2.2  a. The following grammar generates $A$:

$$S \rightarrow RT$$
$$R \rightarrow aR \mid \varepsilon$$
$$T \rightarrow bTc \mid \varepsilon$$

The following grammar generates $B$:

$$S \rightarrow TR$$
$$T \rightarrow aTb \mid \varepsilon$$
$$R \rightarrow cR \mid \varepsilon$$

Both $A$ and $B$ are context-free languages and $A \cap B = \{a^n b^n c^n \mid n \geq 0\}$. We know from Example 2.36 that this language is not context-free. We have found two CFGs whose intersection is not context-free. Therefore the class of context-free languages is not closed under intersection.

b. First, the context-free languages are closed under the union operation. Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be two arbitrary context-free grammars. We construct a grammar $G$ that recognizes their union. Formally, $G = (V, \Sigma, R, S)$ where:

i) $V = V_1 \cup V_2$

ii) $R = R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}$

(Here we assume that $R_1$ and $R_2$ are disjoint, otherwise we change the variable names to ensure disjointness)

Next, we show that the CFGs are not closed under complementation. Assume, for a contradiction, that the CFGs are closed under complementation. Then, if $G_1$ and $G_2$ are context-free grammars, it would follow that $L(G_1)$ and $L(G_2)$ are context-free. We previously showed that context-free languages are closed under union and so $L(G_1) \cup L(G_2)$ is context-free. That, by our assumption, implies that $L(G_1) \cup L(G_1)$ is context-free. But by De Morgan's laws, $L(G_1) \cup L(G_1) = L(G_1) \cap L(G_2)$. However, if $G_1$ and $G_2$ are chosen as in part (a), $L(G_1) \cup L(G_2)$ isn't context free. This contradiction shows that the context-free languages are not closed under complementation.

2.6  b.  

$$S \rightarrow XbXaX \mid T \mid U$$
$$T \rightarrow aTb \mid Tb \mid b$$
$$U \rightarrow aUb \mid aU \mid a$$
$$X \rightarrow aX \mid bX \mid \varepsilon$$

2.13  a. $L(G)$ is the language of strings of 0s and #s that either contain exactly two #s and any number of 0s, or contain exactly one # and the number of 0s on the right-hand side of the #'s twice the number of 0s on the left-hand side of the #.

b. Assume $L(G)$ is regular and obtain a contradiction. Let $A = L(G) \cap 0^*#0^*$. If $L(G)$ is regular, so is $A$. But we can show $A = \{0^k#0^{2k} \mid k \geq 0\}$ is not regular by using a standard pumping lemma argument.

2.19  The grammar generates all strings not of the form $a^k b^k$ for $k \geq 0$. Thus the complement of the language generated is $L(G) = \{a^k b^k \mid k \geq 0\}$. The following grammar generates $L(G)$:

$$S \rightarrow \{S\}, \{a, b\}, \{S \rightarrow aSb \mid \varepsilon\}, S$$
We construct a PDA $P$ recognizing $D$. This PDA guesses corresponding places on which $x$ and $y$ differ. Checking that the places correspond is tricky. Doing so relies on the observation that the two corresponding places are $n/2$ symbols apart, where $n$ is the length of the entire input. Hence, by ensuring that the number of symbols between the guessed places is equal to the number other symbols, the PDA can check that the guessed places do indeed correspond. Here we give a more detailed description of the PDA algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, $P$ rejects on that branch of the computation.

1. Read next input symbol and push 1 onto the stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol $a$ in the finite control.
4. Read next input symbol and pop the stack. Repeat until stack is empty.
5. Read next input symbol and push 1 onto the stack.
6. Nondeterministically jump to either 5 or 7.
7. Reject if current input symbol differs from $a$.
8. Read next input symbol and pop the stack. Repeat until stack is empty.
9. Accept if input is empty.

Alternatively we can give a CFG for this language as follows.

$$S \rightarrow AB \mid BA$$
$$A \rightarrow XAX \mid \epsilon$$
$$B \rightarrow XBX \mid \epsilon$$
$$X \rightarrow 0 \mid 1$$

Algorithm 4 (L)

$L = \{ x \in \{a,b\}^* \mid x \text{ is a palindrome and } \#_a(x) \equiv \#_b(x) \pmod{3} \}$

Facts: Use variables $S_0 \ (k = 0, 1, 2)$ to generate $x$

Setup $k \in \{0, 1, 2\}$ with $x = x^R$ (palindrome) or $x \equiv k \pmod{3}$

Start variable $S_0$:

- $S_0 \rightarrow aS_0a \mid bS_0b$ with $a \in \{ \text{even-length palindromes centered at } a \}$
- $S_0 \rightarrow aS_1a$ with $a \in \{ \text{odd-length palindromes centered at } a \}$
- $S_1 \rightarrow aS_1a \mid bS_0b$
Consider a derivation of $w$. Each application of a rule of the form $A \to BC$ increases the length of the string by 1. So we have $n-1$ steps here. Besides that, we need exactly $n$ applications of terminal rules $A \to a$ to convert the variables into terminals. Therefore, exactly $2n-1$ steps are required.

