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 formalize 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 $(q, x, \alpha) \overset{x}{\overset{\gamma}{\rightarrow}}_{P} (p, y, \beta)$, then for any strings $w$ in $\Sigma^*$ and $\gamma$ in $\Gamma^*$, it is also true that

$$(q, xw, \alpha\gamma) \overset{w}{\overset{\gamma}{\rightarrow}}_{P} (p, yw, \beta\gamma)$$

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 $(q, x, \alpha) \overset{x}{\overset{\gamma}{\rightarrow}}_{P} (p, y, \beta)$ 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. □

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

$$(q, xw, \alpha) \overset{w}{\overset{\gamma}{\rightarrow}}_{P} (p, yw, \beta)$$

then it is also true that $(q, x, \alpha) \overset{w}{\overset{\gamma}{\rightarrow}}_{P} (p, y, \beta)$. □

### 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 \((q, wx) \mapsto (p, x)\) 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
**Exercise 6.3.4:** Convert the PDA of Exercise 6.1.1 to a context-free grammar.

**Exercise 6.3.5:** Below are some context-free languages. For each, devise a PDA that accepts the language by empty stack. You may, if you wish, first construct a grammar for the language, and then convert to a PDA.

a) \( \{a^n b^m c^{2(n+m)} \mid n \geq 0, m \geq 0 \} \).

b) \( \{a^i b^j c^k \mid i = 2j \text{ or } j = 2k \} \).

c) \( \{0^n 1^m \mid n \leq m \leq 2n \} \).

**Exercise 6.3.6:** Show that if \( P \) is a PDA, then there is a one-state PDA \( P_1 \) such that \( N(P_1) = N(P) \).

**Exercise 6.3.7:** Suppose we have a PDA with \( s \) states, \( t \) stack symbols, and no rule in which a replacement stack string has length greater than \( u \). Give a tight upper bound on the number of variables in the CFG that we construct for this PDA by the method of Section 6.3.2.

### 6.4 Deterministic Pushdown Automata

While PDAs are by definition allowed to be nondeterministic, the deterministic subcase is quite important. In particular, parsers generally behave like deterministic PDAs, so the class of languages that can be accepted by these automata is interesting for the insights it gives us into what constructs are suitable for use in programming languages. In this section, we shall define deterministic PDAs and investigate some of the things they can and cannot do.

#### 6.4.1 Definition of a Deterministic PDA

Intuitively, a PDA is deterministic if there is never a choice of move in any situation. These choices are of two kinds. If \( \delta(q,a,X) \) contains more than one pair, then surely the PDA is nondeterministic because we can choose among these pairs when deciding on the next move. However, even if \( \delta(q,a,X) \) is always a singleton, we could still have a choice between using a real input symbol, or making a move on \( \epsilon \). Thus, we define a PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \) to be **deterministic** (a deterministic PDA or DPDA), if and only if the following conditions are met:

1. \( \delta(q,a,X) \) has at most one member for any \( q \) in \( Q \), \( a \) in \( \Sigma \) or \( a = \epsilon \), and \( X \) in \( \Gamma \).

2. If \( \delta(q,a,X) \) is nonempty, for some \( a \) in \( \Sigma \), then \( \delta(q,\epsilon,X) \) must be empty.
one step in the derivation in $G_2$ becomes one or more steps in the derivation of $w$ using the productions of $G_1$. First, if any $X_i$ is a terminal, we know $G_1$ has a corresponding variable $B_i$ and a production $B_i \rightarrow X_i$. Then, if $k > 2$, $G_1$ has productions $A \rightarrow B_1C_1$, $C_1 \rightarrow B_2C_2$, and so on, where $B_i$ is either the introduced variable for terminal $X_i$ or $X_i$ itself, if $X_i$ is a variable. These productions simulate in $G_1$ one step of a derivation of $G_2$ that uses $A \rightarrow X_1X_2 \cdots X_k$. We conclude that there is a derivation of $w$ in $G_1$, so $w$ is in $L(G_1)$.

(If) Suppose $w$ is in $L(G_1)$. Then there is a parse tree in $G_1$, with $S$ at the root and yield $w$. We convert this tree to a parse tree of $G_2$ that also has root $S$ and yield $w$.

