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 \( R = ab(a|b^*c)^*b+a \).

   b) **Show how your regular expression generates the string \( ababcba \).**

   The prefix \( ab \) of \( R \) generates the prefix \( ab \) of the string. Then \( (a|b^*c)^* \) 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 $L$.

![Deterministic finite automaton](image)

All unspecified transitions are to a dead state.

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

4. Consider the context-free grammar $G$: $S \rightarrow S + S \mid S * S \mid a$.

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

![Parse trees](image)
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 $a + a * a$.

```
  S
 / \   /
S +  T
 /   /   /
T a * a
 /   /
 a a
```

c) Construct an SLR(1) parsing table for your grammar.

The sets of LR(0) items for the augmented grammar are:

$I_0 : S' \rightarrow \cdot S$
$I_1 : S' \rightarrow S$.  \hspace{1cm} $I_2 : S \rightarrow T$.

$S \rightarrow \cdot S + T$
$S \rightarrow \cdot T$
$T \rightarrow \cdot a * a$
$T \rightarrow \cdot a$
The parsing table is constructed from these sets of items. Each set of items corresponds to a state. The FOLLOW sets for the nonterminals are

<table>
<thead>
<tr>
<th>NONTERMINAL</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>+ $</td>
</tr>
<tr>
<td>T</td>
<td>+ $</td>
</tr>
</tbody>
</table>

The action and goto tables are shown below. Blank entries are errors. Note all entries are uniquely defined so the grammar is SLR(1), and hence unambiguous.

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>acc</td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>s5</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
</tr>
</tbody>
</table>

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.

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

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.
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
   /\    \
E1  E2  op
```

Then we put parentheses around $E_1.v$ if $E_1.p$ is less than the precedence level of $op$; we put parentheses around $E_2.v$ if $E_2.p$ is less than or equal to the precedence level of $op$. 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 → E; E2 −  { if (E2.p == 1)  
             E2.v = "(" E2.v ");
             E.v = E1.v "−" E2.v;
             E.p = 1; }  

E → E; E2 %  { if (E1.p == 1)  
             E1.v = "(" E1.v ");
             if (E2.p ≤ 2)  
                 E2.v = "(" E2.v ");
             E.v = E1.v "%" E2.v;
             E.p = 2; }  

E → 0      { E.v = "0"; E.p = 3; }  
E → 1      { E.v = "1"; E.p = 3; }  
E → 2      { E.v = "2"; E.p = 3; }  
E → 3      { E.v = "3"; E.p = 3; }  
E → 4      { E.v = "4"; E.p = 3; }  
E → 5      { E.v = "5"; E.p = 3; }  
E → 6      { E.v = "6"; E.p = 3; }  
E → 7      { E.v = "7"; E.p = 3; }  
E → 8      { E.v = "8"; E.p = 3; }  
E → 9      { E.v = "9"; E.p = 3; }
```
Here is the annotated parse tree for the input 123-% using the modified SDTS: