Assignment 8: Regular Languages

Due: 11am, Wednesday, November 14, 2012

- Emails about this assignment should be directed to `cs81help@cs.hmc.edu`.
- Your diagrams may be hand-drawn or computer-drawn via a program of your choice. If you use special-purpose tools such as JAPE to draw your automata, do *not* use it to solve the problems, e.g., converting between NFAs, DFAs, and regular expressions. (After all, you won't be able to use JAPE during the final exam!)
- All work submitted must be your own.

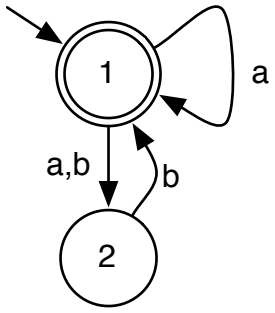
1 Regular Languages

1. Review Chapters 1–3 of Rich (*Automata, Computability, and Complexity*), paying particular attention to all the examples (in the shaded boxes). Then do the same for Chapter 5 up through and including section 5.4.2.

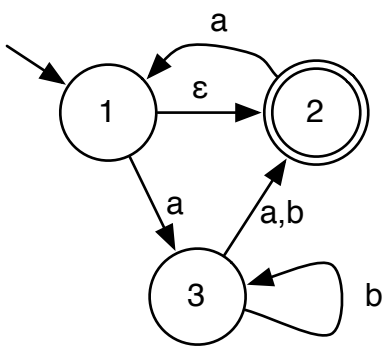
   Come up with two questions related to the reading. These may refer to points where the book is confusing, or simply to some question or conjecture that occurs to you while doing the reading.

2. Turn the following NFAs into DFAs (DFSMs) using the subset construction. It should be clear from your drawing of a DFA how the subset construction was applied.

   (a)



(b)



3. Do Exercise 2(a,c–e,j,o–r,t) on pages 151–152 of Rich.

4. For two languages L and M, let the *shuffle* of L and M be the set

$$\{\, w \mid w = a_1 b_1 \cdots a_k b_k, \text{where } a_1 \cdots a_k \in L \text{ and } b_1 \cdots b_k \in B \,\}$$

   Here $a_1 \cdots a_k$ is a *single* word in A, and the a_i's are pieces of that word; any of the a_i's might be the empty string. The same ideas hold for the b_i's.

   So for example, strings in the shuffle of a^+ and b^+ include abab, aabaabb, aabbaa, ba, babbabbaaa, and so on. The intuition is to take a single string from A and a single string from B and do the equivalent of one "riffle shuffle" of two piles of cards.)

   Prove clearly and convincingly that if L and M are regular then the shuffle of L and M is also regular. (Hint: if L and M are regular, then there are NFAs and DFAs that accept each. Show how these can be used to construct an automaton that can say whether the input is a valid combination of a string in L and a string in M.)

5. A *lexer* (also known as a *tokenizer*) is a program that takes a sequence of characters and splits it up into a sequence of words, or "tokens." Compilers typically do this as a prepass before parsing programs. For example "`if (count == 42) ++n;`" might divide into `if`, `(`, `count`, `==`, `42`, `)`, `++`, `n`, and `;`.

   Regular expressions are a convenient way to describe tokens (e.g., C integer constants) because they are unambiguous and compact. Further, they are easy for a computer to understand: *lexer generators* such as `lex` or `flex` can turn regular expressions into program code for dividing characters into tokens.

In general, there may be many ways to divide the input up into tokens. For example, we might see the input `ifoundit = 1` as starting with a single token `ifoundit` (a variable name) or as starting with the keyword `if` followed immediately by the variable `oundit`. Most commonly, lexers are implemented to be *greedy*: given a choice, they match the longest token. (Hence, `ifoundit` is preferred over `if` as the first token.)

Lexers commonly skip over whitespace and comments. A "traditional" comment in C starts with with the characters `/*` and runs until the next occurrence of `*/`. Comments do not nest; the string `/* a /* b */ c */` consists of a C comment (`/* a /* b */`) followed by non-comment (code) charaters (`c */`).

Your task is to construct a regular expression for these traditional C comments, one suitable for use in a lexer generator.[1]

(a) When I have given this problem in the past, people immediately suggest

`/* ( . | \n )* */`

Explain why this regular expression would not make a lexer skip comments correctly. (Big hint: greed.)

(b) Once the problem with the previous expression is noted, most folks decide that comments should contain only characters that are not stars, plus stars that are not immediately followed by a slash. This leads to the following regular expression:

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

Find a legal 5-character C comment that the regular expression fails to match, and a 7-character ill-formed (non–valid-comment) string that the regular expression erroneously matches.

(c) Draw a finite-state machine that accepts all and only valid traditional C comments.

(d) Provide a correct regular expression that matches all and only valid C traditional comments, by converting your state machine into a regular expression. You may use either method from class, but show your work.

Be sure it's completely clear where you're using * the character and when you're using * the regular expression notation.

[1] Some regular expression implementations supply both "greedy" and "nongreedy" matching operators, e.g., a non-greedy version of the star operator that matches as few times as possible. These makes this problem trivial; our goal here is to find a correct regular expression just using the "classic" operators.