CHAPTER

6

Context-Free Grammars

6.1 | EXAMPLES AND DEFINITIONS

Many of the languages we have considered, both regular and nonregular, can be described by recursive definitions. In our first example, involving a very simple regular language, a slight reformulation of the recursive definition leads us to the idea of a *context-free grammar*. These and other more general grammars that we will later study turn out to be powerful tools for describing and analyzing languages.

Using Grammar Rules to Describe a Language — EXAMPLE 6.1

Let us consider the language $L = \{a, b\}^*$ of all strings over the alphabet $\{a, b\}$. In Example 2.15 we considered a recursive definition of L equivalent to the following:

1. $\Lambda \in L$.
2. For any $S \in L$, $Sa \in L$.
3. For any $S \in L$, $Sb \in L$.
4. No other strings are in L.

We think of S here as a *variable*, representing an arbitrary element of L whose value is to be obtained by some combination of rules 1 to 3. Rule 1, which we write $S \to \Lambda$, indicates that one way of giving S a value is to replace it by Λ. Rules 2 and 3 can be written $S \to Sa$ and $S \to Sb$. This means that S can also be replaced by Sa or Sb; in either case we must then obtain the final value by continuing to use the rules to give a value to the new S.

The symbol $\to$ is used for each of the rules by which a variable is replaced by a string. For two strings α and β, we will use the notation $\alpha \Rightarrow \beta$ to mean that β can be obtained by applying one of these rules to a single variable in the string α. Using this notation in our example, we can write

$$S \Rightarrow Sa \Rightarrow Sba \Rightarrow Sbba \Rightarrow \Lambda bba = bba$$

to describe the sequence of steps (the application of rules 2, 3, 3, and 1) used to obtain, or

derive, the string *bba*. The derivation comes to an end at the point where we replace S by an actual string of alphabet symbols (in this case Λ); each step before that is roughly analogous to a recursive call, since we are replacing S by a string that still contains a variable.

We can simplify the notation even further by introducing the symbol | to mean "or" and writing the first three rules as

$$S \rightarrow \Lambda \mid Sa \mid Sb$$

(In our new notation, we dispense with writing rule 4, even though it is implicitly still in effect.) We note for future reference that in an expression such as $Sa \mid Sb$, the two alternatives are Sa and Sb, not a and S—in other words, the concatenation operation takes precedence over the | operation.

In Example 2.15 we also considered this alternative definition of L:

1. $\Lambda \in L$.
2. $a \in L$.
3. $b \in L$.
4. For every x and y in L, $xy \in L$.
5. No other strings are in L.

Using our new notation, we would summarize the "grammar rules" by writing

$$S \rightarrow \Lambda \mid a \mid b \mid SS$$

With this approach there is more than one way to obtain the string *bba*. Two derivations are shown below:

$$S \Rightarrow SS \Rightarrow bS \Rightarrow bSS \Rightarrow bbS \Rightarrow bba$$
$$S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow bSa \Rightarrow bba$$

The five steps in the first line correspond to rules 4, 3, 4, 3, and 2, and those in the second line to rules 4, 2, 4, 3, and 3.

In both cases in Example 6.1, the formulas obtained from the recursive definition can be interpreted as the grammar rules in a context-free grammar. Before we give the official definition of such a grammar, we consider two more examples. In Example 6.2, although the grammar is perhaps even simpler than that in Example 6.1, the corresponding language is one we know to be nonregular. Example 6.3 is perhaps more typical in that it includes a grammar containing more than one variable.

EXAMPLE 6.2 The Language $\{a^n b^n \mid n \geq 0\}$

The grammar rules

$$S \rightarrow aSb \mid \Lambda$$

are another way of describing the language L defined as follows:

1. $\Lambda \in L$.
2. For every $x \in L$, $axb \in L$.
3. Nothing else is in L.

The language L is easily seen to be the nonregular language $\{a^n b^n \mid n \geq 0\}$. The grammar rules in the first part of Example 6.1, for the language $\{a, b\}^*$, allow a's and b's to be added independently of each other. Here, on the other hand, each time one symbol is added to one end of the string by an application of the grammar rule $S \rightarrow aSb$, the opposite symbol is added simultaneously to the other end. As we have seen, this constraint is not one that can be captured by any regular expression.

EXAMPLE 6.3 Palindromes

Let us consider both the language *pal* of palindromes over the alphabet $\{a, b\}$ and its complement N, the set of all nonpalindromes over $\{a, b\}$. From Example 2.16, we have the following recursive definition of *pal*:

1. $\Lambda, a, b \in pal$.
2. For any $S \in pal$, aSa and bSb are in *pal*.
3. No other strings are in *pal*.

We can therefore describe *pal* by the context-free grammar with the grammar rules

$$S \rightarrow \Lambda \mid a \mid b \mid aSa \mid bSb$$

The language N also obeys rule 2: For any nonpalindrome x, both axa and bxb are nonpalindromes. However, a recursive definition of N cannot be as simple as this definition of *pal*, because there is no finite set of strings that can serve as basis elements in the definition. There is no finite set N_0 so that every nonpalindrome can be obtained from an element of N_0 by applications of rule 2 (Exercise 6.42). Consider a specific nonpalindrome, say

$$abbaaba$$

If we start at the ends and work our way in, trying to match the symbols at the beginning with those at the end, the string looks like a palindrome for the first two steps. What makes it a nonpalindrome is the central portion *baa*, which starts with one symbol and ends with the opposite symbol; the string between those two can be anything. A string is a nonpalindrome if and only if it has a central portion of this type. These are the "basic" nonpalindromes, and we can therefore write the following definition of N:

1. For any $A \in \{a, b\}^*$, aAb and bAa are in N.
2. For any $S \in N$, aSa and bSb are in N.
3. No other strings are in N.

In order to obtain a context-free grammar describing N, we can now simply introduce a second variable A, representing an arbitrary element of $\{a, b\}^*$, and incorporate the grammar rules for this language from Example 6.1:

$$S \rightarrow aAb \mid bAa \mid aSa \mid bSb$$
$$A \rightarrow \Lambda \mid Aa \mid Ab$$

A derivation of the nonpalindrome $abbaaba$, for example, would look like

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbAaba \Rightarrow abbAaaba \Rightarrow abbAaaba = abbaaba$$

It is common, and often necessary, to include several variables in a context-free grammar describing a language L. There will still be one special variable that represents an arbitrary string in L, and it is customary to denote this one by S (the *start* variable). The other variables can then be thought of as representing typical strings in certain auxiliary languages involved in the definition of L. (We can still interpret the grammar as a recursive definition of L, if we extend our notion of recursion slightly to include the idea of *mutual* recursion: rather than one object defined in terms of itself, several objects defined in terms of each other.)

Let us now give the general definition illustrated by these examples.

Definition 6.1 Definition of a Context-Free Grammar

A *context-free grammar* (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$, where V and Σ are disjoint finite sets, S is an element of V, and P is a finite set of formulas of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

The elements of V are called *variables*, or *nonterminal* symbols, and those of the alphabet Σ are called *terminal* symbols, or *terminals*. S is called the *start symbol*; and the elements of P are called *grammar rules*, or *productions*.

Suppose $G = (V, \Sigma, S, P)$ is a CFG. As in the first three examples, we will reserve the symbol $\to$ for individual productions in P. We use the symbol $\Rightarrow$ for steps in a derivation such as those in Examples 6.1 and 6.3. Sometimes it is useful to indicate explicitly that the derivation is with respect to the CFG G, and in this case we write $\Rightarrow_G$.

$$\alpha \Rightarrow_G \beta$$

means that the string β can be obtained from the string α by replacing some variable that appears on the left side of a production in G by the corresponding right side, or that

$$\alpha = \alpha_1 A \alpha_2$$
$$\beta = \alpha_1 \gamma \alpha_2$$

and one of the productions in G is $A \to \gamma$. (We can now understand better the term *context-free*. If at some point in a derivation we have obtained a string α containing the variable A, then we may continue by substituting γ for A, no matter what the strings α_1 and α_2 are—that is, independent of the context.)

In this case, we will say that α derives β, or β is derived from α, in one step. More generally, we write

$$\alpha \Rightarrow_G^* \beta$$

(and shorten it to $\alpha \Rightarrow^* \beta$ if it is clear what grammar is involved) if α derives β in zero or more steps; in other words, either $\alpha = \beta$, or there exist an integer $k \geq 1$ and strings $\alpha_0, \alpha_1, \ldots, \alpha_k$, with $\alpha_0 = \alpha$ and $\alpha_k = \beta$, so that $\alpha_i \Rightarrow_G \alpha_{i+1}$ for every i with $0 \leq i \leq k-1$.

Definition 6.2 The Language Generated by a CFG

Let $G = (V, \Sigma, S, P)$ be a CFG. The language generated by G is

$$L(G) = \{x \in \Sigma^* \mid S \Rightarrow_G^* x\}$$

A language L is a *context-free language* (CFL) if there is a CFG G so that $L = L(G)$.

The Language of Algebraic Expressions

EXAMPLE 6.4

An important language in computer science is the language of legal algebraic expressions. For simplicity we restrict ourselves here to the simple expressions that can be formed from the four binary operators $+$, $-$, $*$, and $/$, left and right parentheses, and the single identifier a. Some of the features omitted, therefore, are unary operators, any binary operators other than these four, numerical literals such as 3.0 or $\sqrt{2}$, expressions involving functional notation, and more general identifiers. Many of these features could be handled simply enough; for example, the language of arbitrary identifiers can be "embedded" within this one by using a variable A instead of the terminal symbol a and introducing productions that allow any identifier to be derived from A. (See Examples 3.5 and 3.6.)

A recursive definition of our language is based on the observation that legal expressions can be formed by joining two legal expressions using one of the four operators or by enclosing a legal expression within parentheses, and that these two operations account for all legal expressions except the single identifier a. The most straightforward way of getting a context-free grammar, therefore, is probably to use the productions

$$S \to S + S \mid S - S \mid S * S \mid S/S \mid (S) \mid a$$

The string $a + (a * a)/a - a$ can be obtained from the derivation

$$\begin{aligned} S &\Rightarrow S - S \Rightarrow S + S - S \Rightarrow a + S - S \Rightarrow a + S/S - S \\ &\Rightarrow a + (S)/S - S \Rightarrow a + (S * S)/S - S \Rightarrow a + (a * S)/S - S \\ &\Rightarrow a + (a * a)/S - S \Rightarrow a + (a * a)/a - S \Rightarrow a + (a * a)/a - a \end{aligned}$$

It is easy to see that there are many other derivations as well. For example,

$$\begin{aligned} S &\Rightarrow S/S \Rightarrow S + S/S \Rightarrow a + S/S \Rightarrow a + (S)/S \\ &\Rightarrow a + (S * S)/S \Rightarrow a + (a * S)/S \Rightarrow a + (a * a)/S \\ &\Rightarrow a + (a * a)/S - S \Rightarrow a + (a * a)/a - S \Rightarrow a + (a * a)/a - a \end{aligned}$$

We would probably say that the first of these is more natural than the second. The first starts with the production

$$S \to S - S$$

and therefore indicates that we are interpreting the original expression as the difference of two other expressions. This seems correct because the expression would normally be evaluated as follows:

1. Evaluate $a * a$, and call its value A.
2. Evaluate A/a, and call its value B.
3. Evaluate $a + B$, and call its value C.
4. Evaluate $C - a$.

The expression "is" the difference of the subexpression with value C and the subexpression a. The second derivation, by contrast, interprets the expression as a quotient. Although there is nothing in the grammar to rule out this derivation, it does not reflect our view of the correct structure of the expression.

One possible conclusion is that the context-free grammar we have given for the language may not be the most appropriate. It does not incorporate in any way the standard conventions, having to do with the precedence of operators and the left-to-right order of evaluation, that we use in evaluating the expression. (Precedence of operators dictates that in the expression $a + b * c$, the multiplication is performed before the addition; and the expression $a - b + c$ means $(a - b) + c$, not $a - (b + c)$.) Moreover, rather than having to choose between two derivations of a string, it is often desirable to select, if possible, a CFG in which a string can *have* only one derivation (except for trivial differences between the order in which two variables in some intermediate string are chosen for replacement). We will return to this question in Section 6.4, when we discuss *ambiguity* in a CFG.

EXAMPLE 6.5 The Syntax of Programming Languages

The language in the previous example and languages like those in Examples 3.5 and 3.6 are relatively simple ingredients of programming languages such as C and Pascal. To a large extent, context-free grammars can be used to describe the overall syntax of such languages.

In C, one might try to formulate grammar rules to specify what constitutes a legal statement. (As you might expect, a complete specification is very involved.) Two types of statements in C are *if* statements and *for* statements; if we represent an arbitrary statement by the variable ⟨statement⟩, the productions involving ⟨statement⟩ might look like

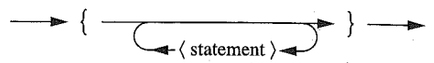
$$\langle \text{statement} \rangle \rightarrow \cdots \mid \langle \textit{if}\text{-statement} \rangle \mid \langle \textit{for}\text{-statement} \rangle \mid \cdots$$

The syntax of these two types of statements can be described by the rules

$$\langle \textit{if}\text{-statement} \rangle \rightarrow \text{if (} \langle \text{expression} \rangle \text{) } \langle \text{statement} \rangle$$
$$\langle \textit{for}\text{-statement} \rangle \rightarrow \text{for (} \langle \text{expression} \rangle \text{; } \langle \text{expression} \rangle \text{; } \langle \text{expression} \rangle \text{) } \langle \text{statement} \rangle$$

where ⟨expression⟩ is another variable, whose productions would also be difficult to describe completely.

Although in both cases the last term on the right side specifies a single statement, the logic of a program often requires more than one. It is therefore necessary to have our definition of ⟨statement⟩ allow a *compound* statement, which is simply a sequence of zero or more statements enclosed within {}. We could easily write a definition for ⟨compound-statement⟩ that would say this. A *syntax diagram* such as the one shown accomplishes the same thing.



A path through the diagram begins with {, ends with }, and can traverse the loop zero or more times.

EXAMPLE 6.6 Grammar Rules for English

The advantage of using high-level programming languages like C and Pascal is that they allow us to write statements that look more like English. If we can use context-free grammars to capture many of the rules of programming languages, what about English itself, which has its own "grammar rules"?

English sentences that are sufficiently simple can be described by CFGs. A great many sentences could be taken care of by the productions

$$\langle \text{declarative sentence} \rangle \rightarrow \langle \text{subject phrase} \rangle \langle \text{verb phrase} \rangle \langle \text{object} \rangle \mid \langle \text{subject phrase} \rangle \langle \text{verb phrase} \rangle$$

if we provided reasonable productions for each of the three variables on the right. Producing a wide variety of reasonable, idiomatic English sentences with context-free grammars, even context-free grammars of manageable size, is not hard; what *is* hard is doing this and at the same time disallowing gibberish. Even harder is disallowing sentences that seem to obey English syntax but that a native English speaker would probably never say, because they don't sound right.

