c) The set of all strings of a's and b's that are not of the form $ww$, that is, not equal to any string repeated.

!! d) The set of all strings with twice as many 0's as 1's.

**Exercise 5.1.2:** The following grammar generates the language of regular expression $0^*1(0 + 1)^*$:

\[
\begin{align*}
S & \to A1B \\
A & \to 0A \mid \epsilon \\
B & \to 0B \mid 1B \mid \epsilon
\end{align*}
\]

Give leftmost and rightmost derivations of the following strings:

* a) 00101.

b) 1001.

c) 00011.

**Exercise 5.1.3:** Show that every regular language is a context-free language. 
*Hint:* Construct a CFG by induction on the number of operators in the regular expression.

**Exercise 5.1.4:** A CFG is said to be right-linear if each production body has at most one variable, and that variable is at the right end. That is, all productions of a right-linear grammar are of the form $A \to wB$ or $A \to w$, where $A$ and $B$ are variables and $w$ some string of zero or more terminals.

a) Show that every right-linear grammar generates a regular language. *Hint:* Construct an ε-NFA that simulates leftmost derivations, using its state to represent the lone variable in the current left-sentential form.

b) Show that every regular language has a right-linear grammar. *Hint:* Start with a DFA and let the variables of the grammar represent states.

**Exercise 5.1.5:** Let $T = \{0, 1, (), +, *, \emptyset, e\}$. We may think of $T$ as the set of symbols used by regular expressions over alphabet \{0, 1\}; the only difference is that we use $e$ for symbol $e$, to avoid potential confusion in what follows. Your task is to design a CFG with set of terminals $T$ that generates exactly the regular expressions with alphabet \{0, 1\}.

**Exercise 5.1.6:** We defined the relation $\xRightarrow{*}$ with a basis “$\alpha \Rightarrow \alpha$” and an induction that says “$\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$ imply $\alpha \Rightarrow \gamma$.” There are several other ways to define $\xRightarrow{*}$ that also have the effect of saying that $\xRightarrow{*}$ is zero or more \Rightarrow steps.” Prove that the following are true:

a) $\alpha \Rightarrow \beta$ if and only if there is a sequence of one or more strings

$$\gamma_1, \gamma_2, \ldots, \gamma_n$$

such that $\alpha = \gamma_1$, $\beta = \gamma_n$, and for $i = 1, 2, \ldots, n - 1$ we have $\gamma_i \Rightarrow \gamma_{i+1}$. 

! **Exercise 5.4.2:** Prove that the grammar of Exercise 5.4.1 generates all and only the strings of $a$’s and $b$’s such that every prefix has at least as many $a$’s as $b$’s.

*! **Exercise 5.4.3:** Find an unambiguous grammar for the language of Exercise 5.4.1.

!! **Exercise 5.4.4:** Some strings of $a$’s and $b$’s have a unique parse tree in the grammar of Exercise 5.4.1. Give an efficient test to tell whether a given string is one of these. The test “try all parse trees to see how many yield the given string” is not adequately efficient.

! **Exercise 5.4.5:** This question concerns the grammar from Exercise 5.1.2, which we reproduce here:

$$
S \rightarrow A1B \\
A \rightarrow 0A | \epsilon \\
B \rightarrow 0B | 1B | \epsilon
$$

a) Show that this grammar is unambiguous.

b) Find a grammar for the same language that is ambiguous, and demonstrate its ambiguity.

*! **Exercise 5.4.6:** Is your grammar from Exercise 5.1.5 unambiguous? If not, redesign it to be unambiguous.

**Exercise 5.4.7:** The following grammar generates prefix expressions with operands $x$ and $y$ and binary operators $+$, $-$, and $\times$:

$$
E \rightarrow EE | *EE | -EE | x | y
$$

a) Find leftmost and rightmost derivations, and a derivation tree for the string $**-xyxy$.

! b) Prove that this grammar is unambiguous.

