MODULE 3

We now turn our attention away from the regular languages to a larger class of languages, called the "context-free languages." These languages have a natural, recursive notation, called "context-free grammars." Context-free grammars have played a central role in compiler technology since the 1960's; they turned the implementation of parsers (functions that discover the structure of a program) from a time-consuming, ad-hoc implementation task into a routine job that can be done in an afternoon. More recently, the context-free grammar has been used to describe document formats, via the so-called document-type definition (DTD) that is used in the XML (extensible markup language) community for information exchange on the Web.

In this chapter, we introduce the context-free grammar notation, and show how grammars define languages. We discuss the "parse tree," a picture of the structure that a grammar places on the strings of its language. The parse tree is the product of a parser for a programming language and is the way that the structure of programs is normally captured.

There is an automaton-like notation, called the "pushdown automaton," that also describes all and only the context-free languages; we introduce the pushdown automaton in Chapter 6. While less important than finite automata, we shall find the pushdown automaton, especially its equivalence to context-free grammars as a language-defining mechanism, to be quite useful when we explore the closure and decision properties of the context-free languages in Chapter 7.

5.1 Context-Free Grammars

We shall begin by introducing the context-free grammar notation informally. After seeing some of the important capabilities of these grammars, we offer formal definitions. We show how to define a grammar formally, and introduce

the process of "derivation," whereby it is determined which strings are in the language of the grammar.

5.1.1 An Informal Example

Let us consider the language of palindromes. A *palindrome* is a string that reads the same forward and backward, such as `otto` or `madamimadam` ("Madam, I'm Adam," allegedly the first thing Eve heard in the Garden of Eden). Put another way, string w is a palindrome if and only if $w = w^R$. To make things simple, we shall consider describing only the palindromes with alphabet $\{0,1\}$. This language includes strings like 0110, 11011, and ϵ, but not 011 or 0101.

It is easy to verify that the language L_{pal} of palindromes of 0's and 1's is not a regular language. To do so, we use the pumping lemma. If L_{pal} is a regular language, let n be the associated constant, and consider the palindrome $w = 0^n10^n$. If L_{pal} is regular, then we can break w into $w = xyz$, such that y consists of one or more 0's from the first group. Thus, xz, which would also have to be in L_{pal} if L_{pal} were regular, would have fewer 0's to the left of the lone 1 than there are to the right of the 1. Therefore xz cannot be a palindrome. We have now contradicted the assumption that L_{pal} is a regular language.

There is a natural, recursive definition of when a string of 0's and 1's is in L_{pal}. It starts with a basis saying that a few obvious strings are in L_{pal}, and then exploits the idea that if a string is a palindrome, it must begin and end with the same symbol. Further, when the first and last symbols are removed, the resulting string must also be a palindrome. That is:

BASIS: ϵ, 0, and 1 are palindromes.

INDUCTION: If w is a palindrome, so are $0w0$ and $1w1$. No string is a palindrome of 0's and 1's, unless it follows from this basis and induction rule.

A context-free grammar is a formal notation for expressing such recursive definitions of languages. A grammar consists of one or more variables that represent classes of strings, i.e., languages. In this example we have need for only one variable P, which represents the set of palindromes; that is the class of strings forming the language L_{pal}. There are rules that say how the strings in each class are constructed. The construction can use symbols of the alphabet, strings that are already known to be in one of the classes, or both.

Example 5.1: The rules that define the palindromes, expressed in the context-free grammar notation, are shown in Fig. 5.1. We shall see in Section 5.1.2 what the rules mean.

The first three rules form the basis. They tell us that the class of palindromes includes the strings ϵ, 0, and 1. None of the right sides of these rules (the portions following the arrows) contains a variable, which is why they form a basis for the definition.

The last two rules form the inductive part of the definition. For instance, rule 4 says that if we take any string w from the class P, then $0w0$ is also in class P. Rule 5 likewise tells us that $1w1$ is also in P. □

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

Figure 5.1: A context-free grammar for palindromes

5.1.2 Definition of Context-Free Grammars

There are four important components in a grammatical description of a language:

1. There is a finite set of symbols that form the strings of the language being defined. This set was $\{0, 1\}$ in the palindrome example we just saw. We call this alphabet the *terminals*, or *terminal symbols*.

2. There is a finite set of *variables*, also called sometimes *nonterminals* or *syntactic categories*. Each variable represents a language; i.e., a set of strings. In our example above, there was only one variable, P, which we used to represent the class of palindromes over alphabet $\{0, 1\}$.

3. One of the variables represents the language being defined; it is called the *start symbol*. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol. In our example, P, the only variable, is the start symbol.

4. There is a finite set of *productions* or *rules* that represent the recursive definition of a language. Each production consists of:

   (a) A variable that is being (partially) defined by the production. This variable is often called the *head* of the production.

   (b) The production symbol $\rightarrow$.

   (c) A string of zero or more terminals and variables. This string, called the *body* of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

   We saw an example of productions in Fig. 5.1.

The four components just described form a *context-free grammar*, or just *grammar*, or *CFG*. We shall represent a CFG G by its four components, that is, $G = (V, T, P, S)$, where V is the set of variables, T the terminals, P the set of productions, and S the start symbol.

Example 5.2: The grammar G_{pal} for the palindromes is represented by

$$G_{pal} = (\{P\}, \{0,1\}, A, P)$$

where A represents the set of five productions that we saw in Fig. 5.1. □

Example 5.3: Let us explore a more complex CFG that represents (a simplification of) expressions in a typical programming language. First, we shall limit ourselves to the operators + and *, representing addition and multiplication. We shall allow arguments to be identifiers, but instead of allowing the full set of typical identifiers (letters followed by zero or more letters and digits), we shall allow only the letters a and b and the digits 0 and 1. Every identifier must begin with a or b, which may be followed by any string in $\{a,b,0,1\}^*$.

We need two variables in this grammar. One, which we call E, represents expressions. It is the start symbol and represents the language of expressions we are defining. The other variable, I, represents identifiers. Its language is actually regular; it is the language of the regular expression

$$\mathbf{(a+b)(a+b+0+1)^*}$$

However, we shall not use regular expressions directly in grammars. Rather, we use a set of productions that say essentially the same thing as this regular expression.

1.	E	$\rightarrow$	I
2.	E	$\rightarrow$	$E+E$
3.	E	$\rightarrow$	$E*E$
4.	E	$\rightarrow$	(E)
5.	I	$\rightarrow$	a
6.	I	$\rightarrow$	b
7.	I	$\rightarrow$	Ia
8.	I	$\rightarrow$	Ib
9.	I	$\rightarrow$	$I0$
10.	I	$\rightarrow$	$I1$

Figure 5.2: A context-free grammar for simple expressions

The grammar for expressions is stated formally as $G = (\{E,I\}, T, P, E)$, where T is the set of symbols $\{+, *, (,), a, b, 0, 1\}$ and P is the set of productions shown in Fig. 5.2. We interpret the productions as follows.

Rule (1) is the basis rule for expressions. It says that an expression can be a single identifier. Rules (2) through (4) describe the inductive case for expressions. Rule (2) says that an expression can be two expressions connected by a plus sign; rule (3) says the same with a multiplication sign. Rule (4) says

Compact Notation for Productions

It is convenient to think of a production as "belonging" to the variable of its head. We shall often use remarks like "the productions for A" or "A-productions" to refer to the productions whose head is variable A. We may write the productions for a grammar by listing each variable once, and then listing all the bodies of the productions for that variable, separated by vertical bars. That is, the productions $A \to \alpha_1$, $A \to \alpha_2, \ldots, A \to \alpha_n$ can be replaced by the notation $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$. For instance, the grammar for palindromes from Fig. 5.1 can be written as $P \to \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$.

that if we take any expression and put matching parentheses around it, the result is also an expression.

Rules (5) through (10) describe identifiers I. The basis is rules (5) and (6); they say that a and b are identifiers. The remaining four rules are the inductive case. They say that if we have any identifier, we can follow it by a, b, 0, or 1, and the result will be another identifier. □

5.1.3 Derivations Using a Grammar

We apply the productions of a CFG to infer that certain strings are in the language of a certain variable. There are two approaches to this inference. The more conventional approach is to use the rules from body to head. That is, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is in the language of the variable in the head. We shall refer to this procedure as *recursive inference*.

There is another approach to defining the language of a grammar, in which we use the productions from head to body. We expand the start symbol using one of its productions (i.e., using a production whose head is the start symbol). We further expand the resulting string by replacing one of the variables by the body of one of its productions, and so on, until we derive a string consisting entirely of terminals. The language of the grammar is all strings of terminals that we can obtain in this way. This use of grammars is called *derivation*.

We shall begin with an example of the first approach — recursive inference. However, it is often more natural to think of grammars as used in derivations, and we shall next develop the notation for describing these derivations.

Example 5.4: Let us consider some of the inferences we can make using the grammar for expressions in Fig. 5.2. Figure 5.3 summarizes these inferences. For example, line (*i*) says that we can infer string a is in the language for I by using production 5. Lines (*ii*) through (*iv*) say we can infer that $b00$

is an identifier by using production 6 once (to get the b) and then applying production 9 twice (to attach the two 0's).

	String Inferred	For language of	Production used	String(s) used
(i)	a	I	5	—
(ii)	b	I	6	—
(iii)	$b0$	I	9	(ii)
(iv)	$b00$	I	9	(iii)
(v)	a	E	1	(i)
(vi)	$b00$	E	1	(iv)
(vii)	$a + b00$	E	2	$(v), (vi)$
$(viii)$	$(a + b00)$	E	4	(vii)
(ix)	$a * (a + b00)$	E	3	$(v), (viii)$

Figure 5.3: Inferring strings using the grammar of Fig. 5.2

Lines (v) and (vi) exploit production 1 to infer that, since any identifier is an expression, the strings a and $b00$, which we inferred in lines (i) and (iv) to be identifiers, are also in the language of variable E. Line (vii) uses production 2 to infer that the sum of these identifiers is an expression; line $(viii)$ uses production 4 to infer that the same string with parentheses around it is also an expression, and line (ix) uses production 3 to multiply the identifier a by the expression we had discovered in line $(viii)$. □

The process of deriving strings by applying productions from head to body requires the definition of a new relation symbol $\Rightarrow$. Suppose $G = (V, T, P, S)$ is a CFG. Let $\alpha A \beta$ be a string of terminals and variables, with A a variable. That is, α and β are strings in $(V \cup T)^*$, and A is in V. Let $A \to \gamma$ be a production of G. Then we say $\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$. If G is understood, we just say $\alpha A \beta \Rightarrow \alpha \gamma \beta$. Notice that one derivation step replaces any variable anywhere in the string by the body of one of its productions.

We may extend the $\Rightarrow$ relationship to represent zero, one, or many derivation steps, much as the transition function δ of a finite automaton was extended to $\hat{\delta}$. For derivations, we use a * to denote "zero or more steps," as follows:

BASIS: For any string α of terminals and variables, we say $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. That is, any string derives itself.

INDUCTION: If $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ and $\beta \underset{G}{\Rightarrow} \gamma$, then $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. That is, if α can become β by zero or more steps, and one more step takes β to γ, then α can become γ. Put another way, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ means that there is a sequence of strings $\gamma_1, \gamma_2, \ldots, \gamma_n$, for some $n \geq 1$, such that

1. $\alpha = \gamma_1$,

2. $\beta = \gamma_n$, and

3. For $i = 1, 2, \ldots, n-1$, we have $\gamma_i \Rightarrow \gamma_{i+1}$.

If grammar G is understood, then we use $\stackrel{*}{\Rightarrow}$ in place of $\stackrel{*}{\underset{G}{\Rightarrow}}$.

Example 5.5: The inference that $a*(a+b00)$ is in the language of variable E can be reflected in a derivation of that string, starting with the string E. Here is one such derivation:

$$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow$$

$$a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow$$

$$a*(a+I) \Rightarrow a*(a+I0) \Rightarrow a*(a+I00) \Rightarrow a*(a+b00)$$

At the first step, E is replaced by the body of production 3 (from Fig. 5.2). At the second step, production 1 is used to replace the first E by I, and so on. Notice that we have systematically adopted the policy of always replacing the leftmost variable in the string. However, at each step we may choose which variable to replace, and we can use any of the productions for that variable. For instance, at the second step, we could have replaced the second E by (E), using production 4. In that case, we would say $E*E \Rightarrow E*(E)$. We could also have chosen to make a replacement that would fail to lead to the same string of terminals. A simple example would be if we used production 2 at the first step, and said $E \Rightarrow E+E$. No replacements for the two E's could ever turn $E+E$ into $a*(a+b00)$.

We can use the $\stackrel{*}{\Rightarrow}$ relationship to condense the derivation. We know $E \stackrel{*}{\Rightarrow} E$ by the basis. Repeated use of the inductive part gives us $E \stackrel{*}{\Rightarrow} E*E$, $E \stackrel{*}{\Rightarrow} I*E$, and so on, until finally $E \stackrel{*}{\Rightarrow} a*(a+b00)$.

