CHAPTER 12

Pushdown Automata

Grammars define context-free languages. We'd also like a computational formalism that is powerful enough to enable us to build an acceptor for every context-free language. In this chapter, we describe such a formalism.

12.1 Definition of a (Nondeterministic) PDA

A pushdown automaton, or PDA, is a finite state machine that has been augmented by a single stack. In a minute, we will present the formal definition of the PDA model that we will use. But, before we do that, one caveat to readers of other books is in order. There are several competing PDA definitions, from which we have chosen one to present here. All are provably equivalent, in the sense that, for all i and j, if there exists a version$_i$ PDA that accepts some language L then there also exists a version$_j$ PDA that accepts L. We'll return to this issue in Section 12.5, where we will mention a few of the other models and sketch an equivalence proof. For now, simply beware of the fact that other definitions are also in widespread use.

We will use the following definition: A ***pushdown automaton*** (or ***PDA***) M is a sextuple $(K, \Sigma, \Gamma, \Delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- Δ is the transition relation. It is a finite subset of:

$$(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*).$$

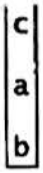
state; input or ε; string of symbols to pop from top of stack; state; string of symbols to push on top of stack

A ***configuration*** of a PDA M is an element of $K \times \Sigma^* \times \Gamma^*$. It captures the three things that can make a difference to M's future behavior:

- its current state,
- the input that is still left to read, and
- the contents of its stack.

The ***initial configuration*** of a PDA M, on input w, is (s, w, ε).

We will use the following notational convention for describing M's stack as a string: The top of the stack is to the left of the string. So:



will be written as cab

If a sequence $c_1c_2 \ldots c_n$ of characters is pushed onto the stack, they will be pushed rightmost first, so if the value of the stack before the push was s, the value after the push will be $c_1c_2 \ldots c_ns$.

Analogously to what we did for FSMs, we define the relation *yields-in-one-step*, written $|\text{-}_M$. *Yields-in-one-step* relates *configuration*$_1$ to *configuration*$_2$ iff M can move from *configuration*$_1$ to *configuration*$_2$ in one step. Let c be any element of $\Sigma \cup \{\varepsilon\}$, let γ_1, γ_2 and γ be any elements of Γ^*, and let w be any element of Σ^*. Then:

$$(q_1, cw, \gamma_1\gamma) |\text{-}_M (q_2, w, \gamma_2\gamma) \text{ iff } ((q_1, c, \gamma_1), (q_2, \gamma_2)) \in \Delta.$$

Note two things about what a transition $((q_1, c, \gamma_1), (q_2, \gamma_2))$ says about how M manipulates its stack:

- M may only take the transition if the string γ_1 matches the current top of the stack. If it does, and the transition is taken, then M pops γ_1 and then pushes γ_2. M cannot "peek" at the top of its stack without popping off the values that it examines.
- If $\gamma_1 = \varepsilon$, then M must match ε against the top of the stack. But ε matches everywhere. So letting γ_1 be ε is equivalent to saying "without bothering to check the current value of the stack." It is not equivalent to saying, "if the stack is empty." In our definition, there is no way to say that directly, although we will see that we can create a way by letting M, before it does anything else, push a special marker onto the stack. Then, whenever that marker is on the top of the stack, the stack is otherwise empty.

The relation *yields*, written $|\text{-}_M^*$, is the reflexive, transitive closure of $|\text{-}_M$. So configuration C_1 yields configuration C_2 iff:

$$C_1 |\text{-}_M^* C_2.$$

A ***computation*** by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε, γ), for some state $q \in K$ and some string γ in Γ^*, and
- $C_0 |\!-_M C_1 |\!-_M C_2 |\!-_M \ldots |\!-_M C_n$.

Note that we have defined the behavior of a PDA M by a transition relation Δ, not a transition function. Thus we allow nondeterminism. If M is in some configuration (q_1, s, γ), it is possible that:

- Δ contains exactly one transition that matches. In that case, M makes the specified move.
- Δ contains more than one transition that matches. In that case, M chooses one of them. Each choice defines one computation that M may perform.
- Δ contains no transition that matches. In that case, the computation that led to that configuration halts.

Let C be a computation of M on input $w \in \Sigma^*$. Then we will say that:

- C is an ***accepting computation*** iff $C = (s, w, \varepsilon) |\!-_M^* (q, \varepsilon, \varepsilon)$, for some $q \in A$. Note the strength of this requirement: A computation accepts only if it runs out of input when it is in an accepting state *and* the stack is empty.
- C is a ***rejecting computation*** iff $C = (s, w, \varepsilon) |\!-_M^* (q, w', \alpha)$, where C is not an accepting computation and where M has no moves that it can make from (q, w', α). A computation can reject only if the criteria for accepting have not been met *and* there are no further moves (including following ε-transitions) that can be taken.

