Chapter 6

Pushdown Automata

The context-free languages have a type of automaton that defines them. This automaton, called a "pushdown automaton," is an extension of the nondeterministic finite automaton with ϵ-transitions, which is one of the ways to define the regular languages. The pushdown automaton is essentially an ϵ-NFA with the addition of a stack. The stack can be read, pushed, and popped only at the top, just like the "stack" data structure.

In this chapter, we define two different versions of the pushdown automaton: one that accepts by entering an accepting state, like finite automata do, and another version that accepts by emptying its stack, regardless of the state it is in. We show that these two variations accept exactly the context-free languages; that is, grammars can be converted to equivalent pushdown automata, and vice-versa. We also consider briefly the subclass of pushdown automata that is deterministic. These accept all the regular languages, but only a proper subset of the CFL's. Since they resemble closely the mechanics of the parser in a typical compiler, it is important to observe what language constructs can and cannot be recognized by deterministic pushdown automata.

6.1 Definition of the Pushdown Automaton

In this section we introduce the pushdown automaton, first informally, then as a formal construct.

6.1.1 Informal Introduction

The pushdown automaton is in essence a nondeterministic finite automaton with ϵ-transitions permitted and one additional capability: a stack on which it can store a string of "stack symbols." The presence of a stack means that, unlike the finite automaton, the pushdown automaton can "remember" an infinite amount of information. However, unlike a general-purpose computer, which also has the ability to remember arbitrarily large amounts of information, the pushdown automaton can only access the information on its stack in a last-in-first-out way.

As a result, there are languages that could be recognized by some computer program, but are not recognizable by any pushdown automaton. In fact, pushdown automata recognize all and only the context-free languages. While there are many languages that *are* context-free, including some we have seen that are not regular languages, there are also some simple-to-describe languages that are not context-free, as we shall see in Section 7.2. An example of a non-context-free language is $\{0^n 1^n 2^n \mid n \geq 1\}$, the set of strings consisting of equal groups of 0's, 1's, and 2's.

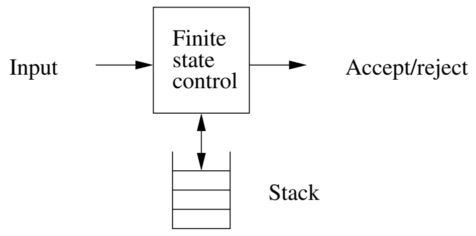

Figure 6.1: A pushdown automaton is essentially a finite automaton with a stack data structure

We can view the pushdown automaton informally as the device suggested in Fig. 6.1. A "finite-state control" reads inputs, one symbol at a time. The pushdown automaton is allowed to observe the symbol at the top of the stack and to base its transition on its current state, the input symbol, and the symbol at the top of stack. Alternatively, it may make a "spontaneous" transition, using ϵ as its input instead of an input symbol. In one transition, the pushdown automaton:

1. Consumes from the input the symbol that it uses in the transition. If ϵ is used for the input, then no input symbol is consumed.

2. Goes to a new state, which may or may not be the same as the previous state.

3. Replaces the symbol at the top of the stack by any string. The string could be ϵ, which corresponds to a pop of the stack. It could be the same symbol that appeared at the top of the stack previously; i.e., no change to the stack is made. It could also replace the top stack symbol by one other symbol, which in effect changes the top of the stack but does not push or pop it. Finally, the top stack symbol could be replaced by two or more symbols, which has the effect of (possibly) changing the top stack symbol, and then pushing one or more new symbols onto the stack.

Example 6.1: Let us consider the language

$$L_{wwr} = \{ww^R \mid w \text{ is in } (\mathbf{0}+\mathbf{1})^*\}$$

This language, often referred to as "w-w-reversed," is the even-length palindromes over alphabet $\{0, 1\}$. It is a CFL, generated by the grammar of Fig. 5.1, with the productions $P \to 0$ and $P \to 1$ omitted.

We can design an informal pushdown automaton accepting L_{wwr}, as follows.[1]

1. Start in a state q_0 that represents a "guess" that we have not yet seen the middle; i.e., we have not seen the end of the string w that is to be followed by its own reverse. While in state q_0, we read symbols and store them on the stack, by pushing a copy of each input symbol onto the stack, in turn.

2. At any time, we may guess that we have seen the middle, i.e., the end of w. At this time, w will be on the stack, with the right end of w at the top and the left end at the bottom. We signify this choice by spontaneously going to state q_1. Since the automaton is nondeterministic, we actually make both guesses: we guess we have seen the end of w, but we also stay in state q_0 and continue to read inputs and store them on the stack.

3. Once in state q_1, we compare input symbols with the symbol at the top of the stack. If they match, we consume the input symbol, pop the stack, and proceed. If they do not match, we have guessed wrong; our guessed w was not followed by w^R. This branch dies, although other branches of the nondeterministic automaton may survive and eventually lead to acceptance.

4. If we empty the stack, then we have indeed seen some input w followed by w^R. We accept the input that was read up to this point.

□

6.1.2 The Formal Definition of Pushdown Automata

Our formal notation for a *pushdown automaton* (PDA) involves seven components. We write the specification of a PDA P as follows:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

The components have the following meanings:

Q: A finite set of *states*, like the states of a finite automaton.

Σ: A finite set of *input symbols*, also analogous to the corresponding component of a finite automaton.

[1] We could also design a pushdown automaton for L_{pal}, which is the language whose grammar appeared in Fig. 5.1. However, L_{wwr} is slightly simpler and will allow us to focus on the important ideas regarding pushdown automata.

No "Mixing and Matching"

There may be several pairs that are options for a PDA in some situation. For instance, suppose $\delta(q, a, X) = \{(p, YZ),\ (r, \epsilon)\}$. When making a move of the PDA, we have to choose one pair in its entirety; we cannot pick a state from one and a stack-replacement string from another. Thus, in state q, with X on the top of the stack, reading input a, we could go to state p and replace X by YZ, or we could go to state r and pop X. However, we cannot go to state p and pop X, and we cannot go to state r and replace X by YZ.

Γ: A finite *stack alphabet*. This component, which has no finite-automaton analog, is the set of symbols that we are allowed to push onto the stack.

δ: The *transition function*. As for a finite automaton, δ governs the behavior of the automaton. Formally, δ takes as argument a triple $\delta(q, a, X)$, where:

1. q is a state in Q.
2. a is either an input symbol in Σ or $a = \epsilon$, the empty string, which is assumed not to be an input symbol.
3. X is a stack symbol, that is, a member of Γ.

The output of δ is a finite set of pairs (p, γ), where p is the new state, and γ is the string of stack symbols that replaces X at the top of the stack. For instance, if $\gamma = \epsilon$, then the stack is popped, if $\gamma = X$, then the stack is unchanged, and if $\gamma = YZ$, then X is replaced by Z, and Y is pushed onto the stack.

q_0: The *start state*. The PDA is in this state before making any transitions.

Z_0: The *start symbol*. Initially, the PDA's stack consists of one instance of this symbol, and nothing else.

F: The set of *accepting states*, or *final states*.

Example 6.2: Let us design a PDA P to accept the language L_{wwr} of Example 6.1. First, there are a few details not present in that example that we need to understand in order to manage the stack properly. We shall use a stack symbol Z_0 to mark the bottom of the stack. We need to have this symbol present so that, after we pop w off the stack and realize that we have seen ww^R on the input, we still have something on the stack to permit us to make a transition to the accepting state, q_2. Thus, our PDA for L_{wwr} can be described as

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

where δ is defined by the following rules:

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially, when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input, and push it onto the stack, leaving Z_0 below to mark the bottom.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four, similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of the stack and leaving the previous top stack symbol alone.

3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$, and $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ϵ input), leaving intact whatever symbol is at the top of the stack.

4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$. Now, in state q_1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.

5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$. Finally, if we expose the bottom-of-stack marker Z_0 and we are in state q_1, then we have found an input of the form ww^R. We go to state q_2 and accept.

□

6.1.3 A Graphical Notation for PDA's

The list of δ facts, as in Example 6.2, is not too easy to follow. Sometimes, a diagram, generalizing the transition diagram of a finite automaton, will make aspects of the behavior of a given PDA clearer. We shall therefore introduce and subsequently use a *transition diagram* for PDA's in which:

a) The nodes correspond to the states of the PDA.

b) An arrow labeled *Start* indicates the start state, and doubly circled states are accepting, as for finite automata.

c) The arcs correspond to transitions of the PDA in the following sense. An arc labeled $a, X/\alpha$ from state q to state p means that $\delta(q, a, X)$ contains the pair (p, α), perhaps among other pairs. That is, the arc label tells what input is used, and also gives the old and new tops of the stack.

The only thing that the diagram does not tell us is which stack symbol is the start symbol. Conventionally, it is Z_0, unless we indicate otherwise.

Example 6.3: The PDA of Example 6.2 is represented by the diagram shown in Fig. 6.2. □

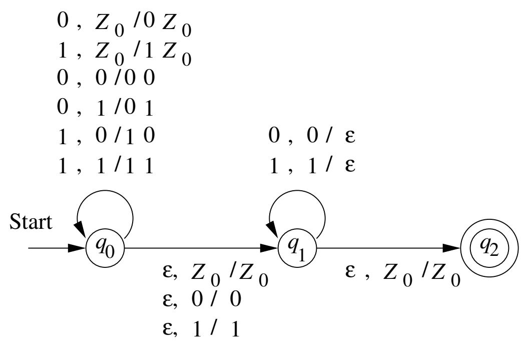

Figure 6.2: Representing a PDA as a generalized transition diagram

6.1.4 Instantaneous Descriptions of a PDA

To this point, we have only an informal notion of how a PDA "computes." Intuitively, the PDA goes from configuration to configuration, in response to input symbols (or sometimes ϵ), but unlike the finite automaton, where the state is the only thing that we need to know about the automaton, the PDA's configuration involves both the state and the contents of the stack. Being arbitrarily large, the stack is often the more important part of the total configuration of the PDA at any time. It is also useful to represent as part of the configuration the portion of the input that remains.

Thus, we shall represent the configuration of a PDA by a triple (q, w, γ), where

1. q is the state,

2. w is the remaining input, and

3. γ is the stack contents.

Conventionally, we show the top of the stack at the left end of γ and the bottom at the right end. Such a triple is called an *instantaneous description*, or ID, of the pushdown automaton.

For finite automata, the $\hat{\delta}$ notation was sufficient to represent sequences of instantaneous descriptions through which a finite automaton moved, since the ID for a finite automaton is just its state. However, for PDA's we need a notation that describes changes in the state, the input, and stack. Thus, we adopt the "turnstile" notation for connecting pairs of ID's that represent one or many moves of a PDA.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Define $\underset{P}{\vdash}$, or just $\vdash$ when P is understood, as follows. Suppose $\delta(q, a, X)$ contains (p, α). Then for all strings w in Σ^* and β in Γ^*:

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

This move reflects the idea that, by consuming a (which may be ϵ) from the input and replacing X on top of the stack by α, we can go from state q to state p. Note that what remains on the input, w, and what is below the top of the stack, β, do not influence the action of the PDA; they are merely carried along, perhaps to influence events later.

