1. (**10 points**) Let $A_1 = \{D_1\#D_2 \mid D_1$ and D_2 describe DFAs and $L(D_1) \subseteq L(D_2)\}$. In English, $D_1\#D_2 \in A_1$ iff D_1 and D_2 describe DFAs and every string that is recognized by the DFA described by D_1 is also recognized by the DFA described by D_2. You can assume that D_1 and D_2 are described as in the Oct. 24 lecture notes, or any other reasonable description. Show that A_1 is Turing decidable.

   Solution: We know that regular languages are closed under union and complement, and the methods presented in class (and in Sipser) for deriving a DFA the recognizes the union or complement of the language(s) recognized by other DFA(s) are straightforward and computable. This problem asks if $\overline{L(D_1)} \cup L(D_2) = \Sigma^*$. To determine this, a TM can derive a DFA for $\overline{L(D_1)} \cup L(D_2) = \Sigma^*$ and verify that every state reachable from the initial state of this DFA is accepting.

2. (**15 points**) Let $A_2 = \{G \mid G$ describe CFG and $L(G) \supseteq L(1^*)\}$. In other words, G generates every string that consists of zero or more 1's (it may generate other strings as well). Show that A_2 is Turing decidable.

   Solution: I'll assume that G is in CNF. If not, a TM can derive an equivalent CNF grammar as described in class. Let n be the number of variables in G. As shown in HW 6, G has a pumping lemma constant, p, that is at most 2^{n-1}. We note that for any $k \geq p$, if G generates 1^k then G also generates $1^{k+p!}$ (proof below). Therefore, if G generates all strings of the form 1^m for $0 \leq m < p! + p$, then G generates Σ^*. A TM can individually check that G generates each of these $p! + p$ strings as shown in class (and in Sipser). If it does, the TM accepts; otherwise it rejects.

   Here's the proof that if G generates 1^k it also generates $1^{k+p!}$. By the pumping lemma, if G generates 1^k, then we divide 1^k into strings u, v, x, y and z such that $uvxyz = 1^k$, $|vxy| \leq p$, $|vy| > 0$ and $uv^ixy^iz \in L(G)$ for any $i \geq 0$. Let $h = |vy|$ and note that $1 \leq h \leq p$. By the pumping lemma, $1^{k+ih} \in L(G)$ for any $i \geq 0$. Because $1 \leq h \leq p$, h is a factor of $p!$, and $p!/h$ is a non-negative integer. Thus $1^{k+(p!/h)h} = 1^{k+p!} \in L(G)$ as claimed.

3. (**15 points**) In class we considered a Java method, `boolean halt(String src, String input)`, that is supposed to return true if the Java program with the source code given by string `src` halts when run with input `input` and returns false otherwise. We showed in class that it is impossible to write such a method. Our proof involved passing the source code for a Jave program as both the `src` and `input` arguments to `halt`.

   Now consider a new method, `boolean haltNoJavaAsInput(String src, String input)`. This method returns `false` if `input` is a syntactically correct Java program. Otherwise, `haltNoJavaAsInput` returns true if the program described by `src` halts when run with input `input` and returns false otherwise (just line `halt` described above). Note that the question of whether or not a program is syntactically correct Java is Turing decidable – this is what a Java compiler does. More formally, `haltNoJavaAsInput` is a decider for the language A_3, with

$$
\begin{aligned}
A_3 \;=\; \{J\#I \mid \quad & J \text{ is the source code for a Java program} \\
\wedge \quad & I \text{ is a string that is \textbf{not} a syntactically correct Java program} \\
\wedge \quad & \text{Program } J \text{ halts when run with input } I \\
\}
\end{aligned}
$$

   Show that it is impossible to write a method `haltNoJavaAsInput` as described above. Equivalently, show that language A_3 is not Turing decidable.

   Solution: All we have to do is modify string `input` so that it won't be a valid Java program and make our counter-example generator undo that change. For example, no Java program can start with the character $\}$. So, we'll make a version program that discards the first character of its input and uses that as `src` and uses the entire string as `input`. Here's the result:

```
boolean halt(String src, String input) {
    /* whatever halt does */
```

```
}
boolean undecidableForHalt(String input) {
    if(halt(input.substring(1), input)) while(1);
    return(true);
}
```

Let S be the source for this program, and invoke undecidableForHalt with the parameter $\} \cdot S$. Note that $\} \cdot S$ is not a syntactically correct Java program. Then, undecidableForHalt will invoke halt(S, $\} \cdot S$). As with the original halting problem, we get a contradiction no matter what halt returns.