Let w be a string that is an element of Σ^*. Then we will say that:

- M ***accepts*** w iff *at least one* of its computations accepts.
- M ***rejects*** w iff all of its computations reject.

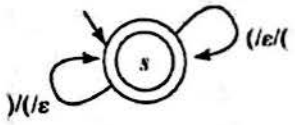
The ***language accepted by*** M, denoted $L(M)$, is the set of all strings accepted by M. Note that it is possible that, on input w, M neither accepts nor rejects.

In all the examples that follow, we will draw a transition $((q_1, c, \gamma_1), (q_2, \gamma_2))$ as an arc from q_1 to q_2, labeled $c/\gamma_1/\gamma_2$. So such a transition should be read to say, "If c matches the input and γ_1 matches the top of the stack, the transition from q_1 to q_2 can be taken, in which case c should be removed from the input, γ_1 should be popped from the stack, and γ_2 should be pushed onto it." If $c = \varepsilon$, then the transition can be taken without consuming any input. If $\gamma_1 = \varepsilon$, the transition can be taken without checking the stack or popping anything. If $\gamma_2 = \varepsilon$, nothing is pushed onto the stack when the transition is taken. As we did with FSMs, we will use a double circle to indicate accepting states.

Even very simple PDAs may be able to accept languages that cannot be accepted by any FSM. The power of such machines comes from the ability of the stack to count.

EXAMPLE 12.1 The Balanced Parentheses Language

Consider again Bal = $\{w \in \{), (\}^* :$ the parentheses are balanced$\}$. The following one-state PDA M accepts Bal. M uses its stack to count the number of left parentheses that have not yet been matched. We show M graphically and then as a sextuple:



$M = (K, \Sigma, \Gamma, \Delta, s, A)$, where:

$K = \{s\}$, (the states)

$\Sigma = \{(,)\}$, (the input alphabet)

$\Gamma = \{(\}$, (the stack alphabet)

$A = \{s\}$, and (the accepting state)

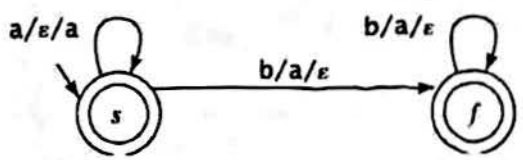
$\Delta = \{((s, (, \varepsilon), (s, ( )),$
$((s,), ( ), (s, \varepsilon))\}$.

If M sees a (, it pushes it onto the stack (regardless of what was already there). If it sees a) and there is a (that can be popped off the stack, M does so. If it sees a) and there is no (to pop, M halts without accepting. If, after consuming its entire input string, M's stack is empty, M accepts. If the stack is not empty, M rejects.

PDAs, like FSMs, can use their states to remember facts about the structure of the string that has been read so far. We see this in the next example.

EXAMPLE 12.2 A^nB^n

Consider again $A^nB^n = \{a^nb^n : n \geq 0\}$. The following PDA M accepts A^nB^n. M uses its states to guarantee that it only accepts strings that belong to a*b*. It uses its stack to count a's so that it can compare them to the b's. We show M graphically:



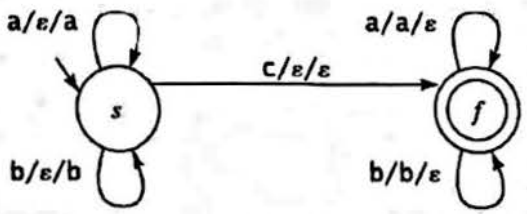
Writing it out, we have $M = (K, \Sigma, \Gamma, \Delta, s, A)$, where:

$$
\begin{aligned}
K &= \{s, f\}, && \text{(the states)}\\
\Sigma &= \{\mathrm{a}, \mathrm{b})\}, && \text{(the input alphabet)}\\
\Gamma &= \{\mathrm{a}\}, && \text{(the stack alphabet)}\\
A &= \{s, f\}, \text{ and} && \text{(the accepting states)}\\
\Delta &= \{((s, \mathrm{a}, \varepsilon), (s, \mathrm{a})),\\
&\quad ((s, \mathrm{b}, \mathrm{a}), (f, \varepsilon)),\\
&\quad ((f, \mathrm{b}, \mathrm{a}), (f, \varepsilon))\}.
\end{aligned}
$$

Remember that M only accepts if, when it has consumed its entire input string, it is in an accepting state *and* its stack is empty. So, for example, M will reject aaa, even though it will be in state s, an accepting state, when it runs out of input. The stack at that point will contain aaa.