First, we “undo” part (b) of the CNF construction. That is, suppose there is a node labeled $A$, with two children labeled $B_1$ and $C_1$, where $C_1$ is one of the variables introduced in part (b). Then this portion of the parse tree must look like Fig. 7.4(a). That is, because these introduced variables each have only one production, there is only one way that they can appear, and all the variables introduced to handle the production $A \rightarrow B_1B_2 \cdots B_k$ must appear together, as shown.

Any such cluster of nodes in the parse tree may be replaced by the production that they represent. The parse-tree transformation is suggested by Fig. 7.4(b).

The resulting parse tree is still not necessarily a parse tree of $G_2$. The reason is that step (a) in the CNF construction introduced other variables that derive single terminals. However, we can identify these in the current parse tree and replace a node labeled by such a variable $A$ and its one child labeled $a$, by a single node labeled $a$. Now, every interior node of the parse tree forms a production of $G_2$. Since $w$ is the yield of a parse tree in $G_2$, we conclude that $w$ is in $L(G_2)$.

### 7.1.6 Exercises for Section 7.1

* **Exercise 7.1.1:** Find a grammar equivalent to

\[
S \rightarrow AB \mid CA \\
A \rightarrow a \\
B \rightarrow BC \mid AB \\
C \rightarrow aB \mid b
\]

with no useless symbols.

* **Exercise 7.1.2:** Begin with the grammar:

\[
S \rightarrow ASB \mid \epsilon \\
A \rightarrow aAS \mid a \\
B \rightarrow SbS \mid A \mid bb
\]
There is another interesting normal form for grammars that we shall not prove. Every nonempty language without $\epsilon$ is $L(G)$ for some grammar $G$ each of whose productions is of the form $A \rightarrow a\alpha$, where $a$ is a terminal and $\alpha$ is a string of zero or more variables. Converting a grammar to this form is complex, even if we simplify the task by, say, starting with a Chomsky-Normal-Form grammar. Roughly, we expand the first variable of each production, until we get a terminal. However, because there can be cycles, where we never reach a terminal, it is necessary to “short-circuit” the process, creating a production that introduces a terminal as the first symbol of the body and has variables following it to generate all the sequences of variables that might have been generated on the way to generation of that terminal.

This form, called Greibach Normal Form, after Sheila Greibach, who first gave a way to construct such grammars, has several interesting consequences. Since each use of a production introduces exactly one terminal into a sentential form, a string of length $n$ has a derivation of exactly $n$ steps. Also, if we apply the PDA construction of Theorem 6.13 to a Greibach-Normal-Form grammar, then we get a PDA with no $\epsilon$-rules, thus showing that it is always possible to eliminate such transitions of a PDA.

a) Eliminate $\epsilon$-productions.
b) Eliminate any unit productions in the resulting grammar.
c) Eliminate any useless symbols in the resulting grammar.
d) Put the resulting grammar into Chomsky Normal Form.

**Exercise 7.1.3:** Repeat Exercise 7.1.2 for the following grammar:

\[
\begin{align*}
S & \rightarrow 0A0 \mid 1B1 \mid BB \\
A & \rightarrow C \\
B & \rightarrow S \mid A \\
C & \rightarrow S \mid \epsilon
\end{align*}
\]

**Exercise 7.1.4:** Repeat Exercise 7.1.2 for the following grammar:

\[
\begin{align*}
S & \rightarrow AAA \mid B \\
A & \rightarrow aA \mid B \\
B & \rightarrow \epsilon
\end{align*}
\]

**Exercise 7.1.5:** Repeat Exercise 7.1.2 for the following grammar:
CHAPTER 7. PROPERTIES OF CONTEXT-FREE LANGUAGES

7.2.4 Exercises for Section 7.2

Exercise 7.2.1: Use the CFL pumping lemma to show each of these languages not to be context-free:

* a) \{a^i b^j c^k \mid i < j < k\}.
   
   b) \{a^n b^n c^n \mid i \leq n\}.
   
   c) \{0^p \mid p \text{ is a prime}\}. \text{Hint: Adapt the same ideas used in Example 4.3, which showed this language not to be regular.}
   
   d) \{0^i 1^j \mid j = i^2\}.
   
   e) \{a^n b^n c^n \mid n \leq i \leq 2n\}.
   
   f) \{wwRw \mid w \text{ is a string of } 0\text{'s and } 1\text{'s}\}. That is, the set of strings consisting of some string w followed by the same string in reverse, and then the string w again, such as 00110001.

