(1)

(a)

[The procedure for converting a DFA to a CFG on page 107 of the textbook can easily be reversed to generate
an NFA from a regular grammar. Here are the steps involved:

1. For each variable \( A \) in the regular grammar add a state \( q_A \) in the NFA.

2. Add a final state \( F \).

3. For a production of the form \( A \rightarrow bC \), add a transition from state \( q_A \) to state \( q_B \) on symbol \( b \).

4. For a production of the form \( A \rightarrow C \), add an \( \epsilon \)-transition from state \( q_A \) to \( q_C \).

5. For a production of the form \( A \rightarrow b \), add a transition from state \( q_A \) to the final state \( F \).

6. For a production of the form \( A \rightarrow \epsilon \), mark \( A \) as one of the final states.

It can be shown that the NFA constructed in this way is equivalent to the original regular grammar, i.e.
they recognize the same language. We use this equivalence repeatedly in the following proofs.]

(b)

Let \( L = \{ \langle G, A \rangle : G \text{ is a regular grammar and } A \text{ is an unreachable variable in } G \} \). \( L \) is decidable. Here
is a description of a Turing machine \( M \) that decides \( L \):

\[ M = \text{"On input } \langle G, A \rangle \text{, where } G \text{ is a regular grammar and } A \text{ is a variable in } G:\]

1. Convert the regular grammar \( G \) into an equivalent NFA \( N \) as above, and let \( q_A \) be the state corresponding to the variable \( A \).

2. Run breadth first search for directed graphs on the transition diagram of \( N \) to find out if there is
a directed path from the start state of \( N \) to the state \( q_A \). Reject if there is one such path; accept otherwise.”

The justification for part 2 is similar to that of part (a).
(c)

Let \( L = \{ \langle G, A \rangle : G \text{ is a regular grammar and } A \text{ is a redundant variable in } G \} \). \( L \) is decidable. Here is a description of a Turing machine \( M \) that gives a mapping reduction from \( L \) to \( EQ_{NFA} \):

\[
M = \text{"On input } \langle G, A \rangle , \text{ where } G \text{ is a regular grammar and } A \text{ is a variable in } G:\]

1. Convert the regular grammar \( G \) into an equivalent NFA \( N_1 \).
2. Obtain a new regular grammar \( G' \) from \( G \) by deleting all productions in which \( A \) appears. Convert \( G' \) into an equivalent NFA \( N_2 \).
3. Output \( \langle N_1, N_2 \rangle \)."

This establishes that \( L \leq_m EQ_{NFA} \). Because \( EQ_{NFA} \) is decidable, so is \( L \).

(2)

(a)

Let \( L = \{ \langle G, A \rangle : G \text{ is a CFG and } A \text{ is an unreachable variable in } G \} \). \( L \) is decidable. Here is a Turing machine \( M \) that decides \( L \):

\[
M = \text{"On input } \langle G, A \rangle , \text{ where } G \text{ is a CFG and } A \text{ is a variable in } G:\]

1. Mark the start variable \( S \) of \( G \).
2. Repeat until no new variables get marked:
   (a) Mark variable \( C \) where \( G \) has a rule \( B \rightarrow w \), where \( w \) contains \( C \), and \( B \) is already marked.
3. If \( A \) is not marked then accept; otherwise, reject.”

(b)

Let \( L = \{ \langle G, A \rangle : G \text{ is a CFG and } A \text{ is an unproductive variable in } G \} \). \( L \) is decidable. Here is a Turing machine \( M \) that decides \( L \):

\[
M = \text{"On input } \langle G, A \rangle , \text{ where } G \text{ is a CFG and } A \text{ is a variable in } G:\]

1. Mark all terminal symbols of the grammar \( G \).
2. Repeat until no new variables get marked:
   (a) Mark variable \( B \) where \( G \) has a rule \( B \rightarrow U_1U_2\ldots U_k \), and all \( U_1, U_2, \ldots, U_k \) are already marked.
3. If \( A \) is not marked then accept; otherwise, reject.”

(c)

Let \( L = \{ \langle G, A \rangle : G \text{ is a CFG and } A \text{ is a redundant variable in } G \} \). \( L \) is undecidable. To prove that we show that \( ALL_{CFG} \leq_m L \).

Consider an instance \( \langle G_1 \rangle \) with start variable \( S_1 \). Form a CFG \( G_2 \) with start variable \( S_2 \) (not a variable of \( G_1 \)) having productions \( S_2 \rightarrow aS_2 \) for \( a \in \Sigma \), and \( S_2 \rightarrow \epsilon \). Note that \( L(G_2) = \Sigma^* \). Then we construct a new CFG \( G \) by adding a new start variable \( S \) to the union of \( G_1 \) and \( G_2 \), along with the production \( S \rightarrow S_1|S_2 \).