We also use the symbol $\overset{*}{\underset{P}{\vdash}}$, or $\overset{*}{\vdash}$ when the PDA P is understood, to represent zero or more moves of the PDA. That is:

BASIS: $I \overset{*}{\vdash} I$ for any ID I.

INDUCTION: $I \overset{*}{\vdash} J$ if there exists some ID K such that $I \vdash K$ and $K \overset{*}{\vdash} J$.

That is, $I \overset{*}{\vdash} J$ if there is a sequence of ID's $K_1, K_2, \ldots, K_n$ such that $I = K_1$, $J = K_n$, and for all $i = 1, 2, \ldots, n-1$, we have $K_i \vdash K_{i+1}$.

Example 6.4: Let us consider the action of the PDA of Example 6.2 on the input 1111. Since q_0 is the start state and Z_0 is the start symbol, the initial ID is $(q_0, 1111, Z_0)$. On this input, the PDA has an opportunity to guess wrongly several times. The entire sequence of ID's that the PDA can reach from the initial ID $(q_0, 1111, Z_0)$ is shown in Fig. 6.3. Arrows represent the $\vdash$ relation.

$$\begin{tikzcd}
(q_0, 1111, Z_0) \arrow[d] \arrow[dr] & & \\
(q_0, 111, 1Z_0) \arrow[d] \arrow[dr] & (q_1, 1111, Z_0) \arrow[r] & (q_2, 1111, Z_0) \\
(q_0, 11, 11Z_0) \arrow[d] \arrow[dr] & (q_1, 111, 1Z_0) \arrow[r] & (q_1, 11, Z_0) \arrow[d] \\
(q_0, 1, 111Z_0) \arrow[d] \arrow[dr] & (q_1, 11, 11Z_0) \arrow[dr] & (q_2, 11, Z_0) \\
(q_0, \varepsilon, 1111Z_0) \arrow[d] & (q_1, 1, 111Z_0) \arrow[d] & (q_1, 1, 1Z_0) \arrow[d] \\
(q_1, \varepsilon, 1111Z_0) & (q_1, \varepsilon, 11Z_0) & (q_1, \varepsilon, Z_0) \arrow[d] \\
 & & (q_2, \varepsilon, Z_0)
\end{tikzcd}$$

Figure 6.3: ID's of the PDA of Example 6.2 on input 1111

Notational Conventions for PDA's

We shall continue using conventions regarding the use of symbols that we introduced for finite automata and grammars. In carrying over the notation, it is useful to realize that the stack symbols play a role analogous to the union of the terminals and variables in a CFG. Thus:

1. Symbols of the input alphabet will be represented by lower-case letters near the beginning of the alphabet, e.g., a, b.
2. States will be represented by q and p, typically, or other letters that are nearby in alphabetical order.
3. Strings of input symbols will be represented by lower-case letters near the end of the alphabet, e.g., w or z.
4. Stack symbols will be represented by capital letters near the end of the alphabet, e.g., X or Y.
5. Strings of stack symbols will be represented by Greek letters, e.g., α or γ.

From the initial ID, there are two choices of move. The first guesses that the middle has not been seen and leads to ID $(q_0, 111, 1Z_0)$. In effect, a 1 has been removed from the input and pushed onto the stack.

The second choice from the initial ID guesses that the middle has been reached. Without consuming input, the PDA goes to state q_1, leading to the ID $(q_1, 1111, Z_0)$. Since the PDA may accept if it is in state q_1 and sees Z_0 on top of its stack, the PDA goes from there to ID $(q_2, 1111, Z_0)$. That ID is not exactly an accepting ID, since the input has not been completely consumed. Had the input been ϵ rather than 1111, the same sequence of moves would have led to ID (q_2, ϵ, Z_0), which would show that ϵ is accepted.

The PDA may also guess that it has seen the middle after reading one 1, that is, when it is in the ID $(q_0, 111, 1Z_0)$. That guess also leads to failure, since the entire input cannot be consumed. The correct guess, that the middle is reached after reading two 1's, gives us the sequence of ID's $(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$. $\Box$

There are three important principles about ID's and their transitions that we shall need in order to reason about PDA's:

1. If a sequence of ID's (*computation*) is legal for a PDA P, then the computation formed by adding the same additional input string to the end of

the input (second component) in each ID is also legal.

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

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

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

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

$$(q, xw, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta\gamma)$$

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

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

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

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

$$(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$$

then it is also true that $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$. $\square$

6.1.5 Exercises for Section 6.1

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

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

ID's for Finite Automata?

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

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

2. $\delta(q, 0, X) = \{(q, XX)\}$.

3. $\delta(q, 1, X) = \{(q, X)\}$.

4. $\delta(q, \epsilon, X) = \{(p, \epsilon)\}$.

5. $\delta(p, \epsilon, X) = \{(p, \epsilon)\}$.

6. $\delta(p, 1, X) = \{(p, XX)\}$.

7. $\delta(p, 1, Z_0) = \{(p, \epsilon)\}$.

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

* a) 01.

b) 0011.

c) 010.

6.2 The Languages of a PDA

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

These two methods are equivalent, in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack. However, for a given PDA P, the languages that P accepts by final state and by empty stack are usually different. We shall show in this section how to convert a PDA accepting L by final state into another PDA that accepts L by empty stack, and vice-versa.

