is usually implemented as a list in which one end is designated as the *top*. Items are always added ("pushed onto the stack") and deleted ("popped off the stack") at this end, and at any time, the only element of the stack that is immediately accessible is the one on top.

In trying to incorporate this algorithm in an abstract machine, it would be reasonable to say that the current "state" of the machine is determined in part by the current contents of the stack. However, this approach would require an infinite set of "states," because the stack needs to be able to hold arbitrarily long strings. It is convenient instead to continue using a finite set of states—although the machine is not a "finite-state machine" in the same way that an FA is, because the current state is not enough to specify the machine's status—and to think of the stack as a simple form of auxiliary memory. This means that a move of our machine will depend not only on the current state and input, but also on the symbol currently on top of the stack. Carrying out the move may change the stack as well as the state.

In this simple example, the set Q of states will contain only three elements, q_0, q_1, and q_2. The state q_0, the initial state, is sufficient for processing the first half of the string. In this state, each input symbol is pushed onto the stack, regardless of what is currently on top. The machine stays in q_0 as long as it has not yet received the symbol c; when that happens, the machine moves to state q_1, leaving the stack unchanged. State q_1 is for processing the second half of the input string. Once the machine enters this state, the only string that can be accepted is the one whose second half (after the c) is the reverse of the string already read. In this state each input symbol is compared to the symbol currently on top of the stack. If they agree, that symbol is popped off the stack and both are discarded; otherwise, the machine will crash and the string will not be accepted. This phase of the processing ends when the stack is empty, provided the machine has not crashed. An empty stack means that every symbol in the first half of the string has been successfully matched with an identical input symbol in the second half, and at that point the machine enters the accepting state q_2.

Now we consider how to describe precisely the abstract machine whose operations we have sketched. Each move of the machine will be determined by three things:

1. The current state
2. The next input
3. The symbol on top of the stack

and will consist of two parts:

1. Changing states (or staying in the same state)
2. Replacing the top stack symbol by a string of zero or more symbols

Describing moves this way allows us to consider the two basic stack moves as special cases: Popping the top symbol off the stack means replacing it by Λ, and pushing Y onto the stack means replacing the top symbol X by YX (assuming that the left end of the string corresponds to the top). We could enforce the stack rules more strictly by requiring that a single move contain only one stack operation, either a push or a pop. However, replacing the stack symbol X by the string α can be accomplished by a sequence of basic moves (a pop, followed by a sequence of zero or more pushes), and allowing the more general move helps to keep the number of distinct moves as small as possible.

In the case of a finite automaton, our transition function took the form

$$\delta : Q \times \Sigma \to Q$$

Here, if we allow the possibility that the stack alphabet Γ (the set of symbols that can appear on the stack) is different from the input alphabet Σ, it looks as though we want

$$\delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$$

For a state q, an input a, and a stack symbol X,

$$\delta(q, a, X) = (p, \alpha)$$

means that in state q, with X on top of the stack, we read the symbol a, move to state p, and replace X on the stack by the string α.

This approach raises a few questions. First, how do we describe a move if the stack is empty ($\delta(q, a, ?)$)? We avoid this problem by saying that initially there is a special *start symbol* Z_0 on the stack, and the machine is not allowed to move when the stack is empty. Provided that Z_0 is never removed from the stack and that no additional copies of Z_0 are pushed onto the stack, saying that Z_0 is on top means that the stack is effectively empty.

Second, how do we describe a move when the input is exhausted ($\delta(q, ?, X)$)? (Remember that in our example we want to move to q_2 if the stack is empty when all the input has been read.) The solution we adopt here is to allow moves that use only Λ as input, corresponding to Λ-transitions in an NFA-Λ. This suggests that what we really want is

$$\delta : Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma \to Q \times \Gamma^*$$

Of course, once we have moves of the form $\delta(q, \Lambda, X)$, we can make them before all the input has been read; if the next input symbol is not read in a move, it is still there to be read subsequently.

We have already said that there may be situations when the machine will crash—that is, when no move is specified. In the case of a finite automaton, when this happened we decided to make $\delta(q, a)$ a *subset* of Q, rather than an element, so that it could have the value $\emptyset$. At the same time we allowed for the possibility that $\delta(q, a)$ might contain more than one element, so that the FA became nondeterministic. Here we do the same thing, except that since $Q \times \Gamma^*$ is an infinite set we should say explicitly that $\delta(q, a, X)$ and $\delta(q, \Lambda, X)$ will always be finite. In our current example the nondeterminism is not necessary, but in many cases it is. Thus we are left with

$$\delta : Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma \to \text{ the set of finite subsets of } Q \times \Gamma^*$$

Now we can give a precise description of our simple-palindrome recognizer. Q will be the set $\{q_0, q_1, q_2\}$, q_0 is the initial state, and q_2 is the only accepting state. The input alphabet Σ is $\{a, b, c\}$, and the stack alphabet Γ is $\{a, b, Z_0\}$. The transition function δ is given by Table 7.1. Remember that when we specify a string to be placed on the stack, the top of the stack corresponds to the left end of the string. This convention may seem odd at first, since if we were to push the symbols on one at a time we would have to do it right-to-left, or in reverse order. The point is that when we get around to processing the symbols on the stack, the order in which we encounter them is the same as the order in which they occurred in the string.

Moves 1 through 6 push the input symbols a and b onto the stack, moves 7 through 9 change state without affecting the stack, moves 10 and 11 match an input symbol with a stack symbol and discard both, and the last move is to accept provided there is nothing except Z_0 on the stack.

Table 7.1 | Transition table for Example 7.1

Move number	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_0, aZ_0)
2	q_0	b	Z_0	(q_0, bZ_0)
3	q_0	a	a	(q_0, aa)
4	q_0	b	a	(q_0, ba)
5	q_0	a	b	(q_0, ab)
6	q_0	b	b	(q_0, bb)
7	q_0	c	Z_0	(q_1, Z_0)
8	q_0	c	a	(q_1, a)
9	q_0	c	b	(q_1, b)
10	q_1	a	a	(q_1, Λ)
11	q_1	b	b	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
	(all other combinations)			none

Let us trace the moves of the machine for three input strings: $abcba$, ab, and $acaa$.

Move number	Resulting state	Unread input	Stack
(initially)	q_0	$abcba$	Z_0
1	q_0	$bcba$	aZ_0
4	q_0	cba	baZ_0
9	q_1	ba	baZ_0
11	q_1	a	aZ_0
10	q_1	—	Z_0
12	q_2	—	Z_0
(accept)			
(initially)	q_0	ab	Z_0
1	q_0	b	aZ_0
4	q_0	—	baZ_0
(crash)			
(initially)	q_0	$acaa$	Z_0
1	q_0	caa	aZ_0
8	q_1	aa	aZ_0
10	q_1	a	Z_0
12	q_2	a	Z_0

Note the last move on input string $acaa$. Although there is no move $\delta(q_1, a, Z_0)$, the machine can take the Λ-transition $\delta(q_1, \Lambda, Z_0)$ before running out of choices. We could say that the portion of the string read so far (i.e., aca) is accepted, since the machine is in an accepting state at this point; however, the entire input string is not accepted, because not all of it has been read.

Figure 7.1 shows a diagram corresponding to Example 7.1, modeled after (but more complicated than) a transition diagram for an FA. Each transition is labeled with an input (either an alphabet symbol or Λ), a stack symbol X, a slash (/), and a string α of stack symbols. The interpretation is that the transition may occur on the specified input and involves

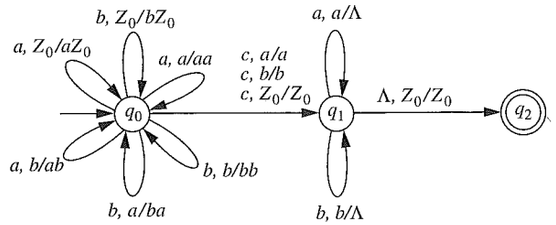


Figure 7.1 |
Transition diagram for the pushdown automaton (PDA) in Example 7.1.

replacing X on the stack by α. Even with the extra information required for labeling an arrow, a diagram of this type does not capture completely the PDA's behavior in the same way that a transition diagram for an FA does. With an FA, you can start at any point with just the diagram and the input symbols and trace the action of the machine by following the arrows. In Figure 7.1, however, you cannot follow the arrows without keeping track of the stack contents—possibly the entire contents—as you go. The number of possible combinations of state and stack contents is infinite, and it is therefore not possible to draw a "finite-state diagram" in the same sense as for an FA. In most cases we will describe pushdown automata in this chapter by transition tables similar to the one in Table 7.1, although it will occasionally also be useful to show a transition diagram.

7.2 | THE DEFINITION OF A PUSHDOWN AUTOMATON

Below is a precise definition of the type of abstract machine illustrated in Example 7.1. Remember that what is being defined is in general nondeterministic.

Definition 7.1 Definition of a PDA

A *pushdown automaton* (PDA) is a 7-tuple $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, where

Q is a finite set of states.
Σ and Γ are finite sets (the input and stack alphabets, respectively).
q_0, the initial state, is an element of Q.
Z_0, the initial stack symbol, is an element of Γ.
A, the set of accepting states, is a subset of Q.
$\delta : Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma \rightarrow$ the set of finite subsets of $Q \times \Gamma^*$.

The function δ is called the transition function of M.

The stack alphabet Γ and the initial stack symbol Z_0 are what make it necessary to have a 7-tuple rather than a 5-tuple. Otherwise, the components of the tuple are the same as in the case of an FA, except that the transition function δ is more complicated.

We can trace the operation of a finite automaton by keeping track of the current state at each step. In order to trace the operation of a PDA M, we must also keep track of the stack contents. If we are interested in what the machine does with a specific input string, it is also helpful to monitor the portion of the string yet to be read. A *configuration* of the PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a triple

$$(q, x, \alpha)$$

where $q \in Q$, $x \in \Sigma^*$, and $\alpha \in \Gamma^*$. Saying that (q, x, α) is the current configuration of M means that q is the current state, x is the string of remaining unread input, and α is the current stack contents, where as usual it is the left end of α that corresponds to the top of the stack.

We write

$$(p, x, \alpha) \vdash_M (q, y, \beta)$$

to mean that one of the possible moves in the first configuration takes M to the second. This can happen in two ways, depending on whether the move consumes an input symbol or is a Λ-transition. In the first case, $x = ay$ for some $a \in \Sigma$, and in the second case $x = y$; we can summarize both cases by saying $x = ay$ for some $a \in \Sigma \cup \{\Lambda\}$. In both cases, the string β of stack symbols is obtained from α by replacing the first symbol X by a string ξ (in other words, $\alpha = X\gamma$ for some $X \in \Gamma$ and some $\gamma \in \Gamma^*$, and $\beta = \xi\gamma$ for some $\xi \in \Gamma^*$), and

$$(q, \xi) \in \delta(p, a, X)$$

More generally, we write

$$(p, x, \alpha) \vdash_M^* (q, y, \beta)$$

if there is a sequence of zero or more moves that takes M from the first configuration to the second. As usual, if there is no possibility of confusion, we shorten $\vdash_M$ to $\vdash$ and $\vdash_M^*$ to $\vdash^*$. Using the new notation, we may define acceptance of a string by a PDA.

Definition 7.2 Acceptance by a PDA

If $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a PDA and $x \in \Sigma^*$, x is *accepted* by M if

$$(q_0, x, Z_0) \vdash_M^* (q, \Lambda, \alpha)$$

for some $\alpha \in \Gamma^*$ and some $q \in A$. (The stack may or may not be empty when x is accepted, because α may or may not be Λ.) A language $L \subseteq \Sigma^*$ is said to be accepted by M if L is precisely the set of strings accepted by M; in this case, we write $L = L(M)$.

Note that whether or not a string is accepted depends only on the current state when the string has been processed, not on the stack contents. We use the phrase

accepting configuration to denote any configuration in which the state is an accepting state. This type of acceptance is sometimes called *acceptance by final state*. It will be convenient in Section 7.5 to look briefly at another type, acceptance by *empty stack*. In this approach, a string is said to be accepted if it allows the PDA to reach a configuration in which the stack is empty, regardless of whether the state is an accepting state. It is not hard to see (Section 7.5 and Exercises 7.41 and 7.42) that the two types of acceptance are equivalent, in the sense that if a language is accepted by some PDA using one mode of acceptance, there is another PDA using the other mode that also accepts the language.

It is worth emphasizing that when we say a string x is accepted by a PDA, we mean that *there is* a sequence of moves that cause the machine to reach an accepting configuration as a result of reading the symbols of x. Since a PDA can be nondeterministic, there may be many other possible sequences of moves that do not lead to an accepting configuration. Each time there is a choice of moves, we may view the PDA as making a guess as to which one to make. Acceptance means that if the PDA guesses right at each step, it can reach an accepting configuration. In our next example, we will see a little more clearly what it means to guess right at each step.

A PDA Accepting the Language of Palindromes — EXAMPLE 7.2

This example involves the language *pal* of palindromes over $\{a, b\}$ (both even-length and odd-length), without the marker in the middle that provides the signal for the PDA to switch from the "pushing-input-onto-stack" state to the "comparing-input-symbol-to-stack-symbol" state. The general strategy for constructing a PDA to recognize this language sounds the same as in Example 7.1: Remember the symbols seen so far, by saving them on the stack, until we are ready to begin matching them with symbols in the second half of the string. This switch from one type of move to the other should happen when we reach the middle of the string, just as in Example 7.1. However, without a symbol marking the middle explicitly, the PDA has no way of knowing that the middle has arrived; it can only guess. Fortunately, there is no penalty for guessing wrong, as long as the guess does not allow a nonpalindrome to be accepted.