Assume $G$ generates a string $w$ using a derivation with at least $2^k$ steps. Let $n$ be the length of $w$. By the results of Problem 2.26, $n \geq \frac{2^k+1}{2} > 2^{b-1}$.

Consider a parse tree of $w$. The right-hand side of each rule contains at most two variables, so each node of the parse tree has at most two children. Additionally, the length of $w$ is at least $2^k$, so the parse tree of $w$ must have height at least $b+1$ to generate a string of length at least $2^k$. Hence, the tree contains a path with at least $b+1$ variables, and therefore some variable is repeated on that path. Using a surgery on trees argument identical to the one used in the proof of the CFL pumping lemma, we can now divide $w$ into pieces $uvxyz$ where $uv^iyz \in G$ for all $i \geq 0$. Therefore, $L(G)$ is infinite.
Problem 8: Recover the 6-tuple definition of a pushdown automaton from its transition function.

(a) \( L = \{ a^n z \mid n \geq 0 \text{ or } x \in \{a, b\}^* \text{ and } 1|x| \leq n \} \)

Idea: Use the stack to remember the length of a prefix of all as, then nondeterministically guess the completion of a prefix \(a^n\) and verify the suffix \(1|x| \leq n\).

\( \delta(q_0, \varepsilon, \varepsilon) \) includes \((q_1, Z_0)\) \( Z_0 \): bottom stack marker

\( \delta(q_1, a, \varepsilon) \) includes \((q_2, A)\) \( q_2 \): keep in memory the length of a prefix of all as

\( \delta(q_1, \varepsilon, \varepsilon) \) includes \((q_2, \varepsilon)\)

Nondeterministically enter \(q_2\) to verify the suffix \(x\) to be \(1|x| \leq n\)

for all \(c \in \{a, b\}\)

\( \delta(q_2, c, A) \) includes \((q_2, \varepsilon)\) \( q_2 \): keep in memory \(1|x| \leq n\)

for all \(z \in \{A, Z_0\}\)

\( \delta(q_2, \varepsilon, Z) \) includes \((\text{Accept}, Z)\)

Nondeterministically guess the end of input

All other combinations \(S\)-value \(\emptyset\)

Question: 1. Check that \(\varepsilon\) is accepted

(if not, modify the \(S\)-function)

2. What happens when \(w \in L\) is an input, will \(w\) be accepted correctly?
(b) \[ L = \{ a^i b^j \mid i, j \geq 0 \text{ and } i \neq j \} \]

Idea: \[ L = \{ a^i b^j \mid i, j \geq 0 \land (i < j \lor i > j) \} \]

\[ \delta(q_0, 0, \varepsilon) \text{ includes } (q_0, 0) \quad q_0 : \text{ loop to remember a prefix of all Os in the stack} \]

\[ \delta(q_0, \varepsilon, \varepsilon) \text{ includes } (q_1, \varepsilon) \quad \text{ nondeterministically enter } q_1 \text{ to guess the end of the prefix } 0^i \]

\[ \delta(q_1, 1, 0) \text{ includes } (q_2, 3) \quad q_1: \text{ loop to check off} \]

\[ \delta(q_2, 0, \varepsilon) \text{ includes } (q_2, \varepsilon) \quad \delta(q_3, 1, \varepsilon) \text{ includes } (q_3, \varepsilon) \]

\[ \delta(q_4, \varepsilon, 0) \text{ includes } (q_4, \varepsilon) \quad \delta(q_5, 1, \varepsilon) \text{ includes } (q_6, \varepsilon) \]

empty the stack (not necessary)

(All other combinations, \( \delta \)-value is \( \phi \))

set \( z \) accepting states: \( \{ q_2, q_3 \} \)

Questions: Similar to part (a).
(c) \( L = \{ w \in \{a,b\}^* \mid w \text{ has twice as many as \( b \)s as } a \text{ s} \} \)
\[ \text{i.e., } \#_a(w) = 2 \#_b(w) \]

Idea: On consumed input \( x \) so far:
- Use no stack to remember the "comparison of \( \#_a(x) \) versus \( 2 \#_b(x) \)."
- The excess/difference of \( \#_a(x) - 2 \#_b(x) \) \((2 \#_b(x) - \#_a(x))\) is indicated by the count of all as (all bs, respectively) in the stack.
- Treat every \( b \) read as if two \( b \)s.

