1. (a) \( 0^*(10^+)^*(\epsilon + 1 + 11(0^+1)^0^*) \) or \( 0^*(10^+)^*(\epsilon + 1 + 11(0^+1)^*) \)

(b) We have to eliminate states \( q \) and \( s \). Suppose we eliminate state \( s \) (without a loop) first. This gives the (generalised) automaton:

\[
\begin{array}{c|cc|cc}
   & a & b & (a+b)a & (a+b)b \\
\hline
   \rightarrow p & p & q & & \\
   q & r & q & & \\
   * r & s & s & q & p \\
\end{array}
\]

Eliminating \( q \) gives the (generalised) automaton:

\[
\begin{array}{c|cc|cc}
   & a & bb^*a & (a+b)ab^*a & (a+b)b \\
\hline
   \rightarrow p & p & r & & \\
   * r & s & s & q & p \\
\end{array}
\]

Converting this to a regular expression gives

\[(a + bb^*a((a+b)ab^*a)^*(a+b)b)bb^*a((a+b)ab^*a)^*)\]

or

\[a^*bb^*a((a+b)ab^*a + (a+b)ba^*bb^*a)^*\]

Of course, we can simplify \( bb^* \) to \( b^+ \).

Eliminating state \( q \) first would give a different solution.

(c) **Proof 1.** Assume that the alphabet is \( \Sigma \). We use induction on the construction of regular expression (RE) to prove that, for any RE \( E \), there is always a RE \( E^R \) for \( L(E)^R : L(E)^R = L(E^R) \). Here \( L(E) \) is the regular language defined by \( E \).

Basis: In this case, \( L = \emptyset, L = \{\epsilon\}, L = \{a\} \) for \( a \in \Sigma \) or \( L = \Sigma \). In each case, it is immediate that \( L^R = L \) and thus \( L^R \) is regular.

Induction: Assume that \( E \) and \( F \) are RE.

Case 1. \( L(E + F)^R = L(E)^R \cup L(F)^R \) is regular.

Case 2. \( L(EF)^R = (L(E)L(F))^R = (L(F)^R)(L(E)^R) \) is regular.

Case 3. \( L(E^*)^R = (L(E)^*)^R = (L(E)^R)^* \) is regular.

(d) See textbook page 152.

(e) Yes, they define the same class of languages.
2. (a) A CFG $G = (V, T, P, S)$ is ambiguous if there is a string $w$ in $T^*$ s.t. $w$ has two different parse trees (or, equivalently, if $w$ has two different leftmost derivations).

The language defined by the given CFG is $L = \{ a^m b^n c^p \mid m = n \land n = p \}$.

The CFG is ambiguous since the string $abc$ has to different parse trees, one in which there are the same number of occurrences of $a$ and $b$, and the other in which there are the same number of occurrences of $b$ and $c$.

(b) Here is a DFA which accepts the language by empty stack:

$$P = (\{q_0, 0, 1\}, \{0, Z_0\}, \delta, q_0, Z_0)$$

where

$$\delta(q_0, 0, Z_0) = (q_0, 0Z_0)$$

$$\delta(q_0, 0, 0) = (q_0, 00)$$

$$\delta(q_0, 1, 0) = (q_0, \epsilon).$$

(c) See the textbook page 286-287

(d) Suppose $L_1$ is context-free. Let $n > 0$ be the pumping lemma constant for $L_1$, and choose $z = a^n b^n c^n$. We may write $z = uvwxy$ s.t. $|vwx| \leq n$ and $vx \neq \epsilon$.

Then we know that $vwx$ cannot involve both $a$ and $c$ since the last $a$ and the first $c$ are separated by $n + 1$ positions. Thus there two possible cases:

Case 1. $vwx$ has no $a$'s: Then $vx$ contains only $b$'s and $c$'s and is not empty. By the pumping lemma, $z' = uv^n wx y \in L_1$. However, $z'$ has $n$ $a$'s but more than $n$ $b$'s. This implies $z' \not\in L_1$, contradiction.

Case 2. $vwx$ has no $c$'s: Then $vx$ contains only $a$ and $b$. Thus $uwxy$ contains $n$ $c$'s but fewer $a$'s or fewer $c$'s, contradiction.

As every case leads to a contradiction, we conclude that $L_1$ is not context-free.

(e) No, the class of DPDA defines a strictly smaller class of languages.

3. (a) We reduce the hello-world problem to the (non)empty-output problem as follows.