EXAMPLE 12.3 WcW^R

Let $\mathrm{WcW}^R = \{wcw^R : w \in \{\mathrm{a}, \mathrm{b}\}^*\}$. The following PDA M accepts WcW^R:



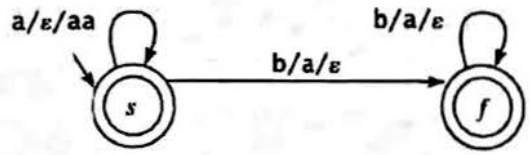
M moves from state s, in which it is recording w, to state f, in which it is checking for w^R, when it sees the character c. Since every string in WcW^R must contain the middle c, state s is not an accepting state.

The definition that we have chosen to use for a PDA is flexible; it allows several symbols to be pushed or popped from the stack in one move. This will turn out to be particularly useful when we attempt to build PDAs that correspond to practical grammars that contain rules like $T \rightarrow T * F$ (the multiplication rule that was part of the arithmetic expression grammar that we defined in Example 11.19). But we illustrate the use of this flexibility here on a simple case.

EXAMPLE 12.4 $\mathrm{A^nB^{2n}}$

Let $\mathrm{A^nB^{2n}} = \{\mathrm{a}^n\mathrm{b}^{2n} : n \geq 0\}$. The following PDA M accepts $\mathrm{A^nB^{2n}}$ by pushing two a's onto the stack for every a in the input string. Then each b pops a single a:

EXAMPLE 12.4 (*Continued*)



12.2 Deterministic and Nondeterministic PDAs

The definition of a PDA that we have presented allows nondeterminism. It sometimes makes sense, however, to restrict our attention to deterministic PDAs. In this section we will define what we mean by a deterministic PDA. We also show some examples of the power of nondeterminism in PDAs. Unfortunately, in contrast to the situation with FSMs, and as we will prove in Theorem 13.13, there exist nondeterministic PDAs for which no equivalent deterministic PDA exists.

12.2.1 Definition of a Deterministic PDA

Define a PDA M to be ***deterministic*** iff there exists no configuration of M in which M has a choice of what to do next. For this to be true, two conditions must hold:

1. Δ_M contains no pairs of transitions that compete with each other.
2. If q is an accepting state of M, then there is no transition $((q, \varepsilon, \varepsilon), (p, a))$ for any p or a. In other words, M is never forced to choose between accepting and continuing. Any transitions out of an accepting state must either consume input (since, if there is remaining input, M does not have the option of accepting) or pop something from the stack (since, if the stack is not empty, M does not have the option of accepting).

So far, all of the PDAs that we have built have been deterministic. So each machine followed only a single computational path.

12.2.2 Exploiting Nondeterminism

But a PDA may be designed to have multiple competing moves from a single configuration. As with FSMs, the easiest way to envision the operation of a nondeterministic PDA M is as a tree, as shown in Figure 12.1. Each node in the tree corresponds to a configuration of M and each path from the root to a leaf node may correspond to one computation that M might perform.

Notice that the state, the stack, and the remaining input can be different along different paths. As a result, it will not be possible to simulate all paths in parallel, the way we did for NDFSMs.

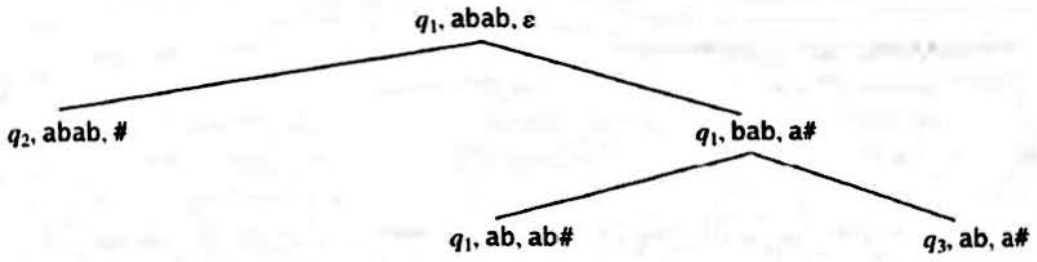
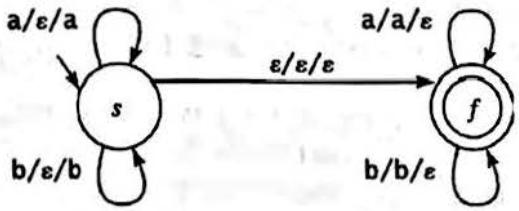


FIGURE 12.1 Viewing nondeterminism as search through a space of computation paths.

EXAMPLE 12.5 Even Length Palindromes

Consider again PalEven = $\{ww^R : w \in \{a, b\}^*\}$, the language of even-length palindromes of a's and b's. The following nondeterministic PDA M accepts PalEven:

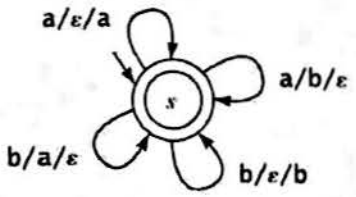


M is nondeterministic because it cannot know when it has reached the middle of its input. Before each character is read, it has two choices: It can guess that it has not yet gotten to the middle. In that case, it stays in state s, where it pushes each symbol it reads. Or it can guess that it has reached the middle. In that case, it takes the ε-transition to state f, where it pops one symbol for each symbol that it reads.

EXAMPLE 12.6 Equal Numbers of a's and b's

Let $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. Now we don't know the order in which the a's and b's will occur. They can be interleaved. So for example, any PDA to accept L must accept aabbba. The only way to count the number of characters that have not yet found their mates is to use the stack. So the stack will sometimes count a's and sometimes count b's. It will count whatever it has seen more of. The following simple PDA accepts L:

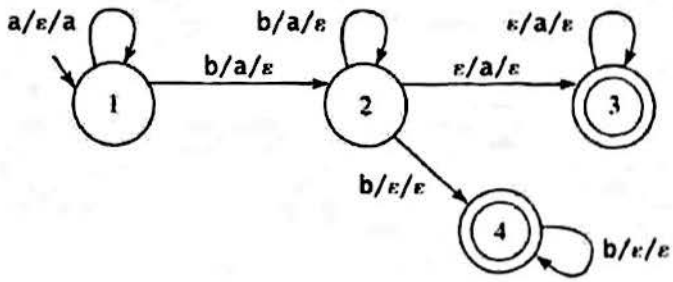
EXAMPLE 12.6 (*Continued*)



This machine is highly nondeterministic. Whenever it sees an a in the input, it can either push it (which is the right thing to do if it should be counting a's) or attempt to pop a b (which is the right thing to do if it should be counting b's). All the computations that make the wrong guess will fail to accept since they will not succeed in clearing the stack. But if $\#_a(w) = \#_b(w)$, there will be one computation that will accept.

EXAMPLE 12.7 The a Region and the b Region are Different

Let $L = \{a^m b^n : m \neq n; m, n > 0\}$. We want to build a PDA M to accept L. It is hard to build a machine that looks for something negative, like $\neq$. But we can break L into two sublanguages: $\{a^m b^n : 0 < m < n\}$ and $\{a^m b^n : 0 < n < m\}$. Either there are more a's or more b's. M must accept any string that is in either of those sublanguages. So M is:



As long as M sees a's, it stays in state 1 and pushes each a onto the stack. When it sees the first b, it goes to state 2. It will accept nothing but b's from that point on. So far, its behavior has been deterministic. But, from state 2, it must make choices. Each time it sees another b and there is an a on the stack, it should consume the b and pop the a and stay in state 2. But, in order to accept, it must eventually either read at least one b that does not have a matching a or pop an a that does not have

a matching b. It should do the former (and go to state 4) if there is a b in the input stream when the stack is empty. But we have no way to specify that a move can be taken only if the stack is empty. It should do the latter (and go to state 3) if there is an a on the stack but the input stream is empty. But we have no way to specify that the input stream is empty.

As a result, in most of its moves in state 2, M will have a choice of three paths to take. All but the correct one will die out without accepting. But a good deal of computational effort will be wasted first.

In the next section, we present techniques for reducing nondeterminism caused by the two problems we've just presented:

- A transition that should be taken only if the stack is empty, and
- A transition that should be taken only if the input stream is empty.

But first we present one additional example of the power of nondeterminism.

EXAMPLE 12.8 $\neg A^nB^nC^n$

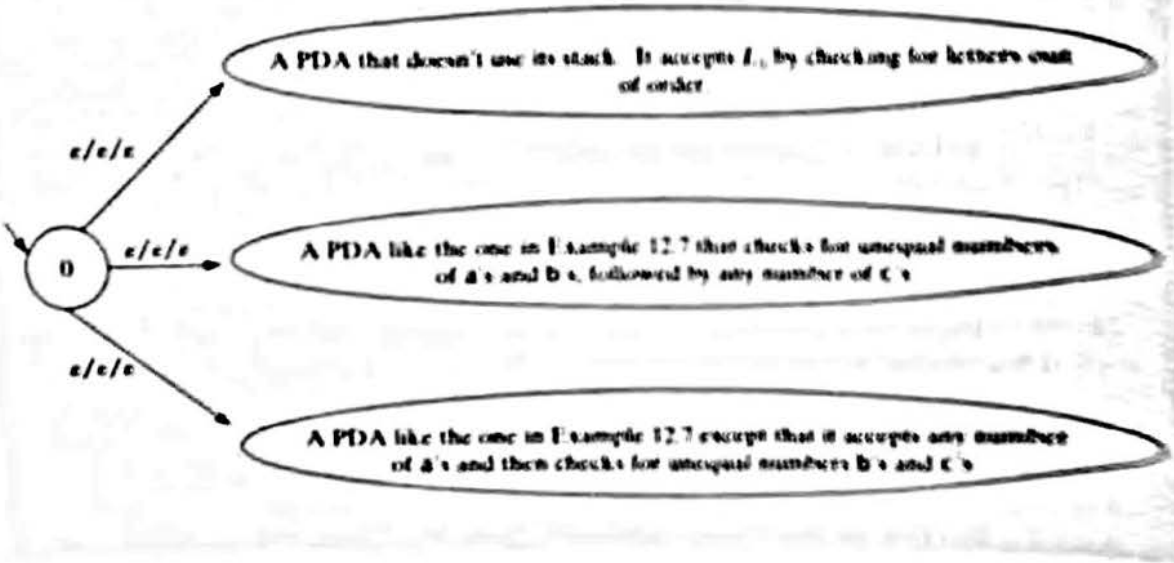
Let's first consider $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$. If we try to think about building a PDA to accept $A^nB^nC^n$, we immediately run into trouble. We can use the stack to count a's and then compare them to the b's. But then the stack will be empty and it won't be possible to compare the c's. We can try to think of something clever to get around this problem, but we will fail. We'll prove in Chapter 13 that no PDA exists to accept this language.

But now let $L = \neg A^nB^nC^n$. There is a PDA that accepts L. $L = L_1 \cup L_2$, where:

- $L_1 = \{w \in \{a, b, c\}^* :$ the letters are out of order$\}$.
- $L_2 = \{a^ib^jc^k : i, j, k \geq 0$ and $(i \neq j$ or $j \neq k)\}$ (in other words, not equal numbers of a's, b's, and c's).

A simple FSM can accept L_1. So we focus on L_2. It turns out to be easier to check for a mismatch in the number of a's, b's, and c's than to check for a match because, to detect a mismatch, it is sufficient to find one thing wrong. It is not necessary to compare everything. So a string w is in L_2 iff *either* (or both) the a's and b's don't match or the b's and c's don't match. We can build PDAs, such as the one we built in Example 12.7, to check each of those conditions. So we can build a straightforward PDA for L. It first guesses which condition to check for. Then submachines do the checking. We sketch a PDA for L here and leave the details as an exercise:

EXAMPLE 12.8 (Continued)



This last example is significant for two reasons:

- It illustrates the power of nondeterminism.
- It proves that the class of languages acceptable by PDAs is not closed under comp plement. We'll have more to say about that in Section 13.4.

An important fact about the context-free languages, in contrast to the regular ones, i that nondeterminism is more than a convenient design tool. In Section 13.5 we will define the ***deterministic context-free languages*** to be those that can be accepted by some deterministic PDA that may exploit an end-of-string marker. Then we will prove that there are context-free languages that are not deterministic in this sense. Thus there exists for the context-free languages, no equivalent of the regular language algorithm *nfa-to-dfa*. There are, however, some techniques that can be used to reduce nondeterminism in many of the kinds of cases that often occur. We'll sketch two of them in the next section.

12.2.3 Techniques for Reducing Nondeterminism ●

In Example 12.7, we saw nondeterminism arising from two very specific circumstances:

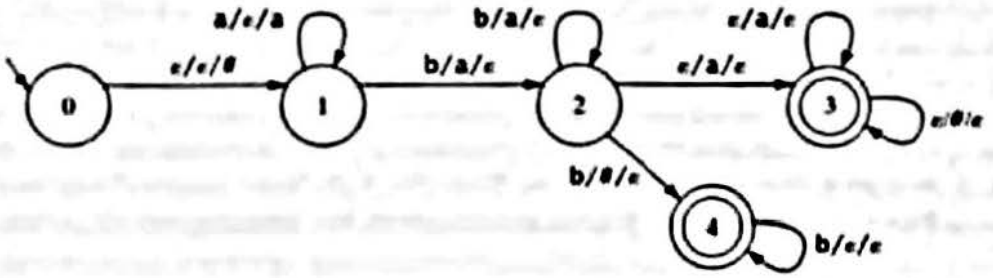
- A transition that should be taken only if the stack is empty competes against one o more moves that require a match of some string on the stack, and
- A transition that should be taken only if the input stream is empty competes against one or more moves that require a match against a specific input characte

Both of these circumstances are common, so we would like to find a way to reduc or eliminate the nondeterminism that they cause.

We first consider the case in which the nondeterminism could be eliminated if it were possible to check for an empty stack. Although our PDA model does not provide a way to do that directly, it is easy to simulate. Any PDA M that would like to be able to check for empty stack can simply, before it does anything else, push a special character onto the stack. The stack is then logically empty iff that special character is at the top of the stack. The only thing we must be careful about is that, before M can accept a string, its stack must be completely empty. So the special character must be popped whenever M reaches an accepting state.

EXAMPLE 12.9 Using a Bottom of Stack Marker

We can use the special, bottom-of-stack marker technique to reduce the nondeterminism in the PDA that we showed in Example 12.7. We'll use # as the marker. When we do that, we get the following PDA M':



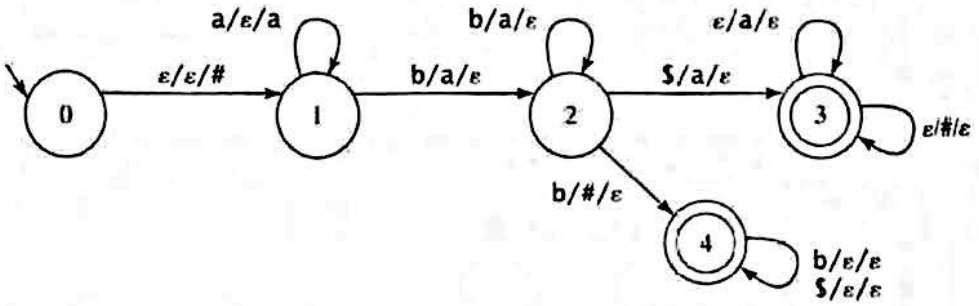
Now the transition back to state 2 no longer competes with the transition to state 4, which can only be taken when the # is the only symbol left on the stack. M' is still nondeterministic though, because the transition back to state 2 competes with the transition to state 3. We still don't have a way to specify that M' should go to state 3 only if it has run out of input.

Next we consider the "out of input" problem. To solve that one, we will make a change to the input language. Instead of building a machine to accept a language L, we'll build one to accept $L\$$, where \$ is a special end-of-string marker. In any practical system, we would probably choose *<newline>* or *<cr>* or *<enter>*, rather than \$, but we'll use \$ here because it is easy to see.

EXAMPLE 12.10 Using an End-of-String Marker

We can use the end-of-string marker technique to eliminate the remaining nondeterminism in the PDAs that we showed in Example 12.7 and Example 12.9. When we do that, we get the following PDA M'':

EXAMPLE 12.10 (*Continued*)



Now the transition back to state 2 no longer competes with the transition to state 3, since the latter can only be taken when the $ is read. Notice that we must be careful to read the $ on all paths, not just the one where we needed it.

Adding an end-of-string marker to the language to be accepted is a powerful tool for reducing nondeterminism. In Section 13.5, we'll define the class of deterministic context-free languages to be exactly the set of context-free languages L such that $L\$$ can be accepted by some deterministic PDA. We'll do that because, for practical reasons, we would like the class of deterministic context-free languages to be as large as possible.

12.3 Equivalence of Context-Free Grammars and PDAs

So far, we have shown PDAs to accept several of the context-free languages for which we wrote grammars in Chapter 11. This is no accident. In this section we'll prove, as usual by construction, that context-free grammars and pushdown automata describe exactly the same class of languages.

12.3.1 Building a PDA from a Grammar

THEOREM 12.1 For Every CFG There Exists an Equivalent PDA

Theorem: Given a context-free grammar $G = (V, \Sigma, R, S)$, there exists a PDA M such that $L(M) = L(G)$.

Proof: The proof is by construction. There are two equally straightforward ways to do this construction, so we will describe both of them. Either of them can be converted to a practical parser (a recognizer that returns a parse tree if it accepts) by

12.3.3 The Equivalence of Context-free Grammars and PDAs

THEOREM 12.3 PDAs and CFGs Describe the Same Class of Languages

Theorem: A language is context-free iff it is accepted by some PDA.

Proof: Theorem 12.1 proves the only if part. Theorem 12.2 proves the if part.

12.4 Nondeterminism and Halting

Recall that a computation C of a PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$ on a string w is an accepting computation iff:

$$C = (s, w, \varepsilon) |\text{-}_M^* (q, \varepsilon, \varepsilon), \text{ for some } q \in A.$$

We'll say that a computation C of M ***halts*** iff at least one of the following conditions holds:

- C is an accepting computation, or
- C ends in a configuration from which there is no transition in Δ that can be taken.

We'll say that M ***halts*** on w iff every computation of M on w halts. If M halts on w and does not accept, then we say that M rejects w.

For every context-free language L, we've proven that there exists a PDA M such that $L(M) = L$. Suppose that we would like to be able to:

- Examine a string and decide whether or not it is in L.
- Examine a string that is in L and create a parse tree for it.
- Examine a string that is in L and create a parse tree for it in time that is linear in the length of the string.
- Examine a string and decide whether or not it is in the complement of L.

Do PDAs provide the tools we need to do those things? When we were at a similar point in our discussion of regular languages, the answer to that question was yes. For every regular language L, there exists a minimal deterministic FSM that accepts it. That minimal DFSM halts on all inputs, accepts all strings that are in L, and rejects all strings that are not in L.

Unfortunately, the facts about context-free languages and PDAs are different from the facts about regular languages and FSMs. Now we must face the following:

1. There are context-free languages for which no deterministic PDA exists. We'll prove this as Theorem 13.13.
2. It is possible that a PDA may
   - not halt, or
   - not ever finish reading its input.

