1. (16 points) Answer the following questions about languages and grammars.

   a) Is the C programming language a context free language? If not, explain in your own words why it is not context free.
   Although the syntax of the C language can be expressed as a CFG, there are many aspects of it which cannot be expressed using a CFG such as def-before-use requirements, function formal and actual parameter type match requirements, etc. These constraints that cannot be checked in the syntax analysis phase are check in the semantic analysis phase using symbol tables.

   b) Can the same language have an ambiguous and an unambiguous grammar at the same time? If yes, explain how this can be so with an example.
   Even when two grammars accept the same language (i.e. the same set of input strings), one can be ambiguous (having multiple parse trees) and the other can be unambiguous (have one parse tree). A canonical example is the dangling else problem which can be defined both ambiguously or unambiguously as was seen on the slides. Another example is grammars such as $E \rightarrow E + E$ that is ambiguous but can be modified to be unambiguous by encoding left associativity into the grammar.

   c) Explain in your own words why ambiguous grammars can never be LL or LR.
   Ambigious grammars have at least one string where there are two or more parse trees. That means for that particular string, there are two sets of rules that can be applied to successfully derive that string. In other words, there are two or more leftmost derivations (in the case of LL), and also two or more rightmost derivations (in the case of LR). Since both LL and LR parsers rely on a single derivation to deterministically choose a single action based on a table, multiple derivations will always result in a conflict in the table.

   d) Explain in your own words why LR(1) is more powerful than LL(1).
   In an LL(k) parser, the next production is predicted on the basis of the current non-terminal at the top of the stack and k terminals of lookahead. Hence, a production is applied without knowing whether the full length of the RHS will be able to match the input string. I just knows the input string will match the first k terminals that comprise the RHS.

   In an LR(k) parser, a reduction is performed only when the full length of the RHS has already been seen. The k terminals of lookahead makes it even more powerful by being able to decide on a reduction taking into consideration k terminals that follow the RHS. In addition, LR parsers have no problem with common left factors or left recursion that would either confuse the prediction of an LL parser or bust its stack.
2. (12 points) Answer the following questions about context free and regular grammars.

a) Write a context free grammar for strings with matching quotes where $\sum = \{a, b, \}'$.
E.g. 'aba', aba''bb, aa'aa'a'.

$$S \rightarrow S' \mid aS \mid bS \mid \varepsilon$$

b) Write a context free grammar for strings with matching parentheses $\sum = \{a, b, (, )\}$.
E.g. (aba), aba(abb, aa(a(a)a).

$$S \rightarrow (S) \mid aS \mid bS \mid \varepsilon$$

c) Is the language expressed in a) a regular language? If so, show that it is a regular language by writing a regular expression, for the language. If not, explain why not.
The regular expression for this grammar is given in 1.b) of HW1. And thus it is a regular language.

d) Is the language expressed in b) a regular language? If so, show that it is a regular language by writing a regular expression, for the language. If not, explain why not.
A regular expression does not exist because it is not a regular language. In order to match the correct number of left parentheses to right parentheses, the FSM will have to count the number of left parentheses for an arbitrary $k$. As we have learned, that requires $k$ states and there is no *finite* state machine that can accept all strings of the language. The difference from c) is that in the case of c), the FSM need not count the number of quotes. It just needs to tell whether there is an even or odd number of quotes in the string. Thus the FSM only needs to have two states: odd quotes and even quotes.

3. (10 points) Given the following grammar, construct the first and follow sets for each non-terminal symbol.

$$A \rightarrow BAc \mid FE$$
$$B \rightarrow bEF \mid g$$
$$E \rightarrow e \mid \varepsilon$$
$$F \rightarrow f \mid EH$$
$$H \rightarrow h$$

First($A$) = \{b, e, f, g, h\}
First($B$) = \{b, g\}
First($E$) = \{e, \varepsilon\}
First($F$) = \{e, f, h\}
First($H$) = \{h\}

Follow($A$) = \{$, c\}
Follow($B$) = \{b, e, f, g, h\}
Follow($E$) = \{$, c, e, f, h\}
Follow($F$) = \{$, b, c, e, f, g, h\}
Follow($H$) = \{$, b, c, e, f, g, h\}

4. (16 points) For each of the below grammars, answer the following questions:
1) Is the grammar LL(1)? If so, write the LL(1) parse table. If not, point out the conflict.
2) Is the grammar LL(k)? If so, show how the extra lookahead resolves the conflict. If not, show why it is not resolved using First and Follow sets.