Here is a simple example that might illustrate the point. Consider the productions

$$\langle \text{declarative sentence} \rangle \rightarrow \langle \text{subject} \rangle \langle \text{verb} \rangle \langle \text{object} \rangle$$
$$\langle \text{subject} \rangle \rightarrow \langle \text{proper noun} \rangle$$
$$\langle \text{proper noun} \rangle \rightarrow \text{John} \mid \text{Jane}$$
$$\langle \text{verb} \rangle \rightarrow \text{reminded}$$
$$\langle \text{object} \rangle \rightarrow \langle \text{proper noun} \rangle \mid \langle \text{reflexive pronoun} \rangle$$
$$\langle \text{reflexive pronoun} \rangle \rightarrow \text{himself} \mid \text{herself}$$

More than one sentence derivable from this grammar does not quite work: "John reminded herself" and "Jane reminded himself," for example. These could be eliminated in a straightforward way (at the cost of complicating the grammar) by introducing productions like

$$\langle \text{declarative sentence} \rangle \rightarrow \langle \text{masculine noun} \rangle \langle \text{verb} \rangle \langle \text{masculine reflexive pronoun} \rangle$$

A slightly more subtle problem is "Jane reminded Jane." Normally we do not say this, unless we have in mind two different people named Jane, but there is no obvious way to prohibit it without also prohibiting "Jane reminded John." (At least, there is no obvious way without essentially using a different production for every sentence we want to end up with. This trivial option is available here, since the language is finite.) To distinguish "Jane reminded John," which is a perfectly good English sentence, from "Jane reminded Jane" requires using *context*, and this is exactly what a context-free grammar does not allow.

6.2 | MORE EXAMPLES

In general, to show that a CFG generates a language, we must show two things: first, that every string in the language can be derived from the grammar, and second, that no other string can. In some of the examples in this section, at least one of these two statements is less obvious.

EXAMPLE 6.7 A CFG for $\{x \mid n_0(x) = n_1(x)\}$

Consider the language

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

where $n_i(x)$ is the number of i's in the string x.

As in Examples 6.1–6.3, we can begin by thinking about a recursive definition of L, and once we find one we can easily turn it into a context-free grammar.

Clearly, $\Lambda \in L$. Given a string x in L, we get a longer string in L by adding one 0 and one 1. (Conversely, any nonnull string in L can be obtained this way.) One way to add the symbols is to add one at each end, producing either $0x1$ or $1x0$. This suggests the productions

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

Not every string in L can be obtained from these productions, because some elements of L begin and end with the same symbol; the strings 0110, 10001101, and 0010111100 are examples. If we look for ways of expressing each of these in terms of simpler elements of L, we might notice that each is the concatenation of two nonnull elements of L (for example, the third string is the concatenation of 001011 and 1100). This observation suggests the production $S \to SS$.

It is reasonably clear that if G is the CFG containing the productions we have so far,

$$S \to \Lambda \mid 0S1 \mid 1S0 \mid SS$$

then derivations in G produce only strings in L. We will prove the converse, that $L \subseteq L(G)$.

It will be helpful to introduce the notation

$$d(x) = n_0(x) - n_1(x)$$

What we must show, therefore, is that for any string x with $d(x) = 0$, $x \in L(G)$. The proof is by mathematical induction on $|x|$.

In the basis step of the proof, we must show that if $|x| = 0$ and $d(x) = 0$ (of course, the second hypothesis is redundant), then $x \in L(G)$. This is true because one of the productions in G is $S \to \Lambda$.

Our induction hypothesis will be that $k \geq 0$ and that for any y with $|y| \leq k$ and $d(y) = 0$, $y \in L(G)$. We must show that if $|x| = k + 1$ and $d(x) = 0$, then $x \in L(G)$.

If x begins with 0 and ends with 1, then $x = 0y1$ for some string y satisfying $d(y) = 0$. By the induction hypothesis, $y \in L(G)$. Therefore, since $S \Rightarrow_G^* y$, we can derive x from S by starting the derivation with the production $S \to 0S1$ and continuing to derive y from the second S. The case when x begins with 1 and ends with 0 is handled the same way, except that the production $S \to 1S0$ is used to begin the derivation.

The remaining case is the one in which x begins and ends with the same symbol. Since $d(x) = 0$, x has length at least 2; suppose for example that $x = 0y0$ for some string y. We would like to show that x has a derivation in G. Such a derivation would have to start with the production $S \to SS$; in order to show that there is such a derivation, we would like to show that $x = wz$, where w and z are shorter strings that can both be derived from S. (It will then follow that we can start the derivation with $S \to SS$, then continue by deriving w from the first S and z from the second.) Another way to express this condition is to say that x has a prefix w so that $0 < |w| < |x|$ and $d(w) = 0$.

Let us consider $d(w)$ for prefixes w of x. The shortest nonnull prefix is 0, and $d(0) = 1$; the longest prefix shorter than x is $0y$, and $d(0y) = -1$ (because the last symbol of x is 0, and $d(x) = 0$). Furthermore, the d-value of a prefix changes by 1 each time an extra symbol is added. It follows that there must be a prefix w, longer than 0 and shorter than $0y$, with $d(w) = 0$. This is what we wanted to prove. The case when $x = 1y1$ is almost the same, and so the proof is concluded.

EXAMPLE 6.8 Another CFG for $\{x \mid n_0(x) = n_1(x)\}$

Let us continue with the language $L = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$ of the last example; this time we construct a CFG with three variables, based on a different approach to a recursive definition of L.

One way to obtain an element of L is to add both symbols to a string already in L. Another way, however, is to add a single symbol to a string that has one extra occurrence of the opposite symbol. Moreover, every element of L can be obtained this way and in fact can be obtained by adding this extra symbol at the beginning. Let us introduce the variables A and B, to represent strings with an extra 0 and an extra 1, respectively, and let us denote these two languages by L_0 and L_1:

$$L_0 = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x) + 1\} = \{x \in \{0, 1\}^* \mid d(x) = 1\}$$
$$L_1 = \{x \in \{0, 1\}^* \mid n_1(x) = n_0(x) + 1\} = \{x \in \{0, 1\}^* \mid d(x) = -1\}$$

where d is the function in Example 6.7 defined by $d(x) = n_0(x) - n_1(x)$. Then it is easy to formulate the productions we need starting with S:

$$S \to 0B \mid 1A \mid \Lambda$$

It is also easy to find one production for each of the variables A and B. If a string in L_0 begins with 0, or if a string in L_1 begins with 1, then the remainder is an element of L. Thus, it is appropriate to add the productions

$$A \to 0S \qquad B \to 1S$$

What remains are the strings in L_0 that start with 1 and the strings in L_1 that start with 0. In the first case, if $x = 1y$ and $x \in L_0$, then y has two more 0's than 1's. If it were true that y could be written as the concatenation of two strings, each with one extra 0, then we could complete the A-productions by adding $A \to 1AA$, and we could handle B similarly.

In fact, the same technique we used in Example 6.7 will work here. If $d(x) = 1$ and $x = 1y$, then Λ is a prefix of y with $d(\Lambda) = 0$, and y itself is a prefix of y with $d(y) = 2$. Therefore, there is some intermediate prefix w of y with $d(w) = 1$, and $y = wz$ where $w, z \in L_0$.

This discussion should make it at least plausible that the context-free grammar with productions

$$S \to 0B \mid 1A \mid \Lambda$$
$$A \to 0S \mid 1AA$$
$$B \to 1S \mid 0BB$$

generates the language L. By taking the start variable to be A or B, we could just as easily think of it as a CFG generating L_0 or L_1. It is possible without much difficulty to give an induction proof; see Exercise 6.50.

The following theorem provides three simple ways of obtaining new CFLs from languages that are known to be context-free.

Theorem 6.1
If L_1 and L_2 are context-free languages, then the languages $L_1 \cup L_2$, L_1L_2, and L_1^* are also CFLs.

Proof
The proof is constructive: Starting with CFGs

$$G_1 = (V_1, \Sigma, S_1, P_1) \quad \text{and} \quad G_2 = (V_2, \Sigma, S_2, P_2)$$

generating L_1 and L_2, respectively, we show how to construct a new CFG for each of the three cases.

A grammar $G_u = (V_u, \Sigma, S_u, P_u)$ generating $L_1 \cup L_2$. First we rename the elements of V_2 if necessary so that $V_1 \cap V_2 = \emptyset$, and we define

$$V_u = V_1 \cup V_2 \cup \{S_u\}$$

where S_u is a new symbol not in V_1 or V_2. Then we let

$$P_u = P_1 \cup P_2 \cup \{S_u \to S_1 \mid S_2\}$$

On the one hand, if x is in either L_1 or L_2, then $S_u \Rightarrow^* x$ in the grammar G_u, because we can start a derivation with either $S_u \to S_1$ or $S_u \to S_2$ and continue with the derivation of x in G_1 or G_2. Therefore,

$$L_1 \cup L_2 \subseteq L(G_u)$$

On the other hand, if x is derivable from S_u in G_u, the first step in any derivation must be

$$S_u \Rightarrow S_1 \quad \text{or} \quad S_u \Rightarrow S_2$$

In the first case, all subsequent productions used must be productions in G_1, because no variables in V_2 are involved, and thus $x \in L_1$; in the second case, $x \in L_2$. Therefore,

$$L(G_u) \subseteq L_1 \cup L_2$$

A grammar $G_c = (V_c, \Sigma, S_c, P_c)$ generating L_1L_2. Again we relabel variables if necessary so that $V_1 \cap V_2 = \emptyset$, and define

$$V_c = V_1 \cup V_2 \cup \{S_c\}$$

This time we let

$$P_c = P_1 \cup P_2 \cup \{S_c \to S_1S_2\}$$

If $x \in L_1L_2$, then $x = x_1x_2$, where $x_i \in L_i$ for each i. We may then derive x in G_c as follows:

$$S_c \Rightarrow S_1S_2 \Rightarrow^* x_1S_2 \Rightarrow^* x_1x_2 = x$$

where the second step is the derivation of x_1 in G_1 and the third step is the derivation of x_2 in G_2. Conversely, if x can be derived from S_c, then since the first step in the derivation must be $S_c \Rightarrow S_1S_2$, x must be derivable from S_1S_2. Therefore, $x = x_1x_2$, where for each i, x_i can be derived from S_i in G_c. Since $V_1 \cap V_2 = \emptyset$, being derivable from S_i in G_c means being derivable from S_i in G_i, and so $x \in L_1L_2$.

A grammar $G^* = (V, \Sigma, S, P)$ generating L_1^*. Let

$$V = V_1 \cup \{S\}$$

where $S \notin V_1$. The language L_1^* contains strings of the form $x = x_1x_2 \cdots x_k$, where each $x_i \in L_1$. Since each x_i can be derived from S_1, then to derive x from S it is enough to be able to derive a string of k S_1's. We can accomplish this by including the productions

$$S \to S_1S \mid \Lambda$$

in P. Therefore, let

$$P = P_1 \cup \{S \to S_1S \mid \Lambda\}$$

The proof that $L_1^* \subseteq L(G^*)$ is straightforward. If $x \in L(G^*)$, on the other hand, then either $x = \Lambda$ or x can be derived from some string of the form S_1^k in G^*. In the second case, since the only productions in G^* beginning with S_1 are those in G_1, we may conclude that

$$x \in L(G_1)^k \subseteq L(G_1)^*$$

Note that it really is necessary in the first two parts of the proof to make sure that $V_1 \cap V_2 = \emptyset$. Consider CFGs having productions

$$S_1 \to XA \qquad X \to c \qquad A \to a$$

and

$$S_2 \to XB \qquad X \to d \qquad B \to b$$

respectively. If we applied the construction in the first part of the proof without relabeling variables, the resulting grammar would allow the derivation

$$S \Rightarrow S_1 \Rightarrow XA \Rightarrow dA \Rightarrow da$$

even though da is not derivable from either of the two original grammars.

Corollary 6.1 Every regular language is a CFL.

Proof
According to Definition 3.1, regular languages over Σ are the languages obtained from $\emptyset$, $\{\Lambda\}$, and $\{a\}$ $(a \in \Sigma)$ by using the operations of union, concatenation, and Kleene *. Each of the primitive languages $\emptyset$, $\{\Lambda\}$, and $\{a\}$ is a context-free language. (In the first case we can use the trivial grammar with no productions, and in the other two cases one production is sufficient.) The corollary therefore follows from Theorem 6.1, using the principle of structural induction. ■

EXAMPLE 6.9 A CFG Equivalent to a Regular Expression

Let L be the language corresponding to the regular expression

$$(011 + 1)^*(01)^*$$

We can take a few obvious shortcuts in the algorithm provided by the proof of Theorem 6.1. The productions

$$A \rightarrow 011 \mid 1$$

generate the language $\{011, 1\}$. Following the third part of the theorem, we can use the productions

$$B \rightarrow AB \mid \Lambda$$
$$A \rightarrow 011 \mid 1$$

with B as the start symbol to generate the language $\{011, 1\}^*$. Similarly, we can use

$$C \rightarrow DC \mid \Lambda$$
$$D \rightarrow 01$$

to derive $\{01\}^*$ from the start symbol C. Finally, we generate the concatenation of the two languages by adding the production $S \rightarrow BC$. The final grammar has start symbol S, auxiliary variables A, B, C, and D, and productions

$$S \rightarrow BC$$
$$B \rightarrow AB \mid \Lambda$$
$$A \rightarrow 011 \mid 1$$
$$C \rightarrow DC \mid \Lambda$$
$$D \rightarrow 01$$

Starting with any regular expression, we can obtain an equivalent CFG using the techniques illustrated in this example. In the next section we will see that any regular language L can also be described by a CFG whose productions all have a very simple form, and that such a CFG can be obtained easily from an FA accepting L.

EXAMPLE 6.10 A CFG for $\{x \mid n_0(x) \neq n_1(x)\}$

Consider the language

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

Neither of the CFGs we found in Examples 6.7 and 6.8 for the complement of L is especially helpful here. As we will see in Chapter 8, there is no general technique for finding a grammar generating the complement of a given CFL—which may in some cases not be a context-free language at all. However, we can express L as the union of the languages L_0 and L_1, where

$$L_0 = \{x \in \{0, 1\}^* \mid n_0(x) > n_1(x)\}$$
$$L_1 = \{x \in \{0, 1\}^* \mid n_1(x) > n_0(x)\}$$

and thus we concentrate on finding a CFG G_0 generating the language L_0. Clearly $0 \in L_0$, and for any $x \in L_0$, both $x0$ and $0x$ are in L_0. This suggests the productions

$$S \rightarrow 0 \mid S0 \mid 0S$$

We also need to be able to add 1's to our strings. We cannot expect that adding a 1 to an element of L_0 will always produce an element of L_0; however, if we have two strings in L_0, concatenating them produces a string with at least two more 0's than 1's, and then adding a single 1 will still yield an element of L_0. We could add it at the left, at the right, or between the two. The corresponding productions are

$$S \rightarrow 1SS \mid SS1 \mid S1S$$

It is not hard to see that any string derived by using the productions

$$S \rightarrow 0 \mid S0 \mid 0S \mid 1SS \mid SS1 \mid S1S$$

is an element of L_0 (see Exercise 6.43). In the converse direction we can do even a little better: if G_0 is the grammar with productions

$$S \rightarrow 0 \mid 0S \mid 1SS \mid SS1 \mid S1S$$

every string in L_0 can be derived in G_0.

The proof is by induction on the length of the string. We consider the case that is probably hardest and leave the others to the exercises. As in previous examples, let $d(x) = n_0(x) - n_1(x)$. The basis step, for a string in L_0 of length 1, is straightforward. Suppose that $k \geq 1$ and that any x for which $|x| \leq k$ and $d(x) > 0$ can be derived in G_0; and consider a string x for which $|x| = k + 1$ and $d(x) > 0$.