We think of the machine as making a sequence of "not yet" guesses as it reads input symbols and pushes them onto the stack. This phase can stop (with a "yes" guess) in two possible ways: The PDA may guess that the next input symbol is the one in the very middle of the string (and that the string is of odd length) and can therefore be discarded since it need not be matched by anything; or it may guess that the input string read so far is the first half of the (even-length) string and that any subsequent input symbols should therefore be used to match stack symbols. In effect, if the string read so far is x, the first "yes" guess that might be made is that another input symbol s should be read and that the input string will be xsx^r. The second possible guess, which can be made without reading another symbol, is that the input string will be xx^r. In either case, from this point on the PDA is committed. It makes no more guesses, attempts to process the remaining symbols as if they belong to the second half, and can accept no string other than the one it has guessed the input string to be.

This approach cannot cause a nonpalindrome to be accepted, because each time the PDA makes a "yes" guess, it is then unable to accept any string not of the form xsx^r or xx^r. On the other hand, the approach allows every palindrome to be accepted: Every palindrome looks like

either xsx^r or xx^r, and in either case, there is a permissible sequence of choices that involves making the correct "yes" guess at just the right time to cause the string to be accepted. It is still possible, of course, that for an input string z that is a palindrome the PDA guesses "yes" at the wrong time or makes the wrong type of "yes" guess; it might end up accepting some palindrome other than z, or simply stop in a nonaccepting state. This does not mean that the PDA is incorrect, but only that the PDA did not choose the particular sequence of moves that *would* have led to acceptance of z.

The transition table for our PDA is shown in Table 7.2. The sets Q, Γ, and A are the same as in Example 7.1, and there are noticeable similarities between the two transition tables. The moves in the first six lines of Table 7.1 show up as *possible* moves in the corresponding lines of Table 7.2, and the last three lines of the two tables (which represent the processing of the second half of the string) are identical.

The fact that the first six lines of Table 7.2 show two possible moves tells us that there is genuine nondeterminism. The two choices in each of these lines are to guess "not yet," as in Table 7.1, and to guess that the input symbol is the middle symbol of the (odd-length) string. The input symbol is read in both cases; the first choice causes it to be pushed onto the stack, and the second choice causes it to be discarded.

However, there is also nondeterminism of a less obvious sort. Suppose for example that the PDA is in state q_0, the top stack symbol is a, and the next input symbol is a, as in line 3. In addition to the two moves shown in line 3, there is a third choice shown in line 8: not to read the input symbol at all, but to execute a Λ-transition to state q_1. This represents the other "yes" guess, the guess that as a result of reading the most recent symbol (now on top of the stack), we have reached the middle of the (even-length) string. This choice is made without even looking at the next input symbol. (Another approach would have been to read the a, use it to match the a on the stack, and move to q_1, all on the same move; however, the moves shown in the table preserve the distinction between the state q_0 in which all the guessing occurs and the state q_1 in which all the comparison-making occurs.)

Note that the Λ-transition in line 8 is not in itself the source of nondeterminism. The move in line 12, for example, is the only possible move from state q_1 if Z_0 is the top stack symbol. Line 8 represents nondeterminism because if the PDA is in state q_0 and a is the top stack symbol, there is a choice between a move that reads an input symbol and one that does not. We will return to this point in Section 7.3.

Table 7.2 | Transition table for Example 7.2

Move number	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_0, aZ_0), (q_1, Z_0)
2	q_0	b	Z_0	(q_0, bZ_0), (q_1, Z_0)
3	q_0	a	a	(q_0, aa), (q_1, a)
4	q_0	b	a	(q_0, ba), (q_1, a)
5	q_0	a	b	(q_0, ab), (q_1, b)
6	q_0	b	b	(q_0, bb), (q_1, b)
7	q_0	Λ	Z_0	(q_1, Z_0)
8	q_0	Λ	a	(q_1, a)
9	q_0	Λ	b	(q_1, b)
10	q_1	a	a	(q_1, Λ)
11	q_1	b	b	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

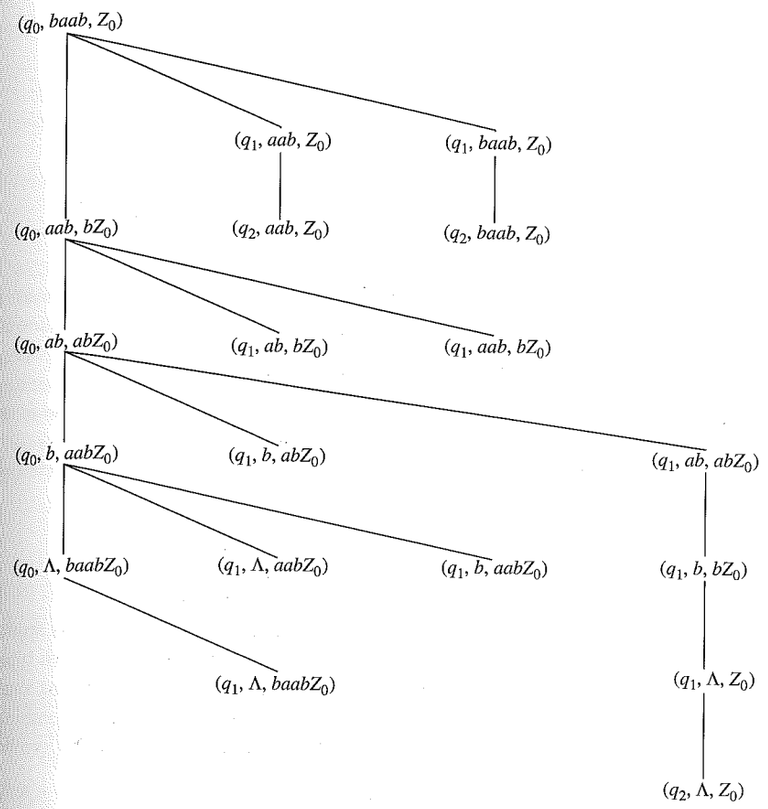


Figure 7.2 |
Computation tree for the PDA in Table 7.2, with input *baab*.

Just as in Section 4.1, we can draw a computation tree for a PDA such as this one, showing the configuration at each step and the possible choices of moves at each step. Figure 7.2 shows such a tree for the string *baab*, which is a palindrome.

Each time there is a choice, the possible moves are shown left-to-right in the order they appear in Table 7.2. In particular, in each configuration along the left edge of Figure 7.2 except the last one, the PDA is in state q_0 and there is at least one unread input symbol. At each of these points, the PDA can choose from three possible moves. Continuing down the left edge of the figure represents a "not yet" guess that reads an input and pushes it onto the stack. The other two possibilities are the two moves to state q_1, one that reads an input symbol and one that does not.

The sequence of moves that leads to acceptance is

$$\begin{aligned}(q_0, baab, Z_0) &\vdash (q_0, aab, bZ_0)\\ &\vdash (q_0, ab, abZ_0)\\ &\vdash (q_1, ab, abZ_0)\\ &\vdash (q_1, b, bZ_0)\\ &\vdash (q_1, \Lambda, Z_0)\\ &\vdash (q_2, \Lambda, Z_0) \quad \text{(accept)}\end{aligned}$$

This sequence of moves is the one in which the "yes" guess of the right type is made at exactly the right time. Paths that deviate from the vertical path too soon terminate before the PDA has finished reading the input; the machine either crashes or enters the accepting state q_2 prematurely (so that the string accepted is a palindrome of length 0 or 1, not the one we have in mind). Paths that follow the vertical path too long cause the PDA either to crash or to run out of input symbols before getting a chance to empty the stack.

7.3 DETERMINISTIC PUSHDOWN AUTOMATA

The PDA in Example 7.1 never has a choice of more than one move, and it is appropriate to call it a deterministic PDA. The one in Example 7.2 illustrates both types of nondeterminism: that in which there are two or more moves involving the same combination of state, stack symbol, and input, and that in which, for some combination of state and stack symbol, the machine has a choice of reading an input symbol or making a Λ-transition.

Definition 7.3 Definition of a deterministic PDA

Let $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ be a pushdown automaton. M is *deterministic* if there is no configuration for which M has a choice of more than one move. In other words, M is deterministic if it satisfies both the following conditions.

1. For any $q \in Q$, $a \in \Sigma \cup \{\Lambda\}$, and $X \in \Gamma$, the set $\delta(q, a, X)$ has at most one element.
2. For any $q \in Q$ and $X \in \Gamma$, if $\delta(q, \Lambda, X) \neq \emptyset$, then $\delta(q, a, X) = \emptyset$ for every $a \in \Sigma$.

A language L is a *deterministic context-free language* (DCFL) if there is a deterministic PDA (DPDA) accepting L.

Note that our definition does not require the transition function to be defined for every combination of state, input, and stack symbol; in a deterministic PDA, it is still possible for one of the sets $\delta(q, a, X)$ to be empty. In this sense, our notion of determinism is a little less strict than in Chapter 4, where we called a finite automaton nondeterministic if there was a pair (q, a) for which $\delta(q, a)$ did not have exactly one element.

The last statement in the definition anticipates to some extent the results of the next two sections, which show that the languages that can be accepted by PDAs are precisely the context-free languages. The last statement also suggests another way in which CFLs are more complicated than regular languages. We did not define a "deterministic regular language" in Chapter 4, although we considered both NFAs and deterministic FAs. The reason is that for any NFA there is an FA recognizing the same language; any regular language can be accepted by a deterministic FA. Not every context-free language, however, can be accepted by a deterministic PDA. It probably seemed obvious in Example 7.2 that the standard approach to accepting the language of palindromes cannot work without nondeterminism; we will be able to show in Theorem 7.1 that no other PDA can do any better, and that the language of palindromes is not a DCFL.

A DPDA Accepting Balanced Strings of Brackets — EXAMPLE 7.3

Consider the language L of all balanced strings involving two types of brackets: { } and [ ]. L is the language generated by the context-free grammar with productions

$$S \rightarrow SS \mid [S] \mid \{S\} \mid \Lambda$$

(It is also possible to describe this type of "balanced" string using the approach of Definition 6.5; see Exercise 7.20.)

Our PDA will have two states: the initial state q_0, which is also the accepting state (note that Λ is one element of L), and another state q_1. Left brackets of either type are saved on the stack, and one is discarded whenever it is on top of the stack and a right bracket of the same type is encountered in the input. The feature of strings in L that makes this approach correct, and therefore makes a stack the appropriate data structure, is that when a right bracket in a balanced string is encountered, the left bracket it matches is the *last* left bracket of the same type that has appeared previously and has not already been matched. The signal that the string read so far is balanced is that the stack has no brackets on it (i.e., Z_0 is the top symbol), and if this happens in state q_1 the PDA will return to the accepting state q_0 via a Λ-transition, leaving the stack unchanged. From this point, if there is more input, the machine proceeds as if from the beginning.

Table 7.3 shows a transition table for such a deterministic PDA. To make it easier to read, the parentheses with which we normally enclose a pair specifying a single move have been omitted.

The input string {[ ]}[ ], for example, results in the following sequence of moves.

$$\begin{aligned}(q_0, \{[]\}[], Z_0) &\vdash (q_1, []\}[], \{Z_0)\\ &\vdash (q_1,]\}[], [\{Z_0)\\ &\vdash (q_1, \}[], \{Z_0)\\ &\vdash (q_1, [], Z_0)\end{aligned}$$

Table 7.3 | Transition table for Example 7.3

Move number	State	Input	Stack symbol	Move
1	q_0	{	Z_0	$q_1, \{Z_0$
2	q_0	[	Z_0	$q_1, [Z_0$
3	q_1	{	{	$q_1, \{\{$
4	q_1	[	{	$q_1, [\{$
5	q_1	{	[	$q_1, \{[$
6	q_1	[	[	$q_1, [[$
7	q_1	}	{	q_1, Λ
8	q_1	]	[	q_1, Λ
9	q_1	Λ	Z_0	q_0, Z_0
(all other combinations)				none

$$\vdash (q_0, [], Z_0)$$
$$\vdash (q_1,], [Z_0)$$
$$\vdash (q_1, \Lambda, Z_0)$$
$$\vdash (q_0, \Lambda, Z_0) \quad \text{(accept)}$$

You may very well have seen stacks used in the way we are using them here, with languages closely related to the set of balanced strings of parentheses. If you have written or studied a computer program that reads an algebraic expression and processes it (to "process" an expression could mean to evaluate it, to convert it to postfix notation, to build an expression tree to store it, or simply to check that it obeys the syntax rules), the program almost certainly involved at least one stack. If the program did not use recursion, the stack was explicit—storing values of subexpressions, perhaps, or parentheses and operators; if the algorithm was recursive, there was a stack behind the scenes, since stacks are the data structures involved whenever recursive algorithms are implemented.

EXAMPLE 7.4 A DPDA to Accept Strings with More *a*'s Than *b*'s

For our last example of a DPDA, we consider

$$L = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$$

The approach we use is similar in some ways to that in the previous example. There we saved left brackets on the stack so that they could eventually be matched by right brackets. Now we save excess symbols of either type, so that they can eventually be matched by symbols of the opposite type. The other obvious difference is that since the null string is not in L, the initial state is not accepting. With those differences in mind, it is easy to understand the DPDA described in Table 7.4. The only two states are the initial state q_0 and the accepting state q_1.

At any point when the stack contains only Z_0, the string read so far has equal numbers of a's and b's. It is almost correct to say that the machine is in the accepting state q_1 precisely when there is at least one a on the stack. This is not quite correct, because input b in state q_1 requires removing a from the stack and returning to q_0, at least temporarily; this guarantees that if the a just removed from the stack was the *only* one on the stack, the input string read so far (which has equal numbers of a's and b's) is not accepted. The Λ-transition is the way the PDA returns to the accepting state once it determines that there are additional a's remaining on the stack.

