1. Briefly explain the essential difference between
   a) call-by-value and call-by-reference. How are parameters passed in C and Java?

   In call-by-value, the actual parameter is evaluated or copied; its value is placed in the location of
   the corresponding formal parameter of the called procedure. In call-by-reference, the address of
   the actual parameter is passed to the callee as the value of the corresponding formal parameter.
   C and Java use call-by-value.

   b) static scope and dynamic scope. How is scoping done in C and Java?

   Scope specifies the textual region of a program in which there is an active association (binding)
   between a name and the object it represents. Static scoping associates the use of a name with
   the closest lexically enclosing declaration. Dynamic scoping chooses the most recent active
   declaration at runtime. C and Java use static scoping.

2. Java compilation.
   a) Draw a block diagram showing how programs are compiled and executed in Java.
b) What is a Java just-in-time compiler?

The intermediate program from the Java translator is a sequence of architecturally neutral bytecodes that are interpreted by the Java virtual machine. A Java just-in-time compiler translates the bytecodes into an equivalent sequence of native code for the target machine in order to achieve faster run-time performance.

3. Let $L$ be the set of strings of the form $abxba$ where $x$ is a string of $a$'s, $b$'s, and $c$'s that does not contain $ba$ as a substring.

   a) Write a regular expression for $L$.

   Let $R = ab(a|b^*)b+a$.

   b) Show how your regular expression generates the string $ababcba$.

   The prefix $ab$ of $R$ generates the prefix $ab$ of the string. Then $(a|b^*)$ generates $abc$. Finally $b^+$ generates $b$ and the final $a$ of $R$ generates the final $a$ of the string.
c) Construct a deterministic finite automaton for $L$.

![AUTOMATON]

All unspecified transitions are to a dead state.

d) Show how your automaton processes the input $ababcba$.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

4. Consider the context-free grammar $G$:

$$S \rightarrow S + S \mid S \ast S \mid a.$$ 

a) Show that $G$ is ambiguous by constructing all parse trees for $a + a \ast a$.

![PARSE TREES]
b) Construct an unambiguous grammar for $L(G)$ in which $+$ is left associative, $*$ is nonassociative and of higher precedence than $+$.

(1) $S \rightarrow S + T$
(2) $S \rightarrow T$
(3) $T \rightarrow a * a$
(4) $T \rightarrow a$

Draw the parse tree in your grammar for the input string $a + a * a$.

![Parse Tree]

5. Syntax-directed translation.

a) Construct an SDTS that maps postfix expressions containing the digits 0, 1, ... , 9 and the binary operators – and % into equivalent infix expressions.
Here is a SDTS using a synthesized attribute \( E.v \) of type string for the nonterminal \( E \). In the semantic rules, we have used juxtaposition as the concatenation operator.

\[
\begin{align*}
E &\rightarrow E_1 \ E_2 \ \{- \ E.v = "(" E_1.v " - " E_2.v ")"; \} \\
E &\rightarrow E_1 \ E_2 \ % \{- \ E.v = "(" E_1.v " % " E_2.v ")"; \} \\
E &\rightarrow 0 \{- \ E.v = "0"; \} \\
E &\rightarrow 1 \{- \ E.v = "1"; \} \\
E &\rightarrow 2 \{- \ E.v = "2"; \} \\
E &\rightarrow 3 \{- \ E.v = "3"; \} \\
E &\rightarrow 4 \{- \ E.v = "4"; \} \\
E &\rightarrow 5 \{- \ E.v = "5"; \} \\
E &\rightarrow 6 \{- \ E.v = "6"; \} \\
E &\rightarrow 7 \{- \ E.v = "7"; \} \\
E &\rightarrow 8 \{- \ E.v = "8"; \} \\
E &\rightarrow 9 \{- \ E.v = "9"; \} \\
\end{align*}
\]

b) Show how your SDTS translates the expression \( 123-\% \).