We consider the case in which $x = 0y0$ for some string y. If x contains only 0's, it can be derived from S using the productions $S \rightarrow 0 \mid 0S$; we assume, therefore, that x contains at least one 1. Our goal is to show that x has the form

$$x = w1z \text{ for some } w \text{ and } z \text{ with } d(w) > 0 \text{ and } d(z) > 0$$

Once we have done this, the induction hypothesis will tell us that both w and z can be derived from S, so that we will be able to derive x by starting with the production

$$S \to S1S$$

To show that x has this form, suppose x contains n 1's, where $n \geq 1$. For each i with $1 \leq i \leq n$, let w_i be the prefix of x up to but not including the ith 1, and z_i the suffix of x that follows this 1. In other words, for each i,

$$x = w_i 1 z_i$$

where the 1 is the ith 1 in x. If $d(w_n) > 0$, then we may let $w = w_n$ and $z = z_n$. The string z_n is 0^j for some $j > 0$ because x ends with 0, and we have the result we want. Otherwise, $d(w_n) \leq 0$. In this case we select the *first* i with $d(w_i) \leq 0$, say $i = m$. Now since x begins with 0, $d(w_1)$ must be > 0, which implies that $m \geq 2$. At this point, we can say that $d(w_{m-1}) > 0$ and $d(w_m) \leq 0$. Because w_m has only one more 1 than w_{m-1}, $d(w_{m-1})$ can be no more than 1. Therefore, $d(w_{m-1}) = 1$. Since $x = w_{m-1}1z_{m-1}$, and $d(x) > 0$, it follows that $d(z_{m-1}) > 0$. This means that we get the result we want by letting $w = w_{m-1}$ and $z = z_{m-1}$. The proof in this case is complete.

For the other two cases, the one in which x starts with 1 and the one in which x ends with 1, see Exercise 6.44.

Now it is easy enough to obtain a context-free grammar G generating L. We use S as our start symbol, A as the start symbol of the grammar we have just derived generating L_0, and B as the start symbol for the corresponding grammar generating L_1. The grammar G then has the productions

$$S \to A \mid B$$
$$A \to 0 \mid 0A \mid 1AA \mid AA1 \mid A1A$$
$$B \to 1 \mid 1B \mid 0BB \mid BB0 \mid B0B$$

EXAMPLE 6.11 Another Application of Theorem 6.1

Let $L = \{0^i 1^j 0^k \mid j > i + k\}$. We try expressing L as the concatenation of CFLs, although what may seem at first like the obvious approach—writing L as a concatenation $L_1L_2L_3$, where these three languages contain strings of 0's, strings of 1's, and strings of 0's, respectively—is doomed to failure. L contains both $0^1 1^3 0^1$ and $0^1 1^4 0^2$, but if we allowed L_1 to contain 0^1, L_2 to contain 1^3, and L_3 to contain 0^2, then $L_1L_2L_3$ would also contain $0^1 1^3 0^2$, and this string is not an element of L.

Observe that

$$0^i 1^{i+k} 0^k = 0^i 1^i \, 1^k 0^k$$

The only difference between this and a string x in L is that x has at least one extra 1 in the middle:

$$x = 0^i 1^i \, 1^m \, 1^k 0^k \quad \text{(for some } m > 0\text{)}$$

A correct formula for L is therefore $L = L_1L_2L_3$, where

$$L_1 = \{0^i 1^i \mid i \geq 0\}$$
$$L_2 = \{1^m \mid m > 0\}$$
$$L_3 = \{1^k 0^k \mid k \geq 0\}$$

The second part of Theorem 6.1, applied twice, reduces the problem to finding CFGs for these three languages.

L_1 is essentially the language in Example 6.2, L_3 is the same with the symbols 0 and 1 reversed, and L_2 can be generated by the productions

$$B \to 1B \mid 1$$

(The second production is $B \to 1$, not $B \to \Lambda$, since we want only nonnull strings.)

The final CFG $G = (V, \Sigma, S, P)$ incorporating these pieces is shown below.

$$V = \{S, A, B, C\} \qquad \Sigma = \{0, 1\}$$
$$P = \{\, S \to ABC$$
$$A \to 0A1 \mid \Lambda$$
$$B \to 1B \mid 1$$
$$C \to 1C0 \mid \Lambda\}$$

A derivation of $01^40^2 = (01)(1)(1^20^2)$, for example, is

$$S \Rightarrow ABC \Rightarrow 0A1BC \Rightarrow 0\Lambda 1BC \Rightarrow 011C$$
$$\Rightarrow 0111C0 \Rightarrow 01111C00 \Rightarrow 01111\Lambda 00 = 0111100$$

6.3 | REGULAR GRAMMARS

The proof of Theorem 6.1 provides an algorithm for constructing a CFG corresponding to a given regular expression. In this section, we consider another way of obtaining a CFG for a regular language L, this time starting with an FA accepting L. The resulting grammar is distinctive in two ways. First, the productions all have a very simple form, closely related to the moves of the FA; second, the construction is reversible, so that a CFG of this simple type can be used to generate a corresponding FA.

We can see how to proceed by looking at an example, the FA in Figure 6.1. It accepts the language $L = \{0, 1\}^*\{10\}$, the set of all strings over $\{0, 1\}$ that end in 10. One element of L is $x = 110001010$. We trace its processing by the FA, as follows.

Substring processed so far	State
Λ	A
1	B
11	B
110	C
1100	A
11000	A
110001	B
1100010	C
11000101	B
110001010	C

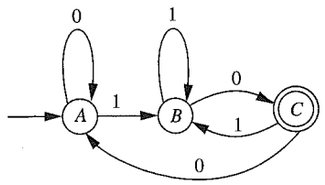


Figure 6.1 |

If we list the lines of this table consecutively, separated by $\Rightarrow$, we obtain

$$A \Rightarrow 1B \Rightarrow 11B \Rightarrow 110C \Rightarrow 1100A \Rightarrow 11000A \Rightarrow 110001B$$
$$\Rightarrow 1100010C \Rightarrow 11000101B \Rightarrow 110001010C$$

This looks like a derivation in a grammar. The grammar can be obtained by specifying the variables to be the states of the FA and starting with the productions

$$A \to 1B$$
$$B \to 1B$$
$$B \to 0C$$
$$C \to 0A$$
$$A \to 0A$$
$$C \to 1B$$

These include every production of the form

$$P \to aQ$$

where

$$P \xrightarrow{a} Q$$

is a transition in the FA. The start symbol is A, the initial state of the FA. To complete the derivation, we must remove the C from the last string. We do this by adding the production $B \to 0$, so that the last step in the derivation is actually

$$11000101B \Rightarrow 110001010$$

Note that the production we have added is of the form

$$P \to a$$

where

$$P \xrightarrow{a} F$$

is a transition from P to an accepting state F.

Any FA leads to a grammar in exactly this way. In our example it is easy to see that the language generated is exactly the one recognized by the FA. In general, we must qualify the statement slightly because the rules we have described for obtaining

productions do not allow Λ-productions; however, it will still be true that the nonnull strings accepted by the FA are precisely those generated by the resulting grammar.

A significant feature of any derivation in such a grammar is that until the last step there is exactly one variable in the current string; we can think of it as the "state of the derivation," and in this sense the derivation simulates the processing of the string by the FA.

Definition 6.3 Regular Grammars

A grammar $G = (V, \Sigma, S, P)$ is *regular* if every production takes one of the two forms

$$B \to aC$$
$$B \to a$$

for variables B and C and terminals a.

Theorem 6.2

For any language $L \subseteq \Sigma^*$, L is regular if and only if there is a regular grammar G so that $L(G) = L - \{\Lambda\}$.

Proof

First, suppose that L is regular, and let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA accepting L. Define the grammar $G = (V, \Sigma, S, P)$ by letting $V = Q$, $S = q_0$, and

$$P = \{B \to aC \mid \delta(B, a) = C\} \cup \{B \to a \mid \delta(B, a) = F \text{ for some } F \in A\}$$

Suppose that $x = a_1 a_2 \cdots a_n$ is accepted by M, and $n \geq 1$. Then there is a sequence of transitions

$$\to q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$$

where $q_n \in A$. By definition of G, we have the corresponding derivation

$$S = q_0 \Rightarrow a_1 q_1 \Rightarrow a_1 a_2 q_2 \Rightarrow \cdots \Rightarrow a_1 a_2 \cdots a_{n-1} q_{n-1} \Rightarrow a_1 a_2 \cdots a_{n-1} a_n$$

Similarly, if x is generated by G, it is clear that x is accepted by M.

Conversely, suppose $G = (V, \Sigma, S, P)$ is a regular grammar generating L. To some extent we can reverse the construction above: We can define states corresponding to all the variables and create a transition

$$B \xrightarrow{a} C$$

for every production of the form $B \to aC$. Note that this is likely to introduce nondeterminism. We can handle the other type of production by adding one extra state f, which will be the only accepting state, and letting every production $B \to a$ correspond to a transition $B \xrightarrow{a} f$.

Our machine $M = (Q, \Sigma, q_0, A, \delta)$ is therefore an NFA. Q is the set $V \cup \{f\}$, q_0 the start symbol S, and A the set $\{f\}$. For any $q \in V$ and $a \in \Sigma$,

$$\delta(q, a) = \begin{cases} \{p \mid \text{the production } q \to ap \text{ is in } P\} & \text{if } q \to a \text{ is not in } P \\ \{p \mid q \to ap \text{ is in } P\} \cup \{f\} & \text{if } q \to a \text{ is in } P \end{cases}$$

There are no transitions out of f.

If $x = a_1 a_2 \cdots a_n \in L = L(G)$, then there is a derivation of the form

$$S \Rightarrow a_1 q_1 \Rightarrow a_1 a_2 q_2 \Rightarrow \cdots \Rightarrow a_1 a_2 \cdots a_{n-1} q_{n-1} \Rightarrow a_1 a_2 \cdots a_{n-1} a_n$$

and according to our definition of M, there is a corresponding sequence of transitions

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} f$$

which implies that x is accepted by M.

On the other hand, if $x = a_1 \cdots a_n$ is accepted by M, then $|x| \geq 1$ because f is the only accepting state of M. The transitions causing x to be accepted look like

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} f$$

These transitions correspond to a derivation of x in the grammar, and it follows that $x \in L(G)$.

Sometimes the term *regular* is applied to grammars that do not restrict the form of the reductions so severely. It can be shown (Exercise 6.12) that a language is regular if and only if it can be generated, except possibly for the null string, by a grammar in which all productions look like this:

$$B \to xC$$
$$B \to x$$

where B and C are variables and x is a nonnull string of terminals. Grammars of this type are also called *linear*. The exercises discuss a few other variations as well.

6.4 | DERIVATION TREES AND AMBIGUITY

In a natural language such as English, understanding a sentence begins with understanding its grammatical structure, which means knowing how it is derived from the grammar rules for the language. Similarly, in a context-free grammar that specifies the syntax of a programming language or the rules for constructing an algebraic expression, interpreting a string correctly requires finding a correct derivation of the string in the grammar. A natural way of exhibiting the structure of a derivation is to draw a *derivation tree*, or *parse tree*. At the root of the tree is the variable with which the derivation begins. Interior nodes correspond to variables that appear in the derivation, and the children of the node corresponding to A represent the symbols in

a string α for which the production $A \to \alpha$ is used in the derivation. (In the case of a production $A \to \Lambda$, the node labeled A has the single child Λ.)

In the simplest case, when the tree is the derivation tree for a string $x \in L(G)$ and there are no "Λ-productions" (of the form $A \to \Lambda$), the leaf nodes of the tree correspond precisely to the symbols of x. If there are Λ-productions, they show up in the tree, so that some of the leaf nodes correspond to Λ; of course, those nodes can be ignored as one scans the leaf nodes to see the string being derived, because Λ's can be interspersed arbitrarily among the terminals without changing the string. In the most general case, we will also allow "derivations" that begin with some variable other than the start symbol of the grammar, and the string being derived may still contain some variables as well as terminal symbols.

In Example 6.4 we considered the CFG with productions

$$S \to S + S \mid S - S \mid S * S \mid S/S \mid (S) \mid a$$

The derivation

$$S \Rightarrow S - S \Rightarrow S * S - S \Rightarrow a * S - S \Rightarrow a * a - S \Rightarrow a * a - a$$

has the derivation tree show in Figure 6.2*a*. The derivation

$$S \Rightarrow S - S \Rightarrow S - S/S \Rightarrow \cdots \Rightarrow a - a/a$$

has the derivation tree shown in Figure 6.2*b*. In general, any derivation of a string in a CFL has a corresponding derivation tree (exactly one).

(There is a technical point here that is worth mentioning. With the two productions $S \to SS \mid a$, the sequence of steps $S \Rightarrow SS \Rightarrow SSS \Rightarrow^* aaa$ can be interpreted two ways, because in the second step it could be either the first S or the second that is replaced by SS. The two interpretations correspond to two different derivation trees. For this reason, we say that specifying a derivation means giving not only the sequence of strings but also the position in each string at which the next substitution occurs. The steps $S \Rightarrow SS \Rightarrow SSS$ already represent two different derivations.)

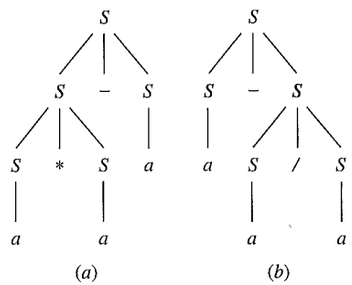


Figure 6.2 |
Derivation trees for two algebraic expressions.

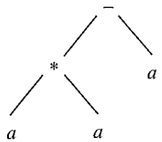


Figure 6.3 | Expression tree corresponding to Figure 6.1*a*.

Algebraic expressions such as the two shown in Figure 6.2 are often represented by *expression trees*—binary trees in which terminal nodes correspond to identifiers or constants and nonterminal nodes correspond to operators. The expression tree in Figure 6.3 conveys the same information as the derivation tree in Figure 6.2*a*, except that only the nodes representing terminal symbols are drawn.

One step in a derivation is the replacement of a variable (to be precise, a particular occurrence of a variable) by the string on the right side of a production. The derivation is the entire sequence of such steps, and in a *sequence* the order of the steps is significant. The derivations

$$S \Rightarrow S+S \Rightarrow a+S \Rightarrow a+a$$

and

$$S \Rightarrow S+S \Rightarrow S+a \Rightarrow a+a$$

are therefore different. However, they are different only in a trivial way: When the current string is $S+S$, the S that is used in the next step is the leftmost S in the first derivation and the rightmost in the second. A precise way to say that these derivations are not significantly different is to say that their derivation trees are the same. A derivation tree specifies completely which productions are used in the derivation, as well as where the right side of each production fits in the string being derived. It does not specify the "temporal" order in which the variables are used, and this order plays no role in using the derivation to interpret the string's structure. Two derivations that correspond to the same derivation tree are essentially the same.

Another way to compare two derivations is to normalize each of them, by requiring that each follow the same rule as to which variable to replace first whenever there is a choice, and to compare the normalized versions. A derivation is a *leftmost* derivation if the variable used in each step is always the leftmost variable of the ones in the current string. If the two derivations being compared are both leftmost and are still different, it seems reasonable to say that they are essentially, or significantly, different.