* Note: LL(k) is LL with arbitrarily long lookahead.

3) Is the grammar ambiguous? If yes, find the input that produces multiple parse trees and give 2 or more left-derivations.

* Note LL(1) ⊂ LL(k) ⊂ L(Unambiguous). So, (2) needs answering only if (1) is false. (3) needs answering only if both (1) and (2) are false.

a) $S \rightarrow [S \mid A$
   $A \rightarrow [A \mid \varepsilon$

   1), 2) The grammar is not LL(k) for arbitrary k. No matter how large k is, First$_k([S)$ and First$_k(A)$ both contain the string $[^k$. Hence there will be a conflict in the parse table and the parser will not be able to decide which RHS to choose given non-terminal S and that input string.

   3) The grammar is not ambiguous since there is exactly one parse tree for every input. The production rule $S \rightarrow [S$ will be applied repeated for the first non-matching [ symbols, and then $S \rightarrow A$ will be applied followed by a series of $A \rightarrow [A$ for the rest of the matching [ and ] symbols.

b) $S \rightarrow ABC$
   $A \rightarrow a \mid \varepsilon$
   $B \rightarrow b \mid \varepsilon$

   1) The grammar is LL(1).

   For $A \rightarrow a \mid \varepsilon$, 
   $First(a) = \{a\}$
   $Follow(A) = \{b, c\}$
   $First(a) \cap Follow(A) = \emptyset$

   For $B \rightarrow b \mid \varepsilon$, 
   $First(b) = \{b\}$
   $Follow(B) = \{c\}$
   $First(b) \cap Follow(B) = \emptyset$

   2) The grammar is LL(k) since it is LL(1).

   3) The grammar is unambiguous since it is LL(1).

c) $S \rightarrow ABBBA$
   $A \rightarrow a \mid \varepsilon$
   $B \rightarrow b \mid \varepsilon$

   1) The grammar is not LL(1).

   For $A \rightarrow a \mid \varepsilon$, 
   $First(a) = \{a\}$
   $Follow(A) = \{a, b, $\}$
   $First(a) \cap Follow(A) = \{a\}$

   For $B \rightarrow b \mid \varepsilon$, 
   $First(b) = \{b\}$
Follow(B) = \{a, b, \$\}
First(b) \cap Follow(B) = \{b\}

2) The grammar is not LL(k). For k \geq 2,
For A \rightarrow a \mid \varepsilon,
First_k(aFollow_k(A)) \ni \{a\$
Follow_k(A) \ni \{a\$
First_k(aFollow_k(A)) \cap Follow_k(A) \ni \{a\$

For B \rightarrow b \mid \varepsilon,
First_k(bFollow_k(B)) \ni \{b\$
Follow_k(B) \ni \{b\$
First_k(bFollow_k(B)) \cap Follow_k(B) = \{b\$

3) The grammar is ambiguous because there are two parse trees for the string ‘a’. You could left-most derive
S \rightarrow ABBA \rightarrow aBB \rightarrow *a\varepsilon \varepsilon, or S \rightarrow *\varepsilon\varepsilon A \rightarrow a.

d) S \rightarrow aAbc|bAc
A \rightarrow b \mid \varepsilon
1), 2) The grammar is not LL(k). For k \geq 3,
First_k(bFollow_k(A)) \ni \{bc\$
Follow_k(A) \ni \{bc\$
First_k(bFollow_k(A)) \cap Follow_k(A) \ni \{bc\$
3) The grammar is unambiguous since there is exactly one parse tree for each of the four strings in the language: abbc, bbc, abc, bc.