The two viewpoints — recursive inference and derivation — are equivalent. That is, a string of terminals w is inferred to be in the language of some variable A if and only if $A \stackrel{*}{\Rightarrow} w$. However, the proof of this fact requires some work, and we leave it to Section 5.2. □

5.1.4 Leftmost and Rightmost Derivations

In order to restrict the number of choices we have in deriving a string, it is often useful to require that at each step we replace the leftmost variable by one of its production bodies. Such a derivation is called a *leftmost derivation*, and we indicate that a derivation is leftmost by using the relations $\underset{lm}{\Rightarrow}$ and $\stackrel{*}{\underset{lm}{\Rightarrow}}$, for one or many steps, respectively. If the grammar G that is being used is not obvious, we can place the name G below the arrow in either of these symbols.

Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies. If so, we call the derivation *rightmost* and use

Notation for CFG Derivations

There are a number of conventions in common use that help us remember the role of the symbols we use when discussing CFG's. Here are the conventions we shall use:

1. Lower-case letters near the beginning of the alphabet, a, b, and so on, are terminal symbols. We shall also assume that digits and other characters such as + or parentheses are terminals.
2. Upper-case letters near the beginning of the alphabet, A, B, and so on, are variables.
3. Lower-case letters near the end of the alphabet, such as w or z, are strings of terminals. This convention reminds us that the terminals are analogous to the input symbols of an automaton.
4. Upper-case letters near the end of the alphabet, such as X or Y, are either terminals or variables.
5. Lower-case Greek letters, such as α and β, are strings consisting of terminals and/or variables.

There is no special notation for strings that consist of variables only, since this concept plays no important role. However, a string named α or another Greek letter might happen to have only variables.

the symbols $\underset{rm}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ to indicate one or many rightmost derivation steps, respectively. Again, the name of the grammar may appear below these symbols if it is not clear which grammar is being used.

Example 5.6: The derivation of Example 5.5 was actually a leftmost derivation. Thus, we can describe the same derivation by:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow}$$

$$a*(E) \underset{lm}{\Rightarrow} a*(E+E) \underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} a*(a+E) \underset{lm}{\Rightarrow}$$

$$a*(a+I) \underset{lm}{\Rightarrow} a*(a+I0) \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00)$$

We can also summarize the leftmost derivation by saying $E \overset{*}{\underset{lm}{\Rightarrow}} a*(a+b00)$, or express several steps of the derivation by expressions such as $E*E \overset{*}{\underset{lm}{\Rightarrow}} a*(E)$.

There is a rightmost derivation that uses the same replacements for each variable, although it makes the replacements in different order. This rightmost derivation is:

$$E \underset{rm}{\Rightarrow} E*E \underset{rm}{\Rightarrow} E*(E) \underset{rm}{\Rightarrow} E*(E+E) \underset{rm}{\Rightarrow}$$

$$E*(E+I) \underset{rm}{\Rightarrow} E*(E+I0) \underset{rm}{\Rightarrow} E*(E+I00) \underset{rm}{\Rightarrow} E*(E+b00) \underset{rm}{\Rightarrow}$$

$$E*(I+b00) \underset{rm}{\Rightarrow} E*(a+b00) \underset{rm}{\Rightarrow} I*(a+b00) \underset{rm}{\Rightarrow} a*(a+b00)$$

This derivation allows us to conclude $E \overset{*}{\underset{rm}{\Rightarrow}} a*(a+b00)$. □

Any derivation has an equivalent leftmost and an equivalent rightmost derivation. That is, if w is a terminal string, and A a variable, then $A \overset{*}{\Rightarrow} w$ if and only if $A \overset{*}{\underset{lm}{\Rightarrow}} w$, and $A \overset{*}{\Rightarrow} w$ if and only if $A \overset{*}{\underset{rm}{\Rightarrow}} w$. We shall also prove these claims in Section 5.2.

5.1.5 The Language of a Grammar

If $G(V,T,P,S)$ is a CFG, the *language* of G, denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol. That is,

$$L(G) = \{w \text{ in } T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

If a language L is the language of some context-free grammar, then L is said to be a *context-free language*, or CFL. For instance, we asserted that the grammar of Fig. 5.1 defined the language of palindromes over alphabet $\{0,1\}$. Thus, the set of palindromes is a context-free language. We can prove that statement, as follows.

Theorem 5.7: $L(G_{pal})$, where G_{pal} is the grammar of Example 5.1, is the set of palindromes over $\{0,1\}$.

PROOF: We shall prove that a string w in $\{0,1\}^*$ is in $L(G_{pal})$ if and only if it is a palindrome; i.e., $w = w^R$.

(If) Suppose w is a palindrome. We show by induction on $|w|$ that w is in $L(G_{pal})$.

BASIS: We use lengths 0 and 1 as the basis. If $|w| = 0$ or $|w| = 1$, then w is ϵ, 0, or 1. Since there are productions $P \to \epsilon$, $P \to 0$, and $P \to 1$, we conclude that $P \overset{*}{\Rightarrow} w$ in any of these basis cases.

INDUCTION: Suppose $|w| \geq 2$. Since $w = w^R$, w must begin and end with the same symbol That is, $w = 0x0$ or $w = 1x1$. Moreover, x must be a palindrome; that is, $x = x^R$. Note that we need the fact that $|w| \geq 2$ to infer that there are two distinct 0's or 1's, at either end of w.

If $w = 0x0$, then we invoke the inductive hypothesis to claim that $P \stackrel{*}{\Rightarrow} x$. Then there is a derivation of w from P, namely $P \Rightarrow 0P0 \stackrel{*}{\Rightarrow} 0x0 = w$. If $w = 1x1$, the argument is the same, but we use the production $P \to 1P1$ at the first step. In either case, we conclude that w is in $L(G_{pal})$ and complete the proof.

(Only-if) Now, we assume that w is in $L(G_{pal})$; that is, $P \stackrel{*}{\Rightarrow} w$. We must conclude that w is a palindrome. The proof is an induction on the number of steps in a derivation of w from P.

BASIS: If the derivation is one step, then it must use one of the three productions that do not have P in the body. That is, the derivation is $P \Rightarrow \epsilon$, $P \Rightarrow 0$, or $P \Rightarrow 1$. Since ϵ, 0, and 1 are all palindromes, the basis is proven.

INDUCTION: Now, suppose that the derivation takes $n+1$ steps, where $n \geq 1$, and the statement is true for all derivations of n steps. That is, if $P \stackrel{*}{\Rightarrow} x$ in n steps, then x is a palindrome.

Consider an $(n+1)$-step derivation of w, which must be of the form

$$P \Rightarrow 0P0 \stackrel{*}{\Rightarrow} 0x0 = w$$

or $P \Rightarrow 1P1 \stackrel{*}{\Rightarrow} 1x1 = w$, since $n+1$ steps is at least two steps, and the productions $P \to 0P0$ and $P \to 1P1$ are the only productions whose use allows additional steps of a derivation. Note that in either case, $P \stackrel{*}{\Rightarrow} x$ in n steps.

By the inductive hypothesis, we know that x is a palindrome; that is, $x = x^R$. But if so, then $0x0$ and $1x1$ are also palindromes. For instance, $(0x0)^R = 0x^R0 = 0x0$. We conclude that w is a palindrome, which completes the proof. □

5.1.6 Sentential Forms

Derivations from the start symbol produce strings that have a special role. We call these "sentential forms." That is, if $G = (V, T, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \stackrel{*}{\Rightarrow} \alpha$ is a *sentential form*. If $S \underset{lm}{\stackrel{*}{\Rightarrow}} \alpha$, then α is a *left-sentential form*, and if $S \underset{rm}{\stackrel{*}{\Rightarrow}} \alpha$, then α is a *right-sentential form*. Note that the language $L(G)$ is those sentential forms that are in T^*; i.e., they consist solely of terminals.

Example 5.8: Consider the grammar for expressions from Fig. 5.2. For example, $E * (I + E)$ is a sentential form, since there is a derivation

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.

As an example of a left-sentential form, consider $a * E$, with the leftmost derivation

The Form of Proofs About Grammars

Theorem 5.7 is typical of proofs that show a grammar defines a particular, informally defined language. We first develop an inductive hypothesis that states what properties the strings derived from each variable have. In this example, there was only one variable, P, so we had only to claim that its strings were palindromes.

We prove the "if" part: that if a string w satisfies the informal statement about the strings of one of the variables A, then $A \stackrel{*}{\Rightarrow} w$. In our example, since P is the start symbol, we stated "$P \stackrel{*}{\Rightarrow} w$" by saying that w is in the language of the grammar. Typically, we prove the "if" part by induction on the length of w. If there are k variables, then the inductive statement to be proved has k parts, which must be proved as a mutual induction.

We must also prove the "only-if" part, that if $A \stackrel{*}{\Rightarrow} w$, then w satisfies the informal statement about the strings derived from variable A. Again, in our example, since we had to deal only with the start symbol P, we assumed that w was in the language of G_{pal} as an equivalent to $P \stackrel{*}{\Rightarrow} w$. The proof of this part is typically by induction on the number of steps in the derivation. If the grammar has productions that allow two or more variables to appear in derived strings, then we shall have to break a derivation of n steps into several parts, one derivation from each of the variables. These derivations may have fewer than n steps, so we have to perform an induction assuming the statement for all values n or less, as discussed in Section 1.4.2.

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

Additionally, the derivation

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

shows that $E * (E + E)$ is a right-sentential form. □

5.1.7 Exercises for Section 5.1

Exercise 5.1.1: Design context-free grammars for the following languages:

* a) The set $\{0^n 1^n \mid n \geq 1\}$, that is, the set of all strings of one or more 0's followed by an equal number of 1's.

*! b) The set $\{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$, that is, the set of strings of a's followed by b's followed by c's, such that there are either a different number of a's and b's or a different number of b's and c's, or both.

! c) 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.

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

Exercise 5.1.2: The following grammar generates the language of regular expression $0^*1(0+1)^*$:

$$\begin{array}{lcl} S & \to & A1B \\ A & \to & 0A \mid \epsilon \\ B & \to & 0B \mid 1B \mid \epsilon \end{array}$$

Give leftmost and rightmost derivations of the following strings:

* a) 00101.

b) 1001.

c) 00011.

! **Exercise 5.1.3:** Show that every regular language is a context-free language. *Hint*: Construct a CFG by induction on the number of operators in the regular expression.

! **Exercise 5.1.4:** A CFG is said to be *right-linear* if each production body has at most one variable, and that variable is at the right end. That is, all productions of a right-linear grammar are of the form $A \to wB$ or $A \to w$, where A and B are variables and w some string of zero or more terminals.

a) Show that every right-linear grammar generates a regular language. *Hint*: Construct an ϵ-NFA that simulates leftmost derivations, using its state to represent the lone variable in the current left-sentential form.

b) Show that every regular language has a right-linear grammar. *Hint*: Start with a DFA and let the variables of the grammar represent states.

*! **Exercise 5.1.5:** Let $T = \{0, 1, (,), +, *, \emptyset, e\}$. We may think of T as the set of symbols used by regular expressions over alphabet $\{0, 1\}$; the only difference is that we use e for symbol ϵ, to avoid potential confusion in what follows. Your task is to design a CFG with set of terminals T that generates exactly the regular expressions with alphabet $\{0, 1\}$.

Exercise 5.1.6: We defined the relation $\stackrel{*}{\Rightarrow}$ with a basis "$\alpha \Rightarrow \alpha$" and an induction that says "$\alpha \stackrel{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$ imply $\alpha \stackrel{*}{\Rightarrow} \gamma$. There are several other ways to define $\stackrel{*}{\Rightarrow}$ that also have the effect of saying that "$\stackrel{*}{\Rightarrow}$ is zero or more $\Rightarrow$ steps." Prove that the following are true:

a) $\alpha \stackrel{*}{\Rightarrow} \beta$ if and only if there is a sequence of one or more strings

$$\gamma_1, \gamma_2, \ldots, \gamma_n$$

such that $\alpha = \gamma_1$, $\beta = \gamma_n$, and for $i = 1, 2, \ldots, n-1$ we have $\gamma_i \Rightarrow \gamma_{i+1}$.

b) If $\alpha \stackrel{*}{\Rightarrow} \beta$, and $\beta \stackrel{*}{\Rightarrow} \gamma$, then $\alpha \stackrel{*}{\Rightarrow} \gamma$. *Hint*: use induction on the number of steps in the derivation $\beta \stackrel{*}{\Rightarrow} \gamma$.

! **Exercise 5.1.7:** Consider the CFG G defined by productions:

$$S \to aS \mid Sb \mid a \mid b$$

a) Prove by induction on the string length that no string in $L(G)$ has ba as a substring.

b) Describe $L(G)$ informally. Justify your answer using part (a).

!! **Exercise 5.1.8:** Consider the CFG G defined by productions:

$$S \to aSbS \mid bSaS \mid \epsilon$$

Prove that $L(G)$ is the set of all strings with an equal number of a's and b's.

