CHAPTER

4

Context-Free Languages

Regular languages and finite automata are too simple and too restrictive to be able to handle languages that are at all complex. Using *context-free grammars* allows us to generate more interesting languages; much of the syntax of a high-level programming language, for example, can be described this way. In this chapter we start with the definition of a context-free grammar and look at a number of examples. A particularly simple type of context-free grammar provides another way to describe the regular languages we discussed in Chapters 2 and 3. Later in the chapter we study derivations in a context-free grammar, how a derivation might be related to the structure of the string being derived, and the presence of ambiguity in a grammar, which can complicate this relationship. We also consider a few ways that a grammar might be simplified to make it easier to answer questions about the strings in the corresponding language.

4.1 USING GRAMMAR RULES TO DEFINE A LANGUAGE

The term *grammar* applied to a language like English refers to the rules for constructing phrases and sentences. For us a grammar is also a set of rules, simpler than the rules of English, by which strings in a language can be generated. In the first few examples in this section, a grammar is another way to write the recursive definitions that we used in Chapter 1 to define languages.

EXAMPLE 4.1 The language *AnBn*

In Example 1.18, we defined the language $AnBn = \{a^n b^n \mid n \geq 0\}$ using this recursive definition:

1. $\Lambda \in AnBn$.
2. for every $S \in AnBn$, $aSb \in AnBn$.

Let us think of S as a *variable*, representing an arbitrary string in *AnBn*. Rule 1, which we rewrite as $S \to \Lambda$, says that the arbitrary string could simply be Λ, obtained by substituting Λ for the variable S. To obtain any other string, we must begin with rule 2, which we write $S \to aSb$. This rule says that a string in *AnBn* can have the form aSb (or that S can be replaced by aSb), where the new occurrence of S represents some other element of *AnBn*. Replacing S by aSb is the first step in a *derivation* of the string, and the remaining steps will be further applications of rules 1 and 2 that will give a value to the new S. The derivation will continue as long as the string contains the variable S, so that in this example the last step will always be to replace S by Λ.

If α and β are strings, and α contains at least one occurrence of S, the notation

$$\alpha \Rightarrow \beta$$

will mean that β is obtained from α by using one of the two rules to replace a single occurrence of S by either Λ or aSb. Using this notation, we would write

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

to describe the sequence of steps (three applications of rule 2, then one application of rule 1) we used to derive the string $aaabbb \in AnBn$.

The notation is simplified further by writing rules 1 and 2 as

$$S \to \Lambda \mid aSb$$

and interpreting | as "or". When we write the rules of a grammar this way, we give concatenation higher precedence than |, which means in our example that the two alternatives in the formula $S \to \Lambda \mid aSb$ are Λ and aSb, not Λ and a.

The Language *Expr* **EXAMPLE 4.2**

In Example 1.19 we considered the language *Expr* of algebraic expressions involving the binary operators $+$ and $*$, left and right parentheses, and a single identifier a. We used the following recursive definition to describe *Expr*:

1. $a \in Expr$.
2. For every x and y in *Expr*, $x + y$ and $x * y$ are in *Expr*.
3. For every $x \in Expr$, $(x) \in Expr$.

Rule 2 involves two different letters x and y, because the two expressions being combined with either $+$ or $*$ are not necessarily the same. However, they both represent elements of *Expr*, and we still need only one variable in the grammar rules corresponding to the recursive definition:

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

To derive the string $a + (a * a)$, for example, we could use the sequence of steps

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + (S) \Rightarrow a + (S * S) \Rightarrow a + (a * S) \Rightarrow a + (a * a)$$

The string $S + S$ that we obtain from the first step of this derivation suggests that the final expression should be interpreted as the sum of two subexpressions. The subexpressions we

end up with are obviously not the same, but they are both elements of *Expr* and can therefore both be derived from S.

A derivation of an expression in *Expr* is related to the way we choose to interpret the expression, and there may be several possible choices. The expression $a + a * a$, for example, has at least these two 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 first steps in these two derivations yield the strings $S + S$ and $S * S$, respectively. The first derivation suggests that the expression is the sum of two subexpressions, the second that it is the product of two subexpressions.

The rules of precedence normally adopted for algebraic expressions say, among other things, that multiplication has higher precedence than addition. If we adopt this precedence rule, then when we evaluate the expression $a + a * a$, we first evaluate the product $a * a$, so that we interpret the expression as a sum, not a product. Because there is nothing in our grammar rules to suggest that the first derivation is preferable to the second, one possible conclusion is that it might be better to use another grammar, in which every string has essentially only one derivation. We will return to this question in Section 4.4, when we discuss *ambiguity* in a grammar.

