4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6. (Actually the procedure given in Theorem 2.9 produces several variables U_i and several rules $U_i \rightarrow \texttt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow \texttt{a}$.)

$$\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid \texttt{a} \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid \texttt{a} \mid SA \mid AS \\
A &\rightarrow \texttt{b} \mid AA_1 \mid UB \mid \texttt{a} \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow \texttt{a} \\
B &\rightarrow \texttt{b}
\end{aligned}$$

2.2 PUSHDOWN AUTOMATA

In this section we introduce a new type of computational model called ***pushdown automata***. These automata are like nondeterministic finite automata but have an extra component called a ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it. Certain languages are more easily described in terms of generators, whereas others are more easily described by recognizers.

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read.

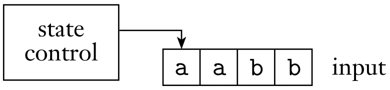


FIGURE 2.11
Schematic of a finite automaton

With the addition of a stack component we obtain a schematic representation of a pushdown automaton, as shown in the following figure.

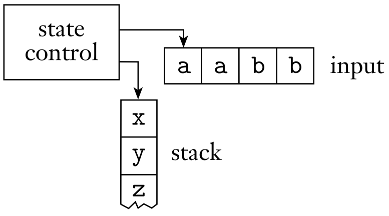


FIGURE 2.12
Schematic of a pushdown automaton

A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up. Writing a symbol on the stack is often referred to as ***pushing*** the symbol, and removing a symbol is referred to as ***popping*** it. Note that all access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a "last in, first out" storage device. If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later information is removed.

Plates on a cafeteria serving counter illustrate a stack. The stack of plates rests on a spring so that when a new plate is placed on top of the stack, the plates below it move down. The stack on a pushdown automaton is like a stack of plates, with each plate having a symbol written on it.

A stack is valuable because it can hold an unlimited amount of information. Recall that a finite automaton is unable to recognize the language $\{0^n1^n | n \geq 0\}$ because it cannot store very large numbers in its finite memory. A PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen. Thus the unlimited nature of a stack allows the PDA to store numbers of unbounded size. The following informal description shows how the automaton for this language works.

> Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If reading the input is finished exactly when the stack becomes empty of 0s, accept the input. If the stack becomes empty while 1s remain or if the 1s are finished while the stack still contains 0s or if any 0s appear in the input following 1s, reject the input.

As mentioned earlier, pushdown automata may be nondeterministic. Deterministic and nondeterministic pushdown automata are *not* equivalent in power.

Nondeterministic pushdown automata recognize certain languages that no deterministic pushdown automata can recognize, as we will see in Section 2.4. We give languages requiring nondeterminism in Examples 2.16 and 2.18. Recall that deterministic and nondeterministic finite automata do recognize the same class of languages, so the pushdown automata situation is different. We focus on nondeterministic pushdown automata because these automata are equivalent in power to context-free grammars.

FORMAL DEFINITION OF A PUSHDOWN AUTOMATON

The formal definition of a pushdown automaton is similar to that of a finite automaton, except for the stack. The stack is a device containing symbols drawn from some alphabet. The machine may use different alphabets for its input and its stack, so now we specify both an input alphabet Σ and a stack alphabet Γ.

At the heart of any formal definition of an automaton is the transition function, which describes its behavior. Recall that $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. The domain of the transition function is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Thus the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton. Either symbol may be ε, causing the machine to move without reading a symbol from the input or without reading a symbol from the stack.

For the range of the transition function we need to consider what to allow the automaton to do when it is in a particular situation. It may enter some new state and possibly write a symbol on the top of the stack. The function δ can indicate this action by returning a member of Q together with a member of Γ_ε, that is, a member of $Q \times \Gamma_\varepsilon$. Because we allow nondeterminism in this model, a situation may have several legal next moves. The transition function incorporates nondeterminism in the usual way, by returning a set of members of $Q \times \Gamma_\varepsilon$, that is, a member of $\mathcal{P}(Q \times \Gamma_\varepsilon)$. Putting it all together, our transition function δ takes the form $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.