Here is an annotated parse tree for \( 123-\% \) with the value of \( E.v \) shown at each node. The output is the infix expression \((1\% (2-3))\), the value of \( E.v \) at the root of the tree..

\[
\begin{align*}
E &\rightarrow E_1 \ E_2 \ % \{- \ E.v = "(" E_1.v " % " E_2.v ")"; \} \\
E &\rightarrow E_1 \ E_2 \ - \{- \ E.v = "(" E_1.v " - " E_2.v ")"; \} \\
E &\rightarrow 0 \{- \ E.v = "0"; \} \\
E &\rightarrow 1 \{- \ E.v = "1"; \} \\
E &\rightarrow 2 \{- \ E.v = "2"; \} \\
E &\rightarrow 3 \{- \ E.v = "3"; \} \\
E &\rightarrow 4 \{- \ E.v = "4"; \} \\
E &\rightarrow 5 \{- \ E.v = "5"; \} \\
E &\rightarrow 6 \{- \ E.v = "6"; \} \\
E &\rightarrow 7 \{- \ E.v = "7"; \} \\
E &\rightarrow 8 \{- \ E.v = "8"; \} \\
E &\rightarrow 9 \{- \ E.v = "9"; \} \\
\end{align*}
\]

c) Modify your SDTS so that it uses the fewest possible number of parentheses in the output.
Here is a modified SDTS that uses two synthesized attributes, \( E.v \) and \( E.p \), for the nonterminal \( E \) in the productions. \( E.v \) is the infix string associated with the nonterminal \( E \) and \( E.p \) is an integer giving the precedence level of the operator associated with \( E \). We assume the precedence level of \( \% \) is 2, and \(-\) is 1. For convenience, we set the precedence level of a digit to 3. To determine whether we need to put parentheses around a subexpression operand, we use the following rule. Suppose we have the parse tree node:

```
\[
E \\
E_1 \\
E_2 \\
\text{op}
\]
```

Then we put parentheses around \( E_1.v \) if \( E_1.p \) is less than the precedence level of \( \text{op} \); we put parentheses around \( E_2.v \) if \( E_2.p \) is less than or equal to the precedence level of \( \text{op} \). Otherwise, we do not add parentheses. The parentheses are there to make sure we evaluate the infix expression in the same order as the postfix expression. Here is the modified SDTS:

\[
E \rightarrow E_1 E_2 \text{ } - \{ \text{ if } (E_2.p == 1) \\
\quad E_2.v = "(\ E_2.v\ )"; \\
\quad E.v = E_1.v "-\" E_2.v; \\
\quad E.p = 1; \}
\]

\[
E \rightarrow E_1 E_2 \text{ } \% \{ \text{ if } (E_1.p == 1) \\
\quad E_1.v = "(\ E_1.v\ )"; \\
\quad \text{if } (E_2.p \leq 2) \\
\quad \quad E_2.v = "(\ E_2.v\ )"; \\
\quad E.v = E_1.v "\%\" E_2.v; \\
\quad E.p = 2; \}
\]

\[
E \rightarrow 0 \quad \{ \ E.v = "0"; \ E.p = 3; \}
\]

\[
E \rightarrow 1 \quad \{ \ E.v = "1"; \ E.p = 3; \}
\]

\[
E \rightarrow 2 \quad \{ \ E.v = "2"; \ E.p = 3; \}
\]

\[
E \rightarrow 3 \quad \{ \ E.v = "3"; \ E.p = 3; \}
\]

\[
E \rightarrow 4 \quad \{ \ E.v = "4"; \ E.p = 3; \}
\]

\[
E \rightarrow 5 \quad \{ \ E.v = "5"; \ E.p = 3; \}
\]

\[
E \rightarrow 6 \quad \{ \ E.v = "6"; \ E.p = 3; \}
\]

\[
E \rightarrow 7 \quad \{ \ E.v = "7"; \ E.p = 3; \}
\]

\[
E \rightarrow 8 \quad \{ \ E.v = "8"; \ E.p = 3; \}
\]

\[
E \rightarrow 9 \quad \{ \ E.v = "9"; \ E.p = 3; \}
\]
Here is the annotated parse tree for the input 123-% using the modified SDTS: