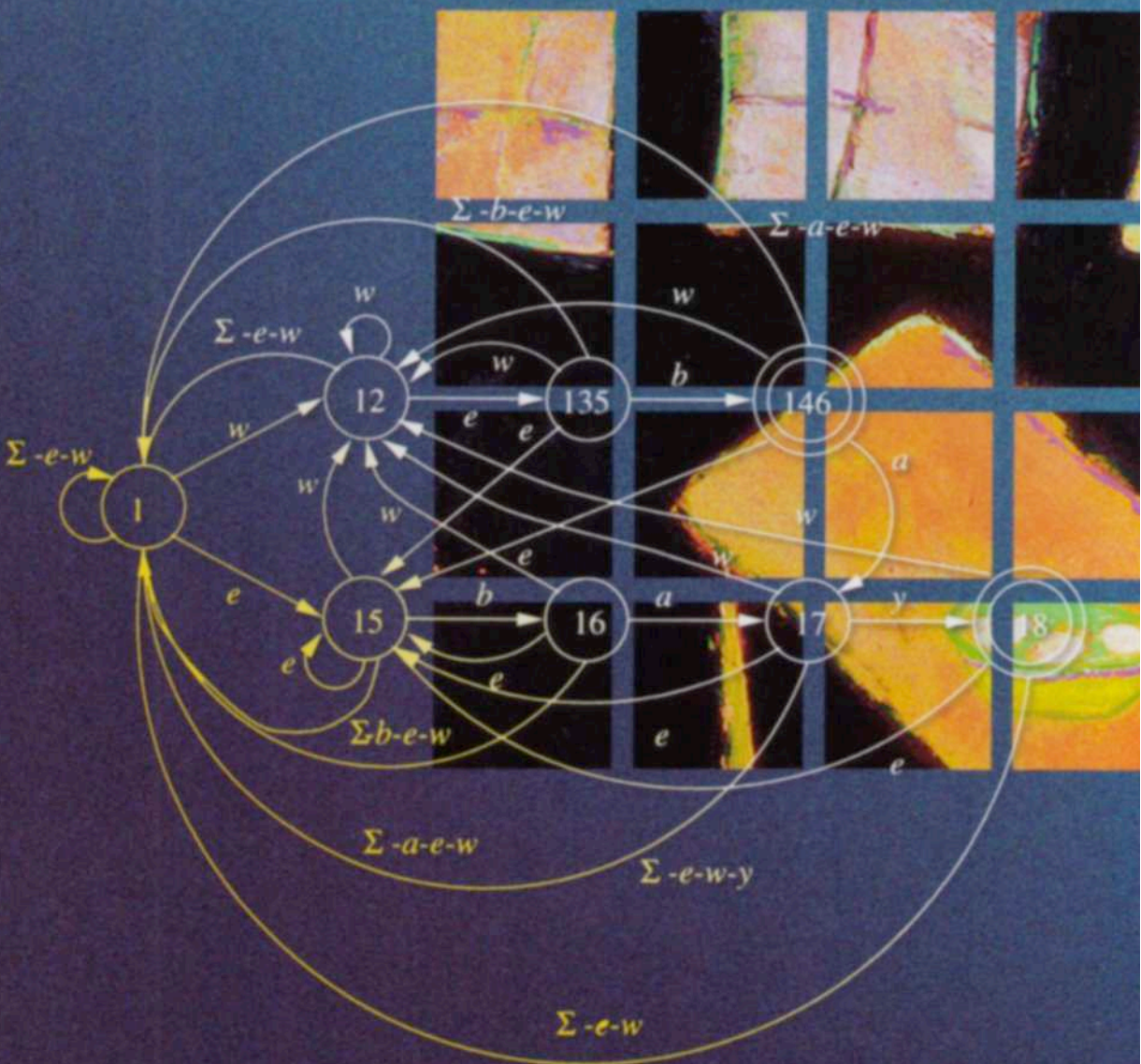
JOHN E. HOPCROFT
RAJEEV MOTWANI
JEFFREY D. ULLMAN

Introduction to Automata Theory, Languages, and Computation



SECOND EDITION

Chapter 5

Context-Free Grammars and Languages

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 $w = 0^n 1 0^n$. 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

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

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 $S \stackrel{*}{\underset{lm}{\Rightarrow}} \alpha$, then α is a *left-sentential form*, and if $S \stackrel{*}{\underset{rm}{\Rightarrow}} \alpha$, 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 $\mathbf{0^*1(0+1)^*}$:

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

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. □

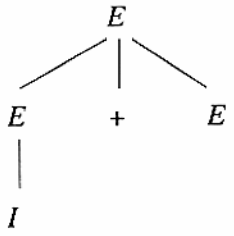


Figure 5.4: A parse tree showing the derivation of $I + E$ from E

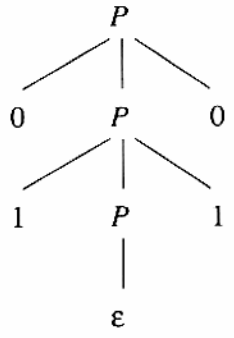


Figure 5.5: A parse tree showing the derivation $P \stackrel{*}{\Rightarrow} 0110$

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. □

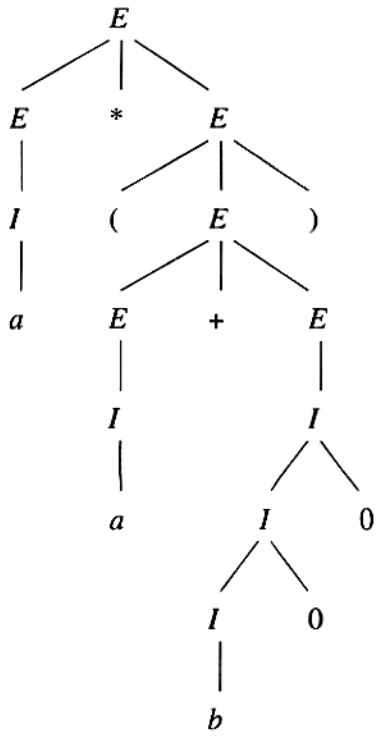


Figure 5.6: Parse tree showing $a * (a + b00)$ is in the language of our expression grammar

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.