DEFINITION 2.13

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.
2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

EXAMPLES OF PUSHDOWN AUTOMATA

EXAMPLE 2.14

The following is the formal description of the PDA (page 112) that recognizes the language $\{0^n 1^n \mid n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$Q = \{q_1, q_2, q_3, q_4\}$,

$\Sigma = \{0,1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

We can also use a state diagram to describe a PDA, as in Figures 2.15, 2.17, and 2.19. Such diagrams are similar to the state diagrams used to describe finite automata, modified to show how the PDA uses its stack when going from state to state. We write "$a,b \to c$" to signify that when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with a c. Any of a, b, and c may be ε. If a is ε, the machine may make this transition without reading any symbol from the input. If b is ε, the machine may make this transition without reading and popping any symbol from the stack. If c is ε, the machine does not write any symbol on the stack when going along this transition.

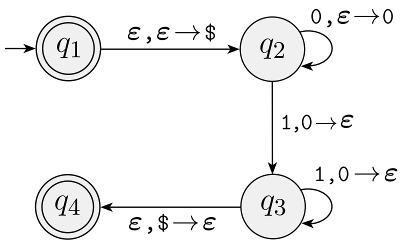


FIGURE 2.15
State diagram for the PDA M_1 that recognizes $\{0^n1^n \mid n \geq 0\}$ ■

The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. This PDA is able to get the same effect by initially placing a special symbol \$ on the stack. Then if it ever sees the \$ again, it knows that the stack effectively is empty. Subsequently, when we refer to testing for an empty stack in an informal description of a PDA, we implement the procedure in the same way.

Similarly, PDAs cannot test explicitly for having reached the end of the input string. This PDA is able to achieve that effect because the accept state takes effect only when the machine is at the end of the input. Thus from now on, we assume that PDAs can test for the end of the input, and we know that we can implement it in the same manner.

EXAMPLE 2.16 ....

This example illustrates a pushdown automaton that recognizes the language

$$\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i,j,k \geq 0 \text{ and } i=j \text{ or } i=k\}.$$

Informally, the PDA for this language works by first reading and pushing the a's. When the a's are done, the machine has all of them on the stack so that it can match, them with either the b's or the c's. This maneuver is a bit tricky because the machine doesn't know in advance whether to match the a's with the b's or the c's. Nondeterminism comes in handy here.

Using its nondeterminism, the PDA can guess whether to match the a's with the b's or with the c's, as shown in Figure 2.17. Think of the machine as having two branches of its nondeterminism, one for each possible guess. If either of them matches, that branch accepts and the entire machine accepts. Problem 2.57 asks you to show that nondeterminism is essential for recognizing this language with a PDA.

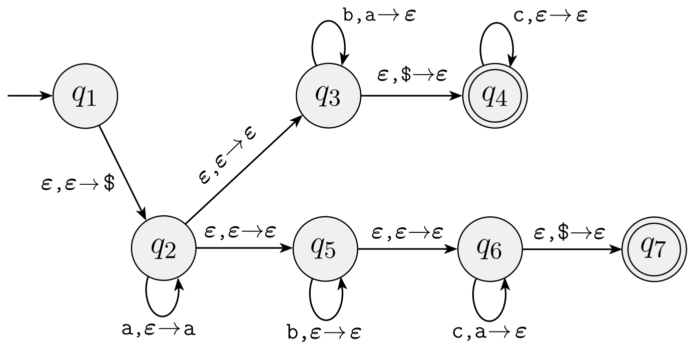


FIGURE 2.17
State diagram for PDA M_2 that recognizes
$\{a^ib^jc^k \mid i,j,k \geq 0 \text{ and } i = j \text{ or } i = k\}$ ■

EXAMPLE 2.18

In this example we give a PDA M_3 recognizing the language $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$. Recall that $w^{\mathcal{R}}$ means w written backwards. The informal description and state diagram of the PDA follow.

Begin by pushing the symbols that are read onto the stack. At each point, nondeterministically guess that the middle of the string has been reached and then change into popping off the stack for each symbol read, checking to see that they are the same. If they were always the same symbol and the stack empties at the same time as the input is finished, accept; otherwise reject.

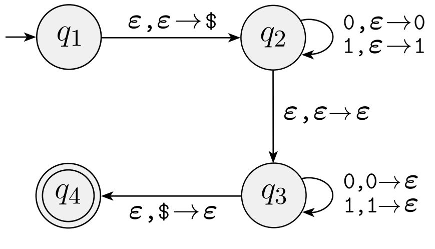


FIGURE 2.19
State diagram for the PDA M_3 that recognizes $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

Problem 2.58 shows that this language requires a nondeterministic PDA. ■

EQUIVALENCE WITH CONTEXT-FREE GRAMMARS

In this section we show that context-free grammars and pushdown automata are equivalent in power. Both are capable of describing the class of context-free languages. We show how to convert any context-free grammar into a pushdown automaton that recognizes the same language and vice versa. Recalling that we defined a context-free language to be any language that can be described with a context-free grammar, our objective is the following theorem.