Exercise 7.2.2: When we try to apply the pumping lemma to a CFL, the “adversary wins,” and we cannot complete the proof. Show what goes wrong when we choose L to be one of the following languages:

   a) \{00, 11\}.
   
   * b) \{0^n 1^n \mid n \geq 1\}.
   
   * c) The set of palindromes over alphabet \{0, 1\}.

Exercise 7.2.3: There is a stronger version of the CFL pumping lemma known as Ogden’s lemma. It differs from the pumping lemma we proved by allowing us to focus on any n “distinguished” positions of a string z and guaranteeing that the strings to be pumped have between 1 and n distinguished positions. The advantage of this ability is that a language may have strings consisting of two parts, one of which can be pumped without producing strings not in the language, while the other does produce strings outside the language when pumped. Without being able to insist that the pumping take place in the latter part, we cannot complete a proof of non-context-freeness. The formal statement of Ogden’s lemma is: If L is a CFL, then there is a constant n, such that if z is any string of length at least n in L, in which we select at least n positions to be distinguished, then we can write z = uvwxy, such that:

1. \text{rvwx} has at most n distinguished positions.
2. \text{vx} has at least one distinguished position.
3. For all i, uv^iwx^iy is in L.

Prove Ogden’s lemma. \text{Hint: The proof is really the same as that of the pumping lemma of Theorem 7.18 if we pretend that the nondistinguished positions of z are not present as we select a long path in the parse tree for z.}
The proofs in both directions are inductions on the number of moves made by the two automata. In the “if” portion, one needs to observe that once the buffer of \( P' \) is nonempty, it cannot read another input symbol and must simulate \( P \), until the buffer has become empty (although when the buffer is empty, it may still simulate \( P \)). We leave further details as an exercise.

Once we accept this relationship between \( P' \) and \( P \), we note that \( P' \) accepts \( h(w) \) if and only if \( P' \) accepts \( w \), because of the way the accepting states of \( P' \) are defined. Thus, \( L(P') = h^{-1}(L(P)) \). \( \square \)

### 7.3.6 Exercises for Section 7.3

**Exercise 7.3.1:** Show that the CFL's are closed under the following operations:

* a) init, defined in Exercise 4.2.6(c). *Hint:* Start with a CNF grammar for the language \( L \).

* b) The operation \( L/a \), defined in Exercise 4.2.2. *Hint:* Again, start with a CNF grammar for \( L \).

* c) cycle, defined in Exercise 4.2.11. *Hint:* Try a PDA-based construction.

**Exercise 7.3.2:** Consider the following two languages:

\[
L_1 = \{a^n b^n c^m \mid n, m \geq 0\} \\
L_2 = \{a^n b^m c^{2m} \mid n, m \geq 0\}
\]

a) Show that each of these languages is context-free by giving grammars for each.

b) Is \( L_1 \cap L_2 \) a CFL? Justify your answer.

**Exercise 7.3.3:** Show that the CFL's are not closed under the following operations:

* a) min, as defined in Exercise 4.2.6(a).

* b) max, as defined in Exercise 4.2.6(b).

* c) half, as defined in Exercise 4.2.8.

* d) alt, as defined in Exercise 4.2.7.

**Exercise 7.3.4:** The shuffle of two strings \( w \) and \( x \) is the set of all strings that one can get by interleaving the positions of \( w \) and \( x \) in any way. More precisely, \( \text{shuffle}(w, x) \) is the set of strings \( z \) such that

1. Each position of \( z \) can be assigned to \( w \) or \( x \), but not both.
**PROOF:** The reason the algorithm finds the correct sets of variables was explained as we introduced the basis and inductive parts of the algorithm. For the running time, note that there are $O(n^2)$ entries to compute, and each involves comparing and computing with $n$ pairs of entries. It is important to remember that, although there can be many variables in each set $X_{ij}$, the grammar $G$ is fixed and the number of its variables does not depend on $n$, the length of the string $w$ whose membership is being tested. Thus, the time to compare two entries $X_{ik}$ and $X_{k+1,j}$, and find variables to go into $X_{ij}$ is $O(1)$. As there are at most $n$ such pairs for each $X_{ij}$, the total work is $O(n^3)$. □