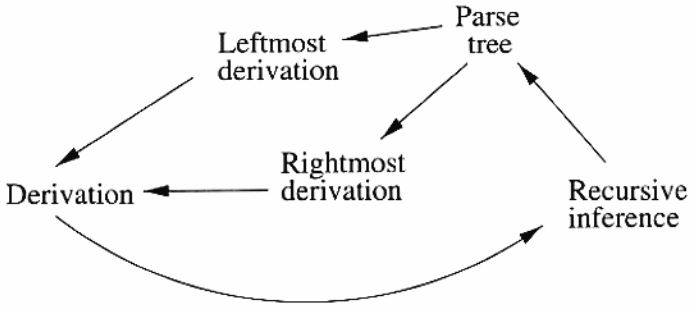


Figure 5.7: Proving the equivalence of certain statements about grammars

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.

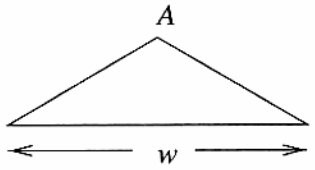


Figure 5.8: Tree constructed in the basis case of Theorem 5.12

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.

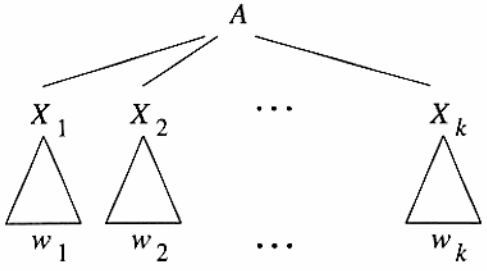


Figure 5.9: Tree used in the inductive part of the proof of Theorem 5.12

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.

5.3 Applications of Context-Free Grammars

Context-free grammars were originally conceived by N. Chomsky as a way to describe natural languages. That promise has not been fulfilled. However, as uses for recursively defined concepts in Computer Science have multiplied, so has the need for CFG's as a way to describe instances of these concepts. We shall sketch two of these uses, one old and one new.

1. Grammars are used to describe programming languages. More importantly, there is a mechanical way of turning the language description as a CFG into a parser, the component of the compiler that discovers the structure of the source program and represents that structure by a parse tree. This application is one of the earliest uses of CFG's; in fact it is one of the first ways in which theoretical ideas in Computer Science found their way into practice.

2. The development of XML (Extensible Markup Language) is widely predicted to facilitate electronic commerce by allowing participants to share conventions regarding the format of orders, product descriptions, and many other kinds of documents. An essential part of XML is the *Document Type Definition* (DTD), which is essentially a context-free grammar that describes the allowable tags and the ways in which these tags may be nested. Tags are the familiar keywords with triangular brackets that you may know from HTML, e.g., `<EM>` and `</EM>` to surround text that needs to be emphasized. However, XML tags deal not with the formatting of text, but with the meaning of text. For instance, one could surround a sequence of characters that was intended to be interpreted as a phone number by `<PHONE>` and `</PHONE>`.

5.3.1 Parsers

Many aspects of a programming language have a structure that may be described by regular expressions. For instance, we discussed in Example 3.9 how identifiers could be represented by regular expressions. However, there are also some very important aspects of typical programming languages that cannot be represented by regular expressions alone. The following are two examples.

Example 5.19: Typical languages use parentheses and/or brackets in a nested and balanced fashion. That is, we must be able to match some left parenthesis against a right parenthesis that appears immediately to its right, remove both of them, and repeat. If we eventually eliminate all the parentheses, then the string was balanced, and if we cannot match parentheses in this way, then it is unbalanced. Examples of strings of balanced parentheses are (()), ()(), (()()), and ϵ, while)(and (() are not.

A grammar $G_{bal} = (\{B\}, \{(,)\}, P, B)$ generates all and only the strings of balanced parentheses, where P consists of the productions:

$$B \to BB \mid (B) \mid \epsilon$$

The first production, $B \to BB$, says that the concatenation of two strings of balanced parentheses is balanced. That assertion makes sense, because we can match the parentheses in the two strings independently. The second production, $B \to (B)$, says that if we place a pair of parentheses around a balanced string, then the result is balanced. Again, this rule makes sense, because if we match

the parentheses in the inner string, then they are all eliminated and we are then allowed to match the first and last parentheses, which have become adjacent. The third production, $B \to \epsilon$ is the basis; it says that the empty string is balanced.

The above informal arguments should convince us that G_{bal} generates all strings of balanced parentheses. We need a proof of the converse — that every string of balanced parentheses is generated by this grammar. However, a proof by induction on the length of the balanced string is not hard and is left as an exercise.