Table 7.4 | Transition table for a DPDA to accept L

Move number	State	Input	Stack symbol	Move
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_0, bZ_0)
3	q_0	a	b	(q_0, Λ)
4	q_0	b	b	(q_0, bb)
5	q_1	a	a	(q_1, aa)
6	q_1	b	a	(q_0, Λ)
7	q_0	Λ	a	(q_1, a)
(all other combinations)				none

Table 7.5 | A DPDA with no Λ-transitions accepting L

Move number	State	Input	Stack symbol	Move
1	q_0	a	Z_0	(q_1, Z_0)
2	q_0	b	Z_0	(q_0, bZ_0)
3	q_0	a	b	(q_0, Λ)
4	q_0	b	b	(q_0, bb)
5	q_1	a	Z_0	(q_1, aZ_0)
6	q_1	b	Z_0	(q_0, Z_0)
7	q_1	a	a	(q_1, aa)
8	q_1	b	a	(q_1, Λ)
(all other combinations)				none

If we can provide a way for the PDA to determine in advance whether an a on the stack is the only one, then we can eliminate the need to leave the accepting state when a is popped from the stack, and thereby eliminate Λ-transitions altogether. There are at least three natural ways we might manage this. One is to say that we will push a's onto the stack only when we have an excess of at least two, so that in state q_1, top stack symbol Z_0 means one extra a, and top stack symbol a means more than one. Another is to use a different stack symbol, say A, for the first extra a. A third is simply to introduce a new state specifically for the case in which there is exactly one extra a. The DPDA shown in Table 7.5 takes the first approach. As before, q_1 is the accepting state. There is no move specified from q_1 with stack symbol b or from q_0 with stack symbol a, because neither of these situations will ever occur.

This PDA may be slightly easier to understand with the transition diagram shown in Figure 7.3.

We illustrate the operation of this machine on the input string *abbabaa*:

$$(q_0, abbabaa, Z_0) \vdash (q_1, bbabaa, Z_0)$$
$$\vdash (q_0, babaa, Z_0)$$
$$\vdash (q_0, abaa, bZ_0)$$
$$\vdash (q_0, baa, Z_0)$$
$$\vdash (q_0, aa, bZ_0)$$
$$\vdash (q_0, a, Z_0)$$
$$\vdash (q_1, \Lambda, Z_0) \quad \text{(accept)}$$

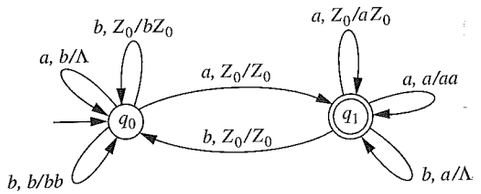


Figure 7.3 |
The DPDA in Table 7.5.

We conclude this section by showing the result we promised at the beginning: Not every language that can be accepted by a PDA can be accepted by a deterministic PDA.

Theorem 7.1
The language $pal = \{x \in \{a, b\}^* \mid x = x^r\}$ cannot be accepted by any DPDA.

Proof
Suppose for the sake of contradiction that $M = (Q, \{a, b\}, \Gamma, q_0, Z_0, A, \delta)$ is a DPDA accepting *pal*. We can easily modify M if necessary (see Exercise 7.12) so that every move is either one of the form

$$\delta(p, s, X) = (q, \Lambda)$$

or one of the form

$$\delta(p, s, X) = (q, \alpha X)$$

where $s \in \{a, b, \Lambda\}$ and $\alpha \in \Gamma^*$. The effect of the modification is that M can still remove a symbol from the stack or place another symbol on the stack, but it cannot do both in the same move.

We observe next that for any string x, M must eventually read every symbol of x. The string xx^r, for example, is a palindrome, and M must read it completely in order to accept it. However, because M is deterministic, the moves it makes while processing x in the course of accepting xx^r are exactly the moves it must make on input x, whether or not x is followed by anything else.

After M has processed a string x, its stack is a certain height, depending on how much M needs to remember about x. We may consider how much shorter the stack can ever get as a result of reading subsequent input symbols. For each x, let y_x be a string for which this resulting stack height is as small as possible. (It cannot be 0, because M must still be able to process longer strings with prefix xy_x.) In other words,

$$(q_0, xy_x, Z_0) \vdash^* (q_x, \Lambda, \alpha_x)$$

for some state q_x and some string $\alpha_x \in \Gamma^*$ with $|\alpha_x| > 0$; and for any string y, any state p, and any string $\beta \in \Gamma^*$,

$$\text{if } (q_0, xy, Z_0) \vdash^* (p, \Lambda, \beta) \text{ then } |\beta| \geq |\alpha_x|$$

Because of our initial assumption about the moves of M, any move that involves removing a stack symbol decreases the stack height. Therefore, once M reaches the configuration (q_x, Λ, α_x), as a result of processing the string xy_x, no symbols of the string α_x will ever be removed from the stack subsequently.

Let A_x be the first symbol of the string α_x. The set S of all strings of the form xy_x is infinite (because we may consider x of any length), and the set of all ordered pairs (q_x, A_x) is finite (because the entire set $Q \times \Gamma$ is finite). Therefore, there must be an infinite subset T of S so that for all the elements xy_x of T, the pairs (q_x, A_x) are equal. In particular, we can choose two different strings $u_1 = xy_x$ and $u_2 = wy_w$ so that

$$(q_0, u_1, Z_0) \vdash^* (q, \Lambda, A\beta_1)$$

and

$$(q_0, u_2, Z_0) \vdash^* (q, \Lambda, A\beta_2)$$

for some $q \in Q$, some $A \in \Gamma$, and some strings $\beta_1, \beta_2 \in \Gamma^*$. If we now look at longer strings of the form u_1z and u_2z, we have

$$(q_0, u_1z, Z_0) \vdash^* (q, z, A\beta_1)$$

and

$$(q_0, u_2z, Z_0) \vdash^* (q, z, A\beta_2)$$

and the symbol A is never removed from the stack as a result of processing the string z. The reason this is useful is that although the stack contents may not be the same in the two cases, the machine can never notice the difference. The ultimate result of processing the string u_1z must be exactly the same as that of processing u_2z: Either both strings are accepted or neither is. Now, however, we have the contradiction we are looking for. On the one hand, M treats u_1z and u_2z the same way; on the other hand, as we showed in the proof of Theorem 3.3, the two strings u_1 and u_2 are distinguishable with respect to *pal*, so that for some z, one of the strings u_1z, u_2z is in *pal* and the other is not. This is impossible if M accepts *pal*.

See Exercise 7.17 for some other examples of CFLs that are not DCFLs, and see Section 8.2 for some other methods of showing that languages are not DCFLs.

7.4 | A PDA CORRESPONDING TO A GIVEN CONTEXT-FREE GRAMMAR

Up to this point, the pushdown automata we have constructed have been based on simple symmetry properties of the strings in the languages being recognized, rather

than on any features of context-free grammars generating the languages. As a result, it may not be obvious that *every* context-free language can be recognized by a PDA. However, that is what we will prove in this section.

Starting with a context-free grammar G, we want to build a PDA that can test an arbitrary string and determine whether it can be derived from G. The basic strategy is to *simulate* a derivation of the string in the given grammar. This will require guessing the steps of the derivation, and our PDA will be nondeterministic. (As we will see in Section 7.6, there are certain types of grammars for which we will be able to modify the PDA, keeping its essential features but eliminating the nondeterminism. The approach will be particularly useful in these cases, because for a string x in the language, following the moves of the machine on input x will not only allow us to confirm x's membership in the language, but also reveal a derivation of x in the grammar. Because of languages like *pal*, however, finding such a deterministic PDA is too much to expect in general.) As the simulation progresses, the machine will test the input string to make sure that it is still consistent with the derivation-in-progress. If the input string does in fact have a derivation from the grammar, and if the PDA's guesses are the ones that correctly simulate this derivation, the tests will confirm this and allow the machine to reach an accepting state.

There are at least two natural ways a PDA can simulate a derivation in the grammar. A step in the simulation corresponds to constructing a portion of the derivation tree, and the two approaches are called *top-down* and *bottom-up* because of the order in which these portions are constructed.

We will begin with the **top-down** approach. The PDA starts by pushing the start symbol S (at the top of the derivation tree) onto the stack, and each subsequent step in the simulated derivation is carried out by replacing a variable on the stack (at a certain node in the tree) by the right side of a production beginning with that variable (in other words, adding the children of that node to the tree.) The stack holds the current string in the derivation, except that as terminal symbols appear at the left of the string they are matched with symbols in the input and discarded.

The two types of moves made by the PDA, after S is placed on the stack, are

1. Replace a variable A on top of the stack by the right side α of some production $A \to \alpha$. This is where the guessing comes in.
2. Pop a terminal symbol from the stack, provided it matches the next input symbol. Both symbols are then discarded.

At each step, the string of input symbols already read (which have been successfully matched with terminal symbols produced at the beginning of the string by the derivation), followed by the contents of the stack, exclusive of Z_0, constitutes the current string in the derivation. When a variable appears on top of the stack, it is because terminal symbols preceding it in the current string have already been matched, and thus it is the leftmost variable in the current string. Therefore, the derivation being simulated is a *leftmost* derivation. If at some point there is no longer any part of the current string remaining on the stack, the attempted derivation must have been successful at producing the input string read so far, and the PDA can accept.

We are now ready to give a more precise description of this top-down PDA and to prove that the strings it accepts are precisely those generated by the grammar.

Definition 7.4 Top-down PDA corresponding to a CFG

Let $G = (V, \Sigma, S, P)$ be a context-free grammar. We define $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ as follows:

$$Q = \{q_0, q_1, q_2\} \quad A = \{q_2\} \quad \Gamma = V \cup \Sigma \cup \{Z_0\} \quad (\text{where } Z_0 \notin V \cup \Sigma)$$

The initial move of M is to place S on the stack and move to q_1: $\delta(q_0, \Lambda, Z_0) = \{(q_1, SZ_0)\}$. The only move to the accepting state q_2 is from q_1, when the stack is empty except for Z_0: $\delta(q_1, \Lambda, Z_0) = \{(q_2, Z_0)\}$. Otherwise, the only moves of M are as follows:

1. For every $A \in V$, $\delta(q_1, \Lambda, A) = \{(q_1, \alpha) \mid A \to \alpha \text{ is a production in } G\}$
2. For every $a \in \Sigma$, $\delta(q_1, a, a) = \{(q_1, \Lambda)\}$

Theorem 7.2

Let $G = (V, \Sigma, S, P)$ be a context-free grammar. Then the top-down PDA M described in Definition 7.4 accepts $L(G)$.

Proof

First, to show $L(G) \subseteq L(M)$. If x is any string in $L(G)$, then the *last* step in a leftmost derivation of x looks like

$$yAz \Rightarrow yy' = x$$

where y, z, and y' are strings of terminals, and a typical intermediate step looks like

$$yA\alpha \Rightarrow yy'\beta$$

where again y and y' are strings of terminals for which yy' is a prefix of x, and the string β begins with a variable. We may take the second case as the representative one, if we say that β either is Λ or begins with a variable.

What we would like to show in this situation is that some sequence of moves of our PDA has the effect of reading the string yy' of terminals, matching it with terminals from the stack, leaving the string β (or actually βZ_0) on the stack, and leaving the PDA in state q_1. It will follow, from the special case in which $\beta = \Lambda$, that with the input string x the PDA can reach the configuration (q_1, Λ, Z_0); therefore, because of the move to q_2 described in the definition, the PDA accepts x.

A precise statement that says this and is suitable for proof by induction is the following:

> For any $n \geq 1$, if $x \in L(G)$ and the nth step in a leftmost derivation of x is $yA\alpha \Rightarrow yy'\beta$, where $x = yy'z$, yy' is a string of terminals, and β either is Λ or begins with a variable, then
>
> $$(q_0, x, Z_0) = (q_0, yy'z, Z_0) \vdash^* (q_1, z, \beta Z_0) \qquad \textbf{(7.1)}$$

For the basis step of the proof, let $n = 1$. The first step in a derivation of x is to use a production of the form $S \to y'\beta$, so that the string y in (7.1)

is null. Using the initial move in the definition of M and a move of type (1), we have

$$(q_0, x, Z_0) = (q_0, y'z, Z_0) \vdash (q_1, y'z, SZ_0) \vdash (q_1, y'z, y'\beta Z_0)$$

However, using as many moves of type (2) as there are symbols in y', the PDA is able to get from the last configuration shown to (q_1, z, β), and we may conclude that

$$(q_0, x, Z_0) \vdash^* (q_1, z, \beta Z_0)$$

Now suppose that $k \geq 1$ and that our statement is true for every $n \leq k$. We wish to show that if the $(k+1)$th step in a leftmost derivation of x is $yA\alpha \Rightarrow yy'\beta$, then (7.1) is true. Let us look at the kth step in the leftmost derivation of x. It is of the form

$$wBy \Rightarrow ww'\beta' = ww'A\alpha$$

where $ww' = y$. This means that $x = ww'(y'z)$, and the induction hypothesis implies that

$$(q_0, x, Z_0) \vdash^* (q_1, y'z, A\alpha Z_0)$$

If the production used in the $(k+1)$th step is $A \rightarrow \alpha'$, so that $\alpha'\alpha = y'\beta$, then using a move of type (1) we get

$$(q_1, y'z, A\alpha Z_0) \vdash (q_1, y'z, \alpha'\alpha Z_0) = (q_1, y'z, y'\beta Z_0)$$

Now, just as in the basis step, we may reach the configuration $(q_1, z, \beta Z_0)$ by using moves of type (2); this completes the proof that $L(G) \subseteq L(M)$.

Next, to show $L(M) \subseteq L(G)$. Roughly speaking, this means we need to show the converse of the previous statement; in other words, the configuration $(q_1, z, \beta Z_0)$ can occur only if some leftmost derivation in the grammar produces the string $y\beta$, where $x = yz$. To be precise, we show the following.

> For any $n \geq 1$, if there is a sequence of n moves that leads from the configuration (q_0, x, Z_0) to the configuration $(q_1, z, \beta Z_0)$, then for some $y \in \Sigma^*$, $x = yz$ and $S \Rightarrow_G^* y\beta$.

The basis step is easy because the only single move producing a configuration of the right form is the initial move of M; in this case $z = x$ and $\beta = S$, so that y can be chosen to be Λ.

Suppose $k \geq 1$ and our statement is true for any $n \geq k$. Suppose also that some sequence of $k+1$ moves produces the configuration $(q_1, z, \beta Z_0)$. We must show that $x = yz$ for some y and that $S \Rightarrow^* y\beta$.