6.2.1 Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α. That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The contents of the stack at that time is irrelevant.

Example 6.7: We have claimed that the PDA of Example 6.2 accepts the language L_{wwr}, the language of strings in $\{0,1\}^*$ that have the form ww^R. Let us see why that statement is true. The proof is an if-and-only-if statement: the PDA P of Example 6.2 accepts string x by final state if and only if x is of the form ww^R.

(If) This part is easy; we have only to show the accepting computation of P. If $x = ww^R$, then observe that

$$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$$

That is, one option the PDA has is to read w from its input and store it on its stack, in reverse. Next, it goes spontaneously to state q_1 and matches w^R on the input with the same string on its stack, and finally goes spontaneously to state q_2.

(Only-if) This part is harder. First, observe that the only way to enter accepting state q_2 is to be in state q_1 and have Z_0 at the top of the stack. Also, any accepting computation of P will start in state q_0, make one transition to q_1, and never return to q_0. Thus, it is sufficient to find the conditions on x such that $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0)$; these will be exactly the strings x that P accepts by final state. We shall show by induction on $|x|$ the slightly more general statement:

- If $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \alpha)$, then x is of the form ww^R.

BASIS: If $x = \epsilon$, then x is of the form ww^R (with $w = \epsilon$). Thus, the conclusion is true, so the statement is true. Note we do not have to argue that the hypothesis $(q_0, \epsilon, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \alpha)$ is true, although it is.

INDUCTION: Suppose $x = a_1 a_2 \cdots a_n$ for some $n > 0$. There are two moves that P can make from ID (q_0, x, α):

1. $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$. Now P can only pop the stack when it is in state q_1. P must pop the stack with every input symbol it reads, and $|x| > 0$. Thus, if $(q_1, x, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \beta)$, then β will be shorter than α and cannot be equal to α.

2. $(q_0, a_1a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1\alpha)$. Now the only way a sequence of moves can end in (q_1, ϵ, α) is if the last move is a pop:

$$(q_1, a_n, a_1\alpha) \vdash (q_1, \epsilon, \alpha)$$

In that case, it must be that $a_1 = a_n$. We also know that

$$(q_0, a_2 \cdots a_n, a_1\alpha) \overset{*}{\vdash} (q_1, a_n, a_1\alpha)$$

By Theorem 6.6, we can remove the symbol a_n from the end of the input, since it is not used. Thus,

$$(q_0, a_2 \cdots a_{n-1}, a_1\alpha) \overset{*}{\vdash} (q_1, \epsilon, a_1\alpha)$$

Since the input for this sequence is shorter than n, we may apply the inductive hypothesis and conclude that $a_2 \cdots a_{n-1}$ is of the form yy^R for some y. Since $x = a_1yy^Ra_n$, and we know $a_1 = a_n$, we conclude that x is of the form ww^R; specifically $w = a_1y$.

The above is the heart of the proof that the only way to accept x is for x to be equal to ww^R for some w. Thus, we have the "only-if" part of the proof, which, with the "if" part proved earlier, tells us that P accepts exactly those strings in L_{wwr}. □

6.2.2 Acceptance by Empty Stack

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, we also define

$$N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$$

for any state q. That is, $N(P)$ is the set of inputs w that P can consume and at the same time empty its stack.[2]

Example 6.8: The PDA P of Example 6.2 never empties its stack, so $N(P) = \emptyset$. However, a small modification will allow P to accept L_{wwr} by empty stack as well as by final state. Instead of the transition $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$, use $\delta(q_1, \epsilon, Z_0) = \{(q_2, \epsilon)\}$. Now, P pops the last symbol off its stack as it accepts, and $L(P) = N(P) = L_{wwr}$. □

Since the set of accepting states is irrelevant, we shall sometimes leave off the last (seventh) component from the specification of a PDA P, if all we care about is the language that P accepts by empty stack. Thus, we would write P as a six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

[2]The N in $N(P)$ stands for "null stack," a synonym for "empty stack."

6.2.3 From Empty Stack to Final State

We shall show that the classes of languages that are $L(P)$ for some PDA P is the same as the class of languages that are $N(P)$ for some PDA P. This class is also exactly the context-free languages, as we shall see in Section 6.3. Our first construction shows how to take a PDA P_N that accepts a language L by empty stack and construct a PDA P_F that accepts L by final state.

Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. 6.4. We use a new symbol X_0, which must not be a symbol of Γ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack. That is, if P_F sees X_0 on top of its stack, then it knows that P_N would empty its stack on the same input.

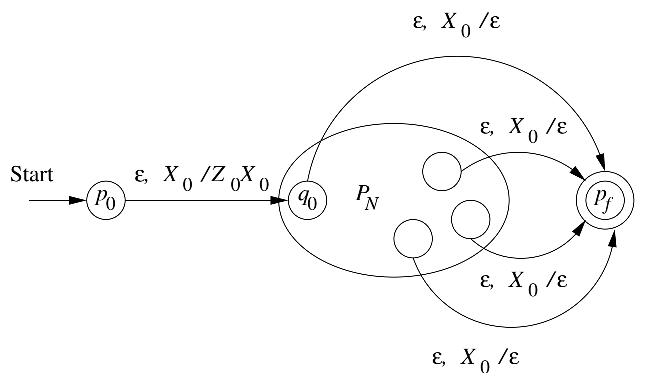

Figure 6.4: P_F simulates P_N and accepts if P_N empties its stack

We also need a new start state, p_0, whose sole function is to push Z_0, the start symbol of P_N, onto the top of the stack and enter state q_0, the start state of P_N. Then, P_F simulates P_N, until the stack of P_N is empty, which P_F detects because it sees X_0 on the top of the stack. Finally, we need another new state, p_f, which is the accepting state of P_F; this PDA transfers to state p_f whenever it discovers that P_N would have emptied its stack.