We mentioned that the set of strings of balanced parentheses is not a regular language, and we shall now prove that fact. If $L(G_{bal})$ were regular, then there would be a constant n for this language from the pumping lemma for regular languages. Consider the balanced string $w = (^n)^n$, that is, n left parentheses followed by n matching right parentheses. If we break $w = xyz$ according to the pumping lemma, then y consists of only left parentheses, and therefore xz has more right parentheses than left. This string is not balanced, contradicting the assumption that the language of balanced parentheses is regular. □

Programming languages consist of more than parentheses, of course, but parentheses are an essential part of arithmetic or conditional expressions. The grammar of Fig. 5.2 is more typical of the structure of arithmetic expressions, although we used only two operators, plus and times, and we included the detailed structure of identifiers, which would more likely be handled by the lexical-analyzer portion of the compiler, as we mentioned in Section 3.3.2. However, the language described in Fig. 5.2 is not regular either. For instance, according to this grammar, $(^n a)^n$ is a legal expression. We can use the pumping lemma to show that if the language were regular, then a string with some of the left parentheses removed and the a and all right parentheses intact would also be a legal expression, which it is not.

There are numerous aspects of a typical programming language that behave like balanced parentheses. There will usually be parentheses themselves, in expressions of all types. Beginnings and endings of code blocks, such as **begin** and **end** in Pascal, or the curly braces $\{\ldots\}$ of C, are examples. That is, whatever curly braces appear in a C program must form a balanced sequence, with { in place of the left parenthesis and } in place of the right parenthesis.

There is a related pattern that appears occasionally, where "parentheses" can be balanced with the exception that there can be unbalanced left parentheses. An example is the treatment of **if** and **else** in C. An if-clause can appear unbalanced by any else-clause, or it may be balanced by a matching else-clause. A grammar that generates the possible sequences of **if** and **else** (represented by i and e, respectively) is:

$$S \to \epsilon \mid SS \mid iS \mid iSeS$$

For instance, $ieie$, iie, and iei are possible sequences of **if**'s and **else**'s, and each of these strings is generated by the above grammar. Some examples of illegal sequences, not generated by the grammar, are ei and $ieeii$.

A simple test (whose correctness we leave as an exercise), for whether a sequence of i's and e's is generated by the grammar is to consider each e, in turn from the left. Look for the first i to the left of the e being considered. If there is none, the string fails the test and is not in the language. If there is such an i, delete this i and the e being considered. Then, if there are no more e's the string passes the test and is in the language. If there are more e's, proceed to consider the next one.

Example 5.20: Consider the string iee. The first e is matched with the i to its left. They are removed, leaving the string e. Since there are more e's we consider the next. However, there is no i to its left, so the test fails; iee is not in the language. Note that this conclusion is valid, since you cannot have more **else**'s than **if**'s in a C program.

For another example, consider $iieie$. Matching the first e with the i to its left leaves iie. Matching the remaining e with the i to its left leaves i. Now there are no more e's, so the test succeeds. This conclusion also makes sense, because the sequence $iieie$ corresponds to a C program whose structure is like that of Fig. 5.10. In fact, the matching algorithm also tells us (and the C compiler) which **if** matches any given **else**. That knowledge is essential if the compiler is to create the control-flow logic intended by the programmer. □

```
if (Condition) {
    ...
    if (Condition) Statement;
    else Statement;
    ...
    if (Condition) Statement;
    else Statement;
    ...
}
```

Figure 5.10: An if-else structure; the two **else**'s match their previous **if**'s, and the first **if** is unmatched

5.3.2 The YACC Parser-Generator

The generation of a parser (function that creates parse trees from source programs) has been institutionalized in the YACC command that appears in all UNIX systems. The input to YACC is a CFG, in a notation that differs only in details from the one we have used here. Associated with each production is an *action*, which is a fragment of C code that is performed whenever a node of the parse tree that (with its children) corresponds to this production is created. Typically, the action is code to construct that node, although in some YACC

applications the tree is not actually constructed, and the action does something else, such as emit a piece of object code.

Example 5.21: In Fig. 5.11 is a sample of a CFG in the YACC notation. The grammar is the same as that of Fig. 5.2. We have elided the actions, just showing their (required) curly braces and their position in the YACC input.

```
Exp : Id            {...}
    | Exp '+' Exp   {...}
    | Exp '*' Exp   {...}
    | '(' Exp ')'   {...}
    ;
Id  : 'a'           {...}
    | 'b'           {...}
    | Id 'a'        {...}
    | Id 'b'        {...}
    | Id '0'        {...}
    | Id '1'        {...}
    ;
```

Figure 5.11: An example of a grammar in the YACC notation

Notice the following correspondences between the YACC notation for grammars and ours:

- The colon is used as the production symbol, our $\rightarrow$.
- All the productions with a given head are grouped together, and their bodies are separated by the vertical bar. We also allow this convention, as an option.
- The list of bodies for a given head ends with a semicolon. We have not used a terminating symbol.
- Terminals are quoted with single quotes. Several characters can appear within a single pair of quotes. Although we have not shown it, YACC allows its user to define symbolic terminals as well. The occurrence of these terminals in the source program are detected by the lexical analyzer and signaled, through the return-value of the lexical analyzer, to the parser.
- Unquoted strings of letters and digits are variable names. We have taken advantage of this capability to give our two variables more descriptive names — `Exp` and `Id` — although E and I could have been used.