4. (**15 points**) In class, we constructed one example that must cause a proposed function for halt to give the wrong answer or never terminate. Show that for any proposed implementation of halt there must be an infinite number of inputs that cause it to give the wrong answer or never terminate.

Solution: For the sake of contradiction, assume otherwise. In particular, let halt that gives the correct answer for all but a finite number of inputs. Let Σ be the input alphabet and let

$$\begin{array}{rcll} \mathit{ShouldHaveSaidHalts} & \subset & \Sigma^*, & \text{inputs, where the correct answer is halt, but halt looped or returned false.} \\ \mathit{ShouldHaveSaidLoops} & \subset & \Sigma^*, & \text{inputs, where the correct answer is not halt, but halt looped or returned true.} \end{array}$$

Because $\mathit{ShouldHaveSaidHalts}$ and $\mathit{ShouldHaveSaidLoops}$ are finite, both are regular. Thus, there is a DFA D_{halts} that recognizes $\mathit{ShouldHaveSaidHalts}$ and a DFA D_{halts} that recognizes $\mathit{ShouldHaveSaidLoops}$. Now, we can construct a TM (or Java program, etc.) that first checks if DFA D_{halts} accepts the input string and if so, our TM accepts. Second, it checks if DFA D_{loops} accepts the input string, and if so, our TM rejects. Finally, it runs halt on the input string. Because all wrong and looping cases for halt where handled by the two DFAs, halt will give the correct answer.

We have just shown that if there is a TM (or Java method, etc.) that gives decides correctly on all but a finite set of inputs, we can use it to construct a TM (or Java method, etc.) that decides correctly on all inputs. However, we have shown that there is not TM (or Java method, etc.) that decides correctly on all inputs. Thus, there cannot be one that decides correctly on all but a finite set.

Further remarks: Let's write this in "Java" and see what happens. Let's say that we have a Java method

```
boolean halt(String prog, String input) { ... }
```

that gives the right answer on all but a finite set of inputs. Let String halts[][2] be an array where for each i, the Java program described by halts[i][0] halts when run with input halts[i][1]. The array halts holds all examples where halt() loops or gives the wrong answer and should have returned true. Let String loops[][2] be the equivalent array for cases where halt() loops or gives the wrong answer and should have returned false. Now, we write the Java program shown in figure 1. Let s the string that is the source code for this program. What happens the CounterExample program with s as its parameter?

If there is an i such that halts[i][0] == halts[i][1] == s, then the program would loop forever; so we conclude that $[s, s]$ is not in the halts array.

If there is an i such that loops[i][0] == loops[i][1] == s, then the program would halt forever; so we conclude that $[s, s]$ is not in the loops array.

Now, we're back to the original halting problem. The function halt will be called, and whether it returns true or false, the actual program will do the opposite. We conclude that we cannot implement a halt that is correct for all but a finite set of inputs.

5. (**35 points**) Download the program mystery.java from

http://www.ugrad.cs.ubc.ca/~cs421/hw/7/mystery.java

Look over the code, compile it, and run it – I promise that it's not malicious.

```
class CounterExample {
    static String halts[][2] = ...;
    static String loops[][2] = ...;

    static boolean halt(String prog, String input) { ... }

    static boolean correctedHalt(String prog, String input) {
        for(int i = 0; i < halts.length; i++)
            if((halts[i][0] == prog) && (halts[i][1] == input))
                return(true);
        for(int i = 0; i < loops.length; i++)
            if((loops[i][0] == prog) && (loops[i][1] == input))
                return(false);
        return(halt(prog, input));
    }

    public static void main(String args[]) {
        if(correctedHalt(args[0], args[0])) while(1);
        else System.exit(0);
    }
}
```

Figure 1: Example program for question 4

(a) (**5 points**) What does the program do? Just give a one-sentence description of the output that it produces. You'll get to explain *how* it does it in the rest of the question.

Solution: The program prints a copy of its source code to stdout.

(b) (**5 points**) What is string s for?

Solution: The string s hold most of the source code for s as a string.

(c) (**5 points**) What does method x() do?
A one sentence answer is enough. You'll get to explain the details in the next three questions.

Solution: Method x() produces the string that is the source code for Mystery.java. as a string.

(d) (**5 points**) What do the first four buf.append(...)'s in x() do?

Solution: They insert the code before the string initializers for s into the string buffer that will hold the source for the program.

(e) (**5 points**) What does the first for loop in x() do?

Solution: It copies the string initializers for s into the string buffer. It gets this strings from s itself.

(f) (**5 points**) What does the second for loop in x() do?

Solution: It copies the same strings from s into the string buffer. Howver, this time they are appended as source statements and not as quoted strings.