5.2 Parse Trees

There is a tree representation for derivations that has proved extremely useful. This tree shows us clearly how the symbols of a terminal string are grouped into substrings, each of which belongs to the language of one of the variables of the grammar. But perhaps more importantly, the tree, known as a "parse tree" when used in a compiler, is the data structure of choice to represent the source program. In a compiler, the tree structure of the source program facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

In this section, we introduce the parse tree and show that the existence of parse trees is tied closely to the existence of derivations and recursive inferences. We shall later study the matter of ambiguity in grammars and languages, which is an important application of parse trees. Certain grammars allow a terminal string to have more than one parse tree. That situation makes the grammar unsuitable for a programming language, since the compiler could not tell the structure of certain source programs, and therefore could not with certainty deduce what the proper executable code for the program was.

5.2.1 Constructing Parse Trees

Let us fix on a grammar $G = (V, T, P, S)$. The *parse trees* for G are trees with the following conditions:

1. Each interior node is labeled by a variable in V.

2. Each leaf is labeled by either a variable, a terminal, or ϵ. However, if the leaf is labeled ϵ, then it must be the only child of its parent.

Review of Tree Terminology

We assume you have been introduced to the idea of a tree and are familiar with the commonly used definitions for trees. However, the following will serve as a review.

- Trees are collections of *nodes*, with a *parent-child* relationship. A node has at most one parent, drawn above the node, and zero or more children, drawn below. Lines connect parents to their children. Figures 5.4, 5.5, and 5.6 are examples of trees.
- There is one node, the *root*, that has no parent; this node appears at the top of the tree. Nodes with no children are called *leaves*. Nodes that are not leaves are *interior nodes*.
- A child of a child of a $\cdots$ node is a *descendant* of that node. A parent of a parent of a $\cdots$ is an *ancestor*. Trivially, nodes are ancestors and descendants of themselves.
- The children of a node are ordered "from the left," and drawn so. If node N is to the left of node M, then all the descendants of N are considered to be to the left of all the descendants of M.

3. If an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

respectively, from the left, then $A \to X_1X_2\cdots X_k$ is a production in P. Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \to \epsilon$ is a production of G.

Example 5.9: Figure 5.4 shows a parse tree that uses the expression grammar of Fig. 5.2. The root is labeled with the variable E. We see that the production used at the root is $E \to E + E$, since the three children of the root have labels E, $+$, and E, respectively, from the left. At the leftmost child of the root, the production $E \to I$ is used, since there is one child of that node, labeled I. □

Example 5.10: Figure 5.5 shows a parse tree for the palindrome grammar of Fig. 5.1. The production used at the root is $P \to 0P0$, and at the middle child of the root it is $P \to 1P1$. Note that at the bottom is a use of the production $P \to \epsilon$. That use, where the node labeled by the head has one child, labeled ϵ, is the only time that a node labeled ϵ can appear in a parse tree. □

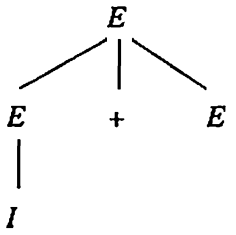

Figure 5.4: A parse tree showing the derivation of $I + E$ from E

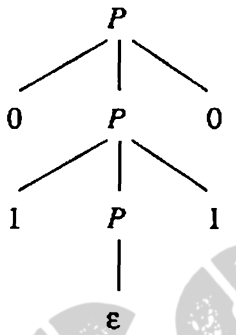

Figure 5.5: A parse tree showing the derivation $P \stackrel{*}{\Rightarrow} 0110$

5.2.2 The Yield of a Parse Tree

If we look at the leaves of any parse tree and concatenate them from the left, we get a string, called the *yield* of the tree, which is always a string that is derived from the root variable. The fact that the yield is derived from the root will be proved shortly. Of special importance are those parse trees such that:

1. The yield is a terminal string. That is, all leaves are labeled either with a terminal or with ϵ.

2. The root is labeled by the start symbol.

These are the parse trees whose yields are strings in the language of the underlying grammar. We shall also prove shortly that another way to describe the language of a grammar is as the set of yields of those parse trees having the start symbol at the root and a terminal string as yield.

Example 5.11: Figure 5.6 is an example of a tree with a terminal string as yield and the start symbol at the root; it is based on the grammar for expressions that we introduced in Fig. 5.2. This tree's yield is the string $a * (a + b00)$ that was derived in Example 5.5. In fact, as we shall see, this particular parse tree is a representation of that derivation. □

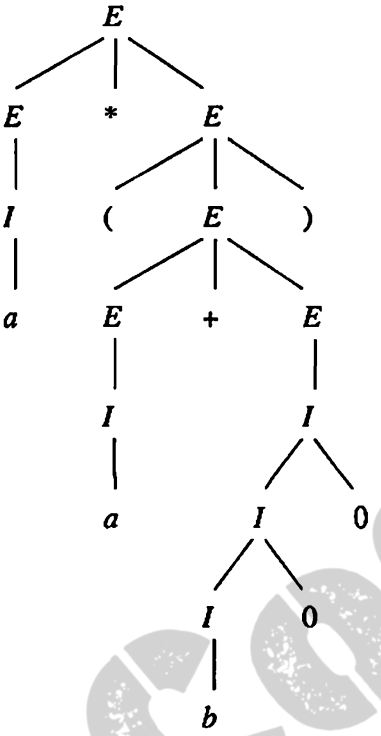

Figure 5.6: Parse tree showing $a * (a + b00)$ is in the language of our expression grammar

5.2.3 Inference, Derivations, and Parse Trees

Each of the ideas that we have introduced so far for describing how a grammar works gives us essentially the same facts about strings. That is, given a grammar $G = (V, T, P, S)$, we shall show that the following are equivalent:

1. The recursive inference procedure determines that terminal string w is in the language of variable A.
2. $A \stackrel{*}{\Rightarrow} w$.
3. $A \underset{lm}{\stackrel{*}{\Rightarrow}} w$.
4. $A \underset{rm}{\stackrel{*}{\Rightarrow}} w$.
5. There is a parse tree with root A and yield w.

In fact, except for the use of recursive inference, which we only defined for terminal strings, all the other conditions — the existence of derivations, leftmost or rightmost derivations, and parse trees — are also equivalent if w is a string that has some variables.

We need to prove these equivalences, and we do so using the plan of Fig. 5.7. That is, each arc in that diagram indicates that we prove a theorem that says if w meets the condition at the tail, then it meets the condition at the head of the arc. For instance, we shall show in Theorem 5.12 that if w is inferred to be in the language of A by recursive inference, then there is a parse tree with root A and yield w.

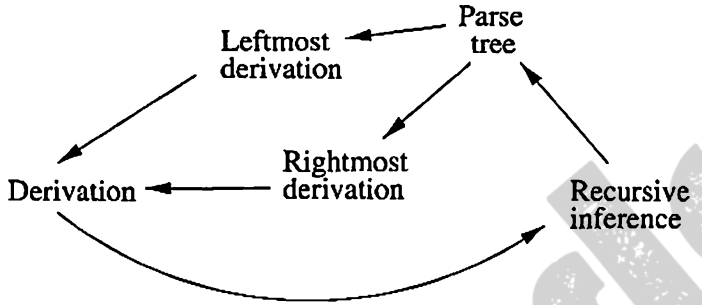

Figure 5.7: Proving the equivalence of certain statements about grammars

Note that two of the arcs are very simple and will not be proved formally. If w has a leftmost derivation from A, then it surely has a derivation from A, since a leftmost derivation *is* a derivation. Likewise, if w has a rightmost derivation, then it surely has a derivation. We now proceed to prove the harder steps of this equivalence.

5.2.4 From Inferences to Trees

Theorem 5.12: Let $G = (V, T, P, S)$ be a CFG. If the recursive inference procedure tells us that terminal string w is in the language of variable A, then there is a parse tree with root A and yield w.

PROOF: The proof is an induction on the number of steps used to infer that w is in the language of A.

BASIS: One step. Then only the basis of the inference procedure must have been used. Thus, there must be a production $A \to w$. The tree of Fig. 5.8, where there is one leaf for each position of w, meets the conditions to be a parse tree for grammar G, and it evidently has yield w and root A. In the special case that $w = \epsilon$, the tree has a single leaf labeled ϵ and is a legal parse tree with root A and yield w.

INDUCTION: Suppose that the fact w is in the language of A is inferred after $n+1$ inference steps, and that the statement of the theorem holds for all strings x and variables B such that the membership of x in the language of B was inferred using n or fewer inference steps. Consider the last step of the inference that w is in the language of A. This inference uses some production for A, say $A \to X_1 X_2 \cdots X_k$, where each X_i is either a variable or a terminal.

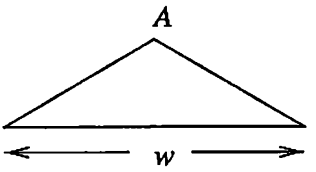

Figure 5.8: Tree constructed in the basis case of Theorem 5.12

We can break w up as $w_1 w_2 \cdots w_k$, where:

1. If X_i is a terminal, then $w_i = X_i$; i.e., w_i consists of only this one terminal from the production.

2. If X_i is a variable, then w_i is a string that was previously inferred to be in the language of X_i. That is, this inference about w_i took at most n of the $n+1$ steps of the inference that w is in the language of A. It cannot take all $n+1$ steps, because the final step, using production $A \to X_1 X_2 \cdots X_k$, is surely not part of the inference about w_i. Consequently, we may apply the inductive hypothesis to w_i and X_i, and conclude that there is a parse tree with yield w_i and root X_i.

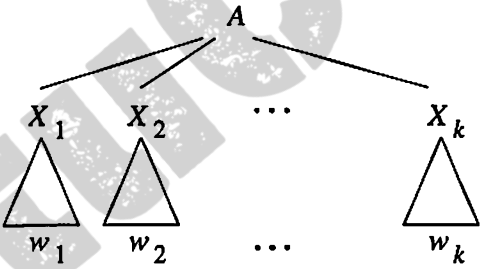

Figure 5.9: Tree used in the inductive part of the proof of Theorem 5.12

We then construct a tree with root A and yield w, as suggested in Fig. 5.9. There is a root labeled A, whose children are $X_1, X_2, \ldots, X_k$. This choice is valid, since $A \to X_1 X_2 \cdots X_k$ is a production of G.

The node for each X_i is made the root of a subtree with yield w_i. In case (1), where X_i is a terminal, this subtree is a trivial tree with a single node labeled X_i. That is, the subtree consists of only this child of the root. Since $w_i = X_i$ in case (1), we meet the condition that the yield of the subtree is w_i.

In case (2), X_i is a variable. Then, we invoke the inductive hypothesis to claim that there is some tree with root X_i and yield w_i. This tree is attached to the node for X_i in Fig. 5.9.

The tree so constructed has root A. Its yield is the yields of the subtrees, concatenated from left to right. That string is $w_1 w_2 \cdots w_k$, which is w. □

5.2.5 From Trees to Derivations

We shall now show how to construct a leftmost derivation from a parse tree. The method for constructing a rightmost derivation uses the same ideas, and we shall not explore the rightmost-derivation case. In order to understand how derivations may be constructed, we need first to see how one derivation of a string from a variable can be embedded within another derivation. An example should illustrate the point.

Example 5.13: Let us again consider the expression grammar of Fig. 5.2. It is easy to check that there is a derivation

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

As a result, for any strings α and β, it is also true that

$$\alpha E\beta \Rightarrow \alpha I\beta \Rightarrow \alpha Ib\beta \Rightarrow \alpha ab\beta$$

The justification is that we can make the same replacements of production bodies for heads in the context of α and β as we can in isolation.[1]

For instance, if we have a derivation that begins $E \Rightarrow E + E \Rightarrow E + (E)$, we could apply the derivation of ab from the second E by treating "$E + ($" as α and "$)$" as β. This derivation would then continue

$$E + (E) \Rightarrow E + (I) \Rightarrow E + (Ib) \Rightarrow E + (ab)$$

□

We are now able to prove a theorem that lets us convert a parse tree to a leftmost derivation. The proof is an induction on the *height* of the tree, which is the maximum length of a path that starts at the root, and proceeds downward through descendants, to a leaf. For instance, the height of the tree in Fig. 5.6 is 7. The longest root-to-leaf path in this tree goes to the leaf labeled b. Note that path lengths conventionally count the edges, not the nodes, so a path consisting of a single node is of length 0.

Theorem 5.14: Let $G = (V, T, P, S)$ be a CFG, and suppose there is a parse tree with root labeled by variable A and with yield w, where w is in T^*. Then there is a leftmost derivation $A \underset{lm}{\overset{*}{\Rightarrow}} w$ in grammar G.

PROOF: We perform an induction on the height of the tree.

[1] In fact, it is this property of being able to make a string-for-variable substitution regardless of context that gave rise originally to the term "context-free." There is a more powerful classes of grammars, called "context-sensitive," where replacements are permitted only if certain strings appear to the left and/or right. Context-sensitive grammars do not play a major role in practice today.

BASIS: The basis is height 1, the least that a parse tree with a yield of terminals can be. In this case, the tree must look like Fig. 5.8, with a root labeled A and children that read w, left-to-right. Since this tree is a parse tree, $A \to w$ must be a production. Thus, $A \underset{lm}{\Rightarrow} w$ is a one-step, leftmost derivation of w from A.

