Please note that there is more than one way to answer most of these questions. The following only represents a sample solution.

(1) A context-free grammar is linear if every rule has at most one variable on the right hand side. A language is linear if it is generated by a linear grammar.

(a) Show that the regular languages form a proper subset of the linear languages.

**Answer:** First, we’ll show containment. Let $L$ be a regular language. Then there exists a regular grammar $G$ that generates $L$, such that every rule of $G$ has the form $A \rightarrow aB$ or $A \rightarrow \epsilon$, for some $A,B \in V$, $a \in \Sigma$. Therefore every rule in $G$ has at most one variable on its right hand side. Therefore, $G$ is a linear grammar. Thus, $L$ is a linear language.

Next we’ll show that the regular languages form a proper subset. Consider the language $L = \{0^n1^n : n \geq 0\}$. Then from Example 1.73 in Sipser, we know that $L$ is not regular. However, there is a linear grammar for $L$ given by

$$S \rightarrow 0S1 | \epsilon.$$ 

Thus, there is a language that is linear, but not regular, so the containment is proper.

(b) Is every linear language a DCFL? Answer yes or no, and explain. (If you answer yes, show why. If you answer no, give an example.)

**Answer:** No, consider the language $A = \{a^ib^jc^k : i \neq j \text{ or } j \neq k\}$. Then $A$ is a linear language, with a linear grammar given by

$$\begin{align*}
S & \rightarrow \ Cc | aA \\
C & \rightarrow \ Cc | R \\
R & \rightarrow \ aRb | aT | bB \\
T & \rightarrow \ aT | \epsilon \\
B & \rightarrow \ bB | \epsilon \\
A & \rightarrow \ aA | U \\
U & \rightarrow \ bUc | bB | cV \\
V & \rightarrow \ cV | \epsilon
\end{align*}$$

However, $A$ is not a DCFL. If it were then $\overline{A} = \{a^ib^jc^k : i \geq 0\} \cup L(a^*b^*c^*)$, would also be a DCFL, because DCFLs are closed under complement. However, then $A' = \overline{A} \cap L(a^*b^*c^*) = \{a^ib^jc^k : i \geq 0\}$ would be a CFL, since CFLs are closed.
under intersection with a regular language. However, \( A' \) is not a CFL as shown in Example 2.36 of Sipser. Therefore, \( A \) cannot be a DCFL. Thus, there is a linear language that is not a DCFL.

(2) I modify the notion of PDA to provide one more ‘stack’ operation. As well as pushing and popping, a transition is permitted to turn the stack upside down (it cannot push in the same transition if it does so). Call this a \textit{flip}-PDA.

(a) Give a formal definition of a flip-PDA, particularly its transition function. (This is asking for a general definition, not an example.)

\textbf{Answer:} A flip-PDA is a 7-tuple, \( P = (Q, \Sigma, \Gamma, f, \delta, q_0, F) \) where

- \( Q \) is a finite set of states,
- \( \Sigma \) is a finite set of the input alphabet,
- \( \Gamma \) is a finite set of the stack alphabet,
- \( f \notin \Gamma \) is the flip symbol,
- \( \delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q \times \Gamma \cup \{\epsilon, f\}) \) is the transition function (note it has been modified in that instead of pushing a symbol from \( \Gamma \cup \{\epsilon\} \) we can instead flip the stack by using the symbol \( f \))
- \( q_0 \in Q \) is the start state, and
- \( F \subseteq Q \) is the set of accept states.