(g) (**5 points**) What does method fix(String) fix?

Solution: It takes care of characters that are "special" in Java strings: double-quote, backslash, and newline. fix converts each of these into the forms that are used in Java string constants.

6. (**20 points**) A 2-PDA is a PDA with two stacks.

(a) (**10 points**) Describe a 2-PDA that recognizes the language $\{w \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* \mid \exists n.\ w = a^n b^n c^n\}$. This shows that a 2-PDA is more powerful than a 1-PDA.

Solution: My 2-PDA processes the strings in the phases described below:

q_0: The 2-PDA starts by pushing special endmarkers, $, onto each stack and moves to state q_1.

q_1: For each a that it reads, the 2-PDA pushes a bullet onto the first stack. The 2-PDA can make an ϵ-move from state q_1 to state q_2.

q_2: For each b that it reads, the 2-PDA pops a $\bullet$ from the first stack and pushes a $\bullet$ onto the second stack. If the first stack doesn't have a $\bullet$, the machine rejects. The 2-PDA can make an ϵ-move from state q_2 to state q_3.

q_3: For each c that it reads, the 2-PDA pops a $\bullet$ from the second stack. If the both stacks have a $ as the top-of-stack symbol, then the machine can make an ϵ move to state q_4 and accept.

q_4: The machine accepts. It can make no further moves from this state.

To summarize, the 2-PDA uses its first stack to verify that the number of a's is equal to the number of b's. It uses its second stack to verify that the number of b's is equal to the number of c's.

(b) (**10 points**) Show that the class of languages recognized by 2-PDAs is exactly the same as the set of Turing recognizable languages. (Hint: Show that any Turing machine can be simulated by a 2-PDA and vice-versa).

Solution: Simulating a 2-PDA with a TM is simple: use a 3-tape, non-deterministic TM. Two of the tapes simulate the two stacks, and the third tape holds the input string. As in Sipser, we allow each head to move left one square, move right one square, or stay at the same position with each step of the full machine. This machine can then move across the input tape one symbol for each step, pushing and/or popping symbols from the two stacks according to the non-deterministic, finite control of the 2-PDA.

Simulating a TM with a 2-PDA is nearly as straightforward. I'll call the two stacks *left* and *right* to hold the tape contents to the left of the current head position and to the right respectively.

The 2-PDA starts by pushing endmarkers, $\vdash$ and $\dashv$ onto the *left* and *right* stacks respectively. It then pushes the input string onto the left stack. This corresponds to scanning the TM head across the input string. The string is now all to the left of the TM head. The 2-PDA now pops symbols off of the left stack and pushes them onto the right stack until it reaches the $\vdash$ endmarker. These are a bunch of ϵ-moves that consume no input. Now, the 2-PDA is has its stacks set-up to correspond to the TM's tape.

At each step of the simulation, the current TM tape symbol is represented by the symbol on the top of the *right* stack. Based on the current state (held in the 2-PDA state) and this symbol, the 2-PDA simulates the TM move. In particular, if the TM moves its head to the right, then the 2-PDA pops the current symbol off of the *right* stack and pushes the symbol that the TM writes at the current square onto the *left* stack. If the new top-of-stack symbol on the right is a $\dashv$ the 2-PDA pushes a blank (e.g. $\square$) onto the *right* stack.

If the TM moves its head to the left with the current move, then the 2-PDA pops the current symbol off of the *right* stack and pushes the symbol that the TM writes at the current square onto the *right* stack. If the top-of-stack symbol on the *left* stack is not a $\vdash$, the 2-PDA pops this symbol off of the *left* stack and pushes it onto the right stack.

If the 2-PDA enters the accept state for the TM, then it enters an accepting state. If it enters the reject state for the TM, then the 2-PDA enters a terminal rejecting state.

7. (**20 points**, *extra credit*) A *ray automaton* consists of an infinite number of DFAs, D_0 D_1, D_2, $\ldots$ arranged in a line. The automata all have the same set of states, Q, the same start state $q_0 \in Q$, and the same transition function $\delta : Q \times Q \times Q \to Q$. A configuration of a ray automaton is a function $\mathcal{C} : \mathbb{Z}^{\geq 0} \to Q$ where $\mathcal{C}(i)$ gives the state of DFA D_i. The automaton moves from configuration $\mathcal{C}$ to configuration $\mathcal{C}'$ iff

$$\begin{array}{rcl} \mathcal{C}'(0) & = & \delta(q_0, \mathcal{C}(0), \mathcal{C}(1)) \\ \mathcal{C}'(i) & = & \delta(\mathcal{C}(i-1), \mathcal{C}(i), \mathcal{C}(i+1)), \quad i > 0 \end{array}$$

In other words, at each step, each DFA makes a transition according to its own state and the states of its left and right neighbours. Because DFA D_0 has no left neighbor, it always uses q_0 as its left input. There is a special state q_f, and the ray automaton halts iff it reaches a configuration, $\mathcal{C}$ where D_0 is in state q_f, i.e. $\mathcal{C}(0) = q_f$.