So, let M be a PDA that accepts some language L. Then, on input w, if $w \in L$ then M will halt and accept. But if $w \notin L$, while M will not accept w, it is possible that it will not reject it either. To see how this could happen, let $\Sigma = \{a\}$ and consider the PDA M, shown in Figure 12.4. $L(M) = \{a\}$. The computation $(1, a, \varepsilon) \vdash (2, a, a) \vdash (3, \varepsilon, \varepsilon)$ will cause M to accept a. But consider any other input except a. Observe that:

- M will never halt. There is no accepting configuration, but there is always at least one computational path that has not yet halted. For example, on input aa, one such path is:
  (1, aa, ε) |- (2, aa, a) |- (1, aa, aa) |- (2, aa, aaa) |-
  (1, aa, aaaa) |- (2, aa, aaaaa) |-...
- M will never finish reading its input unless its input is ε. On input aa, for example, there is no computation that will read the second a.

3. There exists no algorithm to minimize a PDA. In fact, it is undecidable whether a PDA is already minimal.

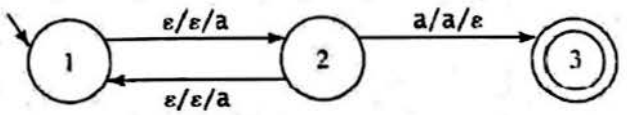


FIGURE 12.4 A PDA that may neither accept nor reject.

Problem 2 is especially critical. This same problem also arose with NDFSMs. But there we had a choice of two solutions:

- Use *ndfsmtodfsm* to convert the NDFSM to an equivalent deterministic one. A DFSM halts on input w in $|w|$ steps.
- Simulate the NDFSM using *ndfsmsimulate*, which ran all computational paths in parallel and handled ε-transitions in a way that guaranteed that the simulation of an NDFSM M on input w halted in $|w|$ steps.

Neither of those approaches works for PDAs. There may not be an equivalent deterministic PDA. And it is not possible to simulate all paths in parallel on a single PDA because each path would need its own stack. So what can we do? Solutions to these problems fall into two classes:

- Formal ones that do not restrict the class of languages that are being considered. Unfortunately, these approaches generally do restrict the *form* of the grammars and PDAs that can be used. For example, they may require that grammars be in Chomsky or Greibach normal form. As a result, parse trees may not make much sense. We'll see some of these techniques in Chapter 14.
- Practical ones that work only on a subclass of the context-free languages. But the subset is large enough to be useful and the techniques can use grammars in their natural forms. We'll see some of these techniques in Chapters 13 and 15.

12.5 Alternative Equivalent Definitions of a PDA ●

We could have defined a PDA somewhat differently. We list here a few reasonable alternative definitions. In all of them a PDA M is a sextuple $(K, \Sigma, \Gamma, \Delta, s, A)$:

- We allow M to pop and to push any string in Γ^*. In some definitions, M may pop only a single symbol but it may push any number of them. In some definitions, M may pop and push only a single symbol.
- In our definition, M accepts its input w only if, when it finishes reading w, it is in an accepting state and its stack is empty. There are two alternatives to this:
  - Accept if, when the input has been consumed, M lands in an accepting state, regardless of the contents of the stack.
  - Accept if, when the input has been consumed, the stack is empty, regardless of the state M is in.

All of these definitions are equivalent in the sense that, if some language L is accepted by a PDA using one definition, it can be accepted by some PDA using each of the other definitions.

We can prove this claim for any pair of definitions by construction. To do so, we show an algorithm that transforms a PDA of one sort into an equivalent PDA of the other sort.

EXAMPLE 12.14 Accepting by Final State Alone

Define a PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$ in exactly the way we have except that it will accept iff it lands in an accepting state, regardless of the contents of the stack. In other words, if $(s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \gamma)$ and $q \in A$, then M accepts.

To show that this model is equivalent to ours, we must show two things: For each of our machines, there exists an equivalent one of these, and, for each of these, there exists an equivalent one of ours. We'll do the first part to show how such a construction can be done. We leave the second as an exercise.

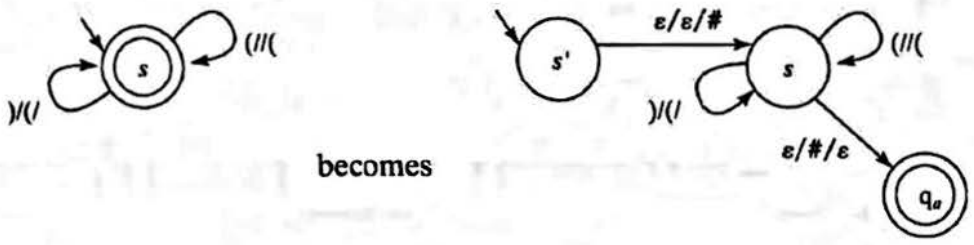
Given a PDA M that accepts by accepting state and empty stack, construct a new PDA M' that accepts by accepting state alone, where $L(M') = L(M)$. M' will have a single accepting state q_a. The only way for M' to get to q_a will be to land in an accepting state of M when the stack is logically empty. But there is no way to check that the stack is empty. So M' will begin by pushing a bottom-of-stack marker #, onto the stack. Whenever # is the top symbol on the stack, the stack is logically empty.