We have made the language *Expr* simple by restricting the expressions in several ways. We could easily add grammar rules to allow other operations besides $+$ and $*$; if we wanted to allow other "atomic" expressions besides the identifier a (more general identifiers, or numeric literals such as 16 and 1.3E−2, or both), we could add another variable, A, to get

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

and then look for grammar rules beginning with A that would generate all the subexpressions we wanted. The next example involves another language L for which grammar rules generating L require more than one variable.

EXAMPLE 4.3 Palindromes and Nonpalindromes

We see from Example 1.18 that the language *Pal* of palindromes over the alphabet $\{a, b\}$ can be generated by the grammar rules

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

What about its complement *NonPal*? The last two grammar rules in the definition of *Pal* still seem to work: For every nonpalindrome x, both axa and bxb are also nonpalindromes. But a recursive definition of *NonPal* cannot be as simple as the one for *Pal*, because there is no finite set of strings comparable to $\{\Lambda, a, b\}$ that can serve as basis elements in the definition (see Exercise 4.6).

To find the crucial feature of a nonpalindrome, let's look at one, say

$$x = abbbbaaba$$

The string x is $abyba$, where $y = bbbaa$. Working our way in from the ends, comparing the symbol on the left with the one on the right, we wouldn't know until we got to y whether x was a palindrome or not, but once we saw that y looked like bza for some string z, it would be clear, even without looking at any symbols of z. Here is a definition of *NonPal*:

1. For every $A \in \{a, b\}^*$, aAb and bAa are elements of *NonPal*;
2. For every S in *NonPal*, aSa and bSb are in *NonPal*.

In order to obtain grammar rules generating *NonPal*, we can introduce A as a second variable, representing an arbitrary element of $\{a, b\}^*$, and use grammar rules starting with A that correspond to the recursive definition in Example 1.17. The complete set of rules for *NonPal* is

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

A derivation of $abbbbaaba$ in this grammar is

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbAaba$$
$$\Rightarrow abbAaaba \Rightarrow abbAbaaba \Rightarrow abbAbbaaba \Rightarrow abbbbaaba$$

In order to generate a language L using the kinds of grammars we are discussing, it is often necessary to include several variables. The *start* variable is distinguished from the others, and we will usually denote it by S. Each remaining variable can be thought of as representing an arbitrary string in some auxiliary language involved in the definition of L (the language of all strings that can be derived from that variable). We can still interpret the grammar as a recursive definition of L, except that we must extend our notion of recursion to include *mutual* recursion: rather than one object defined recursively in terms of itself, several objects defined recursively in terms of each other.

English and Programming-Language Syntax **EXAMPLE 4.4**

You can easily see how grammar rules can be used to describe simple English syntax. Many useful sentences can be generated by the rule

<declarative sentence> → <subject phrase> <verb phrase> <object>

provided that reasonable rules are found for each of the three variables on the right. Three examples are "haste makes waste", "the ends justify the means", and "we must extend our notion" (from the last sentence in Example 4.3). You can also see how difficult it would be to find grammars of a reasonable size that would allow more sophisticated sentences without also allowing gibberish. (Try to formulate some rules to generate the preceding sentence, "You can also see how ... gibberish". Unless your approach is to provide almost as many rules as sentences, the chances are that the rules will also generate strings that aren't really English sentences.)

The syntax of programming languages is much simpler. Two types of statements in C are *if* statements and *for* statements.

<statement> → ... | <*if* − statement> | <*for* − statement>

Assuming we are using "if statements" to include both those with *else* and those without, we might write

$$\begin{aligned} <\textit{if}-\text{statement}> &\rightarrow \text{if (<expr>) <statement> } | \\ &\quad\ \text{if (<expr>) <statement> else <statement>} \\ <\textit{for}-\text{statement}> &\rightarrow \text{for (<expr>; <expr>; <expr>) <statement>} \end{aligned}$$

where <expr> is another variable, for which the productions would need to be described.

The logic of a program often requires that a "statement" include several statements. We can define a compound statement as follows:

$$\begin{aligned} &\text{<compound statement>} \rightarrow \text{\{ <statement-sequence> \}} \\ &\text{<statement-sequence>} \rightarrow \Lambda \mid \text{<statement> <statement-sequence>} \end{aligned}$$

A *syntax diagram* such as the one in Figure 4.5 accomplishes the same thing.

Figure 4.5 |

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

4.2 | CONTEXT-FREE GRAMMARS: DEFINITIONS AND MORE EXAMPLES

Definition 4.6 Context-Free Grammars

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

Elements of Σ are called *terminal symbols*, or *terminals*, and elements of V are *variables*, or *nonterminals*. S is the *start* variable, and elements of P are *grammar rules*, or *productions*.

As in Section 4.1, we will reserve the symbol $\rightarrow$ for productions in a grammar, and we will use $\Rightarrow$ for a