1. (**15 points**) Let boolean halt(String src, String input) be a Java method 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 that will work for all strings src and input. Now, we will add one more restriction: halt fails (i.e. throws an exception) if the string for input is the source code for a Java program. For example, halt could run the Java compiler on input and if it compiles successfully, then halt throws a InputWrittenByASmartAleckException. In all other cases, halt should correctly answer whether the program given by src halts when run with input input.

   Show that even with this restriction, it is impossible to write a method halt that returns true if the program described by src halts when run with input input and returns false otherwise.

   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 $\} \cdot S$ Then, undecidableForHalt will invoke halt(S, $\}\cdot S$). As with the original halting problem, we get a contradiction no matter what halt returns.

2. (**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: Suppose that there were only a finite number of input strings that cause halt to loop or give the wrong answer. We can divide these into two finite subsets: those that describe halting computations and those that don't. Any finite language is regular, and any regular language is decidable. Thus, we can first test to see if the input is in one of these two sets, and if it is, give the appropriate answer. Otherwise, we run the original machine. Note that our initial screening ensures that we never run the original machine on an input where it fails to give the right answer. Thus, this would decide the halting problem. We know that the halting problem is undecidable. Therefore, the set of input string that cause any proposed solution to the halting problem to give a wrong answer or fail to terminate must be infinite.

3. (**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.

1. (**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.

2. (**5 points**) What is string s for?

   Solution: The string s holds strings that match the source code for methods fix(), x(), and main(), in other word, fore everything declared after s itself.

3. (**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.

4. (**5 points**) What do the first four buf.append(...)'s in x() do?

   Solution: They append to buf the line of mystery.java up to the declaration of String[] s but not the values of the initializers for this array.

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

   Solution: It appends to buf the lines of mystery.java that initialize String[] s.

6. (**5 points**) What does the second for loop in x() do?

   Solution: It appends to buf the source code for everything else declared in mystery.java.

7. (**5 points**) What does method fix(String) fix?

   Solution: It translates special characters in Java strings: ", \, and \n into the \... sequences needed by the Java compiler.

4. (**20 points**, Sipser Problem 3.9)
   Let a k-PDA be a pushdown automaton that has k stacks. Thus a 0-PDA is an NFA and a 1-PDA is a conventional PDA. You already know that 1-PDAs are more powerful (recognize a larger class of languages) than 0-PDAs.

   1. (**10 points**) Show that 2-PDAs are more powerful than 1-PDAs.

      Solution: A 2-PDA can recognize the language $\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n$. Clearly it can use its finite control to make sure that its input is in $\mathsf{a}^*\mathsf{b}^*\mathsf{c}^*$. While reading the string of a's, the 2-PDA pushes a marker onto its first stack for each a that it reads. While reading the string of b's, it pops a marker off of its first stack and pushes a marker onto its second stack for each b that it reads. In this way, the 2-PDA checks to make sure that the number of a's matches the number of b's, and it has a record of how many b's it has seen. Finally, while reading the string of c's, it pops a marker off of its second stack for each c that it reads. It does this to confirm that the number of c's is the same as the number of bs.

   2. (**10 points**) Show that 3-PDAs are not more powerful than 2-PDAs.
      (Hint: Simulate a Turing machine tape with two stacks.)

      Solution: A 2-PDA can simulate a Turing machine. Let the first stack represent everything on the tape to the left of the tape head and the second stack represents the current tape symbol and everything to the right. From now on, I'll refer to these as the "left stack" and "right stack."
      Initially, the 2-PDA pushes a special end-marker onto each stack. Next, it reads its input, pushing every symbol that it reads onto the left stack. Then, it can pop all of those symbols off of the left stack and push them onto the right stack. It's now in a configuration where the input string is on the right stack.
      To simulate a move of the Turing machine, the 2-PDA checks the value on the top of the right stack and its current state. If the TM will move to the left for this combination, then the 2-PDA does the following:

- Pop the symbol off the top of the right stack.
- Push the symbol that the TM would write onto its tape onto the right stack.
- Pop a symbol off of the left stack. If this symbol is the endmarker, it just pushes it back onto the left stack. Otherwise, it pushes whatever symbol it encountered onto the right stack.

On the other hand, if the TM will move to the right, then the 2-PDA does the following:

- Pop the symbol off the top of the right stack.
- Push the symbol that the TM would write onto its tape onto the left stack.

If the 2-PDA pops the end-symbol off of its right stack, it pushes the end-symbol back on to the right stack and then pushes a blank onto the right stack. This simulates having an infinite number of blanks on the tape.

If the 2-PDA ever reaches an accepting state for the TM, then it accepts the string. If it reaches a rejecting state, it rejects the string. Note that a 2-PDA can loop, just like the Turing machine it is simulating.

OK, now we've shown that a 2-PDA can simulate a TM that has a single tape. We know that a TM with a single tape can simulate a TM with multiple tapes. It's simple to simulate a 3-PDA using a TM with four tapes: one tape for the input, and the other three for the three stacks. Thus, a 2-PDA can simulate a 3-PDA, and therefore a 3-PDA is no more powerful than a 2-PDA.

Note: A shorter answer would show in part (a) that a 2-PDA can simulate a TM, and just state that TMs can recognize languages that 1-PDAs cannot.