So the construction proceeds as follows:

1. Initially, let $M' = M$.
2. Create a new start state s'. Add the transition $((s', \varepsilon, \varepsilon), (s, \#))$.
3. Create a new accepting state q_a.
4. For each accepting state a in M do:
   Add the transition $((a, \varepsilon, \#), (q_a, \varepsilon))$.
5. Make q_a the only accepting state in M'.

It is easy to see that M' lands in its only accepting state (q_a) iff M lands in some accepting state with an empty stack. Thus M' and M accept the same strings.

As an example, we apply this algorithm to the PDA we built for the balanced parentheses language Bal:



Notice, by the way, that while M is deterministic, M' is not.

12.6 Alternatives that are Not Equivalent to the PDA ●

We defined a PDA to be a finite state machine to which we add a single stack. We mention here two variants of that definition, each of which turns out to define a more powerful class of machine. In both cases, we'll still start with an FSM.

For the first variation, we add a first-in, first-out (FIFO) queue in place of the stack. Such machines are called tag systems or Post machines. As we'll see in Section 18.2.3, tag systems are equivalent to Turing machines in computational power.

For the second variation, we add two stacks instead of one. Again, the resulting machines are equivalent in computational power to Turing machines, as we'll see in Section 17.5.2.

Exercises

1. Build a PDA to accept each of the following languages L:
   a. BalDelim = $\{w$: where w is a string of delimiters: (,), [,], {, }, that are properly balanced$\}$.
   b. $\{a^i b^j : 2i = 3j + 1\}$.
   c. $\{w \in \{a, b\}^* : \#_a(w) = 2 \cdot \#_b(w)\}$.
   d. $\{a^n b^m : m \leq n \leq 2m\}$.
   e. $\{w \in \{a, b\}^* : w = w^R\}$.
   f. $\{a^i b^j c^k : i, j, k \geq 0$ and $(i \neq j$ or $j \neq k)\}$.
   g. $\{w \in \{a, b\}^*$: every prefix of w has at least as many a's as b's$\}$.
   h. $\{a^n b^m a^n : n, m \geq 0$ and m is even$\}$.
   i. $\{xc^n : x \in \{a, b\}^*, \#_a(x) = n$ or $\#_b(x) = n\}$.
   j. $\{a^n b^m : m \geq n, m\text{-}n$ is even$\}$.
   k. $\{a^m b^n c^p d^q : m, n, p, q \geq 0$ and $m + n = p + q\}$.

l. $\{b_i \# b_{i+1}{}^R : b_i$ is the binary representation of some integer i, i ≥ 0, without leading zeros$\}$. (For example 101#011 $\in L$.)

m. $\{x^R \# y : x, y \in \{0,1\}^*$ and x is a substring of $y\}$.

n. L_1^*, where $L_1 = \{xx^R : x \in \{a,b\}^*\}$.

2. Complete the PDA that we sketched, in Example 12.8, for $\neg A^n B^n C^n$, where $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$.

3. Let $L = \{ba^{m_1} ba^{m_2} ba^{m_3} \ldots ba^{m_n} : n \geq 2, m_1, m_2, \ldots, m_n \geq 0$, and $m_i \neq m_j$ for some $i, j\}$.

   a. Show a PDA that accepts L.

   b. Show a context-free grammar that generates L.

   c. Prove that L is not regular.

4. Consider the language $L = L_1 \cap L_2$, where $L_1 = \{ww^R : w \in \{a, b\}^*\}$ and $L_2 = \{a^n b^* a^n : n \geq 0\}$.

   a. List the first four strings in the lexicographic enumeration of L.

   b. Write a context-free grammar to generate L.

   c. Show a natural PDA for L. (In other words, don't just build it from the grammar using one of the two-state constructions presented in this chapter.)

   d. Prove that L is not regular.

5. Build a deterministic PDA to accept each of the following languages:

   a. $L\$$, where $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$.

   b. $L\$$ where $L = \{a^n b^+ a^m : n \geq 0$ and $\exists k \geq 0\ (m = 2k + n)\}$.

6. Complete the proof that we started in Example 12.14. Specifically, show that if M is a PDA that accepts by accepting state alone, then there exists a PDA M' that accepts by accepting state and empty stack (our definition) where $L(M') = L(M)$.