Assignment 7: Regular Languages

Due: 1:15pm, Tuesday, October 26

- Emails about this assignment should be directed to `cs81help@cs.hmc.edu`.
- Grutor office hours in Platt are Sundays 8-10pm and Mondays 9-11pm; Prof. Stone's office hours in Olin 1251 are MW 4-5pm and TR 3-4pm and by appointment.
- The usual collaboration rules apply. You may *discuss* an exercise with any other student(s) currently taking CS 81 as long as:
  - You contribute equally;
  - You come away from this discussion only with *understanding in your head* — no written materials or computer notes may be retained;
  - Your submission is authored solely by you, on a separate occasion.
- You should refer only to materials from this semester of CS 81 (lecture notes, handouts, textbooks, grutors, profs, etc.).
- Bring a writeup/printout to class on the due date. Illegible answers will get no credit.
- Make sure your submission includes your name!

---

1. Read pp. 31–76 of Sipser. Come up with (at least) two questions about the reading such that you're not sure of the answer. These may relate to points where the book is confusing, or simply to some related question or conjecture that occurs to you while doing the reading.
2. Construct the NFAs in Exercise 1.7(a–h), page 84. (We only want your NFA, not any intermediate work.)
3. Construct the DFAs in Exercise 1.16(a–b), page 86. (It should be clear from the DFA you provide how the subset construction was applied.)
4. Using either method from class, convert the automata to regular expressions in Exercise 1.21(a–b), page 86. (Show your work.)

...continued on back...

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.

   (a) Explain why the most-obvious answer

   $$/*\big(.\big|\backslash n\big)^**/$$

   would not make a lexer skip comments correctly. (Big hint: greedy matching)

   (b) Once the problem with the previous expression is noted, the second-most obvious solution seems to be:

   $$/*\big([\hat{}*]\big|*[\hat{}/]\big)^**/$$

   Find a legal 5-character comment that this regular expression fails to match.

   (c) Provide a correct regular expression that matches all and only valid C traditional comments, and carefully justify your solution. (Note: the previous regular expression has more than one problem, so it's not convincing to just show that you've fixed the problems you're aware of—there might be other problems, or your patches may have introduced new ones. You are better off starting with a blank slate.)

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