The specification of P_F is as follows:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

where δ_F is defined by:

1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N, pushing its start symbol Z_0 onto the stack.

2. For all states q in Q, inputs a in Σ or $a = \epsilon$, and stack symbols Y in Γ, $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$.

3. In addition to rule (2), $\delta_F(q, \epsilon, X_0)$ contains (p_f, ϵ) for every state q in Q.

We must show that w is in $L(P_F)$ if and only if w is in $N(P_N)$.

(If) We are given that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$ for some state q. Theorem 6.5 lets us insert X_0 at the bottom of the stack and conclude $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, X_0)$. Since by rule (2) above, P_F has all the moves of P_N, we may also conclude that $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0)$. If we put this sequence of moves together with the initial and final moves from rules (1) and (3) above, we get:

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0) \underset{P_F}{\vdash} (p_f, \epsilon, \epsilon) \tag{6.1}$$

Thus, P_F accepts w by final state.

(Only-if) The converse requires only that we observe the additional transitions of rules (1) and (3) give us very limited ways to accept w by final state. We must use rule (3) at the last step, and we can only use that rule if the stack of P_F contains only X_0. No X_0's ever appear on the stack except at the bottommost position. Further, rule (1) is only used at the first step, and it *must* be used at the first step.

Thus, any computation of P_F that accepts w must look like sequence (6.1). Moreover, the middle of the computation — all but the first and last steps — must also be a computation of P_N with X_0 below the stack. The reason is that, except for the first and last steps, P_F cannot use any transition that is not also a transition of P_N, and X_0 cannot be exposed or the computation would end at the next step. We conclude that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$. That is, w is in $N(P_N)$. □

Example 6.10: Let us design a PDA that processes sequences of `if`'s and `else`'s in a C program, where i stands for `if` and e stands for `else`. Recall from Section 5.3.1 that there is a problem whenever the number of `else`'s in any prefix exceeds the number of `if`'s, because then we cannot match each `else` against its previous `if`. Thus, we shall use a stack symbol Z to count the difference between the number of i's seen so far and the number of e's. This simple, one-state PDA, is suggested by the transition diagram of Fig. 6.5.

We shall push another Z whenever we see an i and pop a Z whenever we see an e. Since we start with one Z on the stack, we actually follow the rule that if the stack is Z^n, then there have been $n-1$ more i's than e's. In particular, if the stack is empty, then we have seen one more e than i, and the input read so far has just become illegal for the first time. It is these strings that our PDA accepts by empty stack. The formal specification of P_N is:

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

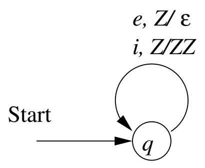

Figure 6.5: A PDA that accepts the if/else errors by empty stack

where δ_N is defined by:

1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$. This rule pushes a Z when we see an i.
2. $\delta_N(q, e, Z) = \{(q, \epsilon)\}$. This rule pops a Z when we see an e.

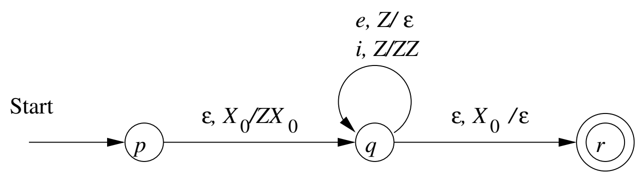

Figure 6.6: Construction of a PDA accepting by final state from the PDA of Fig. 6.5

Now, let us construct from P_N a PDA P_F that accepts the same language by final state; the transition diagram for P_F is shown in Fig. 6.6.[3] We introduce a new start state p and an accepting state r. We shall use X_0 as the bottom-of-stack marker. P_F is formally defined:

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

where δ_F consists of:

1. $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$. This rule starts P_F simulating P_N, with X_0 as a bottom-of-stack-marker.
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$. This rule pushes a Z when we see an i; it simulates P_N.
3. $\delta_F(q, e, Z) = \{(q, \epsilon)\}$. This rule pops a Z when we see an e; it also simulates P_N.
4. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$. That is, P_F accepts when the simulated P_N would have emptied its stack.

□

[3]Do not be concerned that we are using new states p and r here, while the construction in Theorem 6.9 used p_0 and p_f. Names of states are arbitrary, of course.

6.2.4 From Final State to Empty Stack

Now, let us go in the opposite direction: take a PDA P_F that accepts a language L by final state and construct another PDA P_N that accepts L by empty stack. The construction is simple and is suggested in Fig. 6.7. From each accepting state of P_F, add a transition on ϵ to a new state p. When in state p, P_N pops its stack and does not consume any input. Thus, whenever P_F enters an accepting state after consuming input w, P_N will empty its stack after consuming w.

To avoid simulating a situation where P_F accidentally empties its stack without accepting, P_N must also use a marker X_0 on the bottom of its stack. The marker is P_N's start symbol, and like the construction of Theorem 6.9, P_N must start in a new state p_0, whose sole function is to push the start symbol of P_F on the stack and go to the start state of P_F. The construction is sketched in Fig. 6.7, and we give it formally in the next theorem.