## 5.5 Summary of Chapter 5

♯ **Context-Free Grammars:** A CFG is a way of describing languages by recursive rules called productions. A CFG consists of a set of variables, a set of terminal symbols, and a start variable, as well as the productions. Each production consists of a head variable and a body consisting of a string of zero or more variables and/or terminals.

♯ **Derivations and Languages:** Beginning with the start symbol, we derive terminal strings by repeatedly replacing a variable by the body of some production with that variable in the head. The language of the CFG is the set of terminal strings we can so derive; it is called a context-free language.
the input (second component) in each ID is also legal.

2. If a computation is legal for a PDA $P$, then the computation formed by
   adding the same additional stack symbols below the stack in each ID is
   also legal.

3. If a computation is legal for a PDA $P$, and some tail of the input is not
   consumed, then we can remove this tail from the input in each ID, and
   the resulting computation will still be legal.

Intuitively, data that $P$ never looks at cannot affect its computation. We fo-
rmalize points (1) and (2) in a single theorem.

**Theorem 6.5:** If $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA, and $(q, x, \alpha)^\pi_P (p, y, \beta)$,
then for any strings $w$ in $\Sigma^*$ and $\gamma$ in $\Gamma^*$, it is also true that

\[(q, xw, \alpha\gamma)^\pi_P (p, yw, \beta\gamma)\]

Note that if $\gamma = \epsilon$, then we have a formal statement of principle (1) above, and
if $w = \epsilon$, then we have the second principle.

**Proof:** The proof is actually a very simple induction on the number of steps
in the sequence of ID’s that take $(q, xw, \alpha\gamma)$ to $(p, yw, \beta\gamma)$. Each of the moves
in the sequence $(q, x, \alpha)^\pi_P (p, y, \beta)$ is justified by the transitions of $P$ without
using $w$ and/or $\gamma$ in any way. Therefore, each move is still justified when these
strings are sitting on the input and stack. \(\square\)

Incidentally, note that the converse of this theorem is false. There are things
that a PDA might be able to do by popping its stack, using some symbols of $\gamma$,
and then replacing them on the stack, that it couldn’t do if it never looked at
$\gamma$. However, as principle (3) states, we can remove unused input, since it is not
possible for a PDA to consume input symbols and then restore those symbols
to the input. We state principle (3) formally as:

**Theorem 6.6:** If $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA, and

\[(q, xw, \alpha)^\pi_P (p, yw, \beta)\]

then it is also true that $(q, x, \alpha)^\pi_P (p, y, \beta)$. \(\square\)

### 6.1.5 Exercises for Section 6.1

**Exercise 6.1.1:** Suppose the PDA $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$
has the following transition function:

1. $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$. 

**ID’s for Finite Automata?**

One might wonder why we did not introduce for finite automata a notation like the ID’s we use for PDA’s. Although a FA has no stack, we could use a pair \((q, w)\), where \(q\) is the state and \(w\) the remaining input, as the ID of a finite automaton.

While we could have done so, we would not glean any more information from reachability among ID’s than we obtain from the \(\delta\) notation. That is, for any finite automaton, we could show that \(\delta(q, w) = p\) if and only if \((q,wx)^k = (p,x)\) for all strings \(x\). The fact that \(x\) can be anything we wish without influencing the behavior of the FA is a theorem analogous to Theorems 6.5 and 6.6.

2. \(\delta(q, 0, X) = \{(q,XX)\}\).
3. \(\delta(q, 1, X) = \{(q,X)\}\).
4. \(\delta(q, e, X) = \{(p,e)\}\).
5. \(\delta(p,e, X) = \{(p,e)\}\).
6. \(\delta(p, 1, X) = \{(p,XX)\}\).
7. \(\delta(p, 1, Z_0) = \{(p,e)\}\).

Starting from the initial ID \((q, w, Z_0)\), show all the reachable ID’s when the input \(w\) is:

* a) 01.
  * b) 0011.
  * c) 010.