INDUCTION: If the height of the tree is n, where $n > 1$, it must look like Fig 5.9. That is, there is a root labeled A, with children labeled $X_1, X_2, \ldots, X_k$ from the left. The X's may be either terminals or variables.

1. If X_i is a terminal, define w_i to be the string consisting of X_i alone.

2. If X_i is a variable, then it must be the root of some subtree with a yield of terminals, which we shall call w_i. Note that in this case, the subtree is of height less than n, so the inductive hypothesis applies to it. That is, there is a leftmost derivation $X_i \overset{*}{\underset{lm}{\Rightarrow}} w_i$.

Note that $w = w_1 w_2 \cdots w_k$.

We construct a leftmost derivation of w as follows. We begin with the step $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_k$. Then, for each $i = 1, 2, \ldots, k$, in order, we show that

$$A \overset{*}{\underset{lm}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

This proof is actually another induction, this time on i. For the basis, $i = 0$, we already know that $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_k$. For the induction, assume that

$$A \overset{*}{\underset{lm}{\Rightarrow}} w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k$$

a) If X_i is a terminal, do nothing. However, we shall subsequently think of X_i as the terminal string w_i. Thus, we already have

$$A \overset{*}{\underset{lm}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

b) If X_i is a variable, continue with a derivation of w_i from X_i, in the context of the derivation being constructed. That is, if this derivation is

$$X_i \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \cdots \underset{lm}{\Rightarrow} w_i$$

we proceed with

$$\begin{aligned}
&w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&w_1 w_2 \cdots w_{i-1} \alpha_1 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&w_1 w_2 \cdots w_{i-1} \alpha_2 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&\cdots \\
&w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k
\end{aligned}$$

The result is a derivation $A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} \cdots X_k$.

When $i = k$, the result is a leftmost derivation of w from A. □

Example 5.15: Let us construct the leftmost derivation for the tree of Fig. 5.6. We shall show only the final step, where we construct the derivation from the entire tree from derivations that correspond to the subtrees of the root. That is, we shall assume that by recursive application of the technique in Theorem 5.14, we have deduced that the subtree rooted at the first child of the root has leftmost derivation $E \underset{lm}{\Rightarrow} I \underset{lm}{\Rightarrow} a$, while the subtree rooted at the third child of the root has leftmost derivation

$$E \underset{lm}{\Rightarrow} (E) \underset{lm}{\Rightarrow} (E+E) \underset{lm}{\Rightarrow} (I+E) \underset{lm}{\Rightarrow} (a+E) \underset{lm}{\Rightarrow}$$

$$(a+I) \underset{lm}{\Rightarrow} (a+I0) \underset{lm}{\Rightarrow} (a+I00) \underset{lm}{\Rightarrow} (a+b00)$$

To build a leftmost derivation for the entire tree, we start with the step at the root: $A \underset{lm}{\Rightarrow} E * E$. Then, we replace the first E according to its derivation, following each step by $*E$ to account for the larger context in which that derivation is used. The leftmost derivation so far is thus

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

The * in the production used at the root requires no derivation, so the above leftmost derivation also accounts for the first two children of the root. We complete the leftmost derivation by using the derivation of $E \underset{lm}{\overset{*}{\Rightarrow}} (a+b00)$, in a context where it is preceded by $a*$ and followed by the empty string. This derivation actually appeared in Example 5.6; it is:

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E \underset{lm}{\Rightarrow}$$

$$a * (E) \underset{lm}{\Rightarrow} a * (E+E) \underset{lm}{\Rightarrow} a * (I+E) \underset{lm}{\Rightarrow} a * (a+E) \underset{lm}{\Rightarrow}$$

$$a * (a+I) \underset{lm}{\Rightarrow} a * (a+I0) \underset{lm}{\Rightarrow} a * (a+I00) \underset{lm}{\Rightarrow} a * (a+b00)$$

□

A similar theorem lets us convert a tree to a rightmost derivation. The construction of a rightmost derivation from a tree is almost the same as the construction of a leftmost derivation. However, after starting with the step $A \underset{rm}{\Rightarrow} X_1 X_2 \cdots X_k$, we expand X_k first, using a rightmost derivation, then expand X_{k-1}, and so on, down to X_1. Thus, we shall state without further proof:

Theorem 5.16: Let $G = (V, T, P, S)$ be a CFG, and suppose there is a parse tree with root labeled by variable A and with yield w, where w is in T^*. Then there is a rightmost derivation $A \underset{rm}{\overset{*}{\Rightarrow}} w$ in grammar G. □

5.2.6 From Derivations to Recursive Inferences

We now complete the loop suggested by Fig. 5.7 by showing that whenever there is a derivation $A \stackrel{*}{\Rightarrow} w$ for some CFG, then the fact that w is in the language of A is discovered in the recursive inference procedure. Before giving the theorem and proof, let us observe something important about derivations.

Suppose that we have a derivation $A \Rightarrow X_1X_2\cdots X_k \stackrel{*}{\Rightarrow} w$. Then we can break w into pieces $w = w_1w_2\cdots w_k$ such that $X_i \stackrel{*}{\Rightarrow} w_i$. Note that if X_i is a terminal, then $w_i = X_i$, and the derivation is zero steps. The proof of this observation is not hard. You can show by induction on the number of steps of the derivation, that if $X_1X_2\cdots X_k \stackrel{*}{\Rightarrow} \alpha$, then all the positions of α that come from expansion of X_i are to the left of all the positions that come from expansion of X_j, if $i < j$.

If X_i is a variable, we can obtain the derivation of $X_i \stackrel{*}{\Rightarrow} w_i$ by starting with the derivation $A \stackrel{*}{\Rightarrow} w$, and stripping away:

a) All the positions of the sentential forms that are either to the left or right of the positions that are derived from X_i, and

b) All the steps that are not relevant to the derivation of w_i from X_i.

An example should make this process clear.

Example 5.17: Using the expression grammar of Fig. 5.2, consider the derivation

$$E \Rightarrow E*E \Rightarrow E*E+E \Rightarrow I*E+E \Rightarrow I*I+E \Rightarrow$$

$$I*I+I \Rightarrow a*I+I \Rightarrow a*b+I \Rightarrow a*b+a$$

Consider the third sentential form, $E*E+E$, and the middle E in this form.[2]

Starting from $E*E+E$, we may follow the steps of the above derivation, but strip away whatever positions are derived from the $E*$ to the left of the central E or derived from the $+E$ to its right. The steps of the derivation then become E, E, I, I, I, b, b. That is, the next step does not change the central E, the step after that changes it to I, the next two steps leave it as I, the next changes it to b, and the final step does not change what is derived from the central E.

If we take only the steps that change what comes from the central E, the sequence of strings E, E, I, I, I, b, b becomes the derivation $E \Rightarrow I \Rightarrow b$. That derivation correctly describes how the central E evolves during the complete derivation. □

Theorem 5.18: Let $G = (V, T, P, S)$ be a CFG, and suppose there is a derivation $A \stackrel{*}{\underset{G}{\Rightarrow}} w$, where w is in T^*. Then the recursive inference procedure applied to G determines that w is in the language of variable A.

[2]Our discussion of finding subderivations from larger derivations assumed we were concerned with a variable in the second sentential form of some derivation. However, the idea applies to a variable in any step of a derivation.

PROOF: The proof is an induction on the length of the derivation $A \stackrel{*}{\Rightarrow} w$.

BASIS: If the derivation is one-step, then $A \to w$ must be a production. Since w consists of terminals only, the fact that w is in the language of A will be discovered in the basis part of the recursive inference procedure.

INDUCTION: Suppose the derivation takes $n+1$ steps, and assume that for any derivation of n or fewer steps, the statement holds. Write the derivation as $A \Rightarrow X_1X_2\cdots X_k \stackrel{*}{\Rightarrow} w$. Then, as discussed prior to the theorem, we can break w as $w = w_1w_2\cdots w_k$, where:

a) If X_i is a terminal, then $w_i = X_i$.

b) If X_i is a variable, then $X_i \stackrel{*}{\Rightarrow} w_i$. Since the first step of the derivation $A \stackrel{*}{\Rightarrow} w$ is surely not part of the derivation $X_i \stackrel{*}{\Rightarrow} w_i$, we know that this derivation is of n or fewer steps. Thus, the inductive hypothesis applies to it, and we know that w_i is inferred to be in the language of X_i.

Now, we have a production $A \to X_1X_2\cdots X_k$, with w_i either equal to X_i or known to be in the language of X_i. In the next round of the recursive inference procedure, we shall discover that $w_1w_2\cdots w_k$ is in the language of A. Since $w_1w_2\cdots w_k = w$, we have shown that w is inferred to be in the language of A. □

5.2.7 Exercises for Section 5.2

Exercise 5.2.1: For the grammar and each of the strings in Exercise 5.1.2, give parse trees.

! **Exercise 5.2.2:** Suppose that G is a CFG without any productions that have ϵ as the right side. If w is in $L(G)$, the length of w is n, and w has a derivation of m steps, show that w has a parse tree with $n+m$ nodes.

! **Exercise 5.2.3:** Suppose all is as in Exercise 5.2.2, but G may have some productions with ϵ as the right side. Show that a parse tree for a string w other than ϵ may have as many as $n+2m-1$ nodes, but no more.

! **Exercise 5.2.4:** In Section 5.2.6 we mentioned that if $X_1X_2\cdots X_k \stackrel{*}{\Rightarrow} \alpha$, then all the positions of α that come from expansion of X_i are to the left of all the positions that come from expansion of X_j, if $i < j$. Prove this fact. *Hint*: Perform an induction on the number of steps in the derivation.

5.4 Ambiguity in Grammars and Languages

As we have seen, applications of CFG's often rely on the grammar to provide the structure of files. For instance, we saw in Section 5.3 how grammars can be used to put structure on programs and documents. The tacit assumption was that a grammar uniquely determines a structure for each string in its language. However, we shall see that not every grammar does provide unique structures.

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

5.4.1 Ambiguous Grammars

Let us return to our running example: the expression grammar of Fig. 5.2. This grammar lets us generate expressions with any sequence of $*$ and $+$ operators, and the productions $E \to E + E \mid E * E$ allow us to generate these expressions in any order we choose.

Example 5.25: For instance, consider the sentential form $E + E * E$. It has two derivations from E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$

Notice that in derivation (1), the second E is replaced by $E * E$, while in derivation (2), the first E is replaced by $E + E$. Figure 5.17 shows the two parse trees, which we should note are distinct trees.

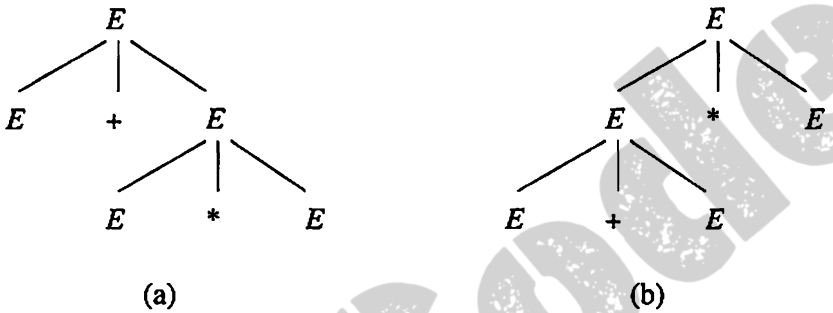

Figure 5.17: Two parse trees with the same yield

The difference between these two derivations is significant. As far as the structure of the expressions is concerned, derivation (1) says that the second and third expressions are multiplied, and the result is added to the first expression, while derivation (2) adds the first two expressions and multiplies the result by the third. In more concrete terms, the first derivation suggests that $1 + 2 * 3$ should be grouped $1 + (2 * 3) = 7$, while the second derivation suggests the same expression should be grouped $(1 + 2) * 3 = 9$. Obviously, the first of these, and not the second, matches our notion of correct grouping of arithmetic expressions.

Since the grammar of Fig. 5.2 gives two different structures to any string of terminals that is derived by replacing the three expressions in $E + E * E$ by identifiers, we see that this grammar is not a good one for providing unique structure. In particular, while it can give strings the correct grouping as arithmetic expressions, it also gives them incorrect groupings. To use this expression grammar in a compiler, we would have to modify it to provide only the correct groupings. □

On the other hand, the mere existence of different derivations for a string (as opposed to different parse trees) does not imply a defect in the grammar. The following is an example.

Example 5.26: Using the same expression grammar, we find that the string $a + b$ has many different derivations. Two examples are:

1. $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
2. $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

However, there is no real difference between the structures provided by these derivations; they each say that a and b are identifiers, and that their values are to be added. In fact, both of these derivations produce the same parse tree if the construction of Theorems 5.18 and 5.12 are applied. □

The two examples above suggest that it is not a multiplicity of derivations that cause ambiguity, but rather the existence of two or more parse trees. Thus, we say a CFG $G = (V, T, P, S)$ is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labeled S and yield w. If each string has at most one parse tree in the grammar, then the grammar is *unambiguous*.