There are two possibilities for the $(k+1)$th move in this sequence. One is that the next-to-last configuration is $(q_1, az, a\beta Z_0)$ and the last move is of type (1). In this case the induction hypothesis implies that for some $y' \in \Sigma^*$, $x = y'az$ and $S \Rightarrow^* y'a\beta$. We get the conclusion immediately by letting $y = y'a$.

The other possibility is that the next-to-last configuration is $(q_1, z, A\gamma Z_0)$ for some variable A for which there is a production $A \rightarrow \alpha$ with $\alpha\gamma = \beta$. In this case, the induction hypothesis tells us that $x = yz$ for some $y \in \Sigma^*$ and $S \Rightarrow^* yA\gamma$. But then

$$S \Rightarrow^* yA\gamma \Rightarrow y\alpha\gamma = y\beta$$

This completes the induction proof.

Now, if x is any string in $L(M)$,

$$(q_0, x, Z_0) \vdash^* (q_1, \Lambda, Z_0) \vdash (q_2, \Lambda, Z_0)$$

because M can enter q_2 only via a Λ-transition with Z_0 on top of the stack. Letting $y = x$ and $z = \beta = \Lambda$ in the statement we proved by induction, we may conclude that $S \Rightarrow_G^* x$ and therefore that $x \in L(G)$.

A Top-down PDA for the Strings with More *a*'s than *b*'s

EXAMPLE 7.5

Consider the language

$$L = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$$

for which we constructed a DPDA in Example 7.4. From Example 6.10, we know that one context-free grammar for L is the one with productions

$$S \rightarrow a \mid aS \mid bSS \mid SSb \mid SbS$$

Following the construction in the proof of Theorem 7.2, we obtain the PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $\Gamma = \{S, a, b, Z_0\}$, $A = \{q_2\}$, and the transition function δ is defined by this table.

State	Input	Stack symbol	Move(s)
q_0	Λ	Z_0	(q_1, SZ_0)
q_1	Λ	S	(q_1, a), (q_1, aS), (q_1, bSS), (q_1, SSb), (q_1, SbS)
q_1	a	a	(q_1, Λ)
q_1	b	b	(q_1, Λ)
q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)			none

We consider the string $x = abbaaa \in L$ and compare the moves made by M in accepting x with a leftmost derivation of x in the grammar. Each move in which a variable is replaced on the stack by a string corresponds to a step in a leftmost derivation of x, and that step is shown to the right of the move. Observe that at each step, the stack contains (in addition to Z_0) the portion of the current string in the derivation that remains after removing the initial string of

terminals read so far.

$$
\begin{array}{lll}
(q_0, abbaaa, Z_0) & & \\
\vdash (q_1, abbaaa, SZ_0) & S & \\
\vdash (q_1, abbaaa, SbSZ_0) & \Rightarrow SbS & \\
\vdash (q_1, abbaaa, abSZ_0) & \Rightarrow abS & \\
\vdash (q_1, bbaaa, bSZ_0) & & \\
\vdash (q_1, baaa, SZ_0) & & \\
\vdash (q_1, baaa, bSSZ_0) & \Rightarrow abbSS & \\
\vdash (q_1, aaa, SSZ_0) & & \\
\vdash (q_1, aaa, aSZ_0) & \Rightarrow abbaS & \\
\vdash (q_1, aa, SZ_0) & & \\
\vdash (q_1, aa, aSZ_0) & \Rightarrow abbaaS & \\
\vdash (q_1, a, SZ_0) & & \\
\vdash (q_1, a, aZ_0) & \Rightarrow abbaaa & \\
\vdash (q_1, \Lambda, Z_0) & & \\
\vdash (q_2, \Lambda, Z_0) & &
\end{array}
$$

You may wish to trace the other possible sequences of moves by which x can be accepted, corresponding to other possible leftmost derivations of x in this CFG.

The opposite approach to top-down is **bottom-up**. In this approach, there are opposite counterparts to both types of moves in the top-down PDA. Instead of replacing a variable A on the stack by the right side α of a production $A \rightarrow \alpha$ (which effectively extends the tree downward), the PDA removes α from the stack and replaces it by (or "reduces it to") A, so that the tree is extended upward. In both approaches, the contents of the stack represents a portion of the current string in the derivation being simulated; instead of removing a terminal symbol from the beginning of this portion (which appeared on the stack as a result of applying a production), the PDA "shifts" a terminal symbol from the input to the end of this portion, in order to prepare for a reduction.

Note that because shifting input symbols onto the stack reverses their order, the string α that is to be reduced to A will appear on the stack in reverse; thus the PDA begins the reduction with the *last* symbol of α on top of the stack.

Note also that while the top-down approach requires only one move to apply a production $A \rightarrow \alpha$, the corresponding reduction in the bottom-up approach requires a sequence of moves, one for each symbol in the string α. We are interested primarily in the sequence as a whole, and with the natural correspondence between a production in the grammar and the sequence of moves that accomplishes the reduction.

The process terminates when the start symbol S, left on the stack by the last reduction, is popped off the stack and Z_0 is the only thing left. The entire process simulates a derivation, in reverse order, of the input string. At each step, the current string in the derivation is formed by the contents of the stack (in reverse), followed by the string of unread input; because after each reduction the variable on top of the stack is the rightmost one in the current string, the derivation being simulated in reverse is a *rightmost* derivation.

A Bottom-up PDA for Simple Algebraic Expressions

EXAMPLE 7.6

Let us illustrate this approach using the grammar G, a simplified version of the one in Section 6.4, with productions

$$
\begin{array}{ll}
(1) & S \rightarrow S + T \\
(2) & S \rightarrow T \\
(3) & T \rightarrow T * a \\
(4) & T \rightarrow a
\end{array}
$$

The reason for numbering the productions will be seen presently. Suppose the input string is $a + a * a\$$, which has the rightmost derivation

$$
\begin{aligned}
S &\Rightarrow S + T \\
&\Rightarrow S + T * a \\
&\Rightarrow S + a * a \\
&\Rightarrow T + a * a \\
&\Rightarrow a + a * a
\end{aligned}
$$

The corresponding steps or groups of steps executed by the bottom-up PDA as the string $a+a*a$ is processed are shown in Table 7.6. Remember that at each point, the reverse of the string on the stack (omitting Z_0), followed by the string of unread input, constitutes the current string in the derivation, and the reductions occur in the opposite order from the corresponding steps in the derivation. For example, since the last step in the derivation is to replace T by a, the first reduction replaces a on the stack by T.

In Table 7.7 we show the details of the nondeterministic bottom-up PDA that carries out these moves. The shift moves allow the next input to be shifted onto the stack, regardless of the current stack symbol. The sequence of moves in a reduction can begin when the top stack symbol is the last symbol of α, for some string α in a production $A \rightarrow \alpha$. If $|\alpha| > 1$, the moves in the sequence proceed on the assumption that the symbols below the top one are in fact the previous symbols of α; they remove these symbols, from back to front, and place A

Table 7.6 | Processing of $a + a * a$ by the bottom-up PDA corresponding to G

Move	Production	Stack	Unread input
–		Z_0	$a + a * a$
shift		aZ_0	$+a * a$
reduce	$T \rightarrow a$	TZ_0	$+a * a$
reduce	$S \rightarrow T$	SZ_0	$+a * a$
shift		$+SZ_0$	$a * a$
shift		$a + SZ_0$	$*a$
reduce	$T \rightarrow a$	$T + SZ_0$	$*a$
shift		$*T + SZ_0$	a
shift		$a * T + SZ_0$	–
reduce	$T \rightarrow T * a$	$T + SZ_0$	–
reduce	$S \rightarrow S + T$	SZ_0	–
(pop S)		Z_0	–
(accept)			

Table 7.7 | The nondeterministic bottom-up PDA for G

State	Input	Stack symbol	Move(s)
Shift moves (σ and X are arbitrary)			
q	σ	X	$(q, \sigma X)$
Moves to reduce $S + T$ to S			
q	Λ	T	$(q_{1,1}, \Lambda)$
$q_{1,1}$	Λ	$+$	$(q_{1,2}, \Lambda)$
$q_{1,2}$	Λ	S	(q, S)
Moves to reduce T to S			
q	Λ	T	(q, S)
Moves to reduce $T * a$ to T			
q	Λ	a	$(q_{3,1}, \Lambda)$
$q_{3,1}$	Λ	$*$	$(q_{3,2}, \Lambda)$
$q_{3,2}$	Λ	T	(q, T)
Moves to reduce a to T			
q	Λ	a	(q, T)
Moves to accept			
q	Λ	S	(q_1, Λ)
q_1	Λ	Z_0	(q_2, Λ)
(all other combinations)			none

on the stack. Once such a sequence is started, a set of states unique to this sequence is what allows the PDA to remember how to complete the sequence. Suppose for example that we want to reduce the string $T * a$ to T. If we begin in some state q, with a on top of the stack, the first step will be to remove a and enter a state that we might call $q_{3,1}$. (Here is where we use the numbering of the productions: The notation is supposed to suggest that the PDA has completed one step of the reduction associated with production 3.) Starting in state $q_{3,1}$, the machine expects to see $*$ on the stack. If it does, it removes it and enters state $q_{3,2}$, from which the only possible move is to remove T from the stack, replace it by T, and return to q. Of course, all the moves of this sequence are Λ-transitions, affecting only the stack.

Apart from the special states used for reductions, the PDA stays in the state q during almost all the processing. When S is on top of the stack, it pops S and moves to q_1, from which it enters the accepting state q_2 if at that point the stack is empty except for Z_0. The input alphabet is $\{a, +, *\}$ and the stack alphabet is $\{a, +, *, S, T, Z_0\}$.

Note that in the shift moves, a number of combinations of input and stack symbol could be omitted. For example, when a string in the language is processed, the symbol $+$ will never occur simultaneously as both the input symbol and stack symbol. It does no harm to include these, however, since no string giving rise to these combinations will be reduced to S.

It can be shown without difficulty that this nondeterministic PDA accepts the language generated by G. Moreover, for any CFG, a nondeterministic PDA can be constructed along the same lines that accepts the corresponding language.

7.5 | A CONTEXT-FREE GRAMMAR CORRESPONDING TO A GIVEN PDA

It will be useful in this section to keep in mind the nondeterministic top-down PDA constructed in the previous section to simulate leftmost derivations in a given context-free grammar. If at some point in a derivation the current string is $x\alpha$, where x is a string of terminals, then at some point in the simulation, the input string read so far is x and the stack contains the string α. The stack alphabet of the PDA is $\Sigma \cup V$. The moves are defined so that variables are removed from the stack and replaced by the right sides of productions, and terminals on the stack are used to match input symbols. In this construction the states are almost incidental; after the initial move, the PDA stays in the same state until it is ready to accept.

Now we consider the opposite problem, constructing a context-free grammar that generates the language accepted by a given PDA. The argument is reasonably complicated, but it will be simplified somewhat if we can assume that the PDA accepts the language *by empty stack* (see Section 7.2). Our first job, therefore, is to convince ourselves that this assumption can be made without loss of generality. We state the result in Theorem 7.3 below and give a brief sketch of the proof, leaving the details to the exercises.

Theorem 7.3

Suppose $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a pushdown automaton accepting the language $L \subseteq \Sigma^*$. Then there is another PDA $M_1 = (Q_1, \Sigma, \Gamma_1, q_1, Z_1, A_1, \delta_1)$ accepting L by empty stack. In other words, for any string x, $x \in L$ if and only if $(q_1, x, Z_1) \vdash^*_{M_1} (q, \Lambda, \Lambda)$ for some state $q \in Q_1$.

Sketch of proof At the point when our original PDA M accepts a string, its stack may not be empty. We would like to construct M_1 so that as the two machines M and M_1 process the same input string, M_1's stack will be empty precisely when M enters an accepting state. If we simply make M_1 a replica of M, but with the ability to empty its stack automatically, without reading any more input, whenever M enters an accepting state, then we will have part of what we want: Any time M enters an accepting state, the string read so far will be accepted by M_1 by empty stack. The reason this is not quite sufficient is that M might crash (in a nonaccepting state) with an empty stack; in this case, since M_1 copies M exactly, its stack will be empty too, and it will therefore accept by empty stack a string that M does not accept. To avoid this, we let M_1 start by placing on its stack a special symbol *under* the start symbol of M. This special symbol will allow M_1 to avoid emptying its stack until it is appropriate to do so.

The way we allow M_1 to empty its stack automatically when M enters an accepting state is to provide it with a Λ-transition from this state to a special "stack-emptying" state, from which there are other Λ-transitions that just keep popping symbols off the stack until it is empty.

Now we may return to the problem we are trying to solve. We have a PDA M that accepts a language L by empty stack (let us represent this fact by writing $L = L_e(M)$), and we would like to construct a CFG generating L. It will be helpful to try to preserve as much as possible the correspondence in Theorem 7.2 between the operation of the PDA and the leftmost derivation being simulated. The current string in the derivation will consist of two parts, the string of input symbols read by the PDA so far and a remaining portion corresponding to the current stack contents. In fact, we will define our CFG so that this remaining portion consists entirely of variables, so as to highlight the correspondence between it and the stack contents: In order to produce a string of terminals, we must eventually eliminate all the variables from the current string, and in order for the input string to be accepted by the PDA (by empty stack), all the symbols on the stack must eventually be popped off.