**Example 7.34:** The following are the productions of a CNF grammar $G$:

\[
\begin{align*}
S & \rightarrow AB \mid BC \\
A & \rightarrow BA \mid a \\
B & \rightarrow CC \mid b \\
C & \rightarrow AB \mid a
\end{align*}
\]

We shall test for membership in $L(G)$ the string $baaba$. Figure 7.14 shows the table filled in for this string.

\[
\begin{array}{cccccc}
{S,A,C} & - & {S,A,C} & - & {B} & {B} \\
{S,A} & {B} & {S,A} & {S,A} \\
{B} & {A,C} & {A,C} & {B} & {A,C} \\
\end{array}
\]

Figure 7.14: The table for string $baaba$ constructed by the CYK algorithm

To construct the first (lowest) row, we use the basis rule. We have only to consider which variables have a production body $a$ (those variables are $A$ and $C$) and which variables have body $b$ (only $B$ does). Thus, above those positions holding $a$ we see the entry $\{A, C\}$, and above the positions holding $b$ we see $\{B\}$. That is, $X_{11} = X_{44} = \{B\}$, and $X_{22} = X_{33} = X_{55} = \{A, C\}$.

In the second row we see the values of $X_{12}$, $X_{23}$, $X_{34}$, and $X_{45}$. For instance, let us see how $X_{12}$ is computed. There is only one way to break the string from positions 1 to 2, which is $ba$, into two nonempty substrings. The first must be position 1 and the second must be position 2. In order for a variable to generate $ba$, it must have a body whose first variable is in $X_{11} = \{B\}$ (i.e., it generates the $b$) and whose second variable is in $X_{22} = \{A, C\}$ (i.e., it generates the $a$). This body can only be $BA$ or $BC$. If we inspect the grammar, we find that the
Given a CFG $G$ and one of its variables $A$, is there any sentential form in which $A$ is the first symbol. \textbf{Note:} Remember that it is possible for $A$ to appear first in the middle of some sentential form but then for all the symbols to its left to derive $\varepsilon$.

\textbf{Exercise 7.4.2:} Use the technique described in Section 7.4.3 to develop linear-time algorithms for the following questions about CFG's:

a) Which symbols appear in some sentential form?

b) Which symbols are nullable (derive $\varepsilon$)?

\textbf{Exercise 7.4.3:} Using the grammar $G$ of Example 7.34, use the CYK algorithm to determine whether each of the following strings is in $L(G)$:

* a) $ababa$.

* b) $baaab$.

* c) $aaabab$.

* \textbf{Exercise 7.4.4:} Show that in any CNF grammar, all parse trees for strings of length $n$ have $2n - 1$ interior nodes (i.e., $2n - 1$ nodes with variables for labels).

* \textbf{Exercise 7.4.5:} Modify the CYK algorithm to report the number of distinct parse trees for the given input, rather than just reporting membership in the language.

\textbf{7.5 Summary of Chapter 7}

\begin{itemize}
  \item \textit{Eliminating Useless Symbols:} A variable can be eliminated from a CFG unless it derives some string of terminals and also appears in at least one string derived from the start symbol. To correctly eliminate such useless symbols, we must first test whether a variable derives a terminal string, and eliminate those that do not, along with all their productions. Only then do we eliminate variables that are not derivable from the start symbol.

  \item \textit{Eliminating $\epsilon$- and Unit-productions:} Given a CFG, we can find another CFG that generates the same language, except for string $\varepsilon$, yet has no $\varepsilon$-productions (those with body $\varepsilon$) or unit productions (those with a single variable as the body).

  \item \textit{Chomsky Normal Form:} Given a CFG that derives at least one nonempty string, we can find another CFG that generates the same language, except for $\varepsilon$, and is in Chomsky Normal Form: there are no useless symbols, and every production body consists of either two variables or one terminal.
\end{itemize}