□

5.3.3 Markup Languages

We shall next consider a family of "languages" called *markup* languages. The "strings" in these languages are documents with certain marks (called *tags*) in them. Tags tell us something about the semantics of various strings within the document.

The markup language with which you are probably most familiar is HTML (HyperText Markup Language). This language has two major functions: creating links between documents and describing the format ("look") of the a document. We shall offer only a simplified view of the structure of HTML, but the following examples should suggest both its structure and how a CFG could be used both to describe the legal HTML documents and to guide the processing (i.e., the display on a monitor or printer) of a document.

Example 5.22: Figure 5.12(a) shows a piece of text, comprising a list of items, and Fig. 5.12(b) shows its expression in HTML. Notice from Fig. 5.12(b) that HTML consists of ordinary text interspersed with tags. Matching tags are of the form <x> and </x> for some string x.[3] For instance, we see the matching tags `<EM>` and `</EM>`, which indicate that the text between them should be emphasized, that is, put in italics or another appropriate font. We also see the matching tags `<OL>` and `</OL>`, indicating an ordered list, i.e., an enumeration of list items.

The things I *hate*:

1. Moldy bread.
2. People who drive too slow in the fast lane.

(a) The text as viewed

```
<P>The things I <EM>hate</EM>:
<OL>
<LI>Moldy bread.
<LI>People who drive too slow
in the fast lane.
</OL>
```

(b) The HTML source

Figure 5.12: An HTML document and its printed version

We also see two examples of unmatched tags: `<P>` and `<LI>`, which introduce paragraphs and list items, respectively. HTML allows, indeed encourages, that

[3] Sometimes the introducing tag <x> has more information in it than just the name x for the tag. However, we shall not consider that possibility in examples.

these tags be matched by </P> and </LI> at the ends of paragraphs and list items, but it does not require the matching. We have therefore left the matching tags off, to provide some complexity to the sample HTML grammar we shall develop. □

There are a number of classes of strings that are associated with an HTML document. We shall not try to list them all, but here are the ones essential to the understanding of text like that of Example 5.22. For each class, we shall introduce a variable with a descriptive name.

1. *Text* is any string of characters that can be literally interpreted; i.e., it has no tags. An example of a *Text* element in Fig 5.12(a) is "Moldy bread."
2. *Char* is any string consisting of a single character that is legal in HTML text. Note that blanks are included as characters.
3. *Doc* represents documents, which are sequences of "elements." We define elements next, and that definition is mutually recursive with the definition of a *Doc*.
4. *Element* is either a *Text* string, or a pair of matching tags and the document between them, or an unmatched tag followed by a document.
5. *ListItem* is the <LI> tag followed by a document, which is a single list item.
6. *List* is a sequence of zero or more list items.

1. *Char* → *a* | *A* | ···
2. *Text* → ϵ | *Char Text*
3. *Doc* → ϵ | *Element Doc*
4. *Element* → *Text* |
   <EM> *Doc* </EM> |
   <P> *Doc* |
   <OL> *List* </OL> | ···
5. *ListItem* → <LI> *Doc*
6. *List* → ϵ | *ListItem List*

Figure 5.13: Part of an HTML grammar

Figure 5.13 is a CFG that describes as much of the structure of the HTML language as we have covered. In line (1) it is suggested that a character can be "a" or "A" or many other possible characters that are part of the HTML character set. Line (2) says, using two productions, that *Text* can be either the empty string, or any legal character followed by more text. Put another way, *Text* is zero or more characters. Note that < and > are not legal characters, although they can be represented by the sequences `<` and `>`, respectively. *Thus, we cannot accidentally get a tag into Text.*

Line (3) says that a document is a sequence of zero or more "elements." An element in turn, we learn at line (4), is either text, an emphasized document, a paragraph-beginning followed by a document, or a list. We have also suggested that there are other productions for *Element*, corresponding to the other kinds of tags that appear in HTML. Then, in line (5) we find that a list item is the `<LI>` tag followed by any document, and line (6) tells us that a list is a sequence of zero or more list elements.

Some aspects of HTML do not require the power of context-free grammars; regular expressions are adequate. For example, lines (1) and (2) of Fig. 5.13 simply say that *Text* represents the same language as does the regular expression $(\mathtt{a} + \mathtt{A} + \cdots)^*$. However, some aspects of HTML *do* require the power of CFG's. For instance, each pair of tags that are a corresponding beginning and ending pair, e.g., `<EM>` and `</EM>`, are like balanced parentheses, which we already know are not regular.

5.3.4 XML and Document-Type Definitions

The fact that HTML is described by a grammar is not in itself remarkable. Essentially all programming languages can be described by their own CFG's, so it would be more surprising if we could *not* so describe HTML. However, when we look at another important markup language, XML (eXtensible Markup Language), we find that the CFG's play a more vital role, as part of the process of using that language.

The purpose of XML is not to describe the formatting of the document; that is the job for HTML. Rather, XML tries to describe the "semantics" of the text. For example, text like "12 Maple St." looks like an address, but is it? In XML, tags would surround a phrase that represented an address; for example:

```
<ADDR>12 Maple St.</ADDR>
```

However, it is not immediately obvious that `<ADDR>` means the address of a building. For instance, if the document were about memory allocation, we might expect that the `<ADDR>` tag would refer to a memory address. To make clear what the different kinds of tags are, and what structures may appear between matching pairs of these tags, people with a common interest are expected to develop standards in the form of a DTD (Document-Type Definition).