We consider first a very simple approach, which is *too* simple to work in general. Take the variables in the grammar to be all possible stack symbols in the PDA, renamed if necessary so that no input symbols are included; take the start symbol to be Z_0; ignore the states of the PDA completely; and for each PDA move that reads a (either Λ or an element of Σ) and replaces A on the stack by $B_1B_2\cdots B_m$, introduce the production

$$A \to aB_1B_2\cdots B_m$$

This approach will give us the correspondence outlined above between the current stack contents and the string of variables remaining in the current string being derived. Moreover, it will allow the grammar to generate all strings accepted by the PDA. The reason it is too simple is that by ignoring the states of the PDA we may be allowing other strings to be derived as well. To see an example, we consider Example 7.1. This PDA accepts the language $\{xcx^r \mid x \in \{a, b\}^*\}$. The acceptance is by final state, rather than by empty stack, but we can fix this, and eliminate the state q_2 as well, by changing move 12 to

$$\delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$$

instead of $\{(q_2, Z_0)\}$. We use A and B as stack symbols instead of a and b. The moves of the PDA include these:

$$\begin{aligned}\delta(q_0, a, Z_0) &= \{(q_0, AZ_0)\}\\ \delta(q_0, c, A) &= \{(q_1, A)\}\\ \delta(q_1, a, A) &= \{(q_1, \Lambda)\}\\ \delta(q_1, \Lambda, Z_0) &= \{(q_1, \Lambda)\}\end{aligned}$$

Using the rule we have tentatively adopted, we obtain the corresponding productions

$$\begin{aligned}Z_0 &\to aAZ_0\\ A &\to cA\\ A &\to a\\ Z_0 &\to \Lambda\end{aligned}$$

The string aca has the leftmost derivation

$$Z_0 \Rightarrow aAZ_0 \Rightarrow acAZ_0 \Rightarrow acaZ_0 \Rightarrow aca$$

corresponding to the sequence of moves

$$(q_0, aca, Z_0) \vdash (q_0, ca, AZ_0) \vdash (q_1, a, AZ_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_1, \Lambda, \Lambda)$$

If we run the PDA on the input string aa instead, the initial move is

$$(q_0, aa, Z_0) \vdash (q_0, a, AZ_0)$$

and at this point the machine crashes, because it is only in state q_1 that it is allowed to read a and replace A on the stack by Λ. However, our grammar also allows the derivation

$$Z_0 \Rightarrow aAZ_0 \Rightarrow aaZ_0 \Rightarrow aa$$

In order to eliminate this problem, we must modify our grammar so as to incorporate the states of the PDA. Rather than using the stack symbols themselves as variables, we try things of the form

$$[p, A, q]$$

where p and q are states. For the variable $[p, A, q]$ to be replaced by a (either Λ or a terminal symbol), it must be the case that there is a PDA move that reads a, pops A from the stack, and takes the machine from state p to state q. More general productions involving the variable $[p, A, q]$ are to be thought of as representing any *sequence* of moves that takes the PDA from state p to state q and has the ultimate effect of removing A from the stack.

If the variable $[p, A, q]$ appears in the current string of a derivation, our goal is to replace it by Λ or a terminal symbol. This will be possible if there is a move that takes the PDA from p to q and pops A from the stack. Suppose instead, however, that there is a move from p to some state p_1, that reads a and replaces A on the stack by $B_1B_2\cdots B_m$. It is appropriate to introduce a into our current string at this point, since we want the initial string of terminals to correspond to the input read so far. But it is now also appropriate to think of our original goal as being modified, as a result of all the new symbols that have been introduced on the stack. The most direct way to eliminate these new symbols $B_1, \ldots, B_m$ is as follows: to start in p_1 and make a sequence of moves—ending up in some state p_2, say—that result in B_1 being removed from the stack; then to make some more moves that remove B_2 and in the process move from p_2 to some other state p_3; ...; to move from p_{m-1} to some p_m and remove B_{m-1}; and finally, to move from p_m to q and remove B_m. The actual moves of the PDA may not accomplish these steps directly, but this is what we want their ultimate effect to be. Because it does not matter what the states $p_2, p_3, \ldots, p_m$ are, we will allow any string of the form

$$a[p_1, B, p_2][p_2, B_2, p_3]\cdots[p_m, B_m, q]$$

to replace $[p, A, q]$ in the current string. In other words, we will introduce the productions

$$[p, A, q] \to a[p_1, B, p_2][p_2, B_2, p_3]\cdots[p_m, B_m, q]$$

for all possible sequences of states $p_2, \ldots, p_m$. Some such sequences will be dead ends, in the sense that there will be no sequence of moves following this sequence of states and having this ultimate effect. But no harm is done by introducing these productions, because for any derivation in which one of these dead-end sequences appears, there will be at least one variable that cannot be eliminated from the string, and so the derivation will not produce a string of terminals. If we denote by S the start symbol of the grammar, the productions that we need to begin are those of the form

$$S \to [q_0, Z_0, q]$$

where q_0 is the initial state. When we accept strings by empty stack the final state is irrelevant, and thus we include a production of this type for every possible state q.

We now present the proof that the CFG we have described generates the language accepted by M.

Theorem 7.4
Let $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ be a pushdown automaton accepting a language L by empty stack; that is, $L = L_e(M)$. Then there is a context-free grammar G with $L(G) = L$.

Proof
We define $G = (V, \Sigma, S, P)$ as follows:

$$V = \{S\} \cup \{[p, A, q] \mid A \in \Gamma,\ p, q \in Q\}$$

The set P contains the following productions and only these:

1. For every $q \in Q$, the production $S \to [q_0, Z_0, q]$ is in P.
2. For every $q, q_1 \in Q$, $a \in \Sigma \cup \{\Lambda\}$, and $A \in \Gamma$, if $\delta(q, a, A)$ contains (q_1, Λ), then the production $[q, A, q_1] \to a$ is in P.
3. For every $q, q_1 \in Q$, $a \in \Sigma \cup \{\Lambda\}$, $A \in \Gamma$, and $m \geq 1$, if $\delta(q, a, A)$ contains $(q_1, B_1 B_2 \cdots B_m)$ for some $B_1, \ldots, B_m \in \Gamma$, then for every choice of $q_2, \ldots, q_{m+1} \in Q$, the production

$$[q, A, q_{m+1}] \to a[q_1, B_1, q_2][q_2, B_2, q_3] \ldots [q_m, B_m, q_{m+1}]$$

is in P.

The main idea of the proof is to characterize the strings of terminals that can be derived from a variable $[q, A, q']$; specifically, to show that for any $q, q' \in Q$, $A \in \Gamma$, and $x \in \Sigma^*$,

(1) $[q, A, q'] \Rightarrow_G^* x$ if and only if $(q, x, A) \vdash_M^* (q', \Lambda, \Lambda)$

From this result, the theorem will follow. On the one hand, if $x \in L_e(M)$, then $(q_0, x, Z_0) \vdash_M^* (q, \Lambda, \Lambda)$ for some $q \in Q$; then (1) implies that $[q_0, Z_0, q] \Rightarrow_G^* x$; therefore, $x \in L(G)$, because we can start a derivation with a production of type 1. On the other hand, if $x \in L(G)$, then the first step in any derivation of x must be $S \Rightarrow [q_0, Z_0, q]$, for some $q \in Q$, which means that $[q_0, Z_0, q] \Rightarrow_G^* x$. It then follows from (1) that $x \in L_e(M)$.

Both parts of (1) are proved using mathematical induction. Let us introduce the notations $\Rightarrow^n$ and $\vdash^n$ (where n is a nonnegative integer) to refer to n-step derivations in the grammar and sequences of n moves of the PDA, respectively.

First we show that for every $n \geq 1$,

(2) If $[q, A, q'] \Rightarrow_G^n x$, then $(q, x, A) \vdash_M^* (q', \Lambda, \Lambda)$

For the basis step of the proof, suppose that $[q, A, q'] \Rightarrow_G^1 x$. The only production that can allow this one-step derivation is one of type 2, and this can happen only if x is either Λ or an element of Σ and $\delta(q, x, A)$ contains (q', Λ). In this case, it is obviously true that $(q, x, A) \vdash (q', \Lambda, \Lambda)$.

For the induction step, suppose that $k \geq 1$ and that whenever $[q, A, q'] \Rightarrow_G^n x$ for some $n \leq k$, $(q, x, A) \vdash^* (q', \Lambda, \Lambda)$. Now suppose that $[q, A, q'] \Rightarrow^{k+1} x$. We wish to show that $(q, x, A) \vdash^* (q', \Lambda, \Lambda)$. Since $k \geq 1$, the first step of the derivation of x must be

$$[q, A, q'] \Rightarrow a[q_1, B_1, q_2][q_2, B_2, q_3] \cdots [q_m, B_m, q']$$

for some $m \geq 1$, some $a \in \Sigma \cup \{\Lambda\}$, some sequence $B_1, B_2, \ldots, B_m \in \Gamma$, and some sequence $q_1, q_2, \ldots, q_m \in Q$, so that $\delta(q, a, A)$ contains $(q_1, B_1 \cdots B_m)$. The remaining part of the derivation takes each of the variables $[q_i, B_i, q_{i+1}]$ to some string x_i, and the variable $[q_m, B_m, q']$ to a string x_m. The strings $x_1, \ldots, x_m$ satisfy the formula $ax_1 \cdots x_m = x$, and each x_i is derived from its respective variable in k or fewer steps. The induction hypothesis, therefore, implies that for each i with $1 \leq i \leq m - 1$,

$$(q_i, x_i, B_i) \vdash^* (q_{i+1}, \Lambda, \Lambda)$$

and that

$$(q_m, x_m, B_m) \vdash^* (q', \Lambda, \Lambda)$$

Suppose M is in the configuration $(q, x, A) = (q, ax_1x_2 \cdots x_m, A)$. Because $\delta(q, a, A)$ contains $(q_1, B_1 \cdots B_m)$, M can go in one step to the configuration

$$(q_1, x_1x_2 \cdots x_m, B_1B_2 \cdots B_m)$$

M can then go in a sequence of steps to

$$(q_2, x_2 \cdots x_m, B_2 \cdots B_m)$$

then to $(q_3, x_3 \cdots x_m, B_3 \cdots B_m)$, and ultimately to (q', Λ, Λ). Thus the result (2) follows.

To complete the proof of (1), we show that for every $n \geq 1$,

(3) If $(q, x, A) \vdash^n (q', \Lambda, \Lambda)$, then $[q, A, q'] \Rightarrow^* x$

In the case $n = 1$, a string x satisfying the hypothesis in (3) must be of length 0 or 1, and $\delta(q, x, A)$ must contain (q', Λ). In this case, we may derive x from $[q, A, q']$ using a production of type 2.

For the induction step, we suppose that $k \geq 1$ and that for any $n \leq k$, and any combination of $q, q' \in Q$, $x \in \Sigma^*$, and $A \in \Gamma$, if $(q, x, A) \vdash^n (q', \Lambda, \Lambda)$, then $[q, A, q'] \Rightarrow^* x$. Next we suppose that $(q, x, A) \vdash^{k+1} (q', \Lambda, \Lambda)$, and we wish to show that $[q, A, q'] \Rightarrow^* x$. We know that for some $a \in \Sigma \cup \{\Lambda\}$ and some $y \in \Sigma^*$, $x = ay$ and the first of the $k + 1$ moves is

$$(q, x, A) = (q, ay, A) \vdash (q_1, y, B_1 B_2 \cdots B_m)$$

Here $m \geq 1$, since $k \geq 1$, and the B_i's are elements of Γ. In other words, $\delta(q, a, A)$ contains $(q_1, B_1 \cdots B_m)$. The k subsequent moves end in the configuration (q', Λ, Λ); therefore, for each i with $1 \leq i \leq m$ there must be intermediate points at which the stack contains precisely the string $B_i B_{i+1} \cdots B_m$. For each such i, let q_i be the state M is in the first time the stack contains $B_i \cdots B_m$, and let x_i be the portion of the input string that is consumed in going from q_i to q_{i+1} (or, if $i = m$, in going from q_m to the configuration (q', Λ, Λ)). Then it must be the case that

$$(q_i, x_i, B_i) \vdash^* (q_{i+1}, \Lambda, \Lambda)$$

for each i with $1 \leq i \leq m - 1$, and

$$(q_m, x_m, B_m) \vdash^* (q', \Lambda, \Lambda)$$

where each of the indicated sequences of moves has k or fewer. Therefore, by the induction hypothesis,

$$[q_i, B_i, q_{i+1}] \Rightarrow_G^* x_i$$

for each i with $1 \leq i \leq m - 1$, and

$$[q_m, B_m, q'] \Rightarrow^* x_m$$

Since $\delta(q, a, A)$ contains $(q_1, B_1 \cdots B_m)$, we know that

$$[q, A, q'] \Rightarrow a[q_1, B_1, q_2][q_2, B_2, q_3] \cdots [q_m, B_m, q']$$

(this is a production of type 3), and we may conclude that

$$[q, A, q'] \Rightarrow^* a x_1 x_2 \cdots x_m = x$$

This completes the induction and the proof of the theorem.

EXAMPLE 7.7 Obtaining a CFG from a PDA Accepting Simple Palindromes

We return once more to the language $L = \{xcx^r \mid x \in \{a, b\}^*\}$ of Example 7.1, which we used to introduce the construction in Theorem 7.4. In that discussion we used the PDA whose transition table is shown below. (It is modified from the one in Example 7.1, both in using uppercase letters for stack symbols and in accepting by empty stack.)

Move number	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_0, AZ_0)
2	q_0	b	Z_0	(q_0, BZ_0)
3	q_0	a	A	(q_0, AA)
4	q_0	b	A	(q_0, BA)
5	q_0	a	B	(q_0, AB)
6	q_0	b	B	(q_0, BB)
7	q_0	c	Z_0	(q_1, Z_0)
8	q_0	c	A	(q_1, A)
9	q_0	c	B	(q_1, B)
10	q_1	a	A	(q_1, Λ)
11	q_1	b	B	(q_1, Λ)
12	q_1	Λ	Z_0	(q_1, Λ)
(all other combinations)				none

In the grammar $G = (V, \Sigma, S, P)$ obtained from the construction in Theorem 7.4, V contains S as well as every object of the form $[p, X, q]$, where X is a stack symbol and p and q can each be either q_0 or q_1. Productions of the following types are contained in P:

(0) $S \rightarrow [q_0, Z_0, q]$
(1) $[q_0, Z_0, q] \rightarrow a[q_0, A, p][p, Z_0, q]$
(2) $[q_0, Z_0, q] \rightarrow b[q_0, B, p][p, Z_0, q]$
(3) $[q_0, A, q] \rightarrow a[q_0, A, p][p, A, q]$
(4) $[q_0, A, q] \rightarrow b[q_0, B, p][p, A, q]$
(5) $[q_0, B, q] \rightarrow a[q_0, A, p][p, B, q]$
(6) $[q_0, B, q] \rightarrow b[q_0, B, p][p, B, q]$
(7) $[q_0, Z_0, q] \rightarrow c[q_1, Z_0, q]$
(8) $[q_0, A, q] \rightarrow c[q_1, A, q]$
(9) $[q_0, B, q] \rightarrow c[q_1, B, q]$
(10) $[q_1, A, q_1] \rightarrow a$
(11) $[q_1, B, q_1] \rightarrow b$
(12) $[q_1, Z_0, q_1] \rightarrow \Lambda$

Allowing all combinations of p and q gives 35 productions in all.