For instance, Example 5.25 almost demonstrated the ambiguity of the grammar of Fig. 5.2. We have only to show that the trees of Fig. 5.17 can be completed to have terminal yields. Figure 5.18 is an example of that completion.

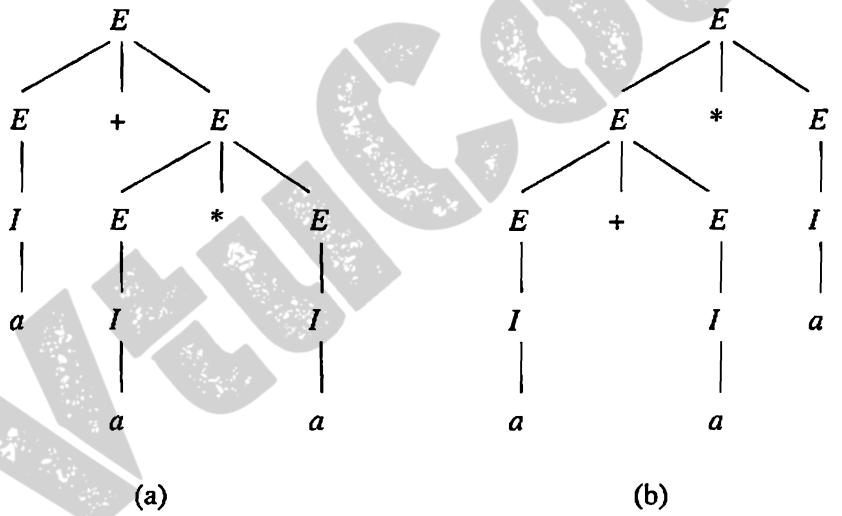

Figure 5.18: Trees with yield $a + a * a$, demonstrating the ambiguity of our expression grammar

5.4.2 Removing Ambiguity From Grammars

In an ideal world, we would be able to give you an algorithm to remove ambiguity from CFG's, much as we were able to show an algorithm in Section 4.4 to remove unnecessary states of a finite automaton. However, the surprising fact is, as we shall show in Section 9.5.2, that there is no algorithm whatsoever that can even tell us whether a CFG is ambiguous in the first place. Moreover, we

Ambiguity Resolution in YACC

If the expression grammar we have been using is ambiguous, we might wonder whether the sample YACC program of Fig. 5.11 is realistic. True, the underlying grammar is ambiguous, but much of the power of the YACC parser-generator comes from providing the user with simple mechanisms for resolving most of the common causes of ambiguity. For the expression grammar, it is sufficient to insist that:

a) $*$ takes precedence over $+$. That is, $*$'s must be grouped before adjacent $+$'s on either side. This rule tells us to use derivation (1) in Example 5.25, rather than derivation (2).

b) Both $*$ and $+$ are left-associative. That is, group sequences of expressions, all of which are connected by $*$, from the left, and do the same for sequences connected by $+$.

YACC allows us to state the precedence of operators by listing them in order, from lowest to highest precedence. Technically, the precedence of an operator applies to the use of any production of which that operator is the rightmost terminal in the body. We can also declare operators to be left- or right-associative with the keywords `%left` and `%right`. For instance, to declare that $+$ and $*$ were both left associative, with $*$ taking precedence over $+$, we would put ahead of the grammar of Fig. 5.11 the statements:

```
%left '+'
%left '*'
```

shall see in Section 5.4.4 that there are context-free languages that have nothing but ambiguous CFG's; for these languages, removal of ambiguity is impossible.

Fortunately, the situation in practice is not so grim. For the sorts of constructs that appear in common programming languages, there are well-known techniques for eliminating ambiguity. The problem with the expression grammar of Fig. 5.2 is typical, and we shall explore the elimination of its ambiguity as an important illustration.

First, let us note that there are two causes of ambiguity in the grammar of Fig. 5.2:

1. The precedence of operators is not respected. While Fig. 5.17(a) properly groups the $*$ before the $+$ operator, Fig 5.17(b) is also a valid parse tree and groups the $+$ ahead of the $*$. We need to force only the structure of Fig. 5.17(a) to be legal in an unambiguous grammar.

2. A sequence of identical operators can group either from the left or from the right. For example, if the $*$'s in Fig. 5.17 were replaced by $+$'s, we would see two different parse trees for the string $E + E + E$. Since addition and multiplication are associative, it doesn't matter whether we group from the left or the right, but to eliminate ambiguity, we must pick one. The conventional approach is to insist on grouping from the left, so the structure of Fig. 5.17(b) is the only correct grouping of two $+$-signs.

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of "binding strength." Specifically:

1. A *factor* is an expression that cannot be broken apart by any adjacent operator, either a $*$ or a $+$. The only factors in our expression language are:
   (a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   (b) Any parenthesized expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent what is inside from becoming the operand of any operator outside the parentheses.
2. A *term* is an expression that cannot be broken by the $+$ operator. In our example, where $+$ and $*$ are the only operators, a term is a product of one or more factors. For instance, the term $a * b$ can be "broken" if we use left associativity and place $a1*$ to its left. That is, $a1 * a * b$ is grouped $(a1 * a) * b$, which breaks apart the $a * b$. However, placing an additive term, such as $a1+$, to its left or $+a1$ to its right cannot break $a * b$. The proper grouping of $a1 + a * b$ is $a1 + (a * b)$, and the proper grouping of $a * b + a1$ is $(a * b) + a1$.
3. An *expression* will henceforth refer to any possible expression, including those that can be broken by either an adjacent $*$ or an adjacent $+$. Thus, an expression for our example is a sum of one or more terms.

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

Figure 5.19: An unambiguous expression grammar

Example 5.27: Figure 5.19 shows an unambiguous grammar that generates the same language as the grammar of Fig. 5.2. Think of F, T, and E as the

variables whose languages are the factors, terms, and expressions, as defined above. For instance, this grammar allows only one parse tree for the string $a + a * a$; it is shown in Fig. 5.20.

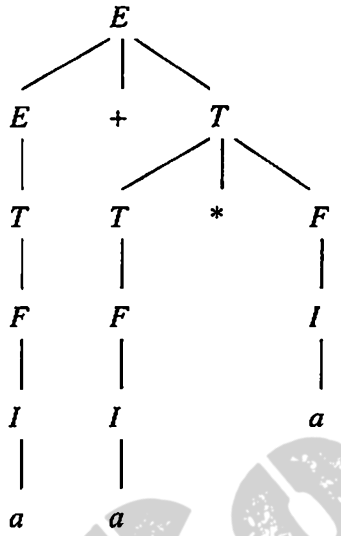

Figure 5.20: The sole parse tree for $a + a * a$

The fact that this grammar is unambiguous may be far from obvious. Here are the key observations that explain why no string in the language can have two different parse trees.

- Any string derived from T, a term, must be a sequence of one or more factors, connected by $*$'s. A factor, as we have defined it, and as follows from the productions for F in Fig. 5.19, is either a single identifier or any parenthesized expression.

- Because of the form of the two productions for T, the only parse tree for a sequence of factors is the one that breaks $f_1 * f_2 * \cdots * f_n$, for $n > 1$ into a term $f_1 * f_2 * \cdots * f_{n-1}$ and a factor f_n. The reason is that F cannot derive expressions like $f_{n-1} * f_n$ without introducing parentheses around them. Thus, it is not possible that when using the production $T \to T * F$, the F derives anything but the last of the factors. That is, the parse tree for a term can only look like Fig. 5.21.

- Likewise, an expression is a sequence of terms connected by $+$. When we use the production $E \to E + T$ to derive $t_1 + t_2 + \cdots + t_n$, the T must derive only t_n, and the E in the body derives $t_1 + t_2 + \cdots + t_{n-1}$. The reason, again, is that T cannot derive the sum of two or more terms without putting parentheses around them.

□

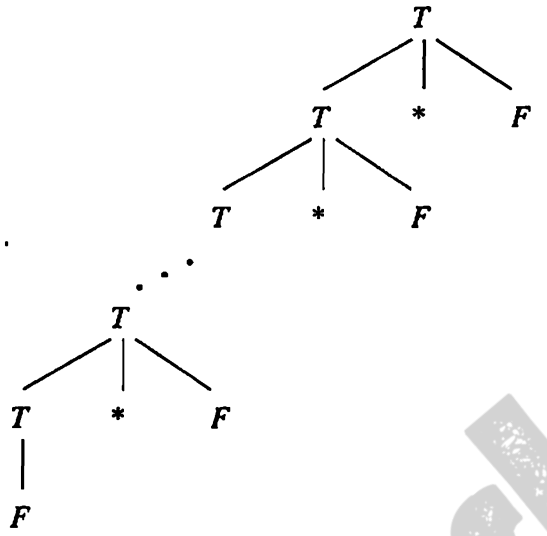

Figure 5.21: The form of all parse trees for a term

5.4.3 Leftmost Derivations as a Way to Express Ambiguity

While derivations are not necessarily unique, even if the grammar is unambiguous, it turns out that, in an unambiguous grammar, leftmost derivations will be unique, and rightmost derivations will be unique. We shall consider leftmost derivations only, and state the result for rightmost derivations.

Example 5.28: As an example, notice the two parse trees of Fig. 5.18 that each yield $E + E * E$. If we construct leftmost derivations from them we get the following leftmost derivations from trees (a) and (b), respectively:

a) $E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} I+E \underset{lm}{\Rightarrow} a+E \underset{lm}{\Rightarrow} a+E*E \underset{lm}{\Rightarrow} a+I*E \underset{lm}{\Rightarrow} a+a*E \underset{lm}{\Rightarrow} a+a*I \underset{lm}{\Rightarrow} a+a*a$

b) $E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} E+E*E \underset{lm}{\Rightarrow} I+E*E \underset{lm}{\Rightarrow} a+E*E \underset{lm}{\Rightarrow} a+I*E \underset{lm}{\Rightarrow} a+a*E \underset{lm}{\Rightarrow} a+a*I \underset{lm}{\Rightarrow} a+a*a$

Note that these two leftmost derivations differ. This example does not prove the theorem, but demonstrates how the differences in the trees force different steps to be taken in the leftmost derivation. □

Theorem 5.29: For each grammar $G = (V, T, P, S)$ and string w in T^*, w has two distinct parse trees if and only if w has two distinct leftmost derivations from S.

PROOF: (Only-if) If we examine the construction of a leftmost derivation from a parse tree in the proof of Theorem 5.14, we see that wherever the two parse trees first have a node at which different productions are used, the leftmost derivations constructed will also use different productions and thus be different derivations.

(If) While we have not previously given a direct construction of a parse tree from a leftmost derivation, the idea is not hard. Start constructing a tree with only the root, labeled S. Examine the derivation one step at a time. At each step, a variable will be replaced, and this variable will correspond to the leftmost node in the tree being constructed that has no children but that has a variable as its label. From the production used at this step of the leftmost derivation, determine what the children of this node should be. If there are two distinct derivations, then at the first step where the derivations differ, the nodes being constructed will get different lists of children, and this difference guarantees that the parse trees are distinct. □

5.4.4 Inherent Ambiguity

A context-free language L is said to be *inherently ambiguous* if all its grammars are ambiguous. If even one grammar for L is unambiguous, then L is an unambiguous language. We saw, for example, that the language of expressions generated by the grammar of Fig. 5.2 is actually unambiguous. Even though that grammar is ambiguous, there is another grammar for the same language that is unambiguous — the grammar of Fig. 5.19.

We shall not prove that there are inherently ambiguous languages. Rather we shall discuss one example of a language that can be proved inherently ambiguous, and we shall explain intuitively why every grammar for the language must be ambiguous. The language L in question is:

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $\mathbf{a^+b^+c^+d^+}$ such that either:

1. There are as many a's as b's and as many c's as d's, or
2. There are as many a's as d's and as many b's as c's.

L is a context-free language. The obvious grammar for L is shown in Fig. 5.22. It uses separate sets of productions to generate the two kinds of strings in L.

This grammar is ambiguous. For example, the string *aabbccdd* has the two leftmost derivations:

1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

$$
\begin{array}{lcl}
S & \rightarrow & AB \mid C \\
A & \rightarrow & aAb \mid ab \\
B & \rightarrow & cBd \mid cd \\
C & \rightarrow & aCd \mid aDd \\
D & \rightarrow & bDc \mid bc
\end{array}
$$

Figure 5.22: A grammar for an inherently ambiguous language

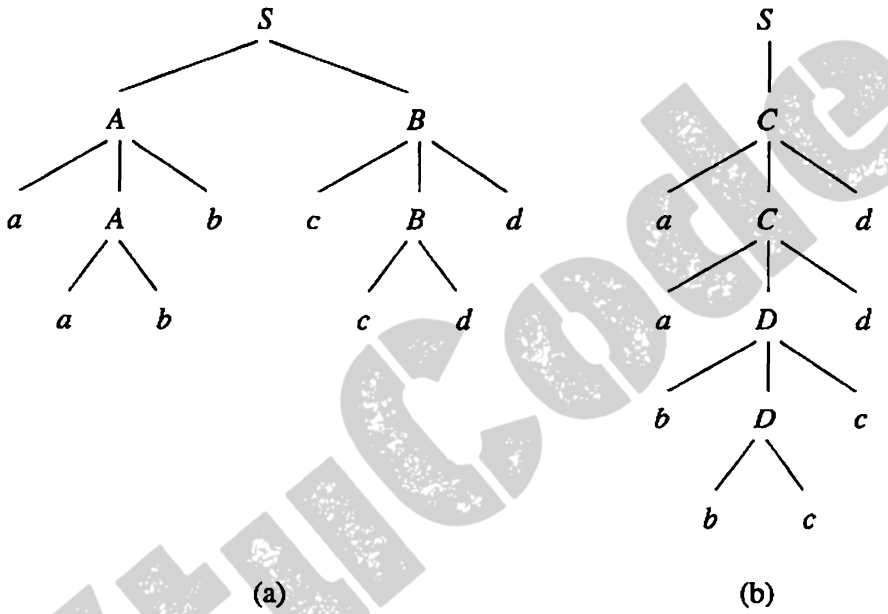

Figure 5.23: Two parse trees for *aabbccdd*

and the two parse trees shown in Fig. 5.23.