A DTD is essentially a context-free grammar, with its own notation for describing the variables and productions. In the next example, we shall show a simple DTD and introduce some of the language used for describing DTD's. The DTD language itself has a context-free grammar, but it is not that grammar we are interested in describing. Rather, the language for describing DTD's is essentially a CFG notation, and we want to see how CFG's are expressed in this language.

The form of a DTD is

```
<!DOCTYPE name-of-DTD [
    list of element definitions
]>
```

An element definition, in turn, has the form

```
<!ELEMENT element-name (description of the element)>
```

Element descriptions are essentially regular expressions. The basis of these expressions are:

1. Other element names, representing the fact that elements of one type can appear within elements of another type, just as in HTML we might find emphasized text within a list.

2. The special term `#PCDATA`, standing for any text that does not involve XML tags. This term plays the role of variable *Text* in Example 5.22.

The allowed operators are:

1. | standing for union, as in the UNIX regular-expression notation discussed in Section 3.3.1.

2. A comma, denoting concatenation.

3. Three variants of the closure operator, as in Section 3.3.1. These are $*$, the usual operator meaning "zero or more occurrences of," $+$, meaning "one or more occurrences of," and ?, meaning "zero or one occurrence of."

Parentheses may group operators to their arguments; otherwise, the usual precedence of regular-expression operators applies.

Example 5.23: Let us imagine that computer vendors get together to create a standard DTD that they can use to publish, on the Web, descriptions of the various PC's that they currently sell. Each description of a PC will have a model number, and details about the features of the model, e.g., the amount of RAM, number and size of disks, and so on. Figure 5.14 shows a hypothetical, very simple, DTD for personal computers.

```
<!DOCTYPE PcSpecs [
    <!ELEMENT PCS (PC*)>
    <!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
    <!ELEMENT MODEL (\#PCDATA)>
    <!ELEMENT PRICE (\#PCDATA)>
    <!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
    <!ELEMENT MANF (\#PCDATA)>
    <!ELEMENT MODEL (\#PCDATA)>
    <!ELEMENT SPEED (\#PCDATA)>
    <!ELEMENT RAM (\#PCDATA)>
    <!ELEMENT DISK (HARDDISK | CD | DVD)>
    <!ELEMENT HARDDISK (MANF, MODEL, SIZE)
    <!ELEMENT SIZE (\#PCDATA)>
    <!ELEMENT CD (SPEED)>
    <!ELEMENT DVD (SPEED)>
]>
```

Figure 5.14: A DTD for personal computers

The name of the DTD is PcSpecs. The first element, which is like the start symbol of a CFG, is PCS (list of PC specifications). Its definition, PC*, says that a PCS is zero or more PC entries.

We then see the definition of a PC element. It consists of the concatenation of five things. The first four are other elements, corresponding to the model, price, processor type, and RAM of the PC. Each of these must appear once, in that order, since the comma represents concatenation. The last constituent, DISK+, tells us that there will be one or more disk entries for a PC.

Many of the constituents are simply text; MODEL, PRICE, and RAM are of this type. However, PROCESSOR has more structure. We see from its definition that it consists of a manufacturer, model, and speed, in that order; each of these elements is simple text.

A DISK entry is the most complex. First, a disk is either a hard disk, CD, or DVD, as indicated by the rule for element DISK, which is the OR of three other elements. Hard disks, in turn, have a structure in which the manufacturer, model, and size are specified, while CD's and DVD's are represented only by their speed.

Figure 5.15 is an example of an XML document that conforms to the DTD of Fig. 5.14. Notice that each element is represented in the document by a tag with the name of that element and a matching tag at the end, with an extra slash, just as in HTML. Thus, in Fig. 5.15 we see at the outermost level the tag <PCS>...</PCS>. Inside these tags appears a list of entries, one for each PC sold by this manufacturer; we have only shown one such entry explicitly.

Within the illustrated <PC> entry, we can easily see that the model number

```
<PCS>
    <PC>
        <MODEL>4560</MODEL>
        <PRICE>$2295</PRICE>
        <PROCESSOR>
            <MANF>Intel</MANF>
            <MODEL>Pentium</MODEL>
            <SPEED>800MHz</SPEED>
        </PROCESSOR>
        <RAM>256</RAM>
        <DISK><HARDDISK>
            <MANF>Maxtor</MANF>
            <MODEL>Diamond</MODEL>
            <SIZE>30.5Gb</SIZE>
        </HARDDISK></DISK>
        <DISK><CD>
            <SPEED>32x</SPEED>
        </CD></DISK>
    </PC>
    <PC>
        ...
    </PC>
</PCS>
```

Figure 5.15: Part of a document obeying the structure of the DTD in Fig. 5.14

is 4560, the price is $2295, and it has an 800MHz Intel Pentium processor. It has 256Mb of RAM, a 30.5Gb Maxtor Diamond hard disk, and a 32x CD-ROM reader. What is important is not that we can read these facts, but that a program could read the document, and guided by the grammar in the DTD of Fig. 5.14 that it has also read, could interpret the numbers and names in Fig. 5.15 properly. □

You may have noticed that the rules for the elements in DTD's like Fig. 5.14 are not quite like productions of context-free grammars. Many of the rules are of the correct form. For instance,

```
<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
```

