1. (Regular languages)

(a) Here is an equivalent DFA, constructed by the subset construction.

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow { p } )</td>
<td>( { q, s } )</td>
<td>( { q } )</td>
</tr>
<tr>
<td>( * { q, s } )</td>
<td>( { r } )</td>
<td>( { q, r } )</td>
</tr>
<tr>
<td>( * { q } )</td>
<td>( { r } )</td>
<td>( { q, r } )</td>
</tr>
<tr>
<td>( { r } )</td>
<td>( { s } )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>( * { q, r } )</td>
<td>( { r, s } )</td>
<td>( { p, q, r } )</td>
</tr>
<tr>
<td>( * { s } )</td>
<td>( \emptyset )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>( * { r, s } )</td>
<td>( { s } )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>( * { p, q, r } )</td>
<td>( { q, r, s } )</td>
<td>( { p, q, r } )</td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

(I see no easy way to define this language, except by a rather complex regular expression.)

(b) Eliminating state \( r \) gives:

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( a + ba*b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow * { p } )</td>
<td>( s )</td>
<td>( p )</td>
<td>( a + ba*b )</td>
</tr>
<tr>
<td>( q )</td>
<td>( p )</td>
<td>( s )</td>
<td></td>
</tr>
<tr>
<td>( s )</td>
<td></td>
<td>( q )</td>
<td></td>
</tr>
</tbody>
</table>

Eliminating state \( q \) gives:

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( (a + ba*b)b )</th>
<th>( (a + ba*b)a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow * { p } )</td>
<td>( s )</td>
<td>( p )</td>
<td>( (a + ba*b)b )</td>
<td>( (a + ba*b)a )</td>
</tr>
<tr>
<td>( s )</td>
<td></td>
<td>( s )</td>
<td>( p )</td>
<td></td>
</tr>
</tbody>
</table>

Eliminating state \( s \) gives:

<table>
<thead>
<tr>
<th></th>
<th>( b + a((a + ba<em>b)b)(a + ba</em>b)a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow * { p } )</td>
<td>( p )</td>
</tr>
</tbody>
</table>

Transforming this to a regular expression gives:

\((b + a((a + ba*b)b)(a + ba*b)a)^*\)

If the states were eliminated in the order \( r, s, q \), the result would be the equivalent regular expression:

\((b + a((a + ba*b)b)(a + ba*b)a)^*\)

(Again I see no easy way to define this language, except by a rather complex regular expression.)
(c) Suppose \( L_1 \) is regular. Let \( n \) be the pumping lemma constant. Let \( w = a^n b^n \). By the pumping lemma, \( w = xyz \), where (a) \( |xy| \leq n \), (b) \( |y| \geq 1 \), and (c) for all \( k \geq 0 \), \( xy^k z \in L_1 \). By (a) and (b), \( y = a^p \) for \( p \geq 1 \). By the pumping lemma, \( xy^2z = a^{n+p} b^n \in L_1 \). But \( n + p \not< n \), so \( xy^2z \not\in L_1 \). This is a contradiction, so \( L_1 \) cannot be regular.

2. (Context-free languages)

(a) This should be fairly straightforward. First define \( L = \{ a^i b^i \mid i \geq 0 \} \), then add one or more \( b \)'s, then add any number of \( c \)'s.

\[
\begin{align*}
S & \rightarrow ABC \\
A & \rightarrow \epsilon \mid aAb \\
B & \rightarrow b \mid bB \\
C & \rightarrow \epsilon \mid cC
\end{align*}
\]

(b) The grammar generates all strings of \( a \)'s and \( b \)'s such that every prefix has at least as many \( a \)'s as \( b \)'s.

The language is ambiguous because the string \( aab \) has two distinct leftmost derivations:

\[
\begin{align*}
S & \rightarrow aS \Rightarrow aaSbS \Rightarrow aabS \Rightarrow aab \\
S & \rightarrow aSbS \Rightarrow aaSbS \Rightarrow aabS \Rightarrow aab
\end{align*}
\]

This ambiguity can also be demonstrated by drawing a parse tree corresponding to each of the derivations.

(c) The key observation required to ensure the PDA is deterministic is to treat the cases of pushing a left parenthesis onto an empty stack differently from pushing one onto a nonempty stack.