In fact, these two criteria for "essentially the same" are equivalent. On the one hand, leftmost derivations corresponding to different derivation trees are clearly different, because as we have already observed, any derivation corresponds to only one derivation tree. On the other hand, the derivation trees corresponding to two different leftmost derivations are also different, and we can see this as follows. Consider the first step at which the derivations differ; suppose that this step is

$$xA\beta \Rightarrow x\alpha_1\beta$$

in one derivation and

$$xA\beta \Rightarrow x\alpha_2\beta$$

in the other. Here x is a string of terminals, since the derivations are leftmost; A is a variable; and $\alpha_1 \neq \alpha_2$. The two derivation trees must both have a node labeled A, and the respective portions of the two trees to the left of this node must be identical, because the leftmost derivations have been the same up to this point. These two nodes have different sets of children, however, and the trees cannot be the same.

We conclude that a string of terminals has more than one derivation tree if and only if it has more than one leftmost derivation. Notice that in this discussion "leftmost" could just as easily be "rightmost"; the important thing is not what order is followed, only that some clearly defined order be followed consistently, so that the two normalized versions can be compared meaningfully.

As we have already noticed, a string can have two or more essentially different derivations in the same CFG.

Definition 6.4 An Ambiguous CFG

A context-free grammar G is *ambiguous* if there is at least one string in $L(G)$ having two or more distinct derivation trees (or, equivalently, two or more distinct leftmost derivations).

It is not hard to see that the ambiguity defined here is closely related to the ambiguity we encounter every day in written and spoken language. The reporter who wrote the headline "Disabled Fly to See Carter," which appeared during the administration of the thirty-ninth U.S. President, probably had in mind a derivation such as

$$S \rightarrow \langle\text{collective noun}\rangle\langle\text{verb}\rangle \cdots$$

However, one that begins

$$S \rightarrow \langle\text{adjective}\rangle\langle\text{noun}\rangle \cdots$$

might suggest a more intriguing or at least less predictable story. Understanding a sentence or a newspaper headline requires picking the right grammatical derivation for it.

Ambiguity in the CFG in Example 6.4 — EXAMPLE 6.12

Let us return to the algebraic-expression CFG discussed in Example 6.4, with productions

$$S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid (S) \mid a$$

In that example we considered two essentially different derivations of the string

$$a+(a*a)/a-a$$

and in fact the two derivations were both leftmost, which therefore demonstrates the ambiguity of the grammar. This can also be demonstrated using only the productions $S \rightarrow S+S$ and $S \rightarrow a$; the string $a+a+a$ has leftmost derivations

$$S \Rightarrow S+S \Rightarrow a+S \Rightarrow a+S+S \Rightarrow a+a+S \Rightarrow a+a+a$$

and

$$S \Rightarrow S+S \Rightarrow S+S+S \Rightarrow a+S+S \Rightarrow a+a+S \Rightarrow a+a+a$$

The corresponding derivation trees are shown in Figures 6.4*a* and 6.4*b*, respectively.

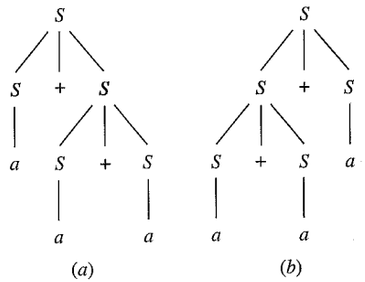


Figure 6.4 |
Two derivation trees for $a + a + a$.

Although the difference in the two interpretations of the string $a + a + a$ is not quite as dramatic as in Example 6.4 (the expression is viewed as the sum of two subexpressions in both cases), the principle is the same. The expression is interpreted as $a + (a + a)$ in one case, and $(a + a) + a$ in the other. The parentheses might be said to remove the ambiguity as to how the expression is to be interpreted. We will examine this property of parentheses more carefully in the next section, when we discuss an unambiguous CFG equivalent to this one.

It is easy to see by studying Example 6.12 that every CFG containing a production of the general form $A \rightarrow A\alpha A$ is ambiguous. However, there are more subtle ways in which ambiguity occurs, and characterizing the ambiguous context-free grammars in any nontrivial way turns out to be difficult or impossible (see Section 11.6).

EXAMPLE 6.13 The "Dangling Else"

A standard example of ambiguity in programming languages is the "dangling else" phenomenon. Consider the productions

⟨statement⟩ → if (⟨expression⟩) ⟨statement⟩ |
if (⟨expression⟩) ⟨statement⟩ else ⟨statement⟩ |
⟨otherstatement⟩

describing the *if* statement of Example 6.5 as well as the related *if-else* statement, both part of the C language. Now consider the statement

```
if (expr1) if (expr2) f(); else g();
```

This can be derived in two ways from the grammar rules. In one, illustrated in Figure 6.5*a*, the *else* goes with the first *if*, and in the other, illustrated in Figure 6.5*b*, it goes with the second. A C compiler should interpret the statement the second way, but not as a result of the syntax rules given; this is additional information with which the compiler must be furnished.

Just as in Example 6.12, parentheses or their equivalent could be used to remove the ambiguity in the statement.

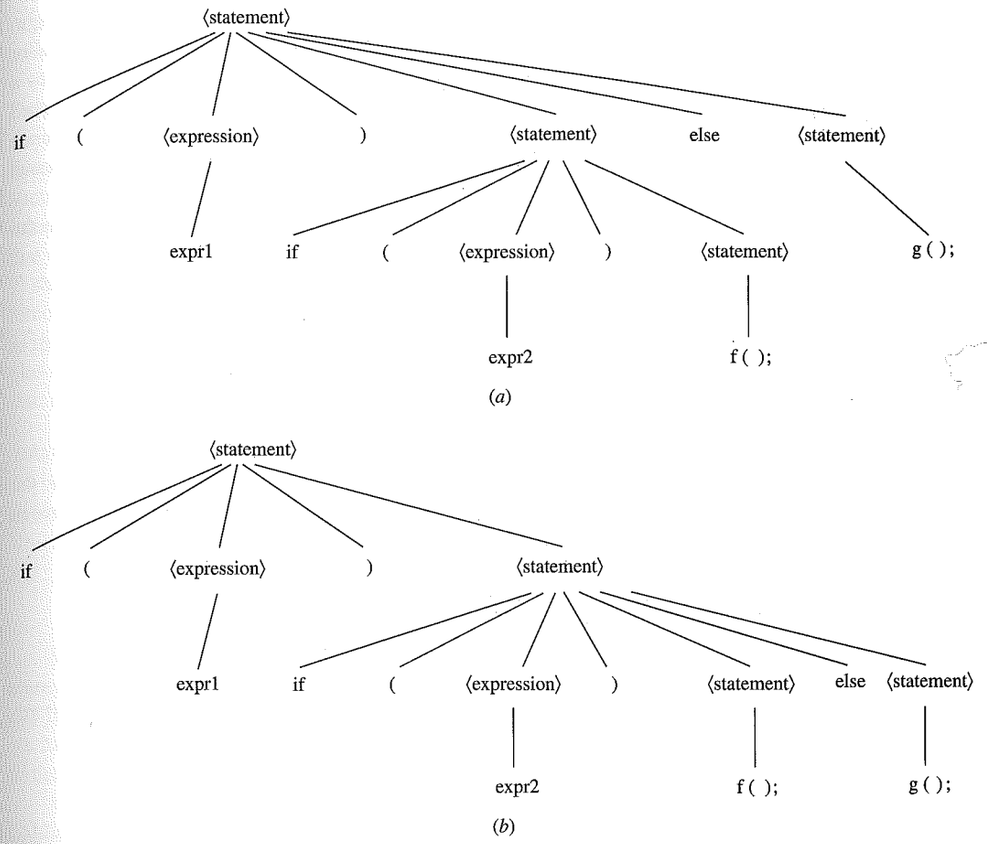


Figure 6.5 |
Two interpretations of a "dangling else."

```
if (expr1) {if (expr2) f();} else g();
```

forces the first interpretation, whereas

```
if (expr1) {if (expr2) f(); else g();}
```

forces the second. In some other languages, the appropriate version of "parentheses" is BEGIN ...END.

It is possible, however, to find grammar rules equivalent to the given ones that incorporate the correct interpretation into the syntax. Consider the formulas

⟨statement⟩ → ⟨st1⟩ | ⟨st2⟩
⟨st1⟩ → if (⟨expression⟩) ⟨st1⟩ else ⟨st1⟩ | ⟨otherstatement⟩
⟨st2⟩ → if (⟨expression⟩) ⟨statement⟩ |
if (⟨expression⟩) ⟨st1⟩ else ⟨st2⟩

These generate the same strings as the original rules and can be shown to be unambiguous. Although we will not present a proof of either fact, you can see the intuitive reason for the second. The variable ⟨ st1 ⟩ represents a statement in which every *if* is matched by a corresponding *else*, while any statement derived from ⟨ st2 ⟩ contains at least one unmatched *if*. The only variable appearing before *else* in these formulas is ⟨ st1 ⟩; since the *else* cannot match any of the *if*s in the statement derived from ⟨ st1 ⟩, it must match the *if* that appeared in the formula with the *else*.

It is interesting to compare both these sets of formulas with the corresponding ones in the official grammar for the Modula-2 programming language:

⟨statement⟩ → IF ⟨expression⟩ THEN ⟨statementsequence⟩ END |
IF ⟨expression⟩ THEN ⟨statementsequence⟩
ELSE ⟨statementsequence⟩ END |
⟨otherstatement⟩

These obviously resemble the rules for C in the first set above. However, the explicit END after each sequence of one or more statements allows the straightforward grammar rule to avoid the "dangling else" ambiguity. The Modula-2 statement corresponding most closely to the tree in Figure 6.5*a* is

```
IF A1 THEN IF A2 THEN S1 END ELSE S2 END
```

while Figure 6.5*b* corresponds to

```
IF A1 THEN IF A2 THEN S1 ELSE S2 END END
```

6.5 AN UNAMBIGUOUS CFG FOR ALGEBRAIC EXPRESSIONS

Although it is possible to prove that some context-free languages are inherently ambiguous, in the sense that they can be produced only by ambiguous grammars, ambiguity is normally a property of the grammar rather than the language. If a CFG is ambiguous, it is often possible and usually desirable to find an equivalent unambiguous CFG. In this section, we will solve this problem in the case of the algebraic-expression grammar discussed in Example 6.4.

For the sake of simplicity we will use only the two operators $+$ and $*$ in our discussion, so that G has productions

$$S \rightarrow S+S \mid S*S \mid (S) \mid a$$

Once we obtain an unambiguous grammar equivalent to this one, it will be easy to reinstate the other operators.

Our final grammar will not have either $S \rightarrow S+S$ or $S \rightarrow S*S$, because either production by itself is enough to produce ambiguity. We will also keep in mind the possibility, mentioned in Example 6.4, of incorporating into the grammar the standard

rules of order and operator precedence: $*$ should have higher precedence than $+$, and $a+a+a$ should "mean" $(a+a)+a$, not $a+(a+a)$.

In trying to eliminate $S \rightarrow S+S$ and $S \rightarrow S*S$, it is helpful to remember Example 2.15, where we discussed possible recursive definitions of L^*. Two possible ways of obtaining new elements of L^* are to concatenate two elements of L^* and to concatenate an element of L^* with an element of L; we observed that the second approach preserves the direct correspondence between one application of the recursive rule and one of the "primitive" strings being concatenated. Here this idea suggests that we replace $S \rightarrow S+S$ by either $S \rightarrow S+T$ or $S \rightarrow T+S$, where the variable T stands for a *term*, an expression that cannot itself be expressed as a sum. If we remember that $a+a+a = (a+a)+a$, we would probably choose $S \rightarrow S+T$ as more appropriate; in other words, an expression consists of (all but the last term) plus the last term. Because an expression can also consist of a single term, we will also need the production $S \rightarrow T$. At this point, we have

$$S \rightarrow S+T \mid T$$

We may now apply the same principle to the set of terms. Terms can be products; however, rather than thinking of a term as a product of terms, we introduce *factors*, which are terms that cannot be expressed as products. The corresponding productions are

$$T \rightarrow T*F \mid F$$

So far we have a hierarchy of levels. Expressions, the most general objects, are sums of one or more terms, and terms are products of one or more factors. This hierarchy incorporates the precedence of multiplication over addition, and the productions we have chosen also incorporate the fact that both the $+$ and $*$ operations associate to the left.

It should now be easy to see where parenthesized expressions fit into the hierarchy. (Although we might say (A) could be an expression or a term or a factor, we should permit ourselves only one way of deriving it, and we must decide which is most appropriate.) A parenthesized expression cannot be expressed directly as either a sum or a product, and it therefore seems most appropriate to consider it a factor. To say it another way, evaluation of a parenthetical expression should take precedence over any operators outside the parentheses; therefore, (A) should be considered a factor, because in our hierarchy factors are evaluated first. What is inside the parentheses should be an *expression*, since it is not restricted at all.

The grammar that we end up with is $G1 = (V, \Sigma, S, P)$, where $V = \{S, T, F\}$ and P contains the productions

$$S \rightarrow S+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (S) \mid a$$

We must now prove two things: first, that $G1$ is indeed equivalent to the original grammar G, and second, that it is unambiguous. To avoid confusion, we relabel the start symbol in $G1$.

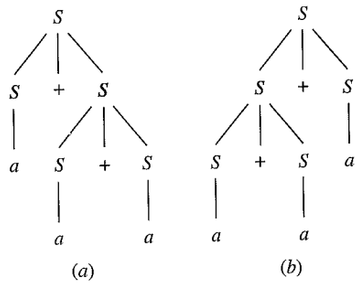


Figure 6.4 |
Two derivation trees for $a + a + a$.

Although the difference in the two interpretations of the string $a + a + a$ is not quite as dramatic as in Example 6.4 (the expression is viewed as the sum of two subexpressions in both cases), the principle is the same. The expression is interpreted as $a + (a + a)$ in one case, and $(a + a) + a$ in the other. The parentheses might be said to remove the ambiguity as to how the expression is to be interpreted. We will examine this property of parentheses more carefully in the next section, when we discuss an unambiguous CFG equivalent to this one.

It is easy to see by studying Example 6.12 that every CFG containing a production of the general form $A \rightarrow A\alpha A$ is ambiguous. However, there are more subtle ways in which ambiguity occurs, and characterizing the ambiguous context-free grammars in any nontrivial way turns out to be difficult or impossible (see Section 11.6).

EXAMPLE 6.13 The "Dangling Else"

A standard example of ambiguity in programming languages is the "dangling else" phenomenon. Consider the productions

⟨statement⟩ → if (⟨expression⟩) ⟨statement⟩ |
if (⟨expression⟩) ⟨statement⟩ else ⟨statement⟩ |
⟨otherstatement⟩

describing the *if* statement of Example 6.5 as well as the related *if-else* statement, both part of the C language. Now consider the statement

```
if (expr1) if (expr2) f(); else g();
```

This can be derived in two ways from the grammar rules. In one, illustrated in Figure 6.5*a*, the *else* goes with the first *if*, and in the other, illustrated in Figure 6.5*b*, it goes with the second. A C compiler should interpret the statement the second way, but not as a result of the syntax rules given; this is additional information with which the compiler must be furnished.

Just as in Example 6.12, parentheses or their equivalent could be used to remove the ambiguity in the statement.

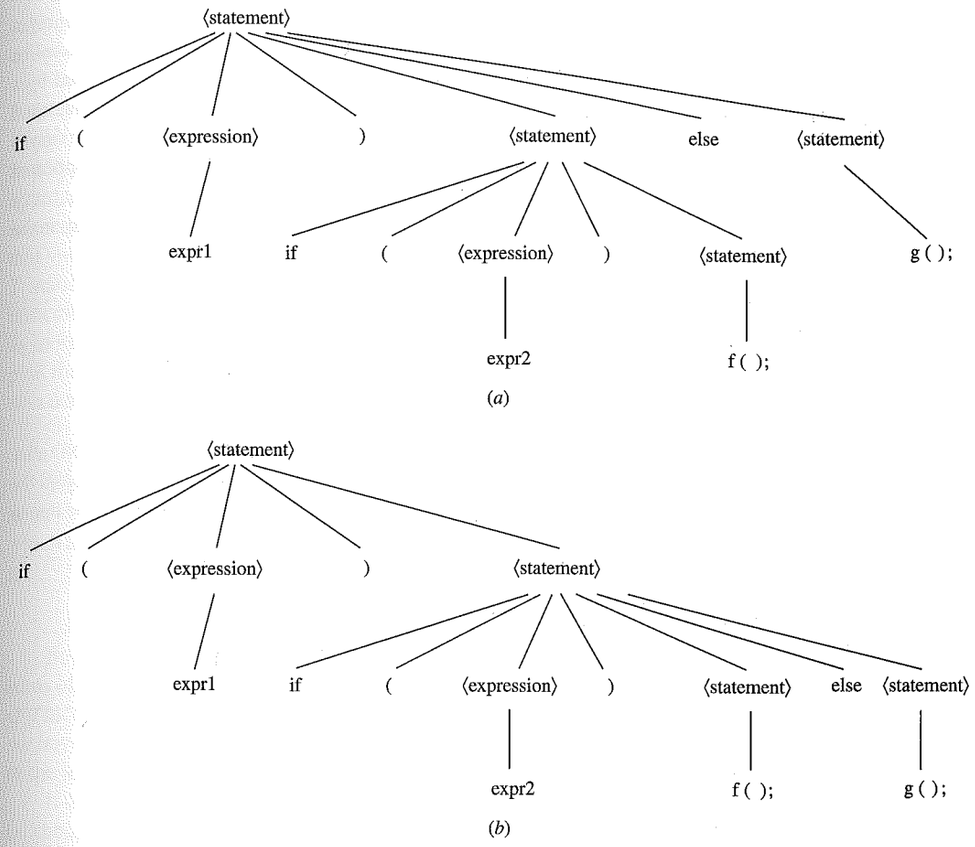


Figure 6.5 |
Two interpretations of a "dangling else."

```
if (expr1) {if (expr2) f();} else g();
```

forces the first interpretation, whereas

```
if (expr1) {if (expr2) f(); else g();}
```

forces the second. In some other languages, the appropriate version of "parentheses" is BEGIN ...END.

It is possible, however, to find grammar rules equivalent to the given ones that incorporate the correct interpretation into the syntax. Consider the formulas

⟨statement⟩ → ⟨st1⟩ | ⟨st2⟩
⟨st1⟩ → if (⟨expression⟩) ⟨st1⟩ else ⟨st1⟩ | ⟨otherstatement⟩
⟨st2⟩ → if (⟨expression⟩) ⟨statement⟩ |
if (⟨expression⟩) ⟨st1⟩ else ⟨st2⟩

These generate the same strings as the original rules and can be shown to be unambiguous. Although we will not present a proof of either fact, you can see the intuitive reason for the second. The variable ⟨ st1 ⟩ represents a statement in which every *if* is matched by a corresponding *else*, while any statement derived from ⟨ st2 ⟩ contains at least one unmatched *if*. The only variable appearing before *else* in these formulas is ⟨ st1 ⟩; since the *else* cannot match any of the *if*s in the statement derived from ⟨ st1 ⟩, it must match the *if* that appeared in the formula with the *else*.

It is interesting to compare both these sets of formulas with the corresponding ones in the official grammar for the Modula-2 programming language:

⟨statement⟩ → IF ⟨expression⟩ THEN ⟨statementsequence⟩ END |
IF ⟨expression⟩ THEN ⟨statementsequence⟩
ELSE ⟨statementsequence⟩ END |
⟨otherstatement⟩

These obviously resemble the rules for C in the first set above. However, the explicit END after each sequence of one or more statements allows the straightforward grammar rule to avoid the "dangling else" ambiguity. The Modula-2 statement corresponding most closely to the tree in Figure 6.5*a* is

```
IF A1 THEN IF A2 THEN S1 END ELSE S2 END
```

while Figure 6.5*b* corresponds to

```
IF A1 THEN IF A2 THEN S1 ELSE S2 END END
```

6.5 | AN UNAMBIGUOUS CFG FOR ALGEBRAIC EXPRESSIONS

Although it is possible to prove that some context-free languages are inherently ambiguous, in the sense that they can be produced only by ambiguous grammars, ambiguity is normally a property of the grammar rather than the language. If a CFG is ambiguous, it is often possible and usually desirable to find an equivalent unambiguous CFG. In this section, we will solve this problem in the case of the algebraic-expression grammar discussed in Example 6.4.

For the sake of simplicity we will use only the two operators $+$ and $*$ in our discussion, so that G has productions

$$S \rightarrow S + S \mid S * S \mid (S) \mid a$$

Once we obtain an unambiguous grammar equivalent to this one, it will be easy to reinstate the other operators.

Our final grammar will not have either $S \rightarrow S + S$ or $S \rightarrow S * S$, because either production by itself is enough to produce ambiguity. We will also keep in mind the possibility, mentioned in Example 6.4, of incorporating into the grammar the standard

rules of order and operator precedence: $*$ should have higher precedence than $+$, and $a + a + a$ should "mean" $(a + a) + a$, not $a + (a + a)$.

In trying to eliminate $S \rightarrow S + S$ and $S \rightarrow S * S$, it is helpful to remember Example 2.15, where we discussed possible recursive definitions of L^*. Two possible ways of obtaining new elements of L^* are to concatenate two elements of L^* and to concatenate an element of L^* with an element of L; we observed that the second approach preserves the direct correspondence between one application of the recursive rule and one of the "primitive" strings being concatenated. Here this idea suggests that we replace $S \rightarrow S + S$ by either $S \rightarrow S + T$ or $S \rightarrow T + S$, where the variable T stands for a *term*, an expression that cannot itself be expressed as a sum. If we remember that $a + a + a = (a + a) + a$, we would probably choose $S \rightarrow S + T$ as more appropriate; in other words, an expression consists of (all but the last term) plus the last term. Because an expression can also consist of a single term, we will also need the production $S \rightarrow T$. At this point, we have

$$S \rightarrow S + T \mid T$$

We may now apply the same principle to the set of terms. Terms can be products; however, rather than thinking of a term as a product of terms, we introduce *factors*, which are terms that cannot be expressed as products. The corresponding productions are

$$T \rightarrow T * F \mid F$$

So far we have a hierarchy of levels. Expressions, the most general objects, are sums of one or more terms, and terms are products of one or more factors. This hierarchy incorporates the precedence of multiplication over addition, and the productions we have chosen also incorporate the fact that both the $+$ and $*$ operations associate to the left.

It should now be easy to see where parenthesized expressions fit into the hierarchy. (Although we might say (A) could be an expression or a term or a factor, we should permit ourselves only one way of deriving it, and we must decide which is most appropriate.) A parenthesized expression cannot be expressed directly as either a sum or a product, and it therefore seems most appropriate to consider it a factor. To say it another way, evaluation of a parenthetical expression should take precedence over any operators outside the parentheses; therefore, (A) should be considered a factor, because in our hierarchy factors are evaluated first. What is inside the parentheses should be an *expression*, since it is not restricted at all.

The grammar that we end up with is $G1 = (V, \Sigma, S, P)$, where $V = \{S, T, F\}$ and P contains the productions

$$S \rightarrow S + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (S) \mid a$$

We must now prove two things: first, that $G1$ is indeed equivalent to the original grammar G, and second, that it is unambiguous. To avoid confusion, we relabel the start symbol in $G1$.

Theorem 6.3
Let G be the context-free grammar with productions

$$S \rightarrow S+S \mid S*S \mid (S) \mid a$$

and let $G1$ be the context-free grammar with productions

$$S1 \rightarrow S1+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (S1) \mid a$$

Then $L(G) = L(G1)$.

Proof

First, to show $L(G1) \subseteq L(G)$. The proof is by induction on the length of a string in $L(G1)$. The basis step is to show that $a \in L(G)$, and this is clear.

In the induction step we assume that $k \geq 1$ and that every y in $L(G1)$ satisfying $|y| \leq k$ is in $L(G)$. We must show that if $x \in L(G1)$ and $|x| = k+1$, then $x \in L(G)$. Since $x \neq a$, any derivation of x in $G1$ must begin in one of these three ways:

$$S1 \Rightarrow S1 + T$$
$$S1 \Rightarrow T \Rightarrow T*F$$
$$S1 \Rightarrow T \Rightarrow F \Rightarrow (S1)$$

We give a proof in the first case, and the other two are very similar. If x has a derivation beginning $S1 \Rightarrow S1+T$, then $x = y+z$, where $S1 \Rightarrow^*_{G1} y$ and $T \Rightarrow^*_{G1} z$. Since $S1 \Rightarrow^*_{G1} T$, it follows that $S1 \Rightarrow^*_{G1} z$. Therefore, since $|y|$ and $|z|$ must be $\leq k$, the induction hypothesis implies that y and z are both in $L(G)$. Since G contains the production $S \rightarrow S+S$, the string $y+z$ is derivable from S in G, and therefore $x \in L(G)$.

Second, to show $L(G) \subseteq L(G1)$. Again we use induction on $|x|$, and just as in the first part, the basis step is straightforward. We assume that $k \geq 1$ and that for every $y \in L(G$ with $|y| \leq k$, $y \in L(G1)$; we wish to show that if $x \in L(G)$ and $|x| = k+1$, then $x \in L(G1)$.

The simplest case is that in which x has a derivation in G beginning $S \rightarrow (S)$. In this case $x = (y)$, for some y in $L(G)$, and it follows from the inductive hypothesis that $y \in L(G1)$. Therefore, we can derive x in $G1$, by starting the derivation $S1 \Rightarrow T \Rightarrow F \Rightarrow (S1)$ and then deriving y from $S1$.

Suppose x has a derivation in G that begins $S \rightarrow S+S$. Then just as before, the induction hypothesis tells us that $x = y+z$, where y and z are both in $L(G1)$. Now, however, in order to conclude that $x \in L(G1)$, we need z to be derivable from T; in other words, we would like z to be a single term, the last of the terms whose sum is x. With this in mind, let

$$x = x_1 + x_2 + \cdots + x_n$$

where each $x_i \in L(G1)$ and n is as large as possible. We have already determined that $n \geq 2$. Because of the way n is defined, none of the x_i's can have a derivation in $G1$ that begins $S1 \Rightarrow S1+T$; therefore, every x_i can be derived from T in $G1$. Let

$$y = x_1 + x_2 + \cdots + x_{n-1} \qquad z = x_n$$

Then y can be derived from $S1$, since $S1 \Rightarrow^*_{G1} T+T+\cdots+T$ ($n-1$ terms), and z can be derived from T. It follows that $x \in L(G1)$, since we can start with the production $S1 \rightarrow S1+T$.

Finally, suppose that every derivation of x in G begins $S \Rightarrow S*S$. Then for some y and z in $L(G)$, $x = y*z$. This time we let

$$x = x_1 * x_2 * \cdots * x_n$$

where each $x_i \in L(G)$ and n is as large as possible. (Note the difference between this statement and the one in the previous case.) Then by the inductive hypothesis, each $x_i \in L(G1)$. What we would like this time is for each x_i to be derivable from F in G_1. We can easily rule out the case where some x_i has a derivation in $G1$ that begins $S1 \Rightarrow T \Rightarrow T*F$. If this were true, x_i would be of the form $y_i * z_i$ for some $y_i, z_i \in L(G1)$; since we know $L(G1) \subseteq L(G)$, this would contradict the maximal property of the number n.

Suppose that some x_i had a derivation in $G1$ beginning $S1 \Rightarrow S1+T$. Then $x_i = y_i + z_i$ for some $y_i, z_i \in L(G1) \subseteq L(G)$. In this case,

$$x = x_1 * x_2 * \cdots * x_{i-1} * y_i + z_i * x_{i+1} * \cdots * x_n$$

This is impossible too. If we let u and v be the substrings before and after the $+$, respectively, we clearly have $u, v \in L(G)$, and therefore $x = u+v$. This means that we could derive x in G using a derivation that begins $S \Rightarrow S+S$, and we have assumed that this is not the case.

We may conclude that each x_i is derivable from F in $G1$. Then, as we did in the previous case, we let

$$y = x_1 * x_2 * \cdots * x_{n-1} \qquad z = x_n$$

The string y is derivable from T in $G1$, since $F*F*\cdots*F$ ($n-1$ factors) is. Therefore, we may derive x from $S1$ in $G1$ by starting the derivation $S1 \Rightarrow T \Rightarrow T*F$, and so $x \in L(G1)$.

In order to show that the grammar $G1$ is unambiguous, it will be helpful to concentrate on the parentheses in a string, temporarily ignoring the other terminal symbols. In a sense we want to convince ourselves that the grammar is unambiguous insofar as it generates strings of parentheses; this will then allow us to demonstrate the unambiguity of the entire grammar. In Exercise 5.34, we defined a string of parentheses to be *balanced* if it is the string of parentheses appearing in some legal algebraic expression. At this point, however, we must be a little more explicit.

Definition 6.5 Balanced Strings of Parentheses

A string of left and right parentheses is *balanced* if it has equal numbers of left and right parentheses, and no prefix has more right than left. The *mate* of a left parenthesis in a balanced string is the first right parenthesis following it for which the string containing those two and everything in between is balanced. If x is a string containing parentheses and other symbols, and the parentheses within x form a balanced string, a symbol σ in x is *within parentheses* if σ appears between some left parenthesis and its mate.

You should spend a minute convincing yourself that every left parenthesis in a balanced string has a mate (Exercise 6.30). The first observation we make is that the string of parentheses in any string obtained from $S1$ in the grammar $G1$ is balanced. Certainly it has equal numbers of left and right parentheses, since they are produced in pairs. Moreover, for every right parenthesis produced by a derivation in $G1$, a left parenthesis appearing before it is produced simultaneously, and so no prefix of the string can have more right parentheses than left.

Secondly, observe that in any derivation in $G1$, the parentheses between and including the pair produced by a single application of the production $F \rightarrow (S1)$ form a balanced string. This is because the parentheses within the string derived from $S1$ do, and because enclosing a balanced string of parentheses within parentheses yields a balanced string.