(a) (**10 points**) Prove that the halting problem for ray automata is undecidable.

Solution: We can simulate a TM, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ running with input w by using a ray-automaton. The key idea is to use the states of the ray automaton to keep track of the symbols on the TM's tape, along with the head position and current state.

The set of states for the ray automaton is $\{0, 1, \dots |w|, q_f\} \cup \Gamma \cup (\Gamma \times Q)$ where 0 denotes the initial state. Let $I = \{1 \dots |w|\}$. DFAs in states in I "count" until they reach their position and then transition to the state for the corresponding symbol of w. Here's the details. Initially, every DFA is in state 0 and transitions to state 1. Let $d(k)$ be a DFA in state $i \in I$.

If $d(k-1) = 0$, then $d(k)$ is the leftmost DFA and it transitions to state (w_1, q_0) to start the computation. w_1 denotes the first symbol of w.

If $d(k-1) \in I$, then $d(k)$ transitions to state $i+1$; in other words, it keeps counting.

If $d(k-1) \in \Gamma$, then $d(k)$ transitions to state w_i where w_i is the i^{th} symbol of w.

If $d(k-1) \in \Gamma \times Q$, then the TM's tape head is at square $k-1$ at this step, and its represented by DFA $k-1$. If the TM's head moves to the right, $d(k)$ transitions to state (w_i, q') where q' is the next state of the TM. Otherwise (the TM's head moves to the left), $d(k)$ transitions to state w_i.

If $i > |w|$ then $d(k)$ transitions to the state corresponding to the blank symbol.

At the end of $|w|+1$ steps, the DFAs have taken on states corresponding to the symbols of the TM's input tape. We have also simulated the fist $|w|-1$ steps of the TM's operation as described below.

After the second step, there will always be exactly one DFA in a state in $\Gamma \times Q$. This corresponds to the current TM tape head position and state. Let this be DFA i and let it be in state (c, q).

If $\delta(q, c) = (q', c', R)$, then DFA i transitions to state c'. If DFA $i+1$ is in state $d \in \Gamma$, then DFA $i+1$ transitions to state (d, q'). Otherwise, DFA $i+1$ must be in state $j \in I$, and it transitions to state (w_j, q').

If $\delta(q, c) = (q', c', L)$, then DFA i transitions to state c', and DFA $i-1$ transitions to state (d, q') where $d \in \Gamma$ is the current state of DFA $i-1$.

These operations simulate TM moves.

If a transtion would bring a DFA to a state of the form (c, q_{accept}) then it goes to state q_f. Furthermore, if the right neighbour of a DFA is in state q_f the DFA transitions to state q_f. This ensures that the leftmost DFA will eventually enter state q_f if any DFA ever enters state q_f.

All other DFAs not covered by situations described above stay in their same state for the next step. They are holding TM tape symbols but aren't at or next to the position corresponding to the TM tape head.

This ray-automaton halts iff the Turing machine being simulated accepts its input string. Thus, every Turing recognizable language can be recognized by a ray-automaton.

(b) (**10 points**) Is the halting problem for ray automata Turing recognizable? Justify your answer.

Solution: Yes. If a ray-automaton halts, it does so after some finite number of steps. Let's call this number n. We note that the leftmost DFA can be affected by at most the next n DFAs to the right in the course of a n step computation. Thus, it is sufficient to simulate a ray-automaton consisting of n DFAs. The challenge is that we don't know before hand how big n is.

Let a bounded ray-automaton be like a ray automaton but with only a fixed number of DFAs. There is a special state $q_\dashv$. The rightmost DFA always uses $q_\dashv$ as its right input. If any DFA has $q_\dashv$ as an input, it transitions to $q_\dashv$ in the next (and therefore all subsequent steps).

A TM can simulate a bounded ray-automaton with one DFA for one step, then one with two DFAs for two steps, and so one. Each such simulation involves a finite number of steps. If the orginal ray automaton halts after n steps, then the TM will eventually simuate an n-DFA automaton for n steps and find that the leftmost DFA is in state q_f and accept. If the original ray automaton loops, then the simulation described above will run forever as well.

Therefore, the halting problem for ray automata is Turing recognizable as claimed.