Assignment 9: Regular Languages

Due: 11am, Wednesday, November 9, 2011

- 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. (After all, you won't be able to use JAPE during the final exam!) All work submitted must be your own.

1 Decision Problems

One of the oldest "NP-Complete" problems is 3SAT; if this could be solved efficiently, then thousands of other interesting and useful problems could be solved efficiently as well.

3SAT is officially a decision problem. The input is a logical proposition in conjunctive normal form, where each clause contains up to three literals. That is, the input is an (arbitrarily long, but finite) "and" of clauses, where each clause is the "or" of one to three literals, e.g.,

$$p_1 \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_1 \vee p_3) \wedge (\neg p_3 \vee \neg p_2 \vee \neg p_{99})$$

has the right form. The decision problem is whether the input is satisfiable, i.e., whether there is an one assignment of truth values to propositional variables that makes the input true. (For this example, the answer is yes.)

1. Viewed in terms of languages, 3SAT would be a set of strings, containing exactly the satisfiable propositions with the correct form. The question would then be whether certain strings are in this set or not. Give a suitable finite alphabet for this problem, and briefly justify your answer. (Don't overthink it.)

2. The decision problem only tells us whether a satisfying assignment exists or not. Show that if we had a "decision procedure" that solves the decision problem, we could use it to find a satisfying assignment. (Hint: if the input involves n propositional variables, then you could find a satisfying assignment while using the decision procedure up to n times, assuming the input is satisfiable.)

2 Regular Languages

1. Review Chapters 1–3 of Rich, 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. Draw NFAs (NDFSMs) for the following languages, with the specified number of states. You may assume the alphabet is always $\{0, 1\}$.

   (a) The language $\{ w \mid w \text{ ends in } 00 \}$, 3 states

   (b) The language $\{ w \mid w \text{ contains the substring } 0101 \}$, five states

   (c) The language $\{ w \mid w \text{ contains an even number of 0s or exactly two 1s} \}$, six states

   (d) The language $\{0\}$, two states

   (e) The language $0^*1^*0^+$, three states

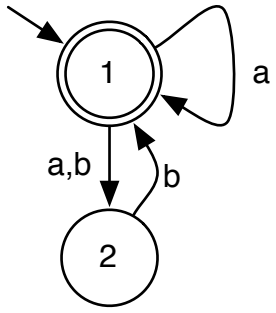
   (f) The language $1^*(001^+)^*$, three states

   (g) The language $\{\epsilon\}$, one state

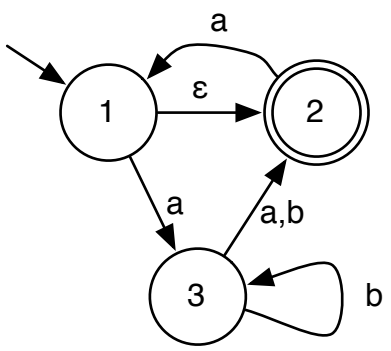
   (h) The language 0^*, one state

3. 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)



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

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 prefer to produce 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 `*/`. Nested comments are forbidden.

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

[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.

(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: greedy[2])

(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.

[2] 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; the goal of this exercise is a correct regular expression just using the "classic" operators.