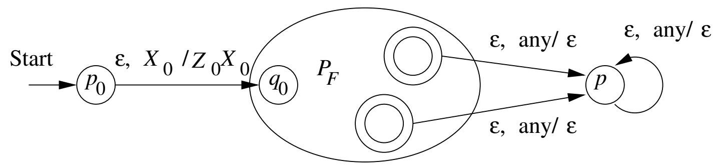

Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

PROOF: The construction is as suggested in Fig. 6.7. Let

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

where δ_N is defined by:

1. $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. We start by pushing the start symbol of P_F onto the stack and going to the start state of P_F.

2. For all states q in Q, input symbols a in Σ or $a = \epsilon$, and Y in Γ, $\delta_N(q, a, Y)$ contains every pair that is in $\delta_F(q, a, Y)$. That is, P_N simulates P_F.

3. For all accepting states q in F and stack symbols Y in Γ or $Y = X_0$, $\delta_N(q, \epsilon, Y)$ contains (p, ϵ). By this rule, whenever P_F accepts, P_N can start emptying its stack without consuming any more input.

4. For all stack symbols Y in Γ or $Y = X_0$, $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$. Once in state p, which only occurs when P_F has accepted, P_N pops every symbol on its stack, until the stack is empty. No further input is consumed.

Now, we must prove that w is in $N(P_N)$ if and only if w is in $L(P_F)$. The ideas are similar to the proof for Theorem 6.9. The "if" part is a direct simulation, and the "only-if" part requires that we examine the limited number of things that the constructed PDA P_N can do.

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

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \epsilon, \epsilon)$$

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

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

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \epsilon, \epsilon)$$

where q is an accepting state of P_F.

Moreover, between ID's (q_0, w, Z_0X_0) and $(q, \epsilon, \alpha X_0)$, all the moves are moves of P_F. In particular, X_0 was never the top stack symbol prior to reaching ID $(q, \epsilon, \alpha X_0)$.[4] Thus, we conclude that the same computation can occur in P_F, without the X_0 on the stack; that is, $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, \alpha)$. Now we see that P_F accepts w by final state, so w is in $L(P_F)$. $\Box$

6.2.5 Exercises for Section 6.2

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

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

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

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

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

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

[4]Although α could be ϵ, in which case P_F has emptied its stack at the same time it accepts.

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

!! **Exercise 6.2.3:** Design a PDA to accept each of the following languages.

a) $\{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$.

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

*! **Exercise 6.2.4:** Let P be a PDA with empty-stack language $L = N(P)$, and suppose that ϵ is not in L. Describe how you would modify P so that it accepts $L \cup \{\epsilon\}$ by empty stack.

Exercise 6.2.5: PDA $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ has the following rules defining δ:

$$\begin{array}{lll}
\delta(q_0, a, Z_0) = (q_1, AAZ_0) & \delta(q_0, b, Z_0) = (q_2, BZ_0) & \delta(q_0, \epsilon, Z_0) = (f, \epsilon) \\
\delta(q_1, a, A) = (q_1, AAA) & \delta(q_1, b, A) = (q_1, \epsilon) & \delta(q_1, \epsilon, Z_0) = (q_0, Z_0) \\
\delta(q_2, a, B) = (q_3, \epsilon) & \delta(q_2, b, B) = (q_2, BB) & \delta(q_2, \epsilon, Z_0) = (q_0, Z_0) \\
\delta(q_3, \epsilon, B) = (q_2, \epsilon) & \delta(q_3, \epsilon, Z_0) = (q_1, AZ_0) &
\end{array}$$

Note that, since each of the sets above has only one choice of move, we have omitted the set brackets from each of the rules.

* a) Give an execution trace (sequence of ID's) showing that string bab is in $L(P)$.

b) Give an execution trace showing that abb is in $L(P)$.

c) Give the contents of the stack after P has read b^7a^4 from its input.

! d) Informally describe $L(P)$.

Exercise 6.2.6: Consider the PDA P from Exercise 6.1.1.

a) Convert P to another PDA P_1 that accepts by empty stack the same language that P accepts by final state; i.e., $N(P_1) = L(P)$.

b) Find a PDA P_2 such that $L(P_2) = N(P)$; i.e., P_2 accepts by final state what P accepts by empty stack.

! **Exercise 6.2.7:** Show that if P is a PDA, then there is a PDA P_2 with only two stack symbols, such that $L(P_2) = L(P)$. *Hint*: Binary-code the stack alphabet of P.

*! **Exercise 6.2.8:** A PDA is called *restricted* if on any transition it can increase the height of the stack by at most one symbol. That is, for any rule $\delta(q, a, Z)$ contains (p, γ), it must be that $|\gamma| \leq 2$. Show that if P is a PDA, then there is a restricted PDA P_3 such that $L(P) = L(P_3)$.

6.3 Equivalence of PDA's and CFG's

Now, we shall demonstrate that the languages defined by PDA's are exactly the context-free languages. The plan of attack is suggested by Fig. 6.8. The goal is to prove that the following three classes of languages:

1. The context-free languages, i.e., the languages defined by CFG's.

2. The languages that are accepted by final state by some PDA.

3. The languages that are accepted by empty stack by some PDA.

are all the same class. We have already shown that (2) and (3) are the same. It turns out to be easiest next to show that (1) and (3) are the same, thus implying the equivalence of all three.