Now suppose that $x \in L(G1)$, and $(_0$ is any left parenthesis in x. The statement that there is only one leftmost derivation of x in $G1$ will follow if we can show that $G1$ is unambiguous, and we will be able to do this very soon. For now, however, the discussion above allows us to say that even if there are several leftmost derivations of x, the right parenthesis produced at the same time as $(_0$ is the same for any of them—it is simply the mate of $(_0$. To see this, let us consider a fixed derivation of x. In this derivation, the step in which $(_0$ is produced also produces a right parenthesis, which we call $)_0$. As we have seen in the previous paragraph, the parentheses in x beginning with $(_0$ and ending with $)_0$ form a balanced string. This implies that the mate of $(_0$ cannot appear after $)_0$, because of how "mate" is defined.

However, the mate of $(_0$ cannot appear before $)_0$ either. Suppose α is the string of parentheses starting just after $(_0$ and ending with the mate of $(_0$. The string α has an excess of right parentheses, because $(_0\alpha$ is balanced. Let β be the string of parentheses strictly between $(_0$ and $)_0$. Then β is balanced. If the mate of $(_0$ appeared before $)_0$, α would be a prefix of β, and this is impossible. Therefore, the mate of $(_0$ coincides with $)_0$.

The point of this discussion is that when we say that something is *within parentheses*, we can be sure that the parentheses it is within are the two parentheses produced by the production $F \rightarrow (S1)$, no matter what derivation we have in mind. This is the ingredient we need for our theorem.

Theorem 6.4

The context-free grammar $G1$ with productions

$$S1 \rightarrow S1 + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (S1) \mid a$$

is unambiguous.

Proof

We wish to show that every string x in $L(G1)$ has only one leftmost derivation from $S1$. The proof will be by mathematical induction on $|x|$, and it will actually be easier to prove something apparently stronger: For any x derivable from one of the variables $S1$, T, or F, x has only one leftmost derivation from that variable.

For the basis step, we observe that a can be derived from any of the three variables, and that in each case there is only one derivation.

In the induction step, we assume that $k \geq 1$ and that for every y derivable from $S1$, T, or F for which $|y| \leq k$, y has only one leftmost derivation from that variable. We wish to show the same result for a string x with $|x| = k+1$.

Consider first the case in which x contains at least one $+$ not within parentheses. Since the only $+$'s in strings derivable from T or F are within parentheses, x can be derived only from $S1$, and any derivation of x must begin $S1 \Rightarrow S1 + T$, where this $+$ is the last $+$ in x that is not within parentheses. Therefore, any leftmost derivation of x from $S1$ has the form

$$S1 \Rightarrow S1 + T \Rightarrow^* y + T \Rightarrow^* y + z$$

where the last two steps represent leftmost derivations of y from $S1$ and z from T, respectively, and the $+$ is still the last one not within parentheses. The induction hypothesis tells us that y has only one leftmost derivation from $S1$ and z has only one from T. Therefore, x has only one leftmost derivation from $S1$.

Next consider the case in which x contains no $+$ outside parentheses but at least one $*$ outside parentheses. This time x can be derived only from $S1$ or T; any derivation from $S1$ must begin $S1 \Rightarrow T \Rightarrow T * F$; and any derivation from T must begin $T \Rightarrow T * F$. In either case, the $*$ must be the last one in x that is not within parentheses. As in the first case, the subsequent steps of any leftmost derivation must be

$$T * F \Rightarrow^* y * F \Rightarrow^* y * z$$

consisting first of a leftmost derivation of y from T and then of a leftmost derivation of z from F. Again the induction hypothesis tells us that there is

only one possible way for these derivations to proceed, and so there is only one leftmost derivation of x from $S1$ or T.

Finally, suppose x contains no $+$'s or $*$'s outside parentheses. Then x can be derived from any of the variables, but the only derivation from $S1$ begins $S1 \Rightarrow T \Rightarrow F \Rightarrow (S1)$, and the only derivation from T or F begins the same way with the first one or two steps omitted. Therefore, $x = (y)$, where $S1 \Rightarrow^* y$. By the induction hypothesis, y has only one leftmost derivation from $S1$, and it follows that x has only one from each of the three variables. This completes the proof.

6.6 SIMPLIFIED FORMS AND NORMAL FORMS

Ambiguity is one undesirable property of a context-free grammar that we might wish to eliminate. In this section we discuss some slightly more straightforward ways of improving a grammar without changing the resulting language: first by eliminating certain types of productions that may be awkward to work with, and then by standardizing the productions so that they all have a certain "normal form."

We begin by trying to eliminate "Λ-productions," those of the form $A \rightarrow \Lambda$, and "unit productions," in which one variable is simply replaced by another. To illustrate how these improvements might be useful, suppose that a grammar contains neither type of production, and consider a derivation containing the step

$$\alpha \Rightarrow \beta$$

If there are no Λ-productions, then the string β must be at least as long as α; if there are no unit productions, α and β can be of equal length only if this step consists of replacing a variable by a single terminal. To say it another way, if l and t represent the length of the current string and the number of terminals in the current string, respectively, then the quantity $l + t$ must increase at each step of the derivation. The value of $l + t$ is 1 for the string S and $2k$ for a string x of length k in the language. We may conclude that a derivation of x can have no more than $2k - 1$ steps. In particular, we now have an algorithm for determining whether a given string x is in the language generated by the grammar: If $|x| = k$, try all possible sequences of $2k - 1$ productions, and see if any of them produces x. Although this is not usually a practical algorithm, at least it illustrates the fact that information about the form of productions can be used to derive conclusions about the resulting language.

In trying to eliminate Λ-productions from a grammar, we must begin with a qualification. We obviously cannot eliminate all productions of this form if the string Λ itself is in the language. This obstacle is only minor, however: We will be able to show that for any context-free language L, $L - \{\Lambda\}$ can be generated by a CFG with no Λ-productions. A preliminary example will help us see how to proceed.

Eliminating Λ-productions from a CFG — EXAMPLE 6.14

Let G be the context-free grammar with productions

$$S \rightarrow ABCBCDA$$
$$A \rightarrow CD$$
$$B \rightarrow Cb$$
$$C \rightarrow a \mid \Lambda$$
$$D \rightarrow bD \mid \Lambda$$

The first thing this example illustrates is probably obvious already: We cannot simply throw away the Λ-productions without adding anything. In this case, if $D \rightarrow \Lambda$ is eliminated then nothing can be derived, because the Λ-production is the only way to remove the variable D from the current string.

Let us consider the production $S \rightarrow ABCBCDA$, which we write temporarily as

$$S \rightarrow ABC_1BC_2DA$$

The three variables C_1, C_2, and D on the right side all begin Λ-productions, and each can also be used to derive a nonnull string. In a derivation we may replace none, any, or all of these three by Λ. Without Λ-productions, we will need to allow for all these options by adding productions of the form $S \rightarrow \alpha$, where α is a string obtained from $ABCBCDA$ by deleting some or all of $\{C_1, C_2, D\}$. In other words, we will need at least the productions

$$S \rightarrow ABBC_2DA \mid ABC_1BDA \mid ABC_1BC_2A \mid ABBDA \mid ABBC_2A \mid ABC_1BA \mid ABBA$$

in addition to the one we started with, in order to make sure of obtaining all the strings that can be obtained from the original grammar.

If we now consider the variable A, we see that these productions are still not enough. Although A does not begin a Λ-production, the string Λ can be derived from A (as can other nonnull strings). Starting with the production $A \rightarrow CD$, we can leave out C or D, using the same argument as before. We cannot leave out both, because we do not want the production $A \rightarrow \Lambda$ in our final grammar. If we add subscripts to the occurrences of A, as we did to those of C, so that the original production is

$$S \rightarrow A_1BC_1BC_2DA_2$$

we need to add productions in which the right side is obtained by leaving out some subset of $\{A_1, A_2, C_1, C_2, D\}$. There are 32 subsets, which means that from this original production we obtain 31 others that will be added to our grammar.

The same reasoning applies to each of the original productions. If we can identify in the production $X \rightarrow \alpha$ all the variables occurring in α from which Λ can be derived, then we can add all the productions $X \rightarrow \alpha'$, where α' is obtained from α by deleting some of these occurrences. In general this procedure might produce new Λ-productions—if so, they are ignored—and it might produce productions of the form $X \rightarrow X$, which also contribute nothing to the grammar and can be omitted.

In this case our final context-free grammar has 40 productions, including the 32 S-productions already mentioned and the ones that follow:

$$A \to CD \mid C \mid D$$
$$B \to Cb \mid b$$
$$C \to a$$
$$D \to bD \mid b$$

The procedure outlined in Example 6.14 is the one that we will show works in general. In presenting it more systematically, we give first a recursive definition of a *nullable* variable (one from which Λ can be derived), and then we give the algorithm suggested by this definition for identifying such variables.

Definition 6.6 Nullable Variables

A *nullable* variable in a CFG $G = (V, \Sigma, S, P)$ is defined as follows.

1. Any variable A for which P contains the production $A \to \Lambda$ is nullable.
2. If P contains the production $A \to B_1 B_2 \dots B_n$ and $B_1, B_2, \dots, B_n$ are nullable variables, then A is nullable.
3. No other variables in V are nullable.

Algorithm FindNull (Finding the nullable variables in a CFG (V, Σ, S, P))

$N_0 = \{A \in V \mid P$ contains the production $A \to \Lambda\}$;
$i = 0$;
do
  $i = i + 1$;
  $N_i = N_{i-1} \cup \{A \mid P$ contains $A \to \alpha$ for some $\alpha \in N_{i-1}^*\}$
while $N_i \neq N_{i-1}$;
N_i is the set of nullable variables. ■

You can easily convince yourself that the variables defined in Definition 6.6 are the variables A for which $A \Rightarrow^* \Lambda$. Obtaining the algorithm FindNull from the definition is straightforward, and a similar procedure can be used whenever we have such a recursive definition (see Exercise 2.70). When we apply the algorithm in Example 6.14, the set N_0 is $\{C, D\}$. The set N_1 also contains A, as a result of the production $A \to CD$. Since no other productions have right sides in $\{A, C, D\}^*$, these three are the only nullable variables in the grammar.

Algorithm 6.1 (Finding an equivalent CFG with no Λ-productions) Given a CFG $G = (V, \Sigma, S, P)$, construct a CFG $G1 = (V, \Sigma, S, P1)$ with no Λ-productions as follows.

1. Initialize $P1$ to be P.
2. Find all nullable variables in V, using Algorithm FindNull.
3. For every production $A \to \alpha$ in P, add to $P1$ every production that can be obtained from this one by deleting from α one or more of the occurrences of nullable variables in α.
4. Delete all Λ-productions from $P1$. Also delete any duplicates, as well as productions of the form $A \to A$. ■

Theorem 6.5

Let $G = (V, \Sigma, S, P)$ be any context-free grammar, and let $G1$ be the grammar obtained from G by Algorithm 6.1. Then $G1$ has no Λ-productions, and $L(G1) = L(G) - \{\Lambda\}$.

Proof

That $G1$ has no Λ-productions is obvious. We show a statement that is slightly stronger than that in the theorem: For any variable $A \in V$, and any nonnull $x \in \Sigma^*$,

$$A \Rightarrow_G^* x \text{ if and only if } A \Rightarrow_{G1}^* x$$

We use the notation $A \Rightarrow_G^k x$ to mean that there is a k-step derivation of x from A in G.

We show first that for any $n \geq 1$, if $A \Rightarrow_G^n x$, then $A \Rightarrow_{G1}^* x$. The proof is by mathematical induction on n. For the basis step, suppose $A \Rightarrow_G^1 x$. Then $A \to x$ is a production in P. Since $x \neq \Lambda$, this production is also in $P1$, and so $A \Rightarrow_{G1}^* x$.

In the induction step we assume that $k \geq 1$ and that any string other than Λ derivable from A in k or fewer steps in G is derivable from A in $G1$. We wish to show that if $x \neq \Lambda$ and $A \Rightarrow_G^{k+1} x$, then $A \Rightarrow_{G1}^* x$. Suppose that the first step in a $(k+1)$-step derivation of x in G is $A \to X_1 X_2 \cdots X_n$, where each X_i is either a variable or a terminal. Then $x = x_1 x_2 \cdots x_n$, where each x_i is either equal to X_i or derivable from X_i in k or fewer steps in G. Any X_i for which the corresponding x_i is Λ is a nullable variable in G. If we delete these X_i's from the string $X_1 X_2 \cdots X_n$, there are still some left, since $x \neq \Lambda$, and the resulting production is an element of $P1$. Furthermore, the induction hypothesis tells us that for each X_i remaining in the right side of this production, $X_i \Rightarrow_{G1}^* x_i$. Therefore, $A \Rightarrow_{G1}^* x$.

Now we show the converse, that for any $n \geq 1$, if $A \Rightarrow_{G1}^n x$, then $A \Rightarrow_G^* x$; again the proof is by induction on n. If $A \Rightarrow_{G1}^1 x$, then $A \to x$ is a production in $P1$. This means that $A \to \alpha$ is a production in P, where x is obtained from α by deleting zero or more nullable variables. It follows that $A \Rightarrow_G^* x$, because we can begin a derivation with the production $A \to \alpha$ and proceed by deriving Λ from each of the nullable variables that was deleted to obtain x.

Suppose that $k \geq 1$ and that any string other than Λ derivable from A in k or fewer steps in $G1$ is derivable from A in G. We wish to show the same result for a string x for which $A \Rightarrow_{G1}^{k+1} x$. Again, let the first step of a $(k+1)$-step derivation of x in $G1$ be $A \rightarrow X_1 X_2 \cdots X_n$, where each X_i is either a variable or a terminal. We may write $x = x_1 x_2 \cdots x_n$, where each x_i is either equal to X_i or derivable from X_i in k or fewer steps in G_1. By the induction hypothesis, $X_i \Rightarrow_G^* x_i$ for each i. By definition of $G1$, there is a production $A \rightarrow \alpha$ in P so that $X_1 X_2 \cdots X_n$ can be obtained from α by deleting certain nullable variables. Since $A \Rightarrow_G^* X_1 X_2 \cdots X_n$, we can derive x from A in G by first deriving $X_1 X_2 \cdots X_n$ and then deriving each x_i from the corresponding X_i.

Eliminating Λ-productions from a grammar is likely to increase the number of productions substantially. We might ask whether any other undesirable properties are introduced. One partial answer is that if the context-free grammar G is unambiguous, then the grammar $G1$ produced by Algorithm 6.1 is also (Exercise 6.64).

The next method of modifying a context-free grammar, eliminating unit productions, is similar enough in principle to what we have just done that we omit many of the details or leave them to the exercises. Just as it was necessary before to consider all nullable variables as well as those that actually begin Λ-productions, here it is necessary to consider all the pairs of variables A, B for which $A \Rightarrow^* B$ as well as the pairs for which there is actually a production $A \rightarrow B$. In order to guarantee that eliminating unit productions does not also eliminate strings in the language, we make sure that whenever $B \rightarrow \alpha$ is a nonunit production and $A \Rightarrow^* B$, we add the production $A \rightarrow \alpha$.

In order to simplify the process of finding all such pairs A, B, we make the simplifying assumption that we have already used Algorithm 6.1 if necessary so that the grammar has no Λ-productions. It follows in this case that one variable can be derived from another only by a sequence of unit productions. For any variable A, we may therefore formulate the following recursive definition of the set of "A-derivable" variables (essentially, variables B other than A for which $A \Rightarrow^* B$), and the definition can easily be adapted to obtain an algorithm.