5. (16 points) Given the following grammar, answer the below questions:
E \rightarrow E + E \mid id

a) Write a new grammar after performing left-recursion removal.
E \rightarrow idE'
E' \rightarrow +EE' \mid \varepsilon

b) Is the grammar in a) now LL(1)? If not, give two conflicting rules that would render it not LL(1), and also give the input symbol that would trigger those rules. The grammar in b) is still not LL(1).
First(+EE') = \{+\}
Follow(E') = \{\$, +\}
First(+EE') \cap Follow(E') = \{+\}
On input symbol +, you would have a conflict on the two rules E' \rightarrow +EE' and E' \rightarrow \varepsilon.

c) Modify the original grammar such that the + operator is left associative, and then perform left-recursion removal. Write the new grammar.
E \rightarrow TE'
E' \rightarrow +TE' \mid \varepsilon
T \rightarrow id
d) Is the grammar in c) now LL(1)? If not, give two conflicting rules that would render it not LL(1), and also give the input symbol that would trigger those rules. Yes, it is now LL(1) since:

First(+TE') = {+}
Follow(E') = {$}
First(+TE') ∩ Follow(E') = ∅

6. (30 points) For each of the below grammars, answer the following questions:
(1) Is the grammar SLR(1)? If so, draw a DFA and the corresponding parse table. If not, point out the conflict.
(2) Is the grammar LALR(1)? If so, show how the lookahead component resolves the conflict. If not, show why it is not resolved.
(3) Is the grammar LR(1)? If so, show how state splitting resolves the conflict. If not, show why it is not resolved.

* Note SLR(1) ⊂ LALR(1) ⊂ LR(1). So, (2) needs answering only if (1) is false. (3) needs answering only if both (1) and (2) are false.

a) (10 points) Σ={v, =, ;, +, (, )}.

S → v = A;
A → P E
P → P v = | ε
E → E + T | T
T → v | ( A )

(1) Is the grammar SLR(1)?
Yes, as shown with the below state machine and parse table.
(2) Is the grammar LALR(1)?
Yes, the grammar is LALR(1) since LALR(1) is stronger than SLR(1).

(3) Is the grammar LR(1)?
Same as above.

b) (10 points) $\Sigma = \{a, b, c\}$.

$$
S \rightarrow bAb \mid Ac \mid ab \\
A \rightarrow a
$$

(1) Is the grammar SLR(1)?
No. Note the shift-reduce conflict on state 3 on input $b$, highlighted in the below table.

(2) Is the grammar LALR(1)?
Yes. Refer to state machine in (3). With LALR(1), the items in state 3 will become:

\[ A \rightarrow a \mid c \]
\[ S \rightarrow a \mid b \mid \$

Now, there will be no shift-reduce conflict since the lookahead for the reduction is $c$. 
(3) Is the grammar LR(1)?
Yes, the grammar is LR(1) since it is LALR(1) as shown below and LR(1) is stronger than LALR(1).

c) (10 points) \( \Sigma = \{a, b, c, d\} \).
\[
S \rightarrow Aa \mid bAc \mid Bc \mid bBa
\]
\[
A \rightarrow d
\]
\[
B \rightarrow d
\]

(1) Is the grammar SLR(1)?
No. Note the reduce-reduce conflict on state 5 on inputs \( a \) and \( c \), highlighted below.

(2) Is the grammar LALR(1)?
No. With LALR(1), the two split states will merge into one again:
\[
A \rightarrow d, a / c
\]
\[
B \rightarrow d, a / c
\]
Now we have reintroduced the reduce-reduce conflict.

(3) Is the grammar LR(1)?
Yes. Refer to state machine in (3). With LR(1), state 5 will split into two states with the following items:
\[
A \rightarrow d, a
\]
\[
B \rightarrow d, c
\]
(The state transitioning out of state 0)
\[
A \rightarrow d, c
\]
\[
B \rightarrow d, a
\]
(The state transitioning out of state 4)
Now, there will be no reduce-reduce conflict since the lookaheads for the reductions are all different.