Let $(P, I)$ be an instance of the hello-world problem. Modify $P$ as follows:

Replace all print statements by statements that append the output strings to an (infinite) list of characters. After every output, check whether the first 12 characters of the list are "hello, world". If they are, print "something" to the output device. Call the resulting program $P'$.

Then $(P, I)$ prints "hello, world" if and only if $(P', I)$ prints anything at all.

So if the (non)empty-output problem were decidable, we could use it (via the above reduction) to decide the hello-world problem. But the hello-world problem is undecidable, so so is the (non)empty-output problem.

(b) Here is a high level description of a Turing machine that doubles a string of $a$'s.

Change all $a$'s to $A$'s.

Return to the first $A$.

While the current symbol is $A$:

Change the $A$ to an $a$ and move right to the first blank.

Change the blank to an $a$. 
Move left until reaching an $A$.
Move left until reaching an $a$.
Move right.
Halt.

Each time we are at the start of the main loop, the tape has the form $a^n | A^m a^n$, where $|$ indicates the position of the tape head.
(For full marks a detailed state transition diagram is required.)

(a) See Section 8.4.4 of the text.

(b) A language is computable if it is the language accepted by some Turing machine that always halts.
A language is computably enumerable if it is the language accepted by some Turing machine.
See the proof of Theorem 9.4 on pp.376–377 of the text.

(c) See the proof of Theorem 9.2 on p.372 of the text.
See the lecture notes on Computability for a slight rewording of the proof.

(d) They define the same class of languages.

4. (a) $\mathcal{NP}$ is the class of languages $L$ such that, for some nondeterministic Turing machine $M$ and polynomial time complexity $T(n)$, $L = L(M)$ and, for every input of length $n$, every computation of $M$ takes at most $T(n)$ steps.

(b) $L_1$ is polynomially reducible to $L_2$ ($L_1 \alpha L_2$) if there exists a polynomial-time algorithm that transforms any instance $I_1$ of $L_1$ to an instance $I_2$ of $L_2$ such that $I_1 \in L_1$ if and only if $I_2 \in L_2$.
If $L_1 \alpha L_2$, we can say that $L_2$ is at least as hard as $L_1$, since, if we could solve $L_2$ in polynomial time, then we could also solve $L_1$ in polynomial time.

(c) A language $L$ is $\mathcal{NP}$-complete if (a) $L$ is in $\mathcal{NP}$, and (b) every language in $\mathcal{NP}$ is polynomially reducible to $L$.
If $L$ is $\mathcal{NP}$-complete, we can say that $L$ is in $\mathcal{NP}$ and is at least as hard as every language in $\mathcal{NP}$, so if we could solve $L$ in polynomial time, we could solve every language in $\mathcal{NP}$ in polynomial time.

(d) Two satisfying truth assignments are $\{x = 1, y = 1, z = 0\}$ and $\{x = 0, y = 1, z = 1\}$.

(e) This requires applying the transformation described in Section 10.3.3, in the proof of Theorem 10.13.
The resulting expression is:
\[
E'_2 = (x + y + p)(x + p + \overline{q})(x + p + q)(\overline{x} + y + r)(\overline{x} + y + r)(x + y + r)(\overline{x} + y + r)(\overline{x} + y + r)(\overline{x} + y + r)(\overline{x} + y + r)(\overline{x} + y + r)
\]
See pp.442–443 of the text for the proof.

(f) To show Subgraph Isomorphism (SI) is $\mathcal{NP}$-complete, we need to show that SI is in $\mathcal{NP}$ and that some $\mathcal{NP}$-complete problem, in this case CLIQUE, is polynomially reducible to SI.
It is clear that SI is in \( \mathcal{NP} \), since if we are given a proposed isomorphism from \( G \) to a subgraph of \( H \), it is easy to check in polynomial time whether or not it is an isomorphism. (A (graph) isomorphism from \( G \) to \( H \) is a 1-1 onto mapping from the nodes of \( G \) to the nodes of \( H \) that preserves the connectivity between nodes.)

To show that CLIQUE \( \alpha \) SI, let \((G, k)\) be an instance of CLIQUE. Let \( K_k \) be the complete graph on \( k \) nodes, and let \((K_k, G)\) be an instance of SI. As \( K_k \) has \( k \) nodes and less than \( k^2 \) edges, it can be constructed in polynomial time. Then it is immediate from the definitions that \( G \) has a clique of size \( k \) if and only if there is an isomorphism from \( K_k \) to a subgraph (the clique) of \( G \).