Consider the string $bacab$. The PDA accepts it by the sequence of moves

$$\begin{aligned}(q_0, bacab, Z_0) &\vdash (q_0, acab, BZ_0)\\ &\vdash (q_0, cab, ABZ_0)\\ &\vdash (q_1, ab, ABZ_0)\\ &\vdash (q_1, b, BZ_0)\\ &\vdash (q_1, \Lambda, Z_0)\\ &\vdash (q_1, \Lambda, \Lambda)\end{aligned}$$

The corresponding leftmost derivation in the grammar is

$$\begin{aligned}S &\Rightarrow [q_0, Z_0, q_1]\\ &\Rightarrow b[q_0, B, q_1][q_1, Z_0, q_1]\end{aligned}$$

$$\Rightarrow ba[q_0, A, q_1][q_1, B, q_1][q_1, Z_0, q_1]$$
$$\Rightarrow bac[q_1, A, q_1][q_1, B, q_1][q_1, Z_0, q_1]$$
$$\Rightarrow baca[q_1, B, q_1][q_1, Z_0, q_1]$$
$$\Rightarrow bacab[q_1, Z_0, q_1]$$
$$\Rightarrow bacab$$

From the sequence of PDA moves, it may look as though there are several choices of leftmost derivations. For example, we might start with the production $S \rightarrow [q_0, Z_0, q_0]$. Remember, however, that $[q_0, Z_0, q]$ represents a sequence of moves from q_0 to q that has the ultimate effect of removing Z_0 from the stack. Since the PDA ends up in state q_1, it is clear that q should be q_1. Similarly, it may seem as if the second step could be

$$[q_0, Z_0, q_1] \Rightarrow b[q_0, B, q_0][q_0, Z_0, q_1]$$

However, the sequence of PDA moves that starts in q_0 and eliminates B from the stack ends with the PDA in state q_1, not q_0. In fact, because every move to state q_0 adds to the stack, the variable $[q_0, B, q_0]$ in this grammar is useless: No string of terminals can be derived from it.

7.6 | PARSING

Suppose that G is a context-free grammar over an alphabet Σ. To *parse* a string $x \in \Sigma^*$ (to find a derivation of x in the grammar G, or to determine that there is none) is often useful. Parsing a statement in a programming language, for example, is necessary in order to classify it according to syntax; parsing an algebraic expression is essentially what allows us to evaluate the expression. The problem of finding efficient parsing algorithms has led to a great deal of research, and there are many specialized techniques that depend on specific properties of the grammar.

In this section we return to the two natural ways presented in Section 7.4 of obtaining a PDA to accept the language $L(G)$. In both cases, the PDA not only accepts a string in $L(G)$ but does it by simulating a derivation of x (in one case a leftmost derivation, in the other case rightmost). Although the official output of a PDA is just a yes-or-no answer, it is easy enough to enhance the machine slightly by allowing it to record its moves, so that any sequence of moves leading to acceptance causes a derivation to be displayed. However, neither construction by itself can be said to produce a parsing algorithm, because both PDAs are inherently nondeterministic. In each case, the simulation proceeds by *guessing* the next step in the derivation; if the guess is correct, its correctness will be confirmed eventually by the PDA.

One approach to obtaining a parsing algorithm would be to consider all possible sequences of guesses the PDA might make, in order to see whether one of them leads to acceptance. Exercise 7.47 asks you to use a backtracking strategy for doing this with a simple CFG. However, it is possible with both types of nondeterministic PDAs to confront the nondeterminism more directly: rather than making an arbitrary choice and then trying to confirm that it was the right one, trying instead to use all the information available in order to select the choice that will be correct. In the

remainder of this section, we concentrate on two simple classes of grammars, for which the next input symbol and the top stack symbol in the corresponding PDA provide enough information at each step to determine the next move in the simulated derivation. In more general grammars, the approach at least provides a starting point for the development of an efficient parser.

7.6.1 Top-down parsing

A Top-down Parser for Balanced Strings of Parentheses — EXAMPLE 7.8

We consider the language of balanced strings of parentheses. For convenience, we modify it slightly by adding a special endmarker \$ to the end of each string. The new language will be denoted L. If we use [ ] as our parentheses, the context-free grammar with productions

$$S \rightarrow T\$$$
$$T \rightarrow [T]T \mid \Lambda$$

is an unambiguous CFG generating L. In the top-down PDA obtained from this grammar, the only nondeterminism arises when the variable T is on the top of the stack, and we have a choice of two moves using input Λ. If the next input symbol is [, then the correct move (or, conceivably, sequence of moves) must produce a [on top of the stack to match it. Replacing T by $[T]T$ will obviously do this; replacing T by Λ would have a chance of being correct only if the symbol below T were either [or T, and it is not hard to see that this never occurs. It appears, therefore, that if T is on top of the stack, T should be replaced by $[T]T$ if the next input symbol is [and by Λ if the next input is] or \$. The nondeterminism can be eliminated by *lookahead*—using the next input symbol as well as the stack symbol to determine the move.

In the case when T is on the stack and the next input symbol is either] or \$, popping T from the stack will lead to acceptance only if the symbol beneath it matches the input; thus the PDA needs to remember the input symbol long enough to match it with the new stack symbol. We can accomplish this by introducing the two states $q_]$ and $q_\$$ to which the PDA can move on the respective input symbol when T is on top of the stack. In either of these states, the only correct move is to pop the corresponding symbol from the stack and return to q_1. For the sake of consistency, we also use a state $q_[$ for the case when T is on top of the stack and the next input is [. Although in this case T is replaced on the stack by the longer string $[T]T$, the move from $q_]$ is also to pop the [from the stack and return to q_1. The alternative, which would be slightly more efficient, would be to replace these two moves by a single one that leaves the PDA in q_1 and replaces T on the stack by $T]T$.

The transition table for the original nondeterministic PDA is shown in Table 7.8, and Table 7.9 describes the deterministic PDA obtained by incorporating lookahead.

The sequence of moves by which the PDA accepts the string [ ]\$, and the corresponding steps in the leftmost derivation of this string, are shown below.

$$(q_0, [\,]\$, Z_0)$$
$$\vdash (q_1, [\,]\$, SZ_0) \qquad S$$
$$\vdash (q_1, [\,]\$, T\$Z_0) \qquad \Rightarrow T\$$$
$$\vdash (q_[,]\$, [T]T\$Z_0) \qquad \Rightarrow [T]T\$$$
$$\vdash (q_1,]\$, T]T\$Z_0)$$

Table 7.8 | A nondeterministic top-down PDA for balanced strings of parentheses

Move number	State	Input	Stack symbol	Move
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	$(q_1, T\$)$
3	q_1	Λ	T	$(q_1, [T]T), (q_1, \Lambda)$
4	q_1	[	[	(q_1, Λ)
5	q_1	]	]	(q_1, Λ)
6	q_1	\$	\$	(q_1, Λ)
7	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

Table 7.9 | Using lookahead to eliminate the nondeterminism from Table 7.8

1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	$(q_1, T\$)$
3	q_1	[	T	$(q_{[}, [T]T)$
4	$q_{[}$	Λ	[	(q_1, Λ)
5	q_1	]	T	$(q_{]}, \Lambda)$
6	$q_{]}$	Λ	]	(q_1, Λ)
7	q_1	\$	T	$(q_{\$}, \Lambda)$
8	$q_{\$}$	Λ	\$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	\$	\$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

$$\begin{aligned}
&\vdash (q_{]}, \$,]T\$Z_0) && \Rightarrow\ ]T\$\\
&\vdash (q_1, \$, T\$Z_0)\\
&\vdash (q_{\$}, \Lambda, \$Z_0) && \Rightarrow\ ]\$\\
&\vdash (q_1, \Lambda, Z_0)\\
&\vdash (q_2, \Lambda, Z_0)
\end{aligned}$$

You can probably see that moves 9, 10, and 11 in the deterministic PDA, which were retained from the nondeterministic machine, will never actually be used. We include them because in a more general example moves of this type may still be necessary. Although we do not give a proof that the deterministic PDA accepts the language, you can convince yourself by tracing the moves for a few longer input strings.

In the top-down PDA obtained from a context-free grammar as in Definition 7.4, looking ahead to the next input may not be enough to determine the next move. Sometimes, however, straightforward modifications of the grammar are enough to establish this property, as the next two examples indicate.

Eliminating Left Recursion in a CFG — EXAMPLE 7.9

Another unambiguous CFG for the language of Example 7.8 is the one with productions

$$S \to T\$$$

$$T \to T[T] \mid \Lambda$$

The standard top-down PDA produced from this grammar is exactly the same as the one in Example 7.8, except for the string replacing T on the stack in the first move of line 3. We can see the potential problem by considering the input string [][][]\$, which has the leftmost derivation

$$S \Rightarrow T\$ \Rightarrow T[T]\$ \Rightarrow T[T][T]\$ \Rightarrow T[T][T][T]\$ \Rightarrow \cdots \Rightarrow [][][]\$$$

The correct sequence of moves for this input string therefore begins

$$\begin{aligned}
(q_0, [][][]\$, Z_0) &\vdash (q_1, [][][]\$, SZ_0)\\
&\vdash (q_1, [][][]\$, T\$Z_0)\\
&\vdash (q_1, [][][]\$, T[T]\$Z_0)\\
&\vdash (q_1, [][][]\$, T[T][T]\$Z_0)\\
&\vdash (q_1, [][][]\$, T[T][T][T]\$Z_0)
\end{aligned}$$

In each of the last four configurations shown, the next input is [and the top stack symbol is T, but the correct sequences of moves beginning at these four points are all different. Since the remaining input is exactly the same in all these configurations, looking ahead to the next input, or even farther ahead, will not help; there is no way to choose the next move on the basis of the input.

The problem arises because of the production $T \to T[T]$, which illustrates the phenomenon of *left recursion*. Because the right side begins with T, the PDA must make a certain number of identical moves before it does anything else, and looking ahead in the input provides no help in deciding how many. In this case we can eliminate the left recursion by modifying the grammar. Suppose in general that a grammar has the T-productions

$$T \to T\alpha \mid \beta$$

where the string β does not begin with T. These allow all the strings $\beta\alpha^n$, for $n \geq 0$, to be obtained from T. If these two productions are replaced by

$$T \to \beta U \qquad U \to \alpha U \mid \Lambda$$

the language is unchanged and the left recursion has been eliminated. In our example, with $\alpha = [T]$ and $\beta = \Lambda$, we replace

$$T \to T[T] \mid \Lambda$$

by

$$T \to U \qquad U \to [T]U \mid \Lambda$$

and the resulting grammar allows us to construct a deterministic PDA much as in Example 7.8.

EXAMPLE 7.10 Factoring in a CFG

Consider the context-free grammar with productions

$$S \to T\$$$
$$T \to [T] \mid [\,]T \mid [T]T \mid [\,]$$

This is the unambiguous grammar obtained from the one in Example 7.8 by removing Λ-productions from the CFG with productions $T \to [T]T \mid \Lambda$; the language is unchanged except that it no longer contains the string \$.

Although there is no left recursion in the CFG, we can tell immediately that knowing the next input symbol will not be enough to choose the nondeterministic PDA's next move when T is on top of the stack. The problem here is that the right sides of all four T-productions begin with the same symbol. An appropriate remedy is to "factor" the right sides, as follows:

$$T \to [U \qquad U \to T] \mid]T \mid T]T \mid]$$

More factoring is necessary because of the U-productions; in the ones whose right side begins with T, we can factor out $T]$. We obtain the productions

$$S \to T\$ \qquad T \to [U$$
$$U \to T]W \mid]W \qquad W \to T \mid \Lambda$$

We can simplify the grammar slightly by eliminating the variable T, and we obtain

$$S \to [U\$ \qquad U \to [U]W \mid]W \qquad W \to [U \mid \Lambda$$

The DPDA we obtain by incorporating lookahead is shown in Table 7.10.

Table 7.10 | A deterministic top-down PDA for Example 7.10

Move number	State	Input	Stack symbol	Move
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	$(q_1, [U\$)$
3	q_1	[	U	$(q_1, [U]W$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	U	$(q_1,]W)$
6	q_1	Λ	]	(q_1, Λ)
7	q_1	[	W	$(q_1, [U)$
8	q_1	]	W	(q_1, Λ)
9	q_1	\$	W	$(q_\$, \Lambda)$
10	$q_\$$	Λ	\$	(q_1, Λ)
11	q_1	[	[	(q_1, Λ)
12	q_1	]	]	(q_1, Λ)
13	q_1	\$	\$	(q_1, Λ)
14	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

In Examples 7.9 and 7.10, we were able by a combination of factoring and eliminating left recursion to transform the CFG into what is called an LL(1) grammar,

meaning that the nondeterministic top-down PDA produced from the grammar can be turned into a deterministic top-down parser by looking ahead to the next symbol. A grammar is LL(k) if looking ahead k symbols in the input is always enough to choose the next move of the PDA. Such a grammar allows the construction of a deterministic top-down parser, and there are systematic methods for determining whether a CFG is LL(k) and for carrying out this construction (see the references).