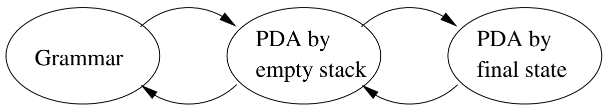

Figure 6.8: Organization of constructions showing equivalence of three ways of defining the CFL's

6.3.1 From Grammars to Pushdown Automata

Given a CFG G, we construct a PDA that simulates the leftmost derivations of G. Any left-sentential form that is not a terminal string can be written as $xA\alpha$, where A is the leftmost variable, x is whatever terminals appear to its left, and α is the string of terminals and variables that appear to the right of A. We call $A\alpha$ the *tail* of this left-sentential form. If a left-sentential form consists of terminals only, then its tail is ϵ.

The idea behind the construction of a PDA from a grammar is to have the PDA simulate the sequence of left-sentential forms that the grammar uses to generate a given terminal string w. The tail of each sentential form $xA\alpha$ appears on the stack, with A at the top. At that time, x will be "represented" by our having consumed x from the input, leaving whatever of w follows its prefix x. That is, if $w = xy$, then y will remain on the input.

Suppose the PDA is in an ID $(q, y, A\alpha)$, representing left-sentential form $xA\alpha$. It guesses the production to use to expand A, say $A \to \beta$. The move of the PDA is to replace A on the top of the stack by β, entering ID $(q, y, \beta\alpha)$. Note that there is only one state, q, for this PDA.

Now $(q, y, \beta\alpha)$ may not be a representation of the next left-sentential form, because β may have a prefix of terminals. In fact, β may have no variables at all, and α may have a prefix of terminals. Whatever terminals appear at the beginning of $\beta\alpha$ need to be removed, to expose the next variable at the top of the stack. These terminals are compared against the next input symbols, to make sure our guesses at the leftmost derivation of input string w are correct; if not, this branch of the PDA dies.

If we succeed in this way to guess a leftmost derivation of w, then we shall eventually reach the left-sentential form w. At that point, all the symbols on the stack have either been expanded (if they are variables) or matched against the input (if they are terminals). The stack is empty, and we accept by empty stack.

The above informal construction can be made precise as follows. Let $G = (V, T, Q, S)$ be a CFG. Construct the PDA P that accepts $L(G)$ by empty stack as follows:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

where transition function δ is defined by:

1. For each variable A,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \rightarrow \beta \text{ is a production of } G\}$$

2. For each terminal a, $\delta(q, a, a) = \{(q, \epsilon)\}$.

Example 6.12: Let us convert the expression grammar of Fig. 5.2 to a PDA. Recall this grammar is:

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ E & \rightarrow & I \mid E * E \mid E + E \mid (E) \end{array}$$

The set of input symbols for the PDA is $\{a, b, 0, 1, (,), +, *\}$. These eight symbols and the symbols I and E form the stack alphabet. The transition function for the PDA is:

a) $\delta(q, \epsilon, I) = \{(q, a),\ (q, b),\ (q, Ia),\ (q, Ib),\ (q, I0),\ (q, I1)\}$.

b) $\delta(q, \epsilon, E) = \{(q, I),\ (q, E + E),\ (q, E * E),\ (q, (E))\}$.

c) $\delta(q, a, a) = \{(q, \epsilon)\}$; $\delta(q, b, b) = \{(q, \epsilon)\}$; $\delta(q, 0, 0) = \{(q, \epsilon)\}$; $\delta(q, 1, 1) = \{(q, \epsilon)\}$; $\delta(q, (, () = \{(q, \epsilon)\}$; $\delta(q,),)) = \{(q, \epsilon)\}$; $\delta(q, +, +) = \{(q, \epsilon)\}$; $\delta(q, *, *) = \{(q, \epsilon)\}$.

Note that (a) and (b) come from rule (1), while the eight transitions of (c) come from rule (2). Also, δ is empty except as defined by (a) through (c). $\square$

Theorem 6.13: If PDA P is constructed from CFG G by the construction above, then $N(P) = L(G)$.

PROOF: We shall prove that w is in $N(P)$ if and only if w is in $L(G)$.

(If) Suppose w is in $L(G)$. Then w has a leftmost derivation

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$$

We show by induction on i that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, where y_i and α_i are a representation of the left-sentential form γ_i. That is, let α_i be the tail of γ_i, and let $\gamma_i = x_i\alpha_i$. Then y_i is that string such that $x_iy_i = w$; i.e., it is what remains when x_i is removed from the input.

BASIS: For $i = 1$, $\gamma_1 = S$. Thus, $x_1 = \epsilon$, and $y_1 = w$. Since $(q, w, S) \overset{*}{\vdash} (q, w, S)$ by 0 moves, the basis is proved.

INDUCTION: Now we consider the case of the second and subsequent left-sentential forms. We assume

$$(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$$

and prove $(q, w, S) \overset{*}{\vdash} (q, y_{i+1}, \alpha_{i+1})$. Since α_i is a tail, it begins with a variable A. Moreover, the step of the derivation $\gamma_i \Rightarrow \gamma_{i+1}$ involves replacing A by one of its production bodies, say β. Rule (1) of the construction of P lets us replace A at the top of the stack by β, and rule (2) then allows us to match any terminals on top of the stack with the next input symbols. As a result, we reach the ID $(q, y_{i+1}, \alpha_{i+1})$, which represents the next left-sentential form γ_{i+1}.

To complete the proof, we note that $\alpha_n = \epsilon$, since the tail of γ_n (which is w) is empty. Thus, $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$, which proves that P accepts w by empty stack.

(Only-if) We need to prove something more general: that if P executes a sequence of moves that has the net effect of popping a variable A from the top of its stack, without ever going below A on the stack, then A derives, in G, whatever input string was consumed from the input during this process. Precisely:

- If $(q, x, A) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon)$, then $A \overset{*}{\underset{G}{\Rightarrow}} x$.