Is analogous to the production

$$Processor \rightarrow Manf\ Model\ Speed$$

However, the rule

```
<!ELEMENT DISK (HARDDISK | CD | DVD)>
```

does not have a definition for DISK that is like a production body. In this case, the extension is simple: we may interpret this rule as three productions, with the vertical bar playing the same role as it does in our shorthand for productions having a common head. Thus, this rule is equivalent to the three productions

$$Disk \rightarrow HardDisk \mid Cd \mid Dvd$$

The most difficult case is

```
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

where the "body" has a closure operator within it. The solution is to replace DISK+ by a new variable, say *Disks*, that generates, via a pair of productions, one or more instances of the variable *Disk*. The equivalent productions are thus:

$$Pc \rightarrow Model\ Price\ Processor\ Ram\ Disks$$
$$Disks \rightarrow Disk \mid Disk\ Disks$$

There is a general technique for converting a CFG with regular expressions as production bodies to an ordinary CFG. We shall give the idea informally; you may wish to formalize both the meaning of CFG's with regular-expression productions and a proof that the extension yields no new languages beyond the CFL's. We show, inductively, how to convert a production with a regular-expression body to a collection of equivalent ordinary productions. The induction is on the size of the expression in the body.

BASIS: If the body is the concatenation of elements, then the production is already in the legal form for CFG's, so we do nothing.

INDUCTION: Otherwise, there are five cases, depending on the final operator used.

1. The production is of the form $A \rightarrow E_1, E_2$, where E_1 and E_2 are expressions permitted in the DTD language. This is the concatenation case. Introduce two new variables, B and C, that appear nowhere else in the grammar. Replace $A \rightarrow E_1, E_2$ by the productions

$$A \rightarrow BC$$
$$B \rightarrow E_1$$
$$C \rightarrow E_2$$

   The first production, $A \rightarrow BC$, is legal for CFG's. The last two may or may not be legal. However, their bodies are shorter than the body of the original production, so we may inductively convert them to CFG form.

2. The production is of the form $A \to E_1 \mid E_2$. For this union operator, replace this production by the pair of productions:

$$A \to E_1$$
$$A \to E_2$$

Again, these productions may or may not be legal CFG productions, but their bodies are shorter than the body of the original. We may therefore apply the rules recursively and eventually convert these new productions to CFG form.

3. The production is of the form $A \to (E_1)^*$. Introduce a new variable B that appears nowhere else, and replace this production by:

$$A \to BA$$
$$A \to \epsilon$$
$$B \to E_1$$

4. The production is of the form $A \to (E_1)^+$. Introduce a new variable B that appears nowhere else, and replace this production by:

$$A \to BA$$
$$A \to B$$
$$B \to E_1$$

5. The production is of the form $A \to (E_1)?$. Replace this production by:

$$A \to \epsilon$$
$$A \to E_1$$

Example 5.24: Let us consider how to convert the DTD rule

```
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

to legal CFG productions. First, we can view the body of this rule as the concatenation of two expressions, the first of which is `MODEL, PRICE, PROCESSOR, RAM` and the second of which is `DISK+`. If we create variables for these two subexpressions, say A and B, respectively, then we can use the productions:

$$Pc \to AB$$
$$A \to Model\ Price\ Processor\ Ram$$
$$B \to Disk^+$$

Only the last of these is not in legal form. We introduce another variable C and the productions:

$$B \to CB \mid C$$
$$C \to Disk$$

In this special case, because the expression that A derives is just a concatenation of variables, and $Disk$ is a single variable, we actually have no need for the variables A or C. We could use the following productions instead:

$$Pc \to Model\ Price\ Processor\ Ram\ B$$
$$B \to Disk\ B \mid Disk$$

□

5.3.5 Exercises for Section 5.3

Exercise 5.3.1: Prove that if a string of parentheses is balanced, in the sense given in Example 5.19, then it is generated by the grammar $B \to BB \mid (B) \mid \epsilon$. *Hint*: Perform an induction on the length of the string.

* **Exercise 5.3.2:** Consider the set of all strings of balanced parentheses of two types, round and square. An example of where these strings come from is as follows. If we take expressions in C, which use round parentheses for grouping and for arguments of function calls, and use square brackets for array indexes, and drop out everything but the parentheses, we get all strings of balanced parentheses of these two types. For example,

```
f(a[i]*(b[i][j],c[g(x)]),d[i])
```

becomes the balanced-parenthesis string `([](([][][()])[])`. Design a grammar for all and only the strings of round and square parentheses that are balanced.

! **Exercise 5.3.3:** In Section 5.3.1, we considered the grammar

$$S \to \epsilon \mid SS \mid iS \mid iSeS$$

and claimed that we could test for membership in its language L by repeatedly doing the following, starting with a string w. The string w changes during repetitions.

1. If the current string begins with e, fail; w is not in L.
2. If the string currently has no e's (it may have i's), succeed; w is in L.
3. Otherwise, delete the first e and the i immediately to its left. Then repeat these three steps on the new string.

Prove that this process correctly identifies the strings in L.

Exercise 5.3.4: Add the following forms to the HTML grammar of Fig. 5.13:

* a) A list item must be ended by a closing tag `</LI>`.

b) An element can be an unordered list, as well as an ordered list. Unordered lists are surrounded by the tag `<UL>` and its closing `</UL>`.