The proof that all grammars for L must be ambiguous is complex. However, the essence is as follows. We need to argue that all but a finite number of the strings whose counts of the four symbols a, b, c, and d, are all equal must be generated in two different ways: one in which the a's and b are generated to be equal and the c's and d's are generated to be equal, and a second way, where the a's and d's are generated to be equal and likewise the b's and c's.

For instance, the only way to generate strings where the a's and b's have the same number is with a variable like A in the grammar of Fig. 5.22. There are variations, of course, but these variations do not change the basic picture. For instance:

- Some small strings can be avoided, say by changing the basis production $A \rightarrow ab$ to $A \rightarrow aaabbb$, for instance.

- We could arrange that A shares its job with some other variables, e.g., by using variables A_1 and A_2, with A_1 generating the odd numbers of a's and A_2 generating the even numbers, as: $A_1 \to aA_2b \mid ab$; $A_2 \to aA_1b \mid ab$.

- We could also arrange that the numbers of a's and b's generated by A are not exactly equal, but off by some finite number. For instance, we could start with a production like $S \to AbB$ and then use $A \to aAb \mid a$ to generate one more a than b's.

However, we cannot avoid some mechanism for generating a's in a way that matches the count of b's.

Likewise, we can argue that there must be a variable like B that generates matching c's and d's. Also, variables that play the roles of C (generate matching a's and d's) and D (generate matching b's and c's) must be available in the grammar. The argument, when formalized, proves that no matter what modifications we make to the basic grammar, it will generate at least *some* of the strings of the form $a^nb^nc^nd^n$ in the two ways that the grammar of Fig. 5.22 does.

5.4.5 Exercises for Section 5.4

*** Exercise 5.4.1:** Consider the grammar

$$S \to aS \mid aSbS \mid \epsilon$$

This grammar is ambiguous. Show in particular that the string aab has two:

a) Parse trees.

b) Leftmost derivations.

c) Rightmost derivations.

! Exercise 5.4.2: Prove that the grammar of Exercise 5.4.1 generates all and only the strings of a's and b's such that every prefix has at least as many a's as b's.

***! Exercise 5.4.3:** Find an unambiguous grammar for the language of Exercise 5.4.1.

!! Exercise 5.4.4: Some strings of a's and b's have a unique parse tree in the grammar of Exercise 5.4.1. Give an efficient test to tell whether a given string is one of these. The test "try all parse trees to see how many yield the given string" is not adequately efficient.

! Exercise 5.4.5: This question concerns the grammar from Exercise 5.1.2, which we reproduce here:

$$\begin{aligned} S &\rightarrow A1B \\ A &\rightarrow 0A \mid \epsilon \\ B &\rightarrow 0B \mid 1B \mid \epsilon \end{aligned}$$

a) Show that this grammar is unambiguous.

b) Find a grammar for the same language that *is* ambiguous, and demonstrate its ambiguity.

*! **Exercise 5.4.6:** Is your grammar from Exercise 5.1.5 unambiguous? If not, redesign it to be unambiguous.

Exercise 5.4.7: The following grammar generates *prefix* expressions with operands x and y and binary operators $+$, $-$, and $*$:

$$E \rightarrow +EE \mid *EE \mid -EE \mid x \mid y$$

a) Find leftmost and rightmost derivations, and a derivation tree for the string +*-xyxy.

! b) Prove that this grammar is unambiguous.

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 $\{0^n1^n2^n \mid n \geq 1\}$, the set of strings consisting of equal groups of 0's, 1's, and 2's.

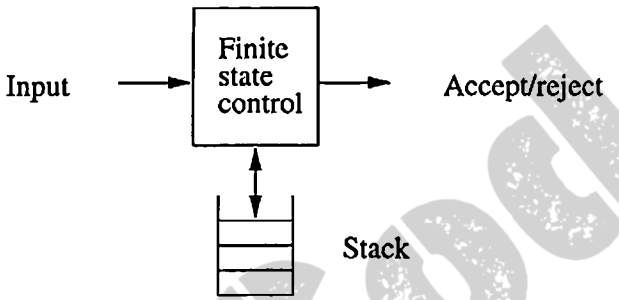

Figure 6.1: A pushdown automaton is essentially a finite automaton with a stack data structure

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

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

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.

[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 chose 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.

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

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.

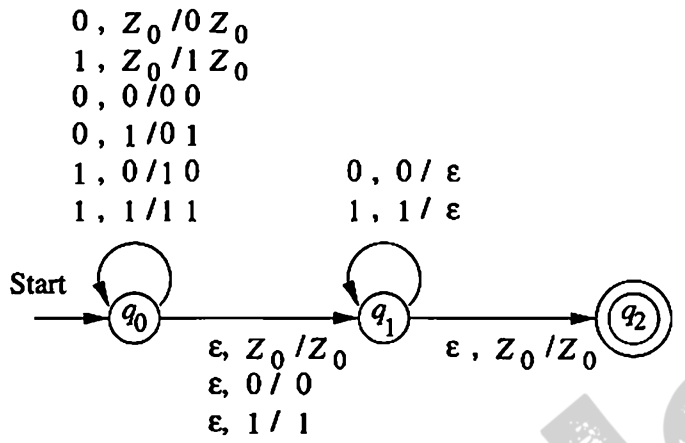

Figure 6.2: Representing a PDA as a generalized transition diagram

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. □

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 $\vdash_P$, 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 $\vdash_P^*$, or $\vdash^*$ when the PDA P is understood, to represent zero or more moves of the PDA. That is:

BASIS: $I \vdash^* I$ for any ID I.

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

That is, $I \vdash^* J$ 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.

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 $(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$. □

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

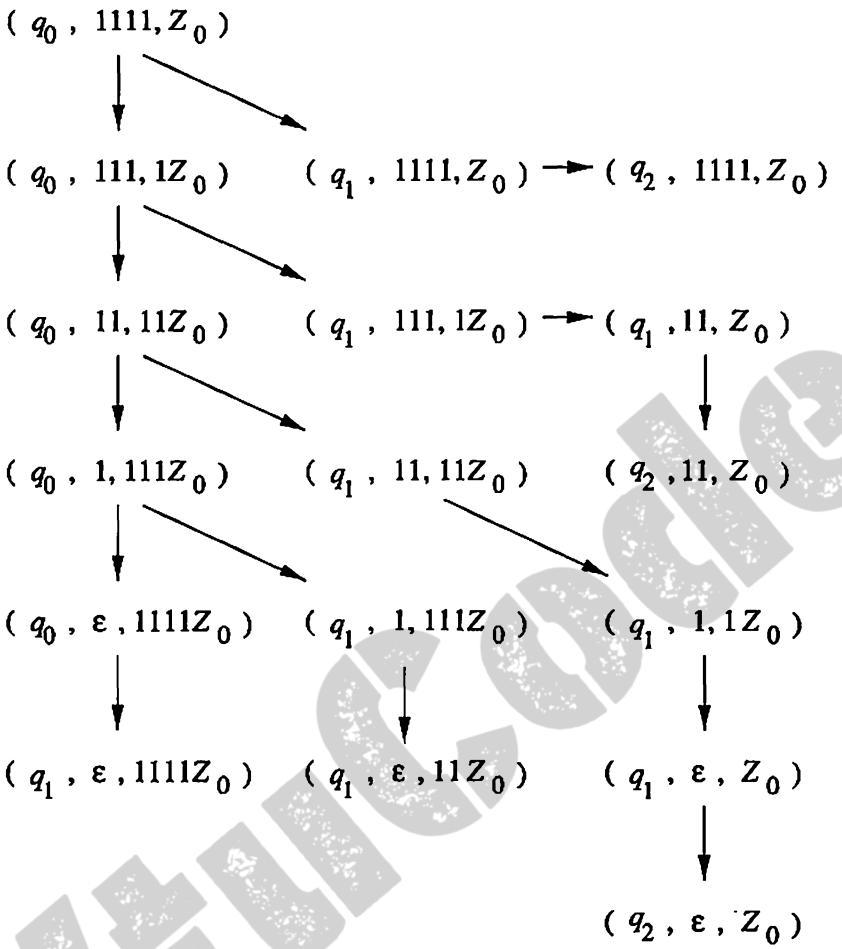

Figure 6.3: ID's of the PDA of Example 6.2 on input 1111

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.

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.

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

$$(q, xw, \alpha\gamma) \stackrel{*}{\vdash}_P (p, yw, \beta\gamma)$$

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

PROOF: The proof is actually a very simple induction on the number of steps in the sequence of ID's that take $(q, xw, \alpha\gamma)$ to $(p, yw, \beta\gamma)$. Each of the moves in the sequence $(q, x, \alpha) \stackrel{*}{\vdash}_P (p, y, \beta)$ 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. □

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:

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 $(q, wx) \vdash^* (p, x)$ 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.

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 $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$. □

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

$$\{w \mid (q_0, w, Z_0) \stackrel{*}{\vdash}_P (q, \epsilon, \alpha)\}$$

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

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

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 $(q_0, x, Z_0) \stackrel{*}{\vdash} (q_1, \epsilon, Z_0)$; 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 $(q_0, x, \alpha) \vdash^* (q_1, \epsilon, \alpha)$, 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 $(q_0, \epsilon, \alpha) \vdash^* (q_1, \epsilon, \alpha)$ 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 $(q_1, x, \alpha) \vdash^* (q_1, \epsilon, \beta)$, then β will be shorter than α and cannot be equal to α.

2. $(q_0, a_1 a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1 \alpha)$. 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

   $$(q_0, a_2 \cdots a_n, a_1 \alpha) \vdash^* (q_1, a_n, a_1 \alpha)$$

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

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

   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 $x = a_1 y y^R a_n$, and we know $a_1 = a_n$, we conclude that x is of the form ww^R; specifically $w = a_1 y$.

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

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

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]

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

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 $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

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.

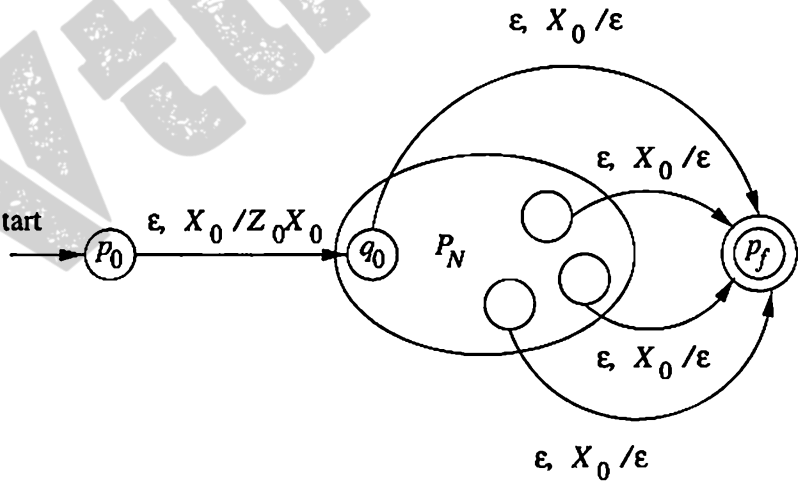

Figure 6.4: P_F simulates P_N and accepts if P_N empties its stack

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.

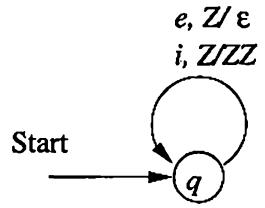

Figure 6.5: A PDA that accepts the if/else errors by empty stack

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, than 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:

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

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.

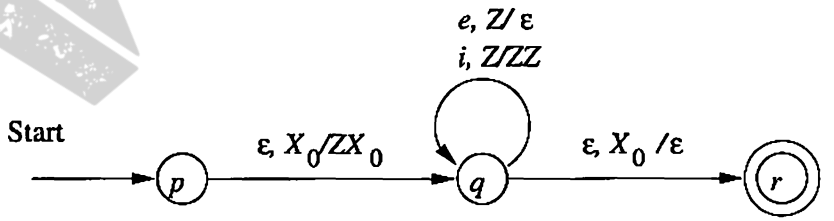

Figure 6.6: Construction of a PDA accepting by final state from the PDA of Fig. 6.5

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

[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.

