Assignment 7: Temporal Logic and Regular Languages

Due: Wednesday, March 30

- 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 Temporal Logic

(CTL was described in class. It also appears in Section 3.4 of Huth & Ryan.)

	Eventually True	Invariably True
Some Path	EF p Possibly	EG p Potentially Always
All Paths	AF p Inevitably	AG p Invariably

Consider the possible execution paths in the following piece of code:

```
int a[100] = { 0, 0, 0, ..., 0 };  // Array is initially all zeros

while (true) {
    int k = user_input();   // Assume this user input will be a number
                            //      between 0 and 99 inclusive!

    if (a[k] == 1) {
         for (int i = 0; i < 100; ++i)
              a[i] = 0;
    } else {
         a[k] = 1;
    }
}
```

Which of the following logical propositions are true (starting at the beginning of the program)? Explain your answer; you do not need to provide a full formal proof, but your explanation should be clear and convincing.

1. $\mathsf{AG}\,(\mathsf{a}[42] = 0)$
2. $\mathsf{AG}\left(\sum_{j=0}^{j<100} \mathsf{a}[j] < 100\right)$
3. $\mathsf{EF}\,(\mathsf{a}[42] = 1)$
4. $\mathsf{AF}\,(\mathsf{a}[42] = 1)$
5. $\mathsf{EG}\,(\mathsf{a}[42] = 0)$
6. $\mathsf{AG}\,(\mathsf{AF}\,(\mathsf{a}[42] = 0))$
7. $\mathsf{AG}\,(\mathsf{EF}\,(\mathsf{a}[42] = 1))$
8. $\mathsf{EF}\,(\mathsf{AG}\,(\mathsf{a}[42] = 0))$

2 Decision Problems

One of the oldest NP-Complete problems is 3SAT; if this could be solved efficiently, then thousands of other interesting 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 finite alphabet suitable for this problem, and briefly justify your answer.

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

3 Regular Languages

1. Read pp. 31–76 of Sipser. Come up with (at least) two questions about the reading; they should be questions where you're not sure of the answer. These may refer 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. 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

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

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

(b) Once the problem with the previous expression is noted, most people next decide that comments should contain only characters that are not stars plus stars that are not immediately followed by a slash. They then write the following regular expression:

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

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

(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 clear when you're using * the character and when you're using * the regular expression notation.