! c) An element can be a table. Tables are surrounded by `<TABLE>` and its closer `</TABLE>`. Inside these tags are one or more rows, each of which is surrounded by `<TR>` and `</TR>`. The first row is the header, with one or more fields, each introduced by the `<TH>` tag (we'll assume these are not closed, although they should be). Subsequent rows have their fields introduced by the `<TD>` tag.

```
<!DOCTYPE CourseSpecs [
    <!ELEMENT COURSES (COURSE+)>
    <!ELEMENT COURSE (CNAME, PROF, STUDENT*, TA?)>
    <!ELEMENT CNAME (#PCDATA)>
    <!ELEMENT PROF (#PCDATA)>
    <!ELEMENT STUDENT (#PCDATA)>
    <!ELEMENT TA (#PCDATA)> ]>
```

Figure 5.16: A DTD for courses

Exercise 5.3.5: Convert the DTD of Fig. 5.16 to a context-free grammar.

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.

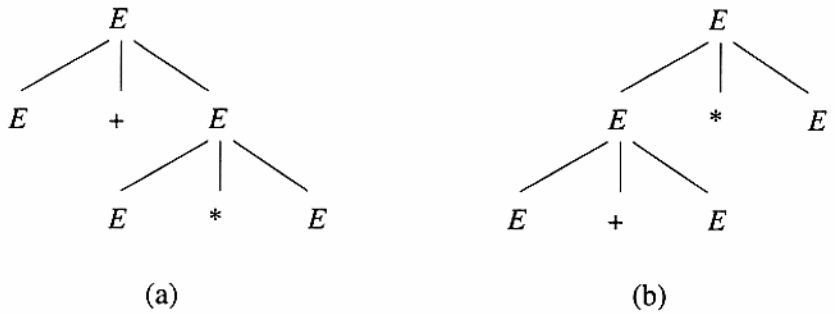


Figure 5.17: Two parse trees with the same yield

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.

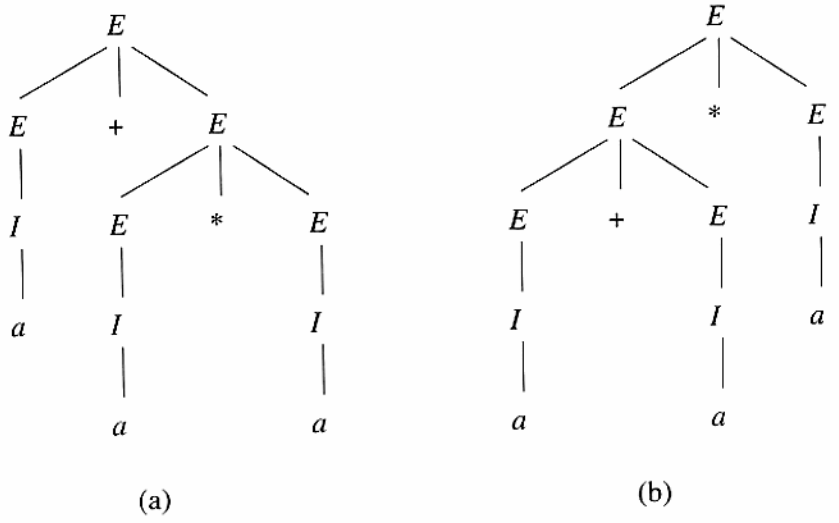


Figure 5.18: Trees with yield $a + a * a$, demonstrating the ambiguity of our expression grammar

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.

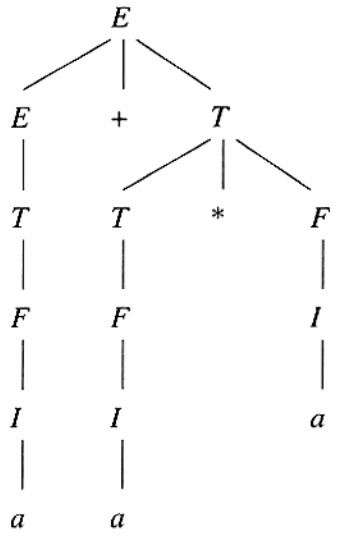


Figure 5.20: The sole parse tree for $a + a * a$

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.