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. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon, \epsilon)\}$. That is, P_F accepts when the simulated P_N would have emptied its stack.

□

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.

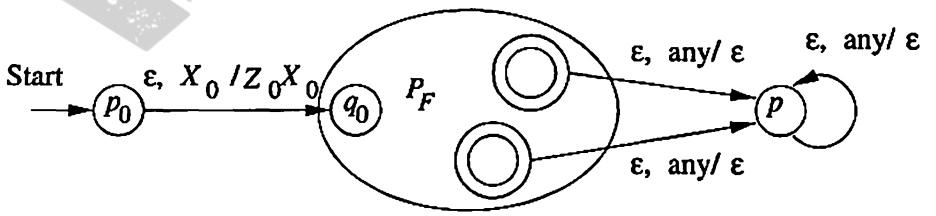

Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

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 $L(P_F)$. □

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

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).

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) $\{a^ib^jc^k \mid i \neq j \text{ or } j \neq k\}$.

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:

$\delta(q_0, a, Z_0) = (q_1, AAZ_0)$	$\delta(q_0, b, Z_0) = (q_2, BZ_0)$	$\delta(q_0, \epsilon, Z_0) = (f, \epsilon)$
$\delta(q_1, a, A) = (q_1, AAA)$	$\delta(q_1, b, A) = (q_1, \epsilon)$	$\delta(q_1, \epsilon, Z_0) = (q_0, Z_0)$
$\delta(q_2, a, B) = (q_3, \epsilon)$	$\delta(q_2, b, B) = (q_2, BB)$	$\delta(q_2, \epsilon, Z_0) = (q_0, Z_0)$
$\delta(q_3, \epsilon, B) = (q_2, \epsilon)$	$\delta(q_3, \epsilon, Z_0) = (q_1, AZ_0)$	

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.

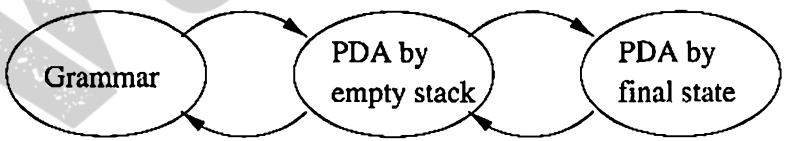

Figure 6.8: Organization of constructions showing equivalence of three ways of defining the CFL's

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,

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

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:

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

The set of terminals 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). □

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},α_{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 $A \to Y_1 Y_2 \cdots Y_k$, 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 $x_1 x_2 \cdots x_k$, 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 $x_1 x_2 x_3$, where $x_2 = a$. Note that in general, if Y_i is a terminal, then x_i must be that terminal.

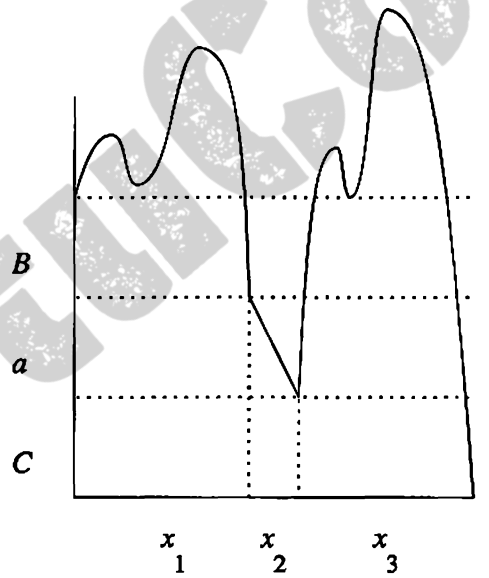

Figure 6.9: The PDA P consumes x and pops BaC from its stack

Formally, we can conclude that $(q, x_i x_{i+1} \cdots x_k, Y_i) \vdash^* (q, x_{i+1} \cdots x_k, \epsilon)$ 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 $Y_i \stackrel{*}{\Rightarrow} x_i$.

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

$Y_i \stackrel{*}{\Rightarrow} x_i$; this time, zero steps are used. Now we have the derivation

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

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

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

6.3.2 From PDA's to Grammars

Now, we complete the proofs of equivalence by showing that for every PDA P, we can find a CFG G whose language is the same language that P accepts by empty stack. The idea behind the proof is to recognize that the fundamental event in the history of a PDA's processing of a given input is the net popping of one symbol off the stack, while consuming some input. A PDA may change state as it pops stack symbols, so we should also note the state that it enters when it finally pops a level off its stack.

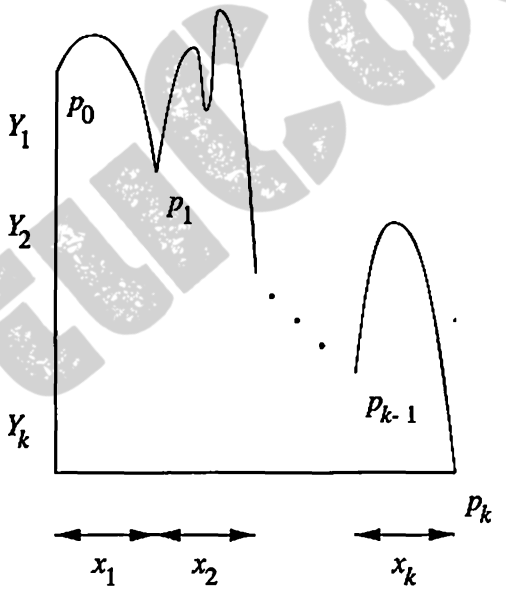

Figure 6.10: A PDA makes a sequence of moves that have the net effect of popping a symbol off the stack

Figure 6.10 suggests how we pop a sequence of symbols $Y_1, Y_2, \ldots Y_k$ off the stack. Some input x_1 is read while Y_1 is popped. We should emphasize that this "pop" is the net effect of (possibly) many moves. For example, the first move may change Y_1 to some other symbol Z. The next move may replace Z by UV, later moves have the effect of popping U, and then other moves pop V.

The net effect is that Y_1 has been replaced by nothing; i.e., it has been popped, and all the input symbols consumed so far constitute x_1.

We also show in Fig. 6.10 the net change of state. We suppose that the PDA starts out in state p_0, with Y_1 at the top of the stack. After all the moves whose net effect is to pop Y_1, the PDA is in state p_1. It then proceeds to (net) pop Y_2, while reading input string x_2 and winding up, perhaps after many moves, in state p_2 with Y_2 off the stack. The computation proceeds until each of the symbols on the stack is removed.

Our construction of an equivalent grammar uses variables each of which represents an "event" consisting of:

1. The net popping of some symbol X from the stack, and
2. A change in state from some p at the beginning to q when X has finally been replaced by ϵ on the stack.

We represent such a variable by the composite symbol $[pXq]$. Remember that this sequence of characters is our way of describing *one* variable; it is not five grammar symbols. The formal construction is given by the next theorem.

Theorem 6.14: Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ be a PDA. Then there is a context-free grammar G such that $L(G) = N(P)$.

PROOF: We shall construct $G = (V, \Sigma, R, S)$, where the set of variables V consists of:

1. The special symbol S, which is the start symbol, and
2. All symbols of the form $[pXq]$, where p and q are states in Q, and X is a stack symbol, in Γ.

The productions of G are as follows:

a) For all states p, G has the production $S \to [q_0 Z_0 p]$. Recall our intuition that a symbol like $[q_0 Z_0 p]$ is intended to generate all those strings w that cause P to pop Z_0 from its stack while going from state q_0 to state p. That is, $(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$. If so, then these productions say that start symbol S will generate all strings w that cause P to empty its stack, after starting in its initial ID.

b) Let $\delta(q, a, X)$ contain the pair $(r, Y_1 Y_2 \cdots Y_k)$, where:

   1. a is either a symbol in Σ or $a = \epsilon$.
   2. k can be any number, including 0, in which case the pair is (r, ϵ).

   Then for all lists of states $r_1, r_2, \ldots, r_k$, G has the production

$$[qXr_k] \to a[rY_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$$

This production says that one way to pop X and go from state q to state r_k is to read a (which may be ϵ), then use some input to pop Y_1 off the stack while going from state r to state r_1, then read some more input that pops Y_2 off the stack and goes from state r_1 to r_2, and so on.

We shall now prove that the informal interpretation of the variables $[qXp]$ is correct:

- $[qXp] \stackrel{*}{\Rightarrow} w$ if and only if $(q, w, X) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$.

(If) Suppose $(q, w, X) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$. We shall show $[qXp] \stackrel{*}{\Rightarrow} w$ by induction on the number of moves made by the PDA.

BASIS: One step. Then (p, ϵ) must be in $\delta(q, w, X)$, and w is either a single symbol or ϵ. By the construction of G, $[qXp] \rightarrow w$ is a production, so $[qXp] \Rightarrow w$.

INDUCTION: Suppose the sequence $(q, w, X) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$ takes n steps, and $n > 1$. The first move must look like

$$(q, w, X) \vdash (r_0, x, Y_1 Y_2 \cdots Y_k) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$$

where $w = ax$ for some a that is either ϵ or a symbol in Σ. It follows that the pair $(r_0, Y_1 Y_2 \cdots Y_k)$ must be in $\delta(q, a, X)$. Further, by the construction of G, there is a production $[qXr_k] \rightarrow a[r_0 Y_1 r_1][r_1 Y_2 r_2] \cdots [r_{k-1} Y_k r_k]$, where:

1. $r_k = p$, and
2. $r_1, r_2, \ldots, r_{k-1}$ are any states in Q.

In particular, we may observe, as was suggested in Fig. 6.10, that each of the symbols $Y_1, Y_2, \ldots, Y_k$ gets popped off the stack in turn, and we may choose p_i to be the state of the PDA when Y_i is popped, for $i = 1, 2, \ldots, k-1$. Let $x = w_1 w_2 \cdots w_k$, where w_i is the input consumed while Y_i is popped off the stack. Then we know that $(r_{i-1}, w_i, Y_i) \stackrel{*}{\vdash} (r_i, \epsilon, \epsilon)$.

As none of these sequences of moves can take as many as n moves, the inductive hypothesis applies to them. We conclude that $[r_{i-1} Y_i r_i] \stackrel{*}{\Rightarrow} w_i$. We may put these derivations together with the first production used to conclude:

$$\begin{aligned}
&[qXr_k] \Rightarrow a[r_0 Y_1 r_1][r_1 Y_1 r_2] \cdots [r_{k-1} Y_k r_k] \stackrel{*}{\Rightarrow} \\
&aw_1[r_1 Y_2 r_2][r_2 Y_3 r_3] \cdots [r_{k-1} Y_k r_k] \stackrel{*}{\Rightarrow} \\
&aw_1 w_2[r_2 Y_3 r_3] \cdots [r_{k-1} Y_k r_k] \stackrel{*}{\Rightarrow} \\
&\cdots \\
&aw_1 w_2 \cdots w_k = w
\end{aligned}$$

where $r_k = p$.

(Only-if) The proof is an induction on the number of steps in the derivation.

BASIS: One step. Then $[qXp] \to w$ must be a production. The only way for this production to exist is if there is a transition of P in which X is popped and state q becomes state p. That is, (p, ϵ) must be in $\delta(q, a, X)$, and $a = w$. But then $(q, w, X) \vdash (p, \epsilon, \epsilon)$.

INDUCTION: Suppose $[qXp] \stackrel{*}{\Rightarrow} w$ by n steps, where $n > 1$. Consider the first sentential form explicitly, which must look like

$$[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow} w$$

where $r_k = p$. This production must come from the fact that $(r_0, Y_1Y_2\cdots Y_k)$ is in $\delta(q, a, X)$.

We can break w into $w = aw_1w_2\cdots w_k$ such that $[r_{i-1}Y_ir_i] \stackrel{*}{\Rightarrow} w_i$ for all $i = 1, 2, \ldots, k$. By the inductive hypothesis, we know that for all i,

$$(r_{i-1}, w_i, Y_i) \stackrel{*}{\vdash} (r_i, \epsilon, \epsilon)$$

If we use Theorem 6.5 to put the correct strings beyond w_i on the input and below Y_i on the stack, we also know that

$$(r_{i-1}, w_iw_{i+1}\cdots w_k, Y_iY_{i+1}\cdots Y_k) \stackrel{*}{\vdash} (r_i, w_{i+1}\cdots w_k, Y_{i+1}\cdots Y_k)$$

If we put all these sequences together, we see that

$$(q, aw_1w_2\cdots w_k, X) \vdash (r_0, w_1w_2\cdots w_k, Y_1Y_2\cdots Y_k) \stackrel{*}{\vdash}$$
$$(r_1, w_2w_3\cdots w_k, Y_2Y_3\cdots Y_k) \stackrel{*}{\vdash} (r_2, w_3\cdots w_k, Y_3\cdots Y_k) \stackrel{*}{\vdash} \cdots \stackrel{*}{\vdash} (r_k, \epsilon, \epsilon)$$

Since $r_k = p$, we have shown that $(q, w, X) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$.