Note: Assume that the machine can push/write a string on the stack in one step of the machine—see text [p609] of [Sipser06].

Proof of Lemma 2.2:

\( S(q_0, x, \epsilon) \) includes \( (q_1, Z_0) \)
\( S(q_1, q, Z_0) \) includes \( (q_1, aZ_0) \) \( q_1: \text{loop to} \)
\( S(q_1, b, Z_0) \) includes \( (q_2, bZ_0) \) remember as \( q_2: \text{treat every } b \text{ read as if two } b \)s
\( S(q_2, a, a) \) includes \( (q_2, \epsilon) \)
\( S(q_2, b, a) \) includes \( (q_3, \epsilon) \)
\( S(q_3, \epsilon, Z_0) \) includes \( (q_4, Z_0) \)
\( S(q_4, a, Z_0) \) includes \( (q_5, Z_0) \) \( q_5: \text{loop} \)
\( S(q_5, b, Z_0) \) includes \( (q_5, bbZ_0) \) check of \( x \)
\( S(q_5, b, b) \) includes \( (q_6, bbb) \) \# \( a \) versus \( 2 \#_b \)
\( S(q_6, a, b) \) includes \( (q_6, \epsilon) \)

(All other combinations, \( S \) -value = \( \epsilon \))

Set \( q \) accepting state \( F \) \( \{ q_5 \} \)
(d) \[ L = (0+1)^* - \{ w w \mid w \in \{0,1\}^* \} \]

Idea: As explained in lecture:

A string \( x \) is of the form \( w w \) for some \( w \in \{0,1\}^* \)

if and only if \( x = \overbrace{w}^{\text{middle}} \overbrace{w}^{\text{middle}} \)

Hence \( x \in L \) iff \( |x| \) is odd or

\( |x| \) is even and \( x = \overbrace{u \downarrow v}^{\text{middle}} \overbrace{u \downarrow v}^{\text{middle}} \)

\[ |u| = |v| \text{ or } \exists \text{ index } i \]

\[ u_i \neq v_i \]

The condition \( |x| \) is odd can be computed by a finite automaton

(hence, a pushdown automaton)

How to check the condition when \( |x| \) is even

but \( u_i \neq v_i \) for some \( i \)?

Example:

\[ \begin{align*}
\text{Example:} & \quad \begin{array}{c}
20 \\
4 \\
16
\end{array} & \quad \begin{array}{c}
20 \\
4 \\
16
\end{array}
\end{align*} \]

\[ \begin{align*}
\text{is viewed as:} & \quad \begin{array}{c}
3 \\
1 \\
3
\end{array} & \quad \begin{array}{c}
13 + 3 = 16 \\
1 \\
16
\end{array}
\end{align*} \]

\[ \text{non-deterministically guess this length at remainder to length in stack} \]

\[ \text{non-deterministically guess this length} \]

\[ \text{check if } u_i \neq v_i \]

\[ \text{Need to consume } \ldots \]

\[ \text{Consume the same length as recorded in stack (or empty the stack)} \]
Problem 9. The given statement is true.

Idea: Given a pushdown automaton $M_1$, we modify $M_1$ to a desired PDA $M_2$ as follows:

**Modification 1:** Have a preliminary sequence of transitions to insert a new stack symbol $U$ as a bottom stack marker or underneath the existing bottom stack marker $Z_0$.

**Modification 2:** Add a new state $q_{new}$ such that:
- Once the state $q_{new}$ is entered in $M_2$, $M_2$ continues to loop back to $q_{new}$ on any subsequent input.
- The new PDA $M_2$ enters the state $q_{new}$ in either of two scenarios:
  - When the top stack is $U$ (which in the original PDA $M_1$ would mean the stack is empty), or
  - As a result of any combination of state, input symbol, and stack symbol for which no transition was defined for $M_1$.

Since these two modifications do not affect sequences of transitions in $M_1$ that end at an accepting state, any sequences do not create any new sequences of transitions leading to an accepting state, $L(M_2) = L(M_1)$.

Now, can you finally write down the 6-tuple definition of $M_2$ based on the 6-tuple definition of $M_1$ (given)?
Problem 10. The given statement is true.

Idea: In a given pushdown automaton $M$, for a transition $(q, B) \in \delta(p, a, A)$ — (1)

where $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$,
$A, B \in \Gamma$

replace the above transition by:

$\delta(p, a, A)$ includes $(q_{\text{new}}, \varepsilon)$ — "pop"

new state with respect to transition (1) above

$\delta(q_{\text{new}}, \varepsilon, \varepsilon)$ includes $(q, B)$ — "push"