5. (**20 points**, Sipser Problem 3.11)

A *Turing machine with doubly infinite tape* is similar to an ordinary Turing machine, but its tape is infinite to the left as well as to the right. The tape is initially filled with blanks except for the portion that contains the input. Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward. Show that this type of Turing machine recognizes the class of Turing-recognizable languages.

Solution: A TM with a doubly infinite tape can be simulated by a TM with two tapes. We'll call the tapes the left tape and the right tape. We'll call the machine with the doubly-infinite tape $M_{\pm infinity}$ and the machine with two tapes M_2. Initially, the right tape of M_2 holds the input string followed by an infinite string of blanks, and the left tape holds an inifinite string of blanks. M_2 first moves every symbol of the input one position to the right and writes a special end-marker in front of the input string on the right tape. It also writes an end marker on the first square of the left tape. It moves the head of the right tape to the first symbol of the input string (or the first blank if the input string is empty), and it moves the head of the left tape to the first blank. M_2 uses its finite state to keep track of whether the left tape or the right tape is the currently active tape. Initially, the right tape is the active one.

To simulate a move of $M_{\pm\infty}$, M_2 checks the symbol under the head of the active tape. If it is an end-marker, it moves the head one position to the right and switches to the other tape as the active tape. Otherwise, if the right tape is active, M_2 does the same thing on that tape as $M_{\pm\infty}$ would do on its tape. Finally, if the left tape is active, M_2 does the same thing on that tape as the $M_{\pm\infty}$ would do on its tape, except that if $M_{\pm\infty}$ would move to the left, M_2 moves to the right, and vice-versa.

Any transition to the accepting state of $M_{\pm\infty}$ causes M_2 to transition to its accepting state and likewise for transitions to the rejecting state of $M_{\pm\infty}$.

6. (**30 points**, Sipser Problem 3.14)

A *queue automaton* is like a push-down automaton except that the stack is repoaced by a queue. A *queue* is a tape allowing symbols to be written only on the left-hand end and read only at the right-hand end. Each write operation (we'll call it a *push*) adds a symbol to the left-hand end of the queue, and each read operation (we'll call it a *pull*) reads and removes a symbol at the right-hand end. As with a PDA, the input is placed on a separate read-only input tape, and the head on the input tape can only move from left to right. The input tape contains a cell with a blank symbol following the input, so that the end of the input can be detected. A queue automaton

accepts its input by entering a special accept state at any time. Show that a language can be recognized by a deterministic queue automaton iff the language is Turing recognizable.

Solution: First, I'll show that a queue automaton can simulate a TM. Then, I'll show that a TM can simulate a queue automaton. This shows that the two types of machines are equivalent.

Initially, the queue automaton marks the first input symbol (possibly a blank). It then pulls three symbols and stores them in its finite state — I'll call these three symbols "left", "middle" and "right". If it encounters a blank before pulling three symbols, it sets the remaining symbols (i.e. right and possibly middle) to blank. The queue machine also keeps track of the TM's state using its finite state. The queue machine repeats the following actions until it reaches the accept state or the reject state:

- if the middle symbol is *not* marked, then the machine pushes its left symbol into the queue, transfers its middle symbol to the left symbol, transfers the right symbol to the middle symbol, and pulls a new right symbol from the queue.
- if the middle symbol *is* marked, then the machine determines what the TM would do for the current state and input symbol:
  - If the TM would move to the left, then the queue automaton marks its left symbol and pushes it into the queue. It transfers the symbol that the TM would write onto its tape to the left symbol, transfers the right symbol to the middle symbol, and pulls a new right symbol from the queue.
  - If the TM would move to the right, then the queue automaton pushes its left symbol into the queue, transfers the symbol that the TM would write onto its tape to the left symbol, transfers the right symbol to the middle symbol and marks it, and pulls a new right symbol from the queue.

  Note that in either case, the queue automaton first pushes a symbol into the queue and then pulls one from the queue. This guarantees that there the queue will be non-empty whenever the automaton attempts a pull.

With these actions, the queueu automaton simulates the TM. Basically, if the TM moves to the right, the queue automaton pushes the left symbol onto the stack, pulls a new right symbol and continues tracking the TM. If the TM moves to the left, then the queue automaton has cycles through the queue to get back to where the TM moved. Marking symbols in the queue to indicate the current head position of the TM allows the queue automaton to know when it has made it to the right place.

Now, I'll show how a TM can simulate a deterministic queue automaton. Of course, the TM uses its tape to simulate a queue. Initially, the TM marks the first symbol of the input to indicate that it is at the head of the queue and the blank after the last symbol to indicate that it is the tail. If the input is empty, then the TM marks the initial blank as being both the head and the tail.

When the queue automaton does a pull, the TM scans its tape to find the tape square marked as the head of the queue. If this square is also marked as the tail, then the queue is empty. The problem didn't say what a queue automaton does if it tries to pull from an empty queue — I'll assume that this is grounds for immediate rejection of the input. Otherwise, the TM records the symbol at the head of the queue in its finite state and moves the head-of-queue marker one position to the right.

When the queue automaton does a push, the TM scans its tape to finde the tape square marked as the tail of the queue. It erases the tail marker and writes the pushed symbol on the next square to the right, marking it as the new tail. Note that this will always be overwriting a blank.

With this construction, the segment of the tape holding the queue contents drifts to the right. This is not a problem because the TM has an infinite amount of tape.