1. If $A \rightarrow B$ is a production, B is A-derivable.
2. If C is A-derivable, $C \rightarrow B$ is a production, and $B \neq A$, then B is A-derivable.
3. No other variables are A-derivable.

(Note that according to our definition, a variable A is A-derivable only if $A \rightarrow A$ is actually a production.)

Algorithm 6.2 (Finding an equivalent CFG with no unit productions) Given a context-free grammar $G = (V, \Sigma, S, P)$ with no Λ-productions, construct a grammar $G1 = (V, \Sigma, S, P1)$ having no unit productions as follows.

1. Initialize $P1$ to be P.
2. For each $A \in V$, find the set of A-derivable variables.

3. For every pair (A, B) such that B is A-derivable, and every nonunit production $B \rightarrow \alpha$, add the production $A \rightarrow \alpha$ to $P1$ if it is not already present in $P1$.
4. Delete all unit productions from $P1$.

Theorem 6.6
Let G be any CFG without Λ-productions, and let $G1$ be the CFG obtained from G by Algorithm 6.2. Then $G1$ contains no unit productions, and $L(G1) = L(G)$.

The proof is omitted (Exercise 6.62). It is worth pointing out, again without proof, that if the grammar G is unambiguous, then the grammar $G1$ obtained from the algorithm is also.

Eliminating Unit Productions — EXAMPLE 6.15

Let G be the algebraic-expression grammar obtained in the previous section, with productions

$$S \rightarrow S + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (S) \mid a$$

The S-derivable variables are T and F, and F is T-derivable. In step 3 of Algorithm 6.2, the productions $S \rightarrow T * F \mid (S) \mid a$ and $T \rightarrow (S) \mid a$ are added to $P1$. When unit productions are deleted, we are left with

$$S \rightarrow S + T \mid T * F \mid (S) \mid a$$
$$T \rightarrow T * F \mid (S) \mid a$$
$$F \rightarrow (S) \mid a$$

In addition to eliminating specific types of productions, such as Λ-productions and unit productions, it may also be useful to impose restrictions upon the form of the remaining productions. Several types of "normal forms" have been introduced; we shall present one of them, the Chomsky normal form.

Definition 6.7 Chomsky Normal Form

A context-free grammar is in *Chomsky normal form* (CNF) if every production is of one of these two types:

$$A \rightarrow BC$$
$$A \rightarrow a$$

where A, B, and C are variables and a is a terminal symbol.

Transforming a grammar $G = (V, \Sigma, S, P)$ into Chomsky normal form may be done in three steps. The first is to apply Algorithms 6.1 and 6.2 to obtain a CFG $G1 = (V, \Sigma, S, P1)$ having neither Λ-productions nor unit productions so that $L(G1) = L(G) - \{\Lambda\}$. The second step is to obtain a grammar $G2 = (V2, \Sigma, S, P2)$, generating the same language as $G1$, so that every production in $P2$ is either of the form

$$A \to B_1 B_2 \cdots B_k$$

where $k \geq 2$ and each B_i is a variable in $V2$, or of the form

$$A \to a$$

for some $a \in \Sigma$.

The construction of $G2$ is very simple. Since $P1$ contains no Λ-productions or unit productions, every production in $P1$ that is not already of the form $A \to a$ looks like $A \to \alpha$ for some string α of length at least 2. For every terminal a appearing in such a string α, we introduce a new variable X_a and a new production $X_a \to a$, and replace a by X_a in all the productions where it appears (except those of the form $A \to a$).

For example, if there were two productions $A \to aAb$ and $B \to ab$, they would be replaced by $A \to X_a A X_b$ and $B \to X_a X_b$, and the productions $X_a \to a$ and $X_b \to b$ would be added. The only X_a-production is $X_a \to a$, and so it is reasonably clear that $G2$ is equivalent to $G1$.

The grammar $G2$ obtained this way now resembles a grammar in CNF, in the sense that the right side of every production is either a single terminal or a string of two or more variables. The last step is to replace each production having more than two variables on the right by an equivalent set of productions, each one having exactly two variables on the right. This process is described best by an example; the production

$$A \to BCDBCE$$

would be replaced by

$$\begin{aligned} A &\to BY_1 \\ Y_1 &\to CY_2 \\ Y_2 &\to DY_3 \\ Y_3 &\to BY_4 \\ Y_4 &\to CE \end{aligned}$$

The new variables Y_1, Y_2, Y_3, Y_4 are specific to this production and would be used nowhere else. Although this may seem wasteful, in terms of the number of variables, at least there is no doubt that the combined effect of this set of five productions is precisely equivalent to the original production. Adding these new variables and productions therefore does not change the language generated.

If we are willing to let these informal arguments suffice, we have obtained the following result.

Theorem 6.7

For any context-free grammar $G = (V, \Sigma, S, P)$, there is a CFG $G' = (V', \Sigma, S, P')$ in Chomsky normal form so that $L(G') = L(G) - \{\Lambda\}$.

Converting a CFG to Chomsky Normal Form

EXAMPLE 6.16

Let G be the grammar with productions

$$\begin{aligned} S &\to AACD \\ A &\to aAb \mid \Lambda \\ C &\to aC \mid a \\ D &\to aDa \mid bDb \mid \Lambda \end{aligned}$$

Let us go through each step of the conversion to CNF.

1. **Eliminating Λ-productions.** The nullable variables are A and D, and Algorithm 6.1 produces the grammar with productions

$$\begin{aligned} S &\to AACD \mid ACD \mid AAC \mid CD \mid AC \mid C \\ A &\to aAb \mid ab \\ C &\to aC \mid a \\ D &\to aDa \mid bDb \mid aa \mid bb \end{aligned}$$

2. **Eliminating unit productions.** Here we may simply add the productions

$$S \to aC \mid a$$

and delete $S \to C$.

3. **Restricting the right sides of productions to single terminals or strings of two or more variables.** This step yields the productions

$$\begin{aligned} S &\to AACD \mid ACD \mid AAC \mid CD \mid AC \mid X_aC \mid a \\ A &\to X_aAX_b \mid X_aX_b \\ C &\to X_aC \mid a \\ D &\to X_aDX_a \mid X_bDX_b \mid X_aX_a \mid X_bX_b \\ X_a &\to a \\ X_b &\to b \end{aligned}$$

4. **The final step to CNF.** There are six productions whose right sides are too long. Applying our algorithm produces the grammar with productions

$$\begin{aligned} &S \to AT_1 \quad T_1 \to AT_2 \quad T_2 \to CD \\ &S \to AU_1 \quad U_1 \to CD \\ &S \to AV_1 \quad V_1 \to AC \end{aligned}$$

$$S \to CD \mid AC \mid X_aC \mid a$$
$$A \to X_aW_1 \qquad W_1 \to AX_b$$
$$A \to X_aX_b$$
$$C \to X_aC \mid a$$
$$D \to X_aY_1 \qquad Y_1 \to DX_a$$
$$D \to X_bZ_1 \qquad Z_1 \to DX_b$$
$$D \to X_aX_b \mid X_bX_b$$
$$X_a \to a \qquad X_b \to b$$

EXERCISES

6.1. In each case, say what language is generated by the context-free grammar with the indicated productions.

a.
$$S \to aSa \mid bSb \mid \Lambda$$

b.
$$S \to aSa \mid bSb \mid a \mid b$$

c.
$$S \to aSb \mid bSa \mid \Lambda$$

d.
$$S \to aSa \mid bSb \mid aAb \mid bAa$$
$$A \to aAa \mid bAb \mid a \mid b \mid \Lambda$$

(See Example 6.3.)

e.
$$S \to aS \mid bS \mid a$$

f.
$$S \to SS \mid bS \mid a$$

g.
$$S \to SaS \mid b$$

h.
$$S \to aT \mid bT \mid \Lambda$$
$$T \to aS \mid bS$$

6.2. Find a context-free grammar corresponding to the "syntax diagram" in Figure 6.6.

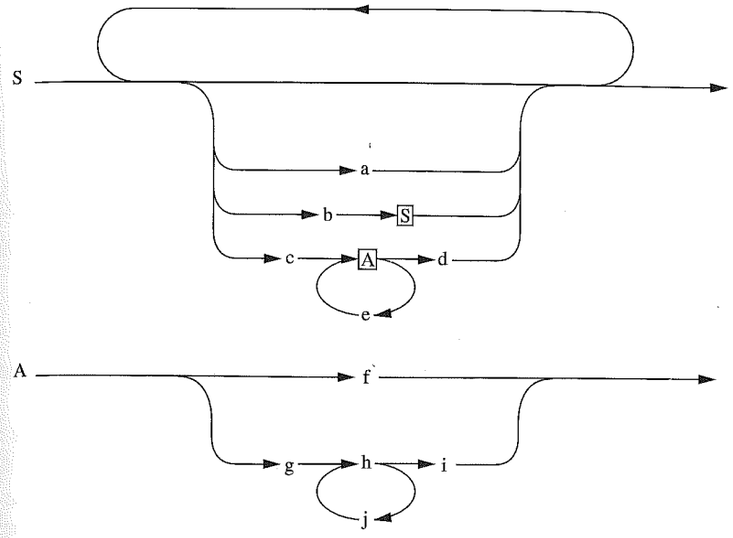


Figure 6.6 |

6.3. A context-free grammar is sometimes specified in the form of BNF rules; the letters are an abbreviation for Backus-Naur Form. In these rules, the symbol ::= corresponds to the usual $\to$, and $\{X\}$ means zero or more occurrences of X. Find a context-free grammar corresponding to the BNF rules shown below. Uppercase letters denote variables, lowercase denote terminals.

$$P ::= p\ I;\ \mid p\ I\ (F\ \{;\ F\})$$
$$F ::= G \mid v\ G \mid f\ G \mid p\ I\ \{,\ I\}$$
$$G ::= I\ \{,\ I\} :\ I$$
$$I ::= L\ \{L_1\}$$
$$L_1 ::= L \mid D$$
$$L ::= a \mid b$$
$$D ::= 0 \mid 1$$

6.4. In each case, find a CFG generating the given language.

a. The set of odd-length strings in $\{a, b\}^*$ with middle symbol a.

b. The set of even-length strings in $\{a, b\}^*$ with the two middle symbols equal.

c. The set of odd-length strings in $\{a, b\}^*$ whose first, middle, and last symbols are all the same.

6.5. In each case, the productions in a CFG are given. Prove that neither one generates the language $L = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$.

a.

$$S \to S01S \mid S10S \mid \Lambda$$

b.

$$S \to 0S1 \mid 1S0 \mid 01S \mid 10S \mid S01 \mid S10 \mid \Lambda$$

6.6. Consider the CFG with productions

$$S \to aSbScS \mid aScSbS \mid bSaScS \mid bScSaS \mid cSaSbS \mid cSbSaS \mid \Lambda$$

Does this generate the language $\{x \in \{a, b, c\}^* \mid n_a(x) = n_b(x) = n_c(x)\}$? Prove your answer.

6.7. Find a context-free grammar generating the language of all regular expressions over an alphabet Σ:

a. If the definition of regular expression is interpreted strictly with regard to parentheses.

b. If the definition is interpreted so as to allow regular expressions that are not "fully parenthesized."

Be careful to distinguish between Λ-productions and productions whose right side is the symbol Λ appearing in a regular expression; use λ in the second case.

6.8. This problem gives proposed alternative constructions for the CFGs G_u, G_c, and G^* in Theorem 6.1. In each case, either prove that the construction works, or give an example of grammars for which it doesn't and say why it doesn't.

a. (For G_u) $V_u = V_1 \cup V_2$; $S_u = S_1$; $P_u = P_1 \cup P_2 \cup \{S_1 \to S_2\}$

b. (For G_c) $V_c = V_1 \cup V_2$; $S_c = S_1$; $P_c = P_1 \cup P_2 \cup \{S_1 \to S_1S_1\}$

c. (For G^*) $V = V_1$; $S = S_1$; $P = P_1 \cup \{S_1 \to S_1S_1 \mid \Lambda\}$

6.9. Find context-free grammars generating each of these languages.

a. $\{a^i b^j c^k \mid i = j + k\}$

b. $\{a^i b^j c^k \mid j = i + k\}$

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

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

e. $\{a^i b^j c^k \mid i < j \text{ or } i > k\}$

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

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

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

6.10. Describe the language generated by each of these grammars.

a. The regular grammar with productions

$$S \to aA \mid bC \mid b$$
$$A \to aS \mid bB$$
$$B \to aC \mid bA \mid a$$
$$C \to aB \mid bS$$

b. The grammar with productions

$$S \to bS \mid aA \mid \Lambda$$
$$A \to aA \mid bB \mid b$$
$$B \to bS$$

6.11. Show that for a language $L \subseteq \Sigma^*$ such that $\Lambda \notin L$, the following statements are equivalent.

a. L is regular.

b. L can be generated by a grammar in which all productions are either of the form $A \to xB$ or of the form $A \to x$ (where A and B are variables and $x \in \Sigma^+$).

c. L can be generated by a grammar in which all productions are either of the form $A \to Bx$ or of the form $A \to x$ (where A and B are variables and $x \in \Sigma^+$).

6.12. Show that for any language $L \subseteq \Sigma^*$, the following statements are equivalent.

a. L is regular.

b. L can be generated by a grammar in which all productions are either of the form $A \to xB$ or of the form $A \to x$ (where A and B are variables and $x \in \Sigma^*$).

c. L can be generated by a grammar in which all productions are either of the form $A \to Bx$ or of the form $A \to x$ (where A and B are variables and $x \in \Sigma^*$).

6.13. Given the FA shown in Figure 6.7, accepting the language L, find a regular grammar generating $L - \{\Lambda\}$.

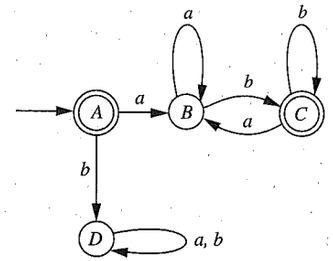


Figure 6.7 |

6.14. Draw an NFA accepting the language generated by the grammar with productions

$$S \to abA \mid bB \mid aba$$
$$A \to b \mid aB \mid bA$$
$$B \to aB \mid aA$$

6.15. Show that if the procedure described in the proof of Theorem 6.2 is applied to an NFA instead of an FA, the result is still a regular grammar generating the language accepted by the NFA.

6.16. Consider the following statement. For any language $L \subseteq \Sigma^*$, L is regular if and only if L can be generated by some grammar in which every production takes one of the four forms $B \to a$, $B \to Ca$, $B \to aC$, or $B \to \Lambda$, where B and C are variables and $a \in \Sigma$. For both the "if" and the "only if" parts, give either a proof or a counterexample.

6.17. A context-free grammar is said to be *self-embedding* if there is some variable A and two nonnull strings of terminals α and β so that $A \Rightarrow^* \alpha A \beta$. Show that a language L is regular if and only if it can be generated by a grammar that is not self-embedding.