The PDA has states \( \{ p, q, r \} \), starting state \( a \), final states \( \{ p, r \} \), stack alphabet \( \{ X, Z_0 \} \), and transition function \( \delta \) defined as follows:

\[
\begin{align*}
\delta(p, '(', Z_0) &= (q, Z_0) \\
\delta(q, '(', Z_0) &= (q, XZ_0) \\
\delta(q, '(', X) &= (q, XX) \\
\delta(q, ')', X) &= (q, \epsilon) \\
\delta(q, ')', Z_0) &= (r, \epsilon) \\
\delta(r, '(', Z_0) &= (q, Z_0)
\end{align*}
\]

(d) Suppose \( L \) is defined by a context-free grammar with rules of the form:

\[
a_i \rightarrow A_{i1} A_{i2} \ldots A_{im_i}
\]

Define a new grammar by reversing the symbols in the right hand side of each such rule:

\[
a_i \rightarrow A_{im_i} A_{im_{i-1}} \ldots A_{i1}
\]

Clearly, by induction on the depth of the parse tree for a given string, the new grammar defines \( L^R \).
3. (Turing machines and computability)

(a) Given an instance \((P, I)\) of the *hello-world* problem, modify program \(P\) to store its input in a (dynamically allocated) array. Further modify \(P\) so that, after each output statement, it checks whether the first 12 characters are "hello, world". If they are, the program should halt. Otherwise, if 12 or more characters have been printed, the program should enter an infinite loop. Further, every (implicit) exit statement in \(P\) should be replaced by another infinite loop. Call the resulting program \(P'\).

Now, program \(P'\) applied to input \(I\) halts if and only if program \(P\) applied to input \(I\) prints "hello, world" as its first 12 characters of output. If we could decide whether or not an arbitrary program \(P'\) applied to input \(I\) halts we could the above procedure to decide whether or not an arbitrary program \(P\) applied to \(I\) prints "hello, world" as its first 12 characters of output. But this is impossible, and hence it is impossible to decide whether an arbitrary program \(P'\) halts on an arbitrary input \(I\). (I.e., the *halting* problem is undecidable.)

(b) A solution from the text by Martin was shown in lectures. Pseudocode for a Turing machine to recognise this language could have the following form:

while true
    if a
        change to B and move right
        while a or b move right
        move L  // if b fail
        if B accept
            if a change to A and move left
                while a or b move left
                move right
            else if b  // similarly, with a and b reversed
                else
                    accept

(c) A *recursive language* (now called a *computable language*) is a language accepted by a Turing machine *that always terminates*.

A *recursively enumerable* (now called a *computably enumerable language*) is a language accepted by a Turing machine (that may not terminate on inputs not in the language).

(See also Theorem 9.3 and its proof.)

(d) Let \(w_1, w_2, \ldots\) be an enumeration of the binary strings. Let \(M_1, M_2, \ldots\) be an enumeration of the Turing machines, where \(M_i\) is encoded as \(w_i\). Define the diagonalisation language \(L_d\) as follows:

\[ L_d = \{ w_i \mid M_i \text{ does not accept } w_i \} \]

Then \(L_d\) is not r.e. (now called c.e.).
4. (Complexity and intractability)

(a) \( \mathcal{NP} \) is the set of problems (or languages) that are accepted by a nondeterministic Turing machine with time-complexity a polynomial function of the length of the input. The time-complexity of a nondeterministic Turing machine \( M \) is the maximum number of steps in any computation of \( M \).

(b) Problem \( L_1 \) is reducible to problem \( L_2 \) in polynomial time if there exists a deterministic Turing machine \( M \) (or, equivalently, an algorithm \( A \)) that maps instances of \( L_1 \) to instances of \( L_2 \) such that

- \( M \) has polynomial time-complexity, and
- for all strings \( w \), \( M(2) \in L_1 \) if and only if \( w \in L_1 \).

(c) Define \( \mathcal{NP} \)-complete problems. A problem \( L \) is \( \mathcal{NP} \)-complete if

- \( L \in \mathcal{NP} \), and
- for every language \( L' \in \mathcal{NP} \), \( L' \) is polynomially reducible to \( L \).

(d) See Theorem 10.4 and its proof (IALC, pp.422–423). Note that \( P_2 \) must belong to \( \mathcal{NP} \) for the result to hold.

(e) This question was rather simple. There is a unique satisfying truth assignment: \( \{ x \rightarrow 1, y \rightarrow 1, z \rightarrow 1 \} \).

(f) Recall the polynomial-time reduction from the problem 3-SAT to the independent set problem (IS). Consider the following instance of 3-SAT.

\[
E_2 = (x + y + z)(\overline{x} + z + w)(x + y + w)(x + \overline{z} + \overline{w})
\]

Show the corresponding instance of IS to which \( E_2 \) is reduced. The corresponding instance consists of a graph \( G \) and an integer \( k = 4 \) (the number of clauses in \( E_2 \)). Graph \( G \) has 12 nodes, one for each literal in \( E_2 \). Each pair of nodes corresponding to a literal in a single clause are connected by an edge, e.g., there are edges between nodes \( x, y \) and \( z \). Each pair of nodes corresponding to complementary literals are connected by an edge, e.g., there is an edge between the literal \( \overline{z} \) in clause 1 and the literal \( z \) in clause 2.