For an LL(1) context-free grammar, a deterministic PDA is one way of formulating the algorithm that decides the next step in the derivation of a string by looking at the next input symbol. The method of *recursive descent* is another way. The name refers to a collection of mutually recursive procedures corresponding to the variables in the grammar.

A Recursive-descent Parser for the LL(1) Grammar in Example 7.10 EXAMPLE 7.11

The context-free grammar is the one with productions

$$S \to [U\$$$
$$U \to]W \mid [U]W$$
$$W \to [U \mid \Lambda$$

We give a C++ version of a recursive-descent parser. The term *recognizer* is really more accurate than *parser*, though it would not be difficult to add output statements to the program that would allow one to reconstruct a derivation of the string being recognized.

The program involves functions `s`, `u`, and `w`, corresponding to the three variables. Calls on these three functions correspond to substitutions for the respective variables during a derivation—or to replacement of those variables on the stack in a PDA implementation. There is a global variable `curr_ch`, whose value is assigned before any of the three functions is called. If the current character is one of those that the function expects, it is *matched*, and the input function is called to read and echo the next character. Otherwise, an error-handling function is called and told what the character should have been; in this case, the program terminates with an appropriate error message.

Note that the program's correctness depends on the grammar's being LL(1), since each of the functions can select the correct action on the basis of the current input symbol.

```
#include <iostream.h>
#include <stdlib.h>

char curr_ch;          // the current symbol

void s(), u(), w();    // recognize S, U, W, respectively
void match(char);      // compares curr_ch to the argument; aborts with
                       // error message if no match, otherwise returns.
void get_ch();         // reads the next symbol into curr_ch, with echo.
void error(char*);     // reports an error and aborts.  String argument.
void error(char);      //                               Character argument.
```

```
void main()
{  get_ch(); s();
   cout << endl << "Parsing complete.  "
        << "The above string is in the language." << endl;
}

void s()                                        // recognizes [U$
{  match('['); u(); match('$'); }

void u()                                        // recognizes ]W  | [U]W
{  switch (curr_ch)
   {  case ']': match(']'); w(); break;   // production ]W
      case '[':                           // production [U]W
               match('['); u(); match(']'); w(); break;
      default : error("[ or ]");
   }
}

void w()                                   // recognizes [U  |  <Lambda>
{  if (curr_ch == '[')    { match('['); u(); }  }

void get_ch()                       // read and echo next nonblank symbol
{  if (cin >> curr_ch)  cout << curr_ch;
   if (cin.eof() && curr_ch != '$')
   {  cout << " (End of Data)"; error("[ or ]");  }
}

void match(char this_ch)
{  if (curr_ch == this_ch)  get_ch();  else error(this_ch);  }

void error(char* some_chars)
{  cout << "\n  ERROR : Expecting one of " << some_chars << ".\n";
   exit(0);
}

void error(char a_char)
{  cout << "\n  ERROR : Expecting " << a_char << ".\n";
   exit(0);
}
```

Here is a sample of the output produced, for the strings `[][[][[]]]$`, `$`, `[]]`, and `[[`, respectively.

```
[][[][[]]]$
Parsing complete.  The above string is in the language.
$
  ERROR : Expecting [.
[]]
  ERROR : Expecting $.
[[ (End of Data)
  ERROR : Expecting one of [ or ].
```

The program is less complete than it might be, in several respects. In the case of a string not in the language, it reads and prints out only the symbols up to the first illegal one—that is, up to the point where the DPDA would crash. In addition, it does not read past the `$`; if the input string were `[]$]`, for example, the program would merely report that `[]$` is in the language. Finally, the error messages may seem slightly questionable. The second error, for example, is detected in the function `s`, after the return from the call on `u`. The production is $S \rightarrow [U\$$; the function "expects" to see $ at this point, although not every symbol other than $ would have triggered the error message. The symbol `[` would be valid here and would have resulted in a different sequence of function calls, so that the program would not have performed the same test at this point in `s`.

7.6.2 Bottom-up parsing

Example 7.12 illustrates one of the simplest ways of obtaining a deterministic bottom-up parser from a nondeterministic bottom-up PDA.

A Deterministic Bottom-up Parser for a CFG — EXAMPLE 7.12

We consider the context-free grammar G with productions

(0) $S \rightarrow S_1\$$
(1) $S_1 \rightarrow S_1 + T$
(2) $S_1 \rightarrow T$
(3) $T \rightarrow T * a$
(4) $T \rightarrow a$

The last four are essentially those in Example 7.6; the endmarker $ introduced in production (0) will be useful here, as it was in the discussion of top-down parsing.

Table 7.11 shows the nondeterministic PDA in Example 7.6 with the additional reduction corresponding to grammar rule (0).

The other slight difference from the PDA in Example 7.6 is that because the start symbol S occurs only in production (0) in the grammar, the PDA can move to the accepting state as soon as it sees S on the stack.

Nondeterminism is present in two ways. First, there may be a choice as to whether to shift an input symbol onto the stack or to try to reduce a string on top of the stack. For example, if T is the top stack symbol, the first choice is correct if it is the T in the right side of $T * a$, and the second is correct if it is the T in one of the S_1-productions. Second, there may be some

Table 7.11 | A nondeterministic bottom-up parser for G

State	Input	Stack symbol	Move(s)
Shift moves (σ and X are arbitrary)			
q	σ	X	$(q, \sigma X)$
Moves to reduce $S_1\$$ to S			
q	Λ	$\$$	$(q_{0,1}, \Lambda)$
$q_{0,1}$	Λ	S_1	(q, S)
Moves to reduce $S_1 + T$ to S_1			
q	Λ	T	$(q_{1,1}, \Lambda)$
$q_{1,1}$	Λ	$+$	$(q_{1,2}, \Lambda)$
$q_{1,2}$	Λ	S_1	(q, S_1)
Moves to reduce T to S_1			
q	Λ	T	(q, S_1)
Moves to reduce $T * a$ to T			
q	Λ	a	$(q_{3,1}, \Lambda)$
$q_{3,1}$	Λ	$*$	$(q_{3,2}, \Lambda)$
$q_{3,2}$	Λ	T	(q, T)
Moves to reduce a to T			
q	Λ	a	(q, T)
Move to accept			
q	Λ	S	(q_1, T)
(all other combinations)			none

doubt as to which reduction is the correct one; for example, there are two productions whose right sides end with a. Answering the second question is easy. When we pop a off the stack, if we find $*$ below it, we should attempt to reduce $T * a$ to T, and otherwise, we should reduce a to T. Either way, the correct reduction is the one that reduces the longest possible string.

Returning to the first question, suppose the top stack symbol is T, and consider the possibilities for the next input. If it is $+$, we should soon have the string $S_1 + T$, in reverse order, on top of the stack, and so the correct move at this point is a reduction of either T or $S_1 + T$ (depending on what is below T on the stack) to S_1. If the next input is $*$, the reduction will be that of $T * a$ to a, and since we have T already, we should shift. Finally, if it is \$, we should reduce either T or $S_1 + T$ to S_1 to allow the reduction of $S_1\$$. In any case, we can make the decision on the basis of the next input symbol. What is true for this example is that there are certain combinations of top stack symbol and input symbol for which a reduction is always appropriate, and a shift is correct for all the other combinations. The set of pairs for which a reduction is correct is an example of a *precedence relation*. (It is a relation from Γ to Σ, in the sense of Section 1.4.) There a number of types of *precedence grammars*, for which precedence relations can be used to obtain a deterministic shift-reduce parser. Our example,

in which the decision to reduce can be made by examining the top stack symbol and the next input, and in which a reduction always reduces the longest possible string, is an example of a *weak precedence grammar*.

A deterministic PDA that acts as a shift-reduce parser for our grammar is shown in Table 7.12. In order to compare it to the nondeterministic PDA, we make a few observations. The stack symbols can be divided into three groups: (1) those whose appearance on top of the stack requires a shift regardless of the next input (these are Z_0, S_1, $+$, and $*$); (2) those that require a reduction or lead to acceptance (a, \$, and S); and (3) one, T, for which the correct choice can be made only by consulting the next input. In the DPDA, shifts in which the top stack symbol is of the second type have been omitted, since they do not lead to acceptance of any string and their presence would introduce nondeterminism. Shifts in which the top stack symbol is of the first or third type are shown, labeled "shift moves." If the top stack symbol is of the second type, the moves in the reduction are all Λ-transitions. If the PDA reads a symbol and decides to reduce, the input symbol will eventually be shifted onto the stack, once the reduction has been completed (the machine must remember the input symbol during the reduction); the eventual shift is shown, not under "shift moves," but farther down in the table, as part of the sequence of reducing moves.

Table 7.12 | A deterministic bottom-up parser for G

Move number	State	Input	Stack symbol	Move
Shift moves				
1	q	σ	X	$(q, \sigma X)$
(σ is arbitrary; X is either Z_0, S_1, $+$, or $*$.)				
2	q	σ	T	$(q, \sigma T)$
(σ is any input symbol other than $+$ or \$.)				
Moves to reduce $S_1\$$ to \$				
3	q	Λ	$\$$	$(q_\$, \Lambda)$
4	$q_\$$	Λ	S_1	(q, S)
Moves to reduce either a or $T * a$ to T				
5	q	Λ	a	$(q_{a,1}, \Lambda)$
6	$q_{a,1}$	Λ	$*$	$(q_{a,2}, \Lambda)$
7	$q_{a,2}$	Λ	T	(q, T)
8	$q_{a,1}$	Λ	X	(q, TX)
(X is any stack symbol other than $*$.)				
Moves to reduce $S_1 + T$ or T to S_1 and shift an input symbol				
9	q	σ	T	$(q_{T,\sigma}, \Lambda)$
10	$q_{T,\sigma}$	Λ	$+$	$(q'_{T,\sigma}, \Lambda)$
11	$q'_{T,\sigma}$	Λ	S_1	$(q, \sigma S_1)$
12	$q_{T,\sigma}$	Λ	X	$(q, \sigma S_1 X)$
(σ is either $+$ or \$; X is any stack symbol other than $+$.)				
Move to accept				
13	q	Λ	S	(q_1, Λ)
(all other combinations)				none

We trace the moves of this PDA on the input string $a + a * a\$$.

$$
\begin{aligned}
(q, a + a * a\$, Z_0) &\vdash (q, +a * a\$, aZ_0) && \text{(move 1)}\\
&\vdash (q_{a,1}, +a * a\$, Z_0) && \text{(move 5)}\\
&\vdash (q, +a * a\$, TZ_0) && \text{(move 8)}\\
&\vdash (q_{T,+}, a * a\$, Z_0) && \text{(move 9)}\\
&\vdash (q, a * a\$, +S_1Z_0) && \text{(move 12)}\\
&\vdash (q, *a\$, a + S_1Z_0) && \text{(move 1)}\\
&\vdash (q_{a,1}, *a\$, +S_1Z_0) && \text{(move 5)}\\
&\vdash (q, *a\$, T + S_1Z_0) && \text{(move 8)}\\
&\vdash (q, a\$, *T + S_1, Z_0) && \text{(move 2)}\\
&\vdash (q, \$, a * T + S_1Z_0) && \text{(move 1)}\\
&\vdash (q_{a,1}, \$, *T + S_1Z_0) && \text{(move 5)}\\
&\vdash (q_{a,2}, \$, T + S_1Z_0) && \text{(move 6)}\\
&\vdash (q, \$, T + S_1Z_0) && \text{(move 7)}\\
&\vdash (q_{T,\$}, \Lambda, +S_1Z_0) && \text{(move 9)}\\
&\vdash (q'_{T,\$}, \Lambda, S_1Z_0) && \text{(move 10)}\\
&\vdash (q, \Lambda, \$S_1Z_0) && \text{(move 11)}\\
&\vdash (q_\$, \Lambda, S_1Z_0) && \text{(move 3)}\\
&\vdash (q, \Lambda, SZ_0) && \text{(move 4)}\\
&\vdash (q_1, \Lambda, Z_0) && \text{(move 13)}\\
&\text{(accept)}
\end{aligned}
$$

EXERCISES

7.1. For the PDA in Example 7.1, trace the sequence of moves made for each of the input strings *bbcbb* and *baca*.

7.2. For the PDA in Example 7.2, draw the computation tree showing all possible sequences of moves for the two input strings *aba* and *aabab*.

7.3. For a string $x \in \{a, b\}^*$ with $|x| = n$, how many possible complete sequences of moves can the PDA in Example 7.2 make, starting with input string x? (By a "complete" sequence of moves, we mean a sequence of moves starting in the initial configuration (q_0, x, Z_0) and terminating in a configuration from which no move is possible.)

7.4. Modify the PDA described in Example 7.2 to accept each of the following subsets of $\{a, b\}^*$.

a. The language of even-length palindromes.

b. The language of odd-length palindromes.

7.5. Give transition tables for PDAs recognizing each of the following languages.

a. The language of all nonpalindromes over $\{a, b\}$.

b. $\{a^n x \mid n \geq 0,\ x \in \{a, b\}^* \text{ and } |x| \leq n\}$.

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

d. $\{x \in \{a, b, c\}^* \mid n_a(x) < n_b(x) \text{ or } n_a(x) < n_c(x)\}$.

7.6. In both cases below, a transition table is given for a PDA with initial state q_0 and accepting state q_2. Describe in each case the language that is accepted.

Move number	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_1, bZ_0)
3	q_1	a	a	$(q_1, a), (q_2, a)$
4	q_1	b	a	(q_1, a)
5	q_1	a	b	(q_1, b)
6	q_1	b	b	$(q_1, b), (q_2, b)$
(all other combinations)				none

Move number	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_0, XZ_0)
2	q_0	b	Z_0	(q_0, XZ_0)
3	q_0	a	X	(q_0, XX)
4	q_0	b	X	(q_0, XX)
5	q_0	c	X	(q_1, X)
6	q_0	c	Z_0	(q_1, Z_0)
7	q_1	a	X	(q_1, Λ)
8	q_1	b	X	(q_1, Λ)
9	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

7.7. Give a transition table for a PDA accepting the language in Example 7.1 and having only two states, the nonaccepting state q_0 and the accepting state q_2. (Use additional stack symbols.)