□

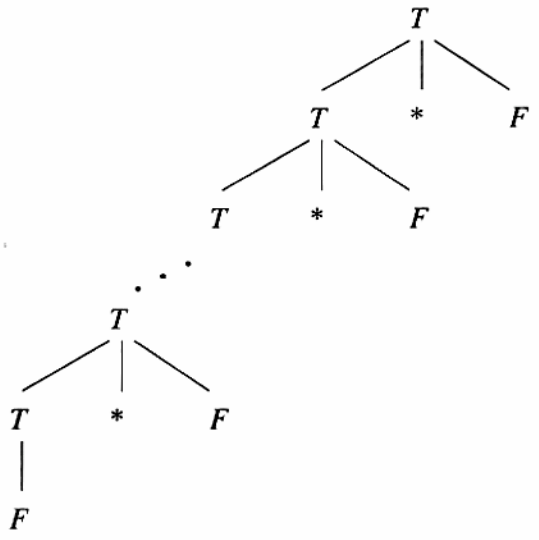


Figure 5.21: The form of all parse trees for a term

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

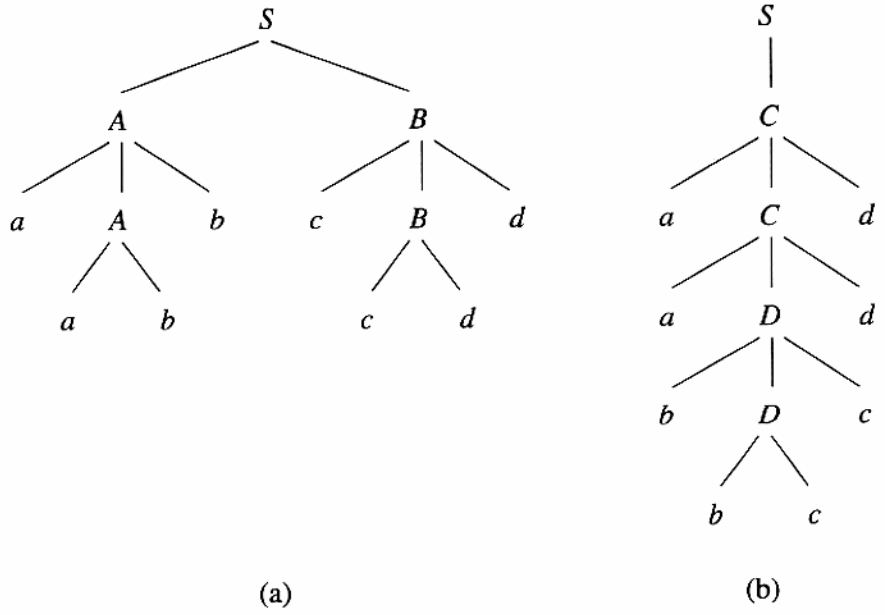


Figure 5.23: Two parse trees for *aabbccdd*

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:

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

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.

5.5 Summary of Chapter 5

- *Context-Free Grammars*: A CFG is a way of describing languages by recursive rules called productions. A CFG consists of a set of variables, a set of terminal symbols, and a start variable, as well as the productions. Each production consists of a head variable and a body consisting of a string of zero or more variables and/or terminals.

- *Derivations and Languages*: Beginning with the start symbol, we derive terminal strings by repeatedly replacing a variable by the body of some production with that variable in the head. The language of the CFG is the set of terminal strings we can so derive; it is called a context-free language.

- *Leftmost and Rightmost Derivations*: If we always replace the leftmost (resp. rightmost) variable in a string, then the resulting derivation is a leftmost (resp. rightmost) derivation. Every string in the language of a CFG has at least one leftmost and at least one rightmost derivation.

- *Sentential Forms*: Any step in a derivation is a string of variables and/or terminals. We call such a string a sentential form. If the derivation is leftmost (resp. rightmost), then the string is a left- (resp. right-) sentential form.

- *Parse Trees*: A parse tree is a tree that shows the essentials of a derivation. Interion nodes are labeled by variables, and leaves are labeled by terminals or ϵ. For each internal node, there must be a production such that the head of the production is the label of the node, and the labels of its children, read from left to right, form the body of that production.
- *Equivalence of Parse Trees and Derivations*: A terminal string is in the language of a grammar if and only if it is the yield of at least one parse tree. Thus, the existence of leftmost derivations, rightmost derivations, and parse trees are equivalent conditions that each define exactly the strings in the language of a CFG.
- *Ambiguous Grammars*: For some CFG's, it is possible to find a terminal string with more than one parse tree, or equivalently, more than one leftmost derivation or more than one rightmost derivation. Such a grammar is called ambiguous.
- *Eliminating Ambiguity*: For many useful grammars, such as those that describe the structure of programs in a typical programming language, it is possible to find an unambiguous grammar that generates the same language. Unfortunately, the unambiguous grammar is frequently more complex than the simplest ambiguous grammar for the language. There are also some context-free languages, usually quite contrived, that are inherently ambiguous, meaning that every grammar for that language is ambiguous.
- *Parsers*: The context-free grammar is an essential concept for the implementation of compilers and other programming-language processors. Tools such as YACC take a CFG as input and produce a parser, the component of a compiler that deduces the structure of the program being compiled.
- *Document Type Definitions*: The emerging XML standard for sharing information through Web documents has a notation, called the DTD, for describing the structure of such documents, through the nesting of semantic tags within the document. The DTD is in essence a context-free grammar whose language is a class of related documents.

5.6 References for Chapter 5

The context-free grammar was first proposed as a description method for natural languages by Chomsky [4]. A similar idea was used shortly thereafter to describe computer languages — Fortran by Backus [2] and Algol by Naur [7]. As a result, CFG's are sometimes referred to as "Backus-Naur form grammars."

Ambiguity in grammars was identified as a problem by Cantor [3] and Floyd [5] at about the same time. Inherent ambiguity was first addressed by Gross [6].

For applications of CFG's in compilers, see [1]. DTD's are defined in the standards document for XML [8].

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

2. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proc. Intl. Conf. on Information Processing* (1959), UNESCO, pp. 125–132.

3. D. C. Cantor, "On the ambiguity problem of Backus systems," *J. ACM* **9**:4 (1962), pp. 477–479.

4. N. Chomsky, "Three models for the description of language," *IRE Trans. on Information Theory* **2**:3 (1956), pp. 113–124.

5. R. W. Floyd, "On ambiguity in phrase-structure languages," *Comm. ACM* **5**:10 (1962), pp. 526–534.

6. M. Gross, "Inherent ambiguity of minimal linear grammars," *Information and Control* **7**:3 (1964), pp. 366–368.

7. P. Naur et al., "Report on the algorithmic language ALGOL 60," *Comm. ACM* **3**:5 (1960), pp. 299–314. See also *Comm. ACM* **6**:1 (1963), pp. 1–17.

8. World-Wide-Web Consortium, `http://www.w3.org/TR/REC-xml` (1998).