The proof is an induction on the number of moves taken by P.

BASIS: One move. The only possibility is that $A \to \epsilon$ is a production of G, and this production is used in a rule of type (1) by the PDA P. In this case, $x = \epsilon$, and we know that $A \Rightarrow \epsilon$.

INDUCTION: Suppose P takes n moves, where $n > 1$. The first move must be of type (1), where A is replaced by one of its production bodies on the top of the stack. The reason is that a rule of type (2) can only be used when there is a terminal on top of the stack. Suppose the production used is $A \to Y_1Y_2 \cdots Y_k$, where each Y_i is either a terminal or variable.

The next $n-1$ moves of P must consume x from the input and have the net effect of popping each of Y_1, Y_2, and so on from the stack, one at a time. We can break x into $x_1x_2\cdots x_k$, where x_1 is the portion of the input consumed until Y_1 is popped off the stack (i.e., the stack first is as short as $k-1$ symbols). Then x_2 is the next portion of the input that is consumed while popping Y_2 off the stack, and so on.

Figure 6.9 suggests how the input x is broken up, and the corresponding effects on the stack. There, we suggest that β was BaC, so x is divided into three parts $x_1x_2x_3$, where $x_2 = a$. Note that in general, if Y_i is a terminal, then x_i must be that terminal.

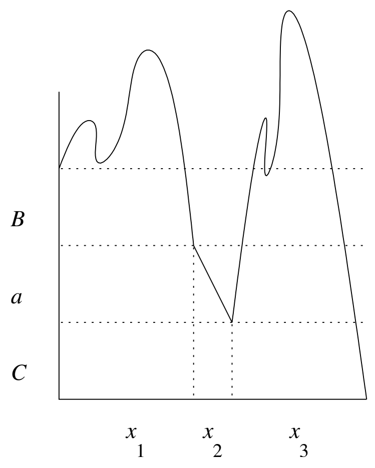

Figure 6.9: The PDA P consumes x and pops BaC from its stack

Formally, we can conclude that $(q, x_ix_{i+1}\cdots x_k, Y_i) \overset{*}{\vdash} (q, x_{i+1}\cdots x_k, \epsilon)$ for all $i = 1, 2, \ldots, k$. Moreover, none of these sequences can be more than $n-1$ moves, so the inductive hypothesis applies if Y_i is a variable. That is, we may conclude $Y_i \overset{*}{\Rightarrow} x_i$.

If Y_i is a terminal, then there must be only one move involved, and it matches the one symbol of x_i against Y_i, which are the same. Again, we can conclude $Y_i \overset{*}{\Rightarrow} x_i$; this time, zero steps are used. Now we have the derivation

$$A \Rightarrow Y_1Y_2\cdots Y_k \overset{*}{\Rightarrow} x_1Y_2\cdots Y_k \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} x_1x_2\cdots x_k$$

That is, $A \overset{*}{\Rightarrow} x$.

To complete the proof, we let $A = S$ and $x = w$. Since we are given that w is in $N(P)$, we know that $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$. By what we have just proved inductively, we have $S \overset{*}{\Rightarrow} w$; i.e., w is in $L(G)$. $\square$

3. From the fact that $\delta_N(q,e,Z)$ contains (q,ϵ), we have production

   $$[qZq] \rightarrow e$$

   Notice that in this case, the list of stack symbols by which Z is replaced is empty, so the only symbol in the body is the input symbol that caused the move.

We may, for convenience, replace the triple $[qZq]$ by some less complex symbol, say A. If we do, then the complete grammar consists of the productions:

$$\begin{array}{l} S \rightarrow A \\ A \rightarrow iAA \mid e \end{array}$$

In fact, if we notice that A and S derive exactly the same strings, we may identify them as one, and write the complete grammar as

$$G = (\{S\}, \{i,e\}, \{S \rightarrow iSS \mid e\}, S)$$

□

6.3.3 Exercises for Section 6.3

* **Exercise 6.3.1:** Convert the grammar

$$\begin{array}{lcl} S & \rightarrow & 0S1 \mid A \\ A & \rightarrow & 1A0 \mid S \mid \epsilon \end{array}$$

to a PDA that accepts the same language by empty stack.

Exercise 6.3.2: Convert the grammar

$$\begin{array}{lcl} S & \rightarrow & aAA \\ A & \rightarrow & aS \mid bS \mid a \end{array}$$

to a PDA that accepts the same language by empty stack.

* **Exercise 6.3.3:** Convert the PDA $P = (\{p,q\}, \{0,1\}, \{X, Z_0\}, \delta, q, Z_0)$ to a CFG, if δ is given by:

1. $\delta(q,1,Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q,1,X) = \{(q, XX)\}$.
3. $\delta(q,0,X) = \{(p, X)\}$.
4. $\delta(q,\epsilon,X) = \{(q, \epsilon)\}$.
5. $\delta(p,1,X) = \{(p, \epsilon)\}$.
6. $\delta(p,0,Z_0) = \{(q, Z_0)\}$.

Exercise 6.3.4: Convert the PDA of Exercise 6.1.1 to a context-free grammar.

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

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

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

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

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

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

6.4 Deterministic Pushdown Automata

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

6.4.1 Definition of a Deterministic PDA

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

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

2. If $\delta(q, a, X)$ is nonempty, for some a in Σ, then $\delta(q, \epsilon, X)$ must be empty.