A flip-PDA \( P \) will accept a string \( w \) if \( w \) can be written as \( w = w_1 \ldots w_n \) where each \( w_i \in \Sigma \cup \{\epsilon\} \) and there exists a sequence of states \( r_0, r_1, \ldots, r_n \in Q \) and strings on the stack \( s_0, s_1, \ldots, s_n \in \Gamma^* \) such that:

- \( r_0 = q_0 \) and \( s_0 = \epsilon \) (that is the flip-PDA starts in the start state with an empty stack),
- \( (r_{i+1}, b) \in \delta(r_i, w_{i+1}, a) \) for all \( 0 \leq i \leq m - 1 \), \( s_i = at \) for some \( a \in \Gamma \cup \{\epsilon\} \), \( t \in \Gamma^* \) and \( s_{i+1} = bt \) if \( b \in \Gamma \cup \{\epsilon\} \) or \( s_{i+1} = \text{reverse}(t) \) if \( b = f \) (that is the stack and state are updated properly according to the transition function. Note here, how the stack is flipped if the flip symbol is used),
- \( r_m \in F \) (after the input is read the flip-PDA is in an accepting state).

(b) Show that there is a language that is recognized by a flip-PDA but not recognized by a PDA.

\textbf{Answer:} Consider the language \( L = \{ww : w \in \{0,1\}^*\} \). Then, \( L \) is not a CFL as shown in Example 2.38 of Sipser. Therefore, there is no PDA that can recognize \( L \). However, the flip-PDA given in the following state diagram will recognize \( L \) by non-deterministically guessing where the middle of the string is, marking that as the bottom to match against and then flipping the stack.
(3) Describe in detail a deterministic Turing machine to recognize \( \{0^a1^b2^c : c \text{ is the remainder when } a \text{ is divided by } b \} \).

**Answer:** First we’ll check that the string has the symbols in proper order and that it does not divide by zero (that is there are no 1s). Then to do the division we will repeatedly subtract (remove) the number of ones from the number of zeroes until there are more 1s than 0s. Then we will check if the number of remaining 0s equals the number of 2s. A detailed algorithm for a TM, \( M \), to do this follows.

\[ M = \text{On input } w \in \{0, 1, 2\}^*: \]

1. **Comment:** Check the string is in proper order and no division by zero occurs.)
   - Examine the symbol under the head:
     - If the symbol is a blank, reject (0 divided by 0 is undefined).
     - If the symbol is a 2, reject (0 divided by 0 is undefined).
     - If the symbol is a 1, scan right to the next non-1 symbol.
       - If the symbol is a blank, accept (0 = 0 mod \( b \)).
       - On any other symbol, reject.
     - If the symbol is a 0, replace it with a special symbol to mark the start of the tape, \( 0^S \), and move right to the first non-0 symbol.
       - If the symbol is not a 1, reject (division by 0).
       - Otherwise, move right to the first non-1.
         * If the symbol is a 0, reject (symbols out of order).
         * If the symbol is a blank, go to (2).
         * If the symbol is a 2, move right to the first non-2.
         * If the symbol is not a blank, reject (symbols out of order).
         * Otherwise, go to (2).

2. **Comment:** If this part is reached, then the input string is in the proper order and contains a 1.)
   Move left until a 0 or \( 0^S \) is seen, then move right (At this point the head will be over the first 1).

3. **Comment:** Steps (3)-(7) does the division, that is repeatedly removes \( b \) zeros until the number of zeroes is less than \( b \).
   - Put a dot over the 1 and move left to a blank, \( 0^S \) or \( \overset{\cdot}{0} \). If a blank or \( \overset{\cdot}{0} \) is seen move right and go to (4). Otherwise, put a dot over \( 0^S \), move right and go to (5).
(4) If a 0 is under the head, put a dot over it and go to (5). If a 1 is seen go to (8) (more 1s than 0s. Time to check for the remainder).

(5) Move right until a 1 or 2 is seen. If a 1 is seen go to (3) (try to match it with a 0). If a 2 is seen first go to (6) (matched all the 1s with a 0).

(6) Move left and remove the dot from every 1, move over the 0s without changing them and replace the 0s with blanks until a blank is seen or 1S. If 1S is seen replace it with a blank, move right and go to (7). Otherwise, a blank is seen, so move right and go to (7).

(7) Move right until the first 1 is seen and then go to (3) (try to match and remove another b 0s).