THEOREM 2.20 ....

A language is context free if and only if some pushdown automaton recognizes it.

As usual for "if and only if" theorems, we have two directions to prove. In this theorem, both directions are interesting. First, we do the easier forward direction.

LEMMA 2.21 ....

If a language is context free, then some pushdown automaton recognizes it.

PROOF IDEA Let A be a CFL. From the definition we know that A has a CFG, G, generating it. We show how to convert G into an equivalent PDA, which we call P.

The PDA P that we now describe will work by accepting its input w, if G generates that input, by determining whether there is a derivation for w. Recall that a derivation is simply the sequence of substitutions made as a grammar generates a string. Each step of the derivation yields an ***intermediate string*** of variables and terminals. We design P to determine whether some series of substitutions using the rules of G can lead from the start variable to w.

One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make. The PDA's nondeterminism allows it to guess the sequence of correct substitutions. At each step of the derivation, one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

The PDA P begins by writing the start variable on its stack. It goes through a series of intermediate strings, making one substitution after another. Eventually it may arrive at a string that contains only terminal symbols, meaning that it has used the grammar to derive a string. Then P accepts if this string is identical to the string it has received as input.

Implementing this strategy on a PDA requires one additional idea. We need to see how the PDA stores the intermediate strings as it goes from one to another. Simply using the stack for storing each intermediate string is tempting. However, that doesn't quite work because the PDA needs to find the variables in the intermediate string and make substitutions. The PDA can access only the top symbol on the stack and that may be a terminal symbol instead of a variable. The way around this problem is to keep only *part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string. Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string. The following figure shows the PDA P.

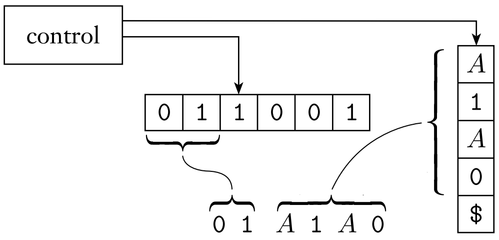


FIGURE 2.22
P representing the intermediate string 01A1A0

The following is an informal description of P.

1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps forever.
   a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
   c. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

PROOF We now give the formal details of the construction of the pushdown automaton $P = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, F)$. To make the construction clearer, we use shorthand notation for the transition function. This notation provides a way to write an entire string on the stack in one step of the machine. We can simulate this action by introducing additional states to write the string one symbol at a time, as implemented in the following formal construction.

Let q and r be states of the PDA and let a be in Σ_ε and s be in Γ_ε. Say that we want the PDA to go from q to r when it reads a and pops s. Furthermore, we want it to push the entire string $u = u_1 \cdots u_l$ on the stack at the same time. We can implement this action by introducing new states $q_1, \ldots, q_{l-1}$ and setting the transition function as follows:

$$\begin{aligned}
&\delta(q, a, s) \text{ to contain } (q_1, u_l),\\
&\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{l-1})\},\\
&\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{l-2})\},\\
&\qquad\vdots\\
&\delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}.
\end{aligned}$$

We use the notation $(r, u) \in \delta(q, a, s)$ to mean that when q is the state of the automaton, a is the next input symbol, and s is the symbol on the top of the stack, the PDA may read the a and pop the s, then push the string u onto the stack and go on to the state r. The following figure shows this implementation.

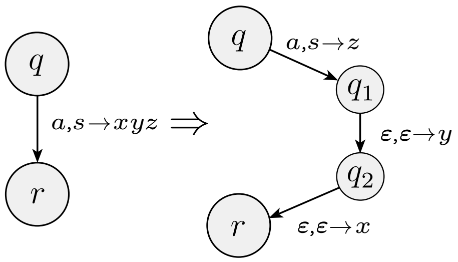

q
$a,s\to xyz$
r
$\Rightarrow$
q
$a,s\to z$
q_1
$\varepsilon,\varepsilon\to y$
q_2
r
$\varepsilon,\varepsilon\to x$

FIGURE 2.23
Implementing the shorthand $(r, xyz) \in \delta(q, a, s)$

The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}.