6.18. Each of the following grammars, though not regular, generates a regular language. In each case, find a regular grammar generating the language.

a. $S \to SSS \mid a \mid ab$

b. $S \to AabB \qquad A \to aA \mid bA \mid \Lambda \qquad B \to Bab \mid Bb \mid ab \mid b$

c. $S \to AAS \mid ab \mid aab \qquad A \to ab \mid ba \mid \Lambda$

d. $S \to AB \qquad A \to aAa \mid bAb \mid a \mid b \qquad B \to aB \mid bB \mid \Lambda$

e. $S \to AA \mid B \qquad A \to AAA \mid Ab \mid bA \mid a \qquad B \to bB \mid b$

6.19. Refer to Example 6.4.

a. Draw the derivation tree corresponding to each of the two given derivations of $a + (a * a)/a - a$.

b. Write the rightmost derivation corresponding to each of the trees in (a).

c. How many distinct leftmost derivations of this string are there?

d. How many derivation trees are there for the string $a + a + a + a + a$?

e. How many derivation trees are there for the string $(a + (a + a)) + (a + a)$?

6.20. Give an example of a CFG and a string of variables and/or terminals derivable from the start symbol for which there is neither a leftmost derivation nor a rightmost derivation.

6.21. Consider the C statements

```
x = 1; if (a > 2) if (a > 4) x = 2; else x = 3;
```

a. What is the resulting value of x if a = 3? If a = 1?

b. Same question as in (a), but this time assume that the statement is interpreted as in Figure 6.5*a*.

6.22. Show that the CFG with productions

$$S \to a \mid Sa \mid bSS \mid SSb \mid SbS$$

is ambiguous.

6.23. Consider the context-free grammar with productions

$$S \to AB \qquad A \to aA \mid \Lambda \qquad B \to ab \mid bB \mid \Lambda$$

Any derivation of a string in this grammar must begin with the production $S \to AB$. Clearly, any string derivable from A has only one derivation from A, and likewise for B. Therefore, the grammar is unambiguous. True or false? Why? (Compare with the proof of Theorem 6.4.)

6.24. In each part of Exercise 6.1, decide whether the grammar is ambiguous or not, and prove your answer.

6.25. For each of the CFGs in Examples 6.3, 6.9, and 6.11, determine whether or not the grammar is ambiguous, and prove your answer.

6.26. In each case, show that the grammar is ambiguous, and find an equivalent unambiguous grammar.

a. $S \to SS \mid a \mid b$

b. $S \to ABA \qquad A \to aA \mid \Lambda \qquad B \to bB \mid \Lambda$

c. $S \to A \mid B \qquad A \to aAb \mid ab \qquad B \to abB \mid \Lambda$

d. $S \to aSb \mid aaSb \mid \Lambda$

e. $S \to aSb \mid abS \mid \Lambda$

6.27. Find an unambiguous context-free grammar equivalent to the grammar with productions

$$S \to aaaaS \mid aaaaaaaS \mid \Lambda$$

(See Exercise 2.50.)

6.28. The proof of Theorem 6.1 shows how to find a regular grammar generating L, given a finite automaton accepting L.

a. Under what circumstances is the grammar obtained this way unambiguous?

b. Describe how the grammar can be modified if necessary in order to make it unambiguous.

6.29. Describe an algorithm for starting with a regular grammar and finding an equivalent unambiguous grammar.

6.30. Show that every left parenthesis in a balanced string has a mate.

6.31. Show that if a is a left parenthesis in a balanced string, and b is its mate, then a is the last left parentheses for which the string consisting of a and b and everything in between is balanced.

6.32. Find an unambiguous context-free grammar for the language of all algebraic expressions involving parentheses, the identifier a, and the four binary operators $+$, $-$, $*$, and $/$.

6.33. Show that the nullable variables defined by Definition 6.6 are precisely those variables A for which $A \Rightarrow^* \Lambda$.

6.34. In each case, find a context-free grammar with no Λ-productions that generates the same language, except possibly for Λ, as the given CFG.

a.

$$S \to AB \mid \Lambda \qquad A \to aASb \mid a \qquad B \to bS$$

b.

$$S \to AB \mid ABC$$
$$A \to BA \mid BC \mid \Lambda \mid a$$
$$B \to AC \mid CB \mid \Lambda \mid b$$
$$C \to BC \mid AB \mid A \mid c$$

6.35. In each case, given the context-free grammar G, find a CFG G' with no Λ-productions and no unit productions that generates the language $L(G) - \{\Lambda\}$.

a. G has productions

$$S \to ABA \qquad A \to aA \mid \Lambda \qquad B \to bB \mid \Lambda$$

b. G has productions

$$S \to aSa \mid bSb \mid \Lambda \qquad A \to aBb \mid bBa \qquad B \to aB \mid bB \mid \Lambda$$

c. G has productions

$$S \to A \mid B \mid C \qquad A \to aAa \mid B \qquad B \to bB \mid bb$$
$$C \to aCaa \mid D \qquad D \to baD \mid abD \mid aa$$

6.36. A variable A in a context-free grammar $G = (V, \Sigma, S, P)$ is *live* if $A \Rightarrow^* x$ for some $x \in \Sigma^*$. Give a recursive definition, and a corresponding algorithm, for finding all live variables in G.

6.37. A variable A in a context-free grammar $G = (V, \Sigma, S, P)$ is *reachable* if $S \Rightarrow^* \alpha A \beta$ for some $\alpha, \beta \in (\Sigma \cup V)^*$. Give a recursive definition, and a corresponding algorithm, for finding all reachable variables in G.

6.38. A variable A is a context-free grammar $G = (V, \Sigma, S, P)$ is *useful* if for some string $x \in \Sigma^*$, there is a derivation of x that takes the form

$$S \Rightarrow^* \alpha A \beta \Rightarrow^* x$$

A variable that is not useful is *useless*. Clearly if a variable is either not live or not reachable (Exercises 6.36–6.37), then it is useless.

a. Give an example in which a variable is both live and reachable but still useless.

b. Let G be a CFG. Suppose $G1$ is obtained by eliminating all dead variables from G and eliminating all productions in which dead variables appear. Suppose $G2$ is then obtained from $G1$ by eliminating all variables unreachable in $G1$, as well as productions in which such variables appear. Show that $G2$ contains no useless variables, and $L(G2) = L(G)$.

c. Show that if the two steps are done in the opposite order, the resulting grammar may still have useless variables.

d. In each case, given the context-free grammar G, find an equivalent CFG with no useless variables.

i. G has productions

$$S \to ABC \mid BaB \qquad A \to aA \mid BaC \mid aaa$$
$$B \to bBb \mid a \qquad C \to CA \mid AC$$

ii. G has productions

$$S \to AB \mid AC \qquad A \to aAb \mid bAa \mid a \qquad B \to bbA \mid aaB \mid AB$$
$$C \to abCa \mid aDb \qquad D \to bD \mid aC$$

6.39. In each case, given the context-free grammar G, find a CFG G' in Chomsky normal form generating $L(G) - \{\Lambda\}$.

a. G has productions $S \to SS \mid (S) \mid \Lambda$

b. G has productions $S \to S(S) \mid \Lambda$

c. G is the CFG in Exercise 6.35c

d. G has productions

$$S \to AaA \mid CA \mid BaB \qquad A \to aaBa \mid CDA \mid aa \mid DC$$
$$B \to bB \mid bAB \mid bb \mid aS \qquad C \to Ca \mid bC \mid D \qquad D \to bD \mid \Lambda$$

6.40. If G is a context-free grammar in Chomsky normal form and $x \in L(G)$ with $|x| = k$, how many steps are there in a derivation of x in G?

MORE CHALLENGING PROBLEMS

6.41. Describe the language generated by the CFG with productions

$$S \to aS \mid aSbS \mid \Lambda$$

One way to understand this language is to replace a and b by left and right parentheses, respectively. However, the language can also be characterized by giving a property that every prefix of a string in the language must have.

6.42. Show that the language of all nonpalindromes over $\{a, b\}$ (see Example 6.3) cannot be generated by any CFG in which $S \to aSa \mid bSb$ are the only productions with variables on the right side.

6.43. Show using mathematical induction that every string produced by the context-free grammar with productions

$$S \to 0 \mid S0 \mid 0S \mid 1SS \mid SS1 \mid S1S$$

has more 0's than 1's.

6.44. Complete the proof in Example 6.10 that every string in $\{0, 1\}^*$ with more 0's than 1's can be generated by the CFG with productions $S \to 0 \mid 0S \mid 1SS \mid SS1 \mid S1S$. (Take care of the two remaining cases.)

6.45. Let L be the language generated by the CFG with productions

$$S \to aSb \mid ab \mid SS$$

Show using mathematical induction that no string in L begins with abb.

6.46. Describe the language generated by the CFG with productions

$$S \to ST \mid \Lambda \qquad T \to aS \mid bT \mid b$$

Prove that your answer is correct.

6.47. Show that the context-free grammar with productions

$$S \to bS \mid aT \mid \Lambda$$
$$T \to aT \mid bU \mid \Lambda$$
$$U \to aT \mid \Lambda$$

generates the language of all strings over the alphabet $\{a, b\}$ that do not contain the substring abb. One approach is to use mathematical induction to prove two three-part statements. In both cases, each part starts with "For every $n \geq 0$, if x is any string of length n,". In the first statement, the three parts end as follows: (i) if $S \Rightarrow^* x$, then x does not contain the substring abb; (ii) if $T \Rightarrow^* x$, then x does not contain the substring bb; (iii) if $U \Rightarrow^* x$, then x does not start with b and does not contain the substring bb. In the second statement, the three parts end with the converses of (i), (ii), and (iii). The reason for using two three-part statements, rather than six separate statements, is that in proving each of the two, the induction hypothesis will say something about all three types of strings: those derivable from S, those derivable from T, and those derivable from U.

6.48. What language over $\{0, 1\}$ does the CFG with productions

$$S \to 00S \mid 11S \mid S00 \mid S11 \mid 01S01 \mid 01S10 \mid 10S10 \mid 10S01 \mid \Lambda$$

generate? Prove your answer.

6.49. Complete the proof of Theorem 6.3, by taking care of the two remaining cases in the first part of the proof.

6.50. Show using mathematical induction that the CFG with productions

$$S \to 0B \mid 1A \mid \Lambda$$
$$A \to 0S \mid 1AA$$
$$B \to 1S \mid 0BB$$

generates the language $L = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$ (See Example 6.8) It would be appropriate to formulate two three-part statements, as in Exercise 6.47, this time involving the variables S, A, and B and the languages L, L_0, and L_1.

6.51. Prove that the CFG with productions $S \to 0S1S \mid 1S0S \mid \Lambda$ generates the language $L = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$.

6.52. a. Describe the language generated by the CFG G with productions

$$S \to SS \mid (S) \mid \Lambda$$

b. Show that the CFG G_1 with productions

$$S_1 \to (S_1)S_1 \mid \Lambda$$

generates the same language. (One inclusion is easy. For the other one, it may be helpful to prove the following statements for a string $x \in L(G)$ with $|x| > 0$. First, if there is no derivation of x beginning with the production $S \to (S)$, then there are strings y and z, both in $L(G)$ and both shorter than x, for which $x = yz$. Second, if there are such strings y and z, and if there are no other such strings y' and z' with y' shorter than y, then there is a derivation of y in G that starts with the production $S \to (S)$.)

6.53. Show that the CFG with productions

$$S \to aSaSbS \mid aSbSaS \mid bSaSaS \mid \Lambda$$

generates the language $\{x \in \{a, b\}^* \mid n_a(x) = 2n_b(x)\}$.

6.54. Does the CFG with productions

$$S \to aSaSb \mid aSbSa \mid bSaSaS \mid \Lambda$$

generate the language of the previous problem? Prove your answer.

6.55. Show that the following CFG generates the language $\{x \in \{a, b\}^* \mid n_a(x) = 2n_b(x)\}$.

$$S \to SS \mid bTT \mid TbT \mid TTb \mid \Lambda \qquad T \to aS \mid SaS \mid Sa \mid a$$

6.56. For alphabets Σ_1 and Σ_2, a *homomorphism* from Σ_1^* to Σ_2^* is defined in Exercise 4.46. Show that if $f : \Sigma_1^* \to \Sigma_2^*$ is a homomorphism and $L \subseteq \Sigma_1^*$ is a context-free language, then $f(L) \subseteq \Sigma_2^*$ is also a CFG.

6.57. Show that the CFG with productions

$$S \to S(S) \mid \Lambda$$

is unambiguous.

6.58. Find context-free grammars generating each of these languages.

a. $\{a^i b^j c^k \mid i \neq j + k\}$

b. $\{a^i b^j c^k \mid j \neq i + k\}$

6.59. Find context-free grammars generating each of these languages, and prove that your answers are correct.

a. $\{a^i b^j \mid i \leq j \leq 3i/2\}$

b. $\{a^i b^j \mid i/2 \leq j \leq 3i/2\}$

6.60. Let G be the context-free grammar with productions

$$S \to aS \mid aSbS \mid c$$

and let G_1 be the one with productions

$$S_1 \to T \mid U \qquad T \to aTbT \mid c \qquad U \to aS_1 \mid aTbU$$

(G_1 is a simplified version of the second grammar in Example 6.13.)

a. Show that G is ambiguous.

b. Show that G and G_1 generate the same language.

c. Show that G_1 is unambiguous.

6.61. Let x be a string of left and right parentheses. A *complete pairing* of x is a partition of the parentheses of x into pairs such that (i) each pair consists of one left parenthesis and one right parenthesis appearing somewhere after it; and (ii) the parentheses *between* those in a pair are themselves the union of pairs. Two parentheses in a pair are said to be *mates* with respect to that pairing.

a. Show that there is at most one complete pairing of a string of parentheses.

b. Show that a string of parentheses has a complete pairing if and only if it is a balanced string, according to Definition 6.5, and in this case the two definitions of mates coincide.

6.62. Give a proof of Theorem 6.6. Suggestion: in order to show that $L(G) \subseteq L(G1)$, show that for every $n \geq 1$, and every string of variables and/or terminals that can be derived from S in G by an n-step leftmost derivation in which the last step is not a unit production can be derived from S in $G1$.

6.63. Show that if a context-free grammar is unambiguous, then the grammar obtained from it by Algorithm 6.1 is also.

6.64. Show that if a context-free grammar with no Λ-productions is unambiguous, then the one obtained from it by Algorithm 6.2 is also.

CHAPTER

7

Pushdown Automata

7.1 INTRODUCTION BY WAY OF AN EXAMPLE

In this chapter we investigate how to extend our finite-state model of computation so that we can recognize context-free languages. In our first example, we consider one of the simplest nonregular context-free languages. Although the abstract machine we describe is not obviously related to a CFG generating the language, we will see later that a machine of the same general type can be used with any CFL, and that one can be constructed very simply from a grammar.

An Abstract Machine to Accept Simple Palindromes — EXAMPLE 7.1

Let G be the context-free grammar having productions

$$S \rightarrow aSa \mid bSb \mid c$$

G generates the language

$$L = \{xcx^r \mid x \in \{a, b\}^*\}$$

The strings in L are odd-length palindromes over $\{a, b\}$ (Example 6.3), except that the middle symbol is c. (We will consider ordinary palindromes shortly. For now, the "marker" in the middle makes it easier to recognize the string.)

It is not hard to design an algorithm for recognizing strings in L, using a single left-to-right pass. We will save the symbols in the first half of the string as we read them, so that once we encounter the c we can begin matching incoming symbols with symbols already read. In order for this to work, we must retrieve the symbols we have saved using the rule "last in, first out" (often abbreviated LIFO): The symbol used to match the next incoming symbol is the one most recently read, or saved. The data structure incorporating the LIFO rule is a *stack*, which