(8) (Comment: The remaining steps will check if the number of 0s remaining is equal to the number of 2s.)
Move left and remove the dot from all 0s until a blank or 1S is seen. If 1S is seen go to (10). If a blank is seen, go to (9).

(9) Move right. If the next symbol is a 0 then put a dot over it and continue to (10). If the next symbol is a 1, go to (12).

(10) Move right until either a 2 or blank is seen. If a blank is seen, reject (c < a mod b). If a 2 is seen, put a dot over it and go to (11).

(11) Move left until a 0 or 1S is seen. Then go to (9).

(12) (Comment: All the 0s have been matched to a 2. Now check that there aren’t any extra 2s.)
Move right until a 2 or blank is seen. If a 2 is seen, reject (c > a mod b). Otherwise, accept (c = a mod b).

(4) A regular expression pattern matcher needs to check whether a string matches a regular expression and answer yes or no. Suppose that we represent a problem of this type as $R\#w$, where $R$ is a regular expression, # is a symbol not used in the alphabet of the regular expression, and $w$ is a finite string. Then define the language $L_{re} = \{ R\#w : w \text{ is in the language described by } R \}$. Give a description of a (deterministic or nondeterministic) Turing machine that recognizes $L_{re}$ and always halts (so that it decides $L_{re}$). You do not need to detail the individual transitions, but provide enough detail so that a not-too-clever programmer could write out the detailed specification of the TM.

Answer: The following TM $M$ decide $L_{re}$ by converting the regular expression into a DFA.

$M = \text{On input } R\#w \text{ where } R \text{ is a regular expression and } w \text{ is a string:}$

(1) Convert $R$ into the description of an equivalent NFA, $N$, as described in the proof of Lemma 1.55 in Sipser.

(2) Convert $N$ into the description of an equivalent DFA, $D$, as described in the proof of Theorem 1.39 in Sipser.

(3) Simulate the computation of $D$ on input $w$ by keeping track of the current state and symbol and following the transition table in the description of $D$. 


(4) If the simulation halts in an accepting state of $D$, accept. Otherwise, if the simulation halts in a non-accepting state of $D$, reject.

(5) **(not to be graded)** Think of the poor grader of the homework assignments. S/he has to look at 82 different DFAs and for each to decide whether it accepts the language that it should. S/he has a correct DFA, but most of the student submissions look quite different. S/he wants to automate the process of deciding whether a student submission recognizes the same language as the sample solution. Can this be done by a Turing machine?

More precisely, we encode each DFA $M$ as a string $\langle M \rangle$. The details of the encoding are not important, but we suppose that we can find each of $(Q, \Sigma, \delta, q_0, F)$ in the encoding, and can find individual states, input symbols, and transitions within these.

Then we want to decide membership in the language $\{\langle M \rangle \# \langle M' \rangle : M$ and $M'$ encode DFAs and $L(M) = L(M')\}$. Can a deterministic TM decide this? (Hint: $L(M) = L(M')$ if and only if $(L(M) \cap \overline{L(M')}) \cup (\overline{L(M)} \cap L(M')) = \emptyset$. Can the TM construct a DFA $M''$ whose language is $(L(M) \cap \overline{L(M')}) \cup (\overline{L(M)} \cap L(M'))$? Can it then decide whether $L(M'') = \emptyset$?)

**Answer:** A Turing Machine can carry out this task. This language is equivalent to $EQ_{DFA}$ for which a TM is given in Theorem 4.5 of Sipser that decides it. Since $L(M)$ and $L(M')$ are regular the TM can follow the procedure in Theorem 1.25 to construct all the unions and intersections and can swap the accept and non-accept states for any complement. Hence a TM can construct $M''$. Furthermore, a TM can then tell if $M''$ accepts any strings as described in Theorem 4.4 of Sipser by marking the start state and every state reachable from it. If none of the states reachable from the start state is a final state, then the language is empty.