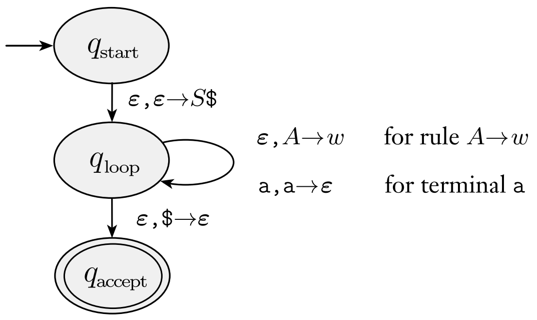
The transition function is defined as follows. We begin by initializing the stack to contain the symbols \$ and S, implementing step 1 in the informal description: $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$. Then we put in transitions for the main loop of step 2.

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) |\text{ where } A \to w \text{ is a rule in } R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

Finally, we handle case (c) wherein the empty stack marker \$ is on the top of the stack. Let $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

The state diagram is shown in Figure 2.24.



q_{start}
$\varepsilon, \varepsilon \to S\$$
q_{loop}
$\varepsilon, A \to w$ for rule $A \to w$
$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal $\mathtt{a}$
$\varepsilon, \$ \to \varepsilon$
q_{accept}

FIGURE 2.24
State diagram of P

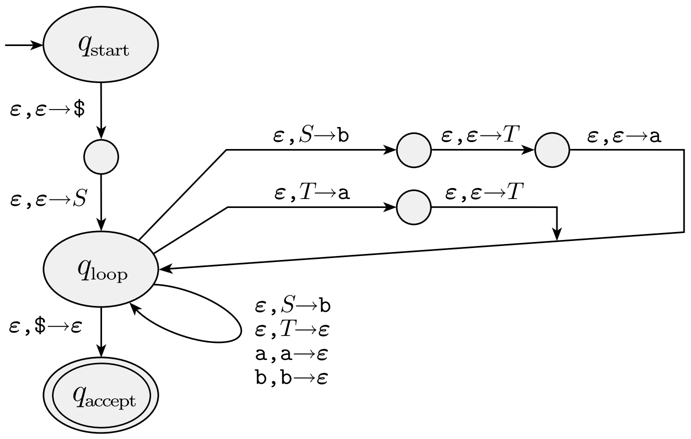
That completes the proof of Lemma 2.21.

EXAMPLE 2.25

We use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G.

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.



q_{start}
$\varepsilon, \varepsilon \to \$$
$\varepsilon, \varepsilon \to S$
$\varepsilon, S \to \mathtt{b}$
$\varepsilon, \varepsilon \to T$
$\varepsilon, \varepsilon \to \mathtt{a}$
$\varepsilon, T \to \mathtt{a}$
$\varepsilon, \varepsilon \to T$
q_{loop}
$\varepsilon, S \to \mathtt{b}$
$\varepsilon, T \to \varepsilon$
$\mathtt{a}, \mathtt{a} \to \varepsilon$
$\mathtt{b}, \mathtt{b} \to \varepsilon$
$\varepsilon, \$ \to \varepsilon$
q_{accept}

FIGURE 2.26
State diagram of P_1

■

Now we prove the reverse direction of Theorem 2.20. For the forward direction, we gave a procedure for converting a CFG into a PDA. The main idea was to design the automaton so that it simulates the grammar. Now we want to give a procedure for going the other way: converting a PDA into a CFG. We design the grammar to simulate the automaton. This task is challenging because "programming" an automaton is easier than "programming" a grammar.

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

PROOF IDEA We have a PDA P, and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state.

To achieve this outcome, we design a grammar that does somewhat more. For each pair of states p and q in P, the grammar will have a variable A_{pq}. This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p.

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

Giving P features 1 and 2 is easy. To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.

To design G so that A_{pq} generates all strings that take P from p to q, starting and ending with an empty stack, we must understand how P operates on these strings. For any such string x, P's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly, the last move on x must be a pop because the stack ends up empty.

Two possibilities occur during P's computation on x. Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack could be empty only at the beginning and end of P's computation on x. If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. We simulate the former possibility with the rule $A_{pq} \to aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the state following p, and s is the state preceding q. We simulate the latter possibility with the rule $A_{pq} \to A_{pr}A_{rq}$, where r is the state when the stack becomes empty.