7.8. Show that every regular language can be accepted by a deterministic PDA M with only two states in which there are no Λ-transitions and no symbols are ever removed from the stack.

7.9. Show that if L is accepted by a PDA in which no symbols are ever removed from the stack, then L is regular.

7.10. Suppose $L \subseteq \Sigma^*$ is accepted by a PDA M, and for some fixed k, and every $x \in \Sigma^*$, no sequence of moves made by M on input x causes the stack to have more than k elements. Show that L is regular.

7.11. Show that if L is accepted by a PDA, then L is accepted by a PDA that never crashes (i.e., for which the stack never empties and no configuration is reached from which there is no move defined).

7.12. Show that if L is accepted by a PDA, then L is accepted by a PDA in which every move either pops something from the stack (i.e., removes a stack symbol without putting anything else on the stack); or pushes a single symbol onto the stack on top of the symbol that was previously on top; or leaves the stack unchanged.

7.13. Give transition tables for deterministic PDAs recognizing each of the following languages.

a. $\{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$

b. $\{x \in \{a, b\}^* \mid n_a(x) \neq n_b(x)\}$

c. $\{x \in \{a, b\}^* \mid n_a(x) < 2n_b(x)\}$

d. $\{a^n b^{n+m} a^m \mid n, m \geq 0\}$

7.14. Suppose M_1 and M_2 are PDAs accepting L_1 and L_2, respectively. Describe a procedure for constructing a PDA accepting each of the following languages. Note that in each case, nondeterminism will be necessary. Be sure to say precisely how the stack of the new machine works; no relationship is assumed between the stack alphabets of M_1 and M_2.

a. $L_1 \cup L_2$

b. $L_1 L_2$

c. L_1^*

7.15. Show that if there are strings x and y in the language L so that x is a prefix of y and $x \neq y$, then no DPDA can accept L by empty stack.

7.16. Show that if there is a DPDA accepting L, and \$ is not one of the symbols in the input alphabet, then there is a DPDA accepting the language $L\{\$\}$ by empty stack.

7.17. Show that none of the following languages can be accepted by a DPDA. (Determine exactly what property of the language *pal* is used in the proof of Theorem 7.1, and show that these languages also have that property.)

a. The set of even-length palindromes over $\{a, b\}$

b. The set of odd-length palindromes over $\{a, b\}$

c. $\{xx^\sim \mid x \in \{0, 1\}^*\}$ (where $x^\sim$ means the string obtained from x by changing 0's to 1's and 1's to 0's)

d. $\{xy \mid x \in \{0, 1\}^*$ and y is either x or $x^\sim\}$

7.18. A *counter automaton* is a PDA with just two stack symbols, A and Z_0, for which the string on the stack is always of the form $A^n Z_0$ for some $n \geq 0$. (In other words, the only possible change in the stack contents is a change in the number of A's on the stack.) For some context-free languages, such as $\{0^i 1^i \mid i \geq 0\}$, the obvious PDA to accept the language is in fact a counter automaton. Construct a counter automaton to accept the given language in each case below.

a. $\{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$

b. $\{x \in \{0, 1\}^* \mid n_0(x) < 2n_1(x)\}$

7.19. Suppose that $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a deterministic PDA accepting a language L. If x is a string in L, then by definition there is a sequence of moves of M with input x in which all the symbols of x are read. It is conceivable, however, that for some strings $y \notin L$, no sequence of moves causes M to read all of y. This could happen in two ways: M could either crash by not being able to move, or it could enter a loop in which there were infinitely many repeated Λ-transitions. Find an example of a DCFL $L \subseteq \{a, b\}^*$, a string $y \notin L$, and a DPDA M accepting L for which M crashes on y by not being able to move. (Say what L is and what y is, and give a transition table for M.) Note that once you have such an M, it can easily be modified so that y causes it to enter an infinite loop of Λ-transitions.

7.20. Give a definition of "balanced string" involving two types of brackets (such as in Example 7.3) corresponding to Definition 6.5.

7.21. In each case below, you are given a CFG and a string x that it generates. For the top-down PDA that is constructed from the grammar as in Definition 7.4, trace a sequence of moves by which x is accepted, showing at each step the state, the unread input, and the stack contents. Show at the same time the corresponding leftmost derivation of x in the grammar. See Example 7.5 for a guide.

a. The grammar has productions

$$S \to S+T \mid T \quad T \to T*F \mid F \quad F \to (S) \mid a$$

and $x = (a + a * a) * a$.

b. The grammar has productions $S \to S + S \mid S * S \mid (S) \mid a$, and $x = (a * a + a)$.

c. The grammar has productions $S \to (S)S \mid \Lambda$, and $x = ()(()())$.

7.22. Let M be the PDA in Example 7.2, except that move number 12 is changed to (q_2, Λ), so that M does in fact accept by empty stack. Let $x = ababa$. Find a sequence of moves of M by which x is accepted, and give the corresponding leftmost derivation in the CFG obtained from M as in Theorem 7.4.

7.23. Under what circumstances is the "nondeterministic" top-down PDA described in Definition 7.4 actually deterministic? (For what kind of language could this happen?)

7.24. In each case below, you are given a CFG and a string x that it generates. For the nondeterministic bottom-up PDA that is constructed from the grammar as in Example 7.6, trace a sequence of moves by which x is accepted, showing at each step the state, the stack contents, and the unread input. Show at the same time the corresponding rightmost derivation of x (in reverse order) in the grammar. See Example 7.6 for a guide.

a. The grammar has productions $S \to S[S] \mid \Lambda$, and $x = [][[]]$.

b. The grammar has productions $S \to [S]S \mid \Lambda$, and $x = [][[]]$.

7.25. If the PDA in Theorem 7.4 is deterministic, what does this tell you about the grammar that is obtained? Can the resulting grammar have this property without the original PDA being deterministic?

7.26. Find the other useless variables in the CFG obtained in Example 7.7.

7.27. In each case, the grammar with the given productions satisfies the LL(1) property. For each one, give a transition table for the deterministic PDA obtained as in Example 7.8.

a. $S \to S_1\$ \quad S_1 \to AS_1 \mid \Lambda \quad A \to aA \mid b$

b. $S \to S_1\$ \quad S_1 \to aA \quad A \to aA \mid bA \mid \Lambda$

c. $S \to S_1\$ \quad S_1 \to aAB \mid bBA \quad A \to bS_1 \mid a \quad B \to aS_1 \mid b$

7.28. In each case, the grammar with the given productions does not satisfy the LL(1) property. Find an equivalent LL(1) grammar by factoring and eliminating left recursion.

a. $S \to S_1\$ \quad S_1 \to aaS_1b \mid ab \mid bb$

b. $S \to S_1\$ \quad S_1 \to S_1A \mid \Lambda \quad A \to Aa \mid b$

c. $S \to S_1\$ \quad S_1 \to S_1T \mid ab \quad T \to aTbb \mid ab$

d. $S \to S_1\$ \quad S_1 \to aAb \mid aAA \mid aB \mid bbA$

$A \to aAb \mid ab \quad B \to bBa \mid ba$

7.29. Show that for the CFG in part (c) of the previous exercise, if the last production were $T \to a$ instead of $T \to ab$, the grammar obtained by factoring and eliminating left recursion would not be LL(1). (Find a string that doesn't work, and identify the point at which looking ahead one symbol in the input isn't enough to decide what move the PDA should make.)

7.30. Consider the CFG with productions

$$S \to S_1\$ \quad S_1 \to S_1 + T \mid T \quad T \to T * F \mid F \quad F \to (S_1) \mid a$$

a. Write the CFG obtained from this one by eliminating left recursion.

b. Give a transition table for a DPDA that acts as a top-down parser for this language.

7.31. Suppose that in a grammar having a variable T, the T-productions are

$$T \to T\alpha_i \quad (1 \le i \le m) \qquad T \to \beta_i \quad (1 \le i \le n)$$

where none of the strings β_i begins with T. Find a set of productions with which these can be replaced, so that the resulting grammar will be equivalent to the original and will have no left recursion involving T.

7.32. Let G be the CFG with productions

$$S \to S_1\$ \quad S_1 \to (S_1 + S_1) \mid (S_1 * S_1) \mid a$$

so that $L(G)$ is the language of all fully parenthesized algebraic expressions involving the operators $+$ and $*$ and the identifier a. Give a transition table for a deterministic bottom-up parser obtained from this grammar as in Example 7.12.

7.33. Let G have productions

$$S \to S_1\$ \quad S_1 \to S_1[S_1] \mid S_1[] \mid [S_1] \mid []$$

and let G_1 have productions

$$S \to S_1\$ \quad S_1 \to [S_1]S_1 \mid [S_1] \mid []S_1 \mid []$$

a. Give a transition table for a deterministic bottom-up parser obtained from G.

b. Show that G_1 is not a weak precedence grammar.

7.34. In the nondeterministic bottom-up parser given for the grammar in Example 7.12, the implicit assumption in the transition table was that the start symbol S did not appear on the right side of any production. Why is there no loss of generality in making this assumption in general?

7.35. In the standard nondeterministic bottom-up parsing PDA for a grammar, obtained as in Example 7.12, consider a configuration in which the right side of a production is currently on top of the stack in reverse, and this string does not appear in the right side of any other production. Why is it always correct to reduce at this point?

7.36. a. Say exactly what the precedence relation is for the grammar in Example 7.12. In other words, for which pairs (X, σ), where X is a stack symbol and σ an input symbol, is it correct to reduce when X is on top of the stack and σ is the next input?

b. Answer the same question for the larger grammar (also a weak precedence grammar) with productions

$$S \to S_1\$ \quad S_1 \to S_1 + T \mid S_1 - T \mid T$$
$$T \to T * F \mid T/F \mid F \quad F \to (S_1) \mid a$$

MORE CHALLENGING PROBLEMS

7.37. Give transition tables for PDAs recognizing each of the following languages.

a. $\{a^i b^j \mid i \le j \le 2i\}$

b. $\{x \in \{a, b\}^* \mid n_a(x) < n_b(x) < 2n_a(x)\}$

7.38. Suppose $L \subseteq \Sigma^*$ is accepted by a PDA M, and for some fixed k, and every $x \in \Sigma^*$, at least one choice of moves allows M to process x completely so that the stack never contains more than k elements. Does it follow that L is regular? Prove your answer.

7.39. Suppose $L \subseteq \Sigma^*$ is accepted by a PDA M, and for some fixed k, and every $x \in L$, at least one choice of moves allows M to accept x in such a way that the stack never contains more than k elements. Does it follow that L is regular? Prove your answer.

7.40. Show that if L is accepted by a DPDA, then there is a DPDA accepting the language $\{x\#y \mid x \in L \text{ and } xy \in L\}$. (The symbol # is assumed not to occur in any of the strings of L.)

7.41. Complete the proof of Theorem 7.3. Give a precise definition of the PDA M_1, and a proof that it accepts the same language as the original PDA M.

7.42. Prove the converse of Theorem 7.3: If there is a PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ accepting L by empty stack (that is, $x \in L$ if and only if $(q_0, x, Z_0) \vdash_M^* (q, \Lambda, \Lambda)$ for some state q), then there is a PDA M_1 accepting L by final state (i.e., the ordinary way).

7.43. Show that in the previous exercise, if M is a deterministic PDA, then M_1 can also be taken to be deterministic.

7.44. Show that if L is accepted by a PDA, then L is accepted by a PDA having at most two states and no Λ-transitions.

7.45. Show that if L is accepted by a PDA, then L is accepted by a PDA in which there are at most two stack symbols in addition to Z_0.

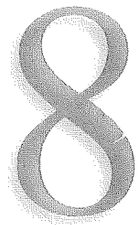
7.46. Show that if M is a DPDA accepting a language $L \subseteq \Sigma^*$, then there is a DPDA M_1 accepting L for which neither of the phenomena in Exercise 7.19 occurs—that is, for every $x \in \Sigma^*$, $(q_0, x, Z_0) \vdash_{M_1}^* (q, \Lambda, \gamma)$ for some state q and some string γ of stack symbols.

7.47. Starting with the top-down nondeterministic PDA constructed as in Definition 7.4, one might try to produce a deterministic parsing algorithm by using a backtracking approach: specifying an order in which to try all the moves possible in a given configuration, and trying sequences of moves in order, backtracking whenever the machine crashes.

a. Describe such a backtracking algorithm in more detail for the grammar in Example 7.8, and trace the algorithm on several strings, including strings derivable from the grammar and strings that are not.

b. What possible problems may arise with such an approach for a general grammar?

CHAPTER



Context-Free and Non-Context-Free Languages

8.1 THE PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

Neither the definition of context-free languages in terms of grammars nor the pushdown-automaton characterization in Chapter 7 makes it immediately obvious that there are formal languages that are not context-free. However, our brief look at natural languages (Example 6.6) has suggested some of the limitations of CFGs. In the first section of this chapter we formulate a principle, similar to the pumping lemma for regular languages (Theorem 5.2a), which will allow us to identify a number of non-context-free languages.

The earlier pumping lemma used the fact that a sufficiently long input string causes a finite automaton to visit some state more than once. Any such string can be written $x = uvw$, where v is a substring that causes the FA to start in a state and return to that state; the result is that all the strings of the form uv^iw are also accepted by the FA. Although we will get the comparable result for CFLs by using grammars instead of automata, the way it arises is similar. Suppose a derivation in a context-free grammar G involves a variable A more than once, in this way:

$$S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vwxyz$$

where $v, w, x, y, z \in \Sigma^*$. Within this derivation, both the strings x and wAy are derived from A. We may write

$$S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vw^2Ay^2z \Rightarrow^* vw^3Ay^3z \Rightarrow^* \cdots$$

and since x can be derived from each of these A's, we may conclude that all the strings $vxz, vwxyz, vw^2xy^2z, \ldots$ are in $L(G)$.