We complete the proof as follows. $S \stackrel{*}{\Rightarrow} w$ if and only if $[q_0Z_0p] \stackrel{*}{\Rightarrow} w$ for some p, because of the way the rules for start symbol S are constructed. We just proved that $[q_0Z_0p] \stackrel{*}{\Rightarrow} w$ if and only if $(q, w, Z_0) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$, i.e., if and only if P accepts x by empty stack. Thus, $L(G) = N(P)$. □

Example 6.15: Let us convert the PDA $P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ from Example 6.10 to a grammar. Recall that P_N accepts all strings that violate, for the first time, the rule that every e (else) must correspond to some preceding i (if). Since P_N has only one state and one stack symbol, the construction is particularly simple. There are only two variables in the grammar G:

a) S, the start symbol, which is in every grammar constructed by the method of Theorem 6.14, and

b) $[qZq]$, the only triple that can be assembled from the states and stack symbols of P_N.

The productions of grammar G are as follows:

1. The only production for S is $S \to [qZq]$. However, if there were n states of the PDA, then there would be n productions of this type, since the last state could be any of the n states. The first state would have to be the start state, and the stack symbol would have to be the start symbol, as in our production above.

2. From the fact that $\delta_N(q, i, Z)$ contains (q, ZZ), we get the production $[qZq] \to i[qZq][qZq]$. Again, for this simple example, there is only one production. However, if there were n states, then this one rule would produce n^2 productions, since the middle two states of the body could be any one state p, and the last states of the head and body could also be any one state. That is, if p and r were any two states of the PDA, then production $[qZp] \to i[qZr][rZp]$ would be produced.

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

$$[qZq] \to e$$

   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:

$$\begin{aligned} S &\to A \\ A &\to iAA \mid e \end{aligned}$$

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

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

□

6.3.3 Exercises for Section 6.3

*** Exercise 6.3.1:** Convert the grammar

$$\begin{aligned} S &\to 0S1 \mid A \\ A &\to 1A0 \mid S \mid \epsilon \end{aligned}$$

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

Exercise 6.3.2: Convert the grammar

$$\begin{aligned} S &\to aAA \\ A &\to aS \mid bS \mid a \end{aligned}$$

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.

Example 6.16: It turns out that the language L_{wwr} of Example 6.2 is a CFL that has no DPDA. However, by putting a "center-marker" c in the middle, we can make the language recognizable by a DPDA. That is, we can recognize the language $L_{wcwr} = \{wcw^R \mid w \text{ is in } (0+1)^*\}$ by a deterministic PDA.

The strategy of the DPDA is to store 0's and 1's on its stack, until it sees the center marker c. it then goes to another state, in which it matches input symbols against stack symbols and pops the stack if they match. If it ever finds a nonmatch, it dies; its input cannot be of the form wcw^R. If it succeeds in popping its stack down to the initial symbol, which marks the bottom of the stack, then it accepts its input.

The idea is very much like the PDA that we saw in Fig. 6.2. However, that PDA is nondeterministic, because in state q_0 it always has the choice of pushing the next input symbol onto the stack or making a transition on ϵ to state q_1; i.e., it has to guess when it has reached the middle. The DPDA for L_{wcwr} is shown as a transition diagram in Fig. 6.11.

This PDA is clearly deterministic. It never has a choice of move in the same state, using the same input and stack symbol. As for choices between using a real input symbol or ϵ, the only ϵ-transition it makes is from q_1 to q_2 with Z_0 at the top of the stack. However, in state q_1, there are no other moves when Z_0 is at the stack top. □

6.4.2 Regular Languages and Deterministic PDA's

The DPDA's accept a class of languages that is between the regular languages and the CFL's. We shall first prove that the DPDA languages include all the regular languages.

Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P.

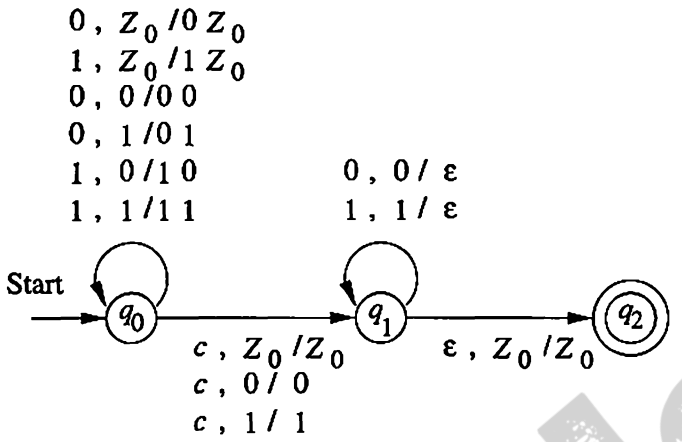

Figure 6.11: A deterministic PDA accepting L_{wcw^r}

PROOF: Essentially, a DPDA can simulate a deterministic finite automaton. The PDA keeps some stack symbol Z_0 on its stack, because a PDA has to have a stack, but really the PDA ignores its stack and just uses its state. Formally, let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA. Construct DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

by defining $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states p and q in Q, such that $\delta_A(q, a) = p$.

We claim that $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, Z_0)$ if and only if $\hat{\delta}_A(q_0, w) = p$. That is, P simulates A using its state. The proofs in both directions are easy inductions on $|w|$, and we leave them for the reader to complete. Since both A and P accept by entering one of the states of F, we conclude that their languages are the same. □

If we want the DPDA to accept by empty stack, then we find that our language-recognizing capability is rather limited. Say that a language L has the *prefix property* if there are no two different strings x and y in L such that x is a prefix of y.

Example 6.18: The language L_{wcw^r} of Example 6.16 has the prefix property. That is, it is not possible for there to be two strings wcw^R and xcx^R, one of which is a prefix of the other, unless they are the same string. To see why, suppose wcw^R is a prefix of xcx^R, and $w \neq x$. Then w must be shorter than x. Therefore, the c in wcw^R comes in a position where xcx^R has a 0 or 1; it is a position in the first x. That point contradicts the assumption that wcw^R is a prefix of xcx^R.

On the other hand, there are some very simple languages that do not have the prefix property. Consider $\{0\}^*$, i.e., the set of all strings of 0's. Clearly,

there are pairs of strings in this language one of which is a prefix of the other, so this language does not have the prefix property. In fact, of *any* two strings, one is a prefix of the other, although that condition is stronger than we need to establish that the prefix property does not hold. □

Note that the language $\{0\}^*$ is a regular language. Thus, it is not even true that every regular language is $N(P)$ for some DPDA P. We leave as an exercise the following relationship:

Theorem 6.19: A language L is $N(P)$ for some DPDA P if and only if L has the prefix property and L is $L(P')$ for some DPDA P'. □

6.4.3 DPDA's and Context-Free Languages

We have already seen that a DPDA can accept languages like L_{wcwr} that are not regular. To see this language is not regular, suppose it were, and use the pumping lemma. If n is the constant of the pumping lemma, then consider the string $w = 0^n c 0^n$, which is in L_{wcwr}. However, when we "pump" this string, it is the first group of 0's whose length must change, so we get in L_{wcwr} strings that have the "center" marker not in the center. Since these strings are not in L_{wcwr}, we have a contradiction and conclude that L_{wcwr} is not regular.

On the other hand, there are CFL's like L_{wwr} that cannot be $L(P)$ for any DPDA P. A formal proof is complex, but the intuition is transparent. If P is a DPDA accepting L_{wwr}, then given a sequence of 0's, it must store them on the stack, or do something equivalent to count an arbitrary number of 0's. For instance, it could store one X for every two 0's it sees, and use the state to remember whether the number was even or odd.

Suppose P has seen n 0's and then sees 110^n. It must verify that there were n 0's after the 11, and to do so it must pop its stack.[5] Now, P has seen 0^n110^n. If it sees an identical string next, it must accept, because the complete input is of the form ww^R, with $w = 0^n110^n$. However, if it sees 0^m110^m for some $m \neq n$, P must *not* accept. Since its stack is empty, it cannot remember what arbitrary integer n was, and must fail to recognize L_{wwr} correctly. Our conclusion is that:

- The languages accepted by DPDA's by final state properly include the regular languages, but are properly included in the CFL's.

6.4.4 DPDA's and Ambiguous Grammars

We can refine the power of the DPDA's by noting that the languages they accept all have unambiguous grammars. Unfortunately, the DPDA languages are not

[5]This statement is the intuitive part that requires a (hard) formal proof; could there be some other way for P to compare equal blocks of 0's?

exactly equal to the subset of the CFL's that are not inherently ambiguous. For instance, L_{wwr} has an unambiguous grammar

$$S \to 0S0 \mid 1S1 \mid \epsilon$$

even though it is not a DPDA language. The following theorems refine the bullet point above.

Theorem 6.20: If $L = N(P)$ for some DPDA P, then L has an unambiguous context-free grammar.

PROOF: We claim that the construction of Theorem 6.14 yields an unambiguous CFG G when the PDA to which it is applied is deterministic. First recall from Theorem 5.29 that it is sufficient to show that the grammar has unique leftmost derivations in order to prove that G is unambiguous.

Suppose P accepts string w by empty stack. Then it does so by a unique sequence of moves, because it is deterministic, and cannot move once its stack is empty. Knowing this sequence of moves, we can determine the one choice of production in a leftmost derivation whereby G derives w. There can never be a choice of which rule of P motivated the production to use. However, a rule of P, say $\delta(q, a, X) = \{(r, Y_1Y_2 \cdots Y_k)\}$ might cause many productions of G, with different states in the positions that reflect the states of P after popping each of $Y_1, Y_2, \ldots, Y_{k-1}$. Because P is deterministic, only one of these sequences of choices will be consistent with what P actually does, and therefore, only one of these productions will actually lead to derivation of w. □

However, we can prove more: even those languages that DPDA's accept by final state have unambiguous grammars. Since we only know how to construct grammars directly from PDA's that accept by empty stack, we need to change the language involved to have the prefix property, and then modify the resulting grammar to generate the original language. We do so by use of an "endmarker" symbol.

Theorem 6.21: If $L = L(P)$ for some DPDA P, then L has an unambiguous CFG.

PROOF: let \$ be an "endmarker" symbol that does not appear in the strings of L, and let $L' = L\$$. That is, the strings of L' are the strings of L, each followed by the symbol \$. Then L' surely has the prefix property, and by Theorem 6.19, $L' = N(P')$ for some DPDA P'.[6] By Theorem 6.20, there is an unambiguous grammar G' generating the language $N(P')$, which is L'.

Now, construct from G' a grammar G such that $L(G) = L$. To do so, we have only to get rid of the endmarker \$ from strings. Thus, treat \$ as a variable

[6]The proof of Theorem 6.19 appears in Exercise 6.4.3, but we can easily see how to construct P' from P. Add a new state q that P' enters whenever P is in an accepting state and the next input is \$. In state q, P' pops all symbols off its stack. Also, P' needs its own bottom-of-stack marker to avoid accidentally emptying its stack as it simulates P.

of G, and introduce production $\$ \to \epsilon$; otherwise, the productions of G' and G are the same. Since $L(G') = L'$, it follows that $L(G) = L$.

We claim that G is unambiguous. In proof, the leftmost derivations in G are exactly the same as the leftmost derivations in G', except that the derivations in G have a final step in which $\$$ is replaced by ϵ. Thus, if a terminal w string had two leftmost derivations in G, then $w\$$ would have two leftmost derivations in G'. Since we know G' is unambiguous, so is G. □

6.4.5 Exercises for Section 6.4

Exercise 6.4.1: For each of the following PDA's, tell whether or not it is deterministic. Either show that it meets the definition of a DPDA or find a rule or rules that violate it.

a) The PDA of Example 6.2.

* b) The PDA of Exercise 6.1.1.

c) The PDA of Exercise 6.3.3.

Exercise 6.4.2: Give deterministic pushdown automata to accept the following languages:

a) $\{0^n1^m \mid n \leq m\}$.

b) $\{0^n1^m \mid n \geq m\}$.

c) $\{0^n1^m0^n \mid n \text{ and } m \text{ are arbitrary}\}$.

Exercise 6.4.3: We can prove Theorem 6.19 in three parts:

* a) Show that if $L = N(P)$ for some DPDA P, then L has the prefix property.

! b) Show that if $L = N(P)$ for some DPDA P, then there exists a DPDA P' such that $L = L(P')$.

*! c) Show that if L has the prefix property and is $L(P')$ for some DPDA P', then there exists a DPDA P such that $L = N(P)$.

!! **Exercise 6.4.4:** Show that the language

$$L = \{0^n1^n \mid n \geq 1\} \cup \{0^n1^{2n} \mid n \geq 1\}$$

is a context-free language that is not accepted by any DPDA. *Hint*: Show that there must be two strings of the form 0^n1^n for different values of n, say n_1 and n_2 that cause a hypothetical DPDA for L to enter the same ID after reading both strings. Intuitively, the DPDA must erase from its stack almost everything it placed there on reading the 0's, in order to check that it has seen the same number of 1's. Thus, the DPDA cannot tell whether or not to accept next after seeing n_1 1's or after seeing n_2 1's.