Then \( \langle G, S_2 \rangle \in L \) if and only if \( \langle G_1 \rangle \in ALL_{CFG} \), as follows. If there is a string \( w \in \Sigma^* \setminus L(G_1) \) then \( S_2 \Rightarrow^* w \) so \( S_2 \) is not redundant in \( G \). If no such \( w \) exists, then \( L(G_1) = L(G) = \Sigma^* \) so \( S_2 \) is redundant in \( G \).

Hence because \( ALL_{CFG} \) is undecidable, so is \( L \).
(3)

(a)
Let \( AP = \{ \langle P \rangle : P \) is an aperiodic instance of the PCP\}. We prove that \( AP \) is not Turing recognizable by showing \( \overline{A_{TM}} \leq_m AP \). On input \( \langle M, w \rangle \) we modify the Turing machine \( M \) such that when \( M \) rejects \( w \), instead of halting it loops. We do this by adding some additional states and transitions to the Turing machine. Now following the steps of the reduction of \( A_{TM} \) to \( \text{MPCP} \) given in the textbook, we see that when \( M \) accepts \( w \) the resulting PCP instance is periodic (i.e. it is not aperiodic). On the other hand if \( M \) does not accept \( w \), \( M \) loops and we get a computation history that is infinitely long. This infinite computation history ensures that we can find an infinite sequence of tiles satisfying the definition of an aperiodic instance.

(b)
Let \( AP = \{ \langle P \rangle : P \) is an aperiodic instance of the PCP\}. The following Turing machine \( M \) recognizes \( \overline{AP} \):

\[
M = \begin{cases} 
\text{On input } \langle P \rangle : \\
1. \text{Initialize a list } L \text{ of sequences with the null sequence.} \\
2. \text{For each sequence } I \text{ in the list } L:\n   \quad (a) \text{Remove } I \text{ from } L. \\
   \quad (b) \text{For each tile } \left[ \frac{a_i}{b_i} \right], 1 \leq i \leq l, \text{ check} \\
   \quad \quad \text{i. If adding this tile to the sequence } I \text{ results in a match then accept. (} \langle P \rangle \text{ is a periodic instance).} \\
   \quad \quad \text{ii. After adding this tile to } I \text{ if concatenation of the top parts of the tiles is a prefix of the} \\
   \quad \quad \text{concatenation of the bottom parts of the tiles, or vice versa, then concatenate } i \text{ to a copy of} \\
   \quad \quad \text{the sequence } I, \text{ and add this new sequence to the list } L. \\
3. \text{If the list } L \text{ is empty then accept. (} \langle P \rangle \text{ is neither periodic nor aperiodic); otherwise go to step 2.} 
\end{cases}
\]

Instance \( \langle P \rangle \) is periodic, aperiodic, or cannot generate any infinite matching sequence. If it is periodic, eventually \( M \) finds a match and accepts because \( \langle P \rangle \notin \overline{AP} \). If it admits no infinite matching sequence, some longest possible entry on the list must exist, and eventually \( M \) exhausts its list and accepts, again because \( \langle P \rangle \notin \overline{AP} \). If it is aperiodic, no periodic solution can be found, and the list cannot be exhausted, so \( M \) runs forever.

(4)
Assume (to the contrary) that \( \rho(n) \) is Turing computable. Let \( M \) be a Turing machine that computes the function \( \rho \). Construct the following Turing machine:

\[
B = \begin{cases} 
\text{On input } w: \\
1. \text{Obtain, via recursion theorem, own description } \langle B \rangle. \\
2. \text{Let } n \text{ be the number of states in } \langle B \rangle. \\
3. \text{Apply } M \text{ to compute } \rho(n). \\
4. \text{Write some (non-blank) symbol on the first } \rho(n) + 1 \text{ cells of the tape.} \\
5. \text{Halt.} 
\end{cases}
\]

\( B \) is a Turing machine with \( n \) states that leaves \( \rho(n) + 1 \) non-blank cells on the tape on termination, a contradiction. So \( \rho(n) \) is not Turing computable.

(If you want more information about problems of this type, look for “busy beaver” Turing machines.)
Let $P$ be any non-trivial property of the language of a Turing machine. Assume (to the contrary) that $R$ is a Turing machine that decides $P$. Since $P$ is non-trivial there is a Turing machine $T_1$ such that $\langle T_1 \rangle \in P$, and a Turing machine $T_2$ such that $\langle T_2 \rangle \notin P$. Now construct a Turing machine $S$:

$$S = \text{"On input } w: \text{"}$$

1. Obtain, via recursion theorem, own description $\langle S \rangle$.
2. Run $R$ on $\langle S \rangle$.
3. If $R$ accepts then run $T_2$ on $w$, and accept if $T_2$ does.
4. If $R$ rejects then run $T_1$ on $w$, and accept if $T_1$ does.

So $S$ simulates $T_2$ when $R$ accepts $\langle S \rangle$, and $T_1$ when $R$ rejects $\langle S \rangle$, i.e. \( L(S) = L(T_2) \) when $\langle S \rangle \in P$, and \( L(S) = L(T_1) \) when $\langle S \rangle \notin P$. Since $P$ is a property of the language of a Turing machine, this is a contradiction.