(b) Now, show the steps performed by Algorithm 3.22 in Dragon. (This is a more precise and succinct description of the second algorithm Cox describes). It is sufficient to show the states in S each time line (3) is executed. You will find it useful to compute the ϵ-closure for each state in the NFA.

(c) Implement Algorithm 3.22. You may use any language you like, provided that your program compiles and runs on the lab machines. I have put a few lines of helper code to help you parse the input data on the web site, should you choose to use Java.

The input to your program will be 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 (which will be 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, the table for the following NFA



would be

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

Several sample input files are also on the web page. Your program should output "yes" or "no" for each string it tests. The name of the file containing the input NFA and the test strings should be passed to your program as command-line arguments, as in:

```
java nfa.NFASimulator ex1.nfa  aabb abab cow
```

NOTE: I am far more interested in clarity and correctness than efficiency. The Dragon book describes a fairly low-level, efficient implementation, but you are not expected to follow that approach. Just use standard `java.util` (or similar) data structures — Stacks, Vectors, Sets, arrays, etc. — to implement a reasonable, straight-forward solution.

(d) 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 `part-d/en.nfa` contain the NFAs for $(\texttt{a}?)^n\texttt{a}^n$.

(e) The previous problem explored converting an NFA to a DFA before simulation. Why is this preferable? What are the downsides to DFA conversion? Do you think the potential issues impact lexical analysis for programming languages substantially?