1. Prove that the grammar G given below is ambiguous by giving two different parse trees of the string \textit{iiaea}. Both of those parse trees are correct, but only one of them is consistent with the usual rule for resolving ambiguity of if-then-else statements. Which one?

\begin{align*}
1. & \quad S \rightarrow a \\
2. & \quad S \rightarrow iS \\
3. & \quad S \rightarrow iSeS
\end{align*}

Using the usual rule of disambiguation, the tree on the left is the correct one.

2. Solve problems given in the handout \textit{finiteAutomata.pdf} associated with the following figures.

   (a) (Figure 7 of the handout.) Draw a minimal DFA equivalent to the NFA shown in (a). Your answer should be the same as (b).

\begin{align*}
Q &= \{0, 1, 2, 3\}, \text{ thus the states of the derived DFA are } 2^Q, \text{ all subsets of } Q. \text{ Here is the transition table of that DFA, where the the states of the DFA, all the subsets of } Q \text{ are listed in canonical order. Of the 16 members of } 2^Q, \text{ only five are reachable. The start state is } 01. \text{ In (b) We do not show the dead state, the empty set. No two reachable states of the DFA are equivalent.}
\end{align*}
(b) (Figure 8 of the handout.) Give a minimal DFA equivalent to this NFA.

We use the same technique we used for the previous problem. All 8 members of $2^Q$ are reachable, and there are no pairs of equivalent states of the DFA. We omit the dead state $\emptyset$ in the figure below.

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>2</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>01</td>
<td>012</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>02</td>
<td>01</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>012</td>
<td>012</td>
<td>02</td>
<td>12</td>
</tr>
</tbody>
</table>

(c) (Figure 10 of the handout.) Write a regular expression for the language accepted by this NFA.

$$a(b + aa)^*$$

(d) (Figure 12 of the handout.) Give a minimal DFA equivalent to this NFA.

There are only four reachable states of the DFA, one of which is the dead state. No pair is equivalent.
(e) (Figure 13 of the handout.) Give a grammar for the language accepted by this NFA.

\[
S \rightarrow bS \mid aA \mid bA \mid aB \mid cB \\
A \rightarrow aA \mid cB \\
B \rightarrow aB \mid cB \mid bS \mid \lambda
\]

3. Write (do not prove) the pumping lemma for regular languages. (You will need to do this on the first examination, so try to understand it as you write it.)

If \( L \) is a regular language there exists a number \( p \), called a pumping length of \( L \), such that, for any \( w \in L \) of length at least \( p \), there exist strings \( x, y, z \) such that the following four statements hold:

1. \( w = xyz \)
2. \( |xy| \leq p \)
3. \( |y| \geq 1 \)
4. For any integer \( i \geq 0 \), \( xy^iz \in L \).

4. Please help me make a very important decision! As you know, the question of whether there is a \( \mathcal{P} \)-time algorithm which finds the factors of an integer is one of the most important unsolved problems in computation theory. In fact, the security of RSA encryption, which is used many times every day, depends on the hypothesis that no such algorithm exists.

After many years of work, I have found an algorithm for the problem. Here is my code:

```cpp
void primefactors(int n) // lists prime factors of n separated by commas
{
    int f = 2;
    while (n%f != 0) f++;
    if(f == n) cout << n << endl; // n is prime
    else if(n%f == 0) // f is a prime factor of n
    {
        cout << f << "",
        primefactors(n/f);
    }
}
```

You can easily see that the running time of the code is \( O(n) \), which is certainly polynomial!

I could either publish this result and become insanely famous, or keep it to myself (I'm sure you won't tell) and become insanely rich breaking RSA encryption for enormous fees. What should I do?

An algorithm is in the class \( \mathcal{P} \)-time if its time complexity is bounded by a polynomial function of the number of bits of input. The number of bits of a the program is \( \Omega(\log n) \). The time complexity of the program is \( \Omega(n) \), which is an exponential function of the number of bits, since \( n = 2^{\log_2 n} \). Realizing this, I canceled my world tour.