### 6.2 The Languages of a PDA

We have assumed that a PDA accepts its input by consuming it and entering an accepting state. We call this approach “acceptance by final state.” There is a second approach to defining the language of a PDA that has important applications. We may also define for any PDA the language “accepted by empty stack,” that is, the set of strings that cause the PDA to empty its stack, starting from the initial ID.

These two methods are equivalent, in the sense that a language \(L\) has a PDA that accepts it by final state if and only if \(L\) has a PDA that accepts it by empty stack. However, for a given PDA \(P\), the languages that \(P\) accepts
Now, we must prove that \( w \) is in \( N(P_N) \) if and only if \( w \) is in \( L(P_F) \). The ideas are similar to the proof for Theorem 6.9. The “if” part is a direct simulation, and the “only-if” part requires that we examine the limited number of things that the constructed PDA \( P_N \) can do.

(If) Suppose \( (q_0, w, Z_0) \xrightarrow{P_F} (q, \epsilon, \alpha) \) for some accepting state \( q \) and stack string \( \alpha \). Using the fact that every transition of \( P_F \) is a move of \( P_N \), and invoking Theorem 6.5 to allow us to keep \( X_0 \) below the symbols of \( \Gamma \) on the stack, we know that \( (q_0, w, Z_0X_0) \xrightarrow{P_N} (q, \epsilon, \alpha X_0) \). Then \( P_N \) can do the following:

\[
(p_0, w, X_0) \xrightarrow{P_N} (q_0, w, Z_0X_0) \xrightarrow{P_N} (q, \epsilon, \alpha X_0) \xrightarrow{P_N} (p, \epsilon, \epsilon)
\]

The first move is by rule (1) of the construction of \( P_N \), while the last sequence of moves is by rules (3) and (4). Thus, \( w \) is accepted by \( P_N \), by empty stack.

(Only-if) The only way \( P_N \) can empty its stack is by entering state \( p \), since \( X_0 \) is sitting at the bottom of stack and \( X_0 \) is not a symbol on which \( P_F \) has any moves. The only way \( P_N \) can enter state \( p \) is if the simulated \( P_F \) enters an accepting state. The first move of \( P_N \) is surely the move given in rule (1). Thus, every accepting computation of \( P_N \) looks like

\[
(p_0, w, X_0) \xrightarrow{P_N} (q_0, w, Z_0X_0) \xrightarrow{P_N} (q, \epsilon, \alpha X_0) \xrightarrow{P_N} (p, \epsilon, \epsilon)
\]

where \( q \) is an accepting state of \( P_F \).

Moreover, between ID’s \( (q_0, w, Z_0X_0) \) and \( (q, \epsilon, \alpha X_0) \), all the moves are moves of \( P_F \). In particular, \( X_0 \) was never the top stack symbol prior to reaching ID \( (q, \epsilon, \alpha X_0) \). Thus, we conclude that the same computation can occur in \( P_F \), without the \( X_0 \) on the stack; that is, \( (q_0, w, Z_0) \xrightarrow{P_F} (q, \epsilon, \alpha) \). Now we see that \( P_F \) accepts \( w \) by final state, so \( w \) is in \( L(P_F) \). \( \square \)

### 6.2.5 Exercises for Section 6.2

**Exercise 6.2.1**: Design a PDA to accept each of the following languages. You may accept either by final state or by empty stack, whichever is more convenient.

* a) \( \{0^n1^n \mid n \geq 1 \} \).

* b) The set of all strings of 0’s and 1’s such that no prefix has more 1’s than 0’s.

* c) The set of all strings of 0’s and 1’s with an equal number of 0’s and 1’s.

**Exercise 6.2.2**: Design a PDA to accept each of the following languages.

* a) \( \{a^i b^j c^k \mid i = j \text{ or } j = k \} \). Note that this language is different from that of Exercise 5.1.1(b).

\(^4\)Although \( \alpha \) could be \( \epsilon \), in which case \( P_F \) has emptied its stack at the same time it accepts.