1. Consider the context-free grammar \( G = (\{S\}, \{a\}, S, P) \):

\[
S \rightarrow SS^+ | SS^* | a
\]

a. Show how the string \( aa + a^* \) can be generated by this grammar.

\[
S \rightarrow SS^* \rightarrow Sa^* \rightarrow SS + a^* \rightarrow Sa + a^* \rightarrow aa + a^*
\]

b. Construct a parse tree for this string.

```
          S
         /|
        / | S
       /  | S
      /   | a
     /    |  *
    /     a
   /      a
```

c. What language does this grammar generate? Justify your answer!

All strings that start with 2 aa’s and that consist of substrings of \( a^* \) and \( a^+ \)
2. Write an unambiguous grammar that leads to correct parse trees for the language consisting of expressions involving the operand id and the operators described below.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Precedence</th>
<th>Binary / Unary</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Left to right</td>
<td>Highest (4)</td>
<td>Binary</td>
</tr>
<tr>
<td>@</td>
<td>Right to left</td>
<td>High (3)</td>
<td>Unary</td>
</tr>
<tr>
<td>^</td>
<td>Right to left</td>
<td>Medium (2)</td>
<td>Binary</td>
</tr>
<tr>
<td>#</td>
<td>Left to right</td>
<td>Lowest (1)</td>
<td>Binary</td>
</tr>
</tbody>
</table>

Parentheses are also allowed, with their usual interpretation.

Solution:

\[
\begin{align*}
A_1 & \rightarrow A_i \# A_2 \mid A_2 \\
A_2 & \rightarrow A_i \wedge A_2 \mid A_3 \\
A_3 & \rightarrow @ A_i \mid A_i \\
A_i & \rightarrow A_i ! A_i \mid A_i \\
A_i & \rightarrow \text{id} \mid (A_i)
\end{align*}
\]

3. The following grammar contains occurrences of left recursion. Rewrite it so that there is no left recursion. Also do left factoring to remove rules for a single non-terminal that start with the same sequence of symbols.

\[
G = (\{A_1, A_2, A_3\}, \{a, b\}, A_1, P)
\]

where the production rules \( P \) are:

\[
\begin{align*}
A_1 & \rightarrow A_1aa \mid A_2b \\
A_2 & \rightarrow A_1aa \mid A_2A_i b \\
A_3 & \rightarrow aA_i b \mid ab
\end{align*}
\]
Solution:

First, remove the immediate left recursion in $A_1$ and $A_2$ by transforming it into right recursion:

\[
A_1 \rightarrow A_2bR_1 \\
R_1 \rightarrow aaR_1 | \epsilon \\
A_2 \rightarrow A_2aaR_2 \\
R_2 \rightarrow A_2bR_2 | \epsilon \\
A_3 \rightarrow aA_3b | ab
\]

There is still a left recursion involving a two-step derivation: $A_2 \Rightarrow A_2bR_1aaR_2$. An algorithm for handling these kinds of non-immediate left recursions is described in the “Systems Software Knights” textbook on pages 285 – 286.

In essence, we keep rewriting the grammar until we only have immediate left recursions left:

\[
A_2 \rightarrow A_2bR_1aaR_2 \\
A_1 \rightarrow A_2bR_1
\]

Now simply transform the remaining recursion:

\[
A_2 \rightarrow R_3 \\
R_3 \rightarrow bR_1aaR_2R_3 | \epsilon
\]

Then, do the left factoring. The only non-terminal with multiple productions with a common prefix is $A_3$, so this rule will have to change to:

\[
A_3 \rightarrow aA_3^* \\
A_3^* \rightarrow A_3b | b
\]

The final transformed grammar looks like this:

\[
A_1 \rightarrow A_2bR_1 \\
R_1 \rightarrow aaR_1 | \epsilon \\
A_2 \rightarrow R_3 \\
R_3 \rightarrow bR_1aaR_2R_3 | \epsilon \\
R_2 \rightarrow A_2bR_2 | \epsilon \\
A_3 \rightarrow aA_3^* \\
A_3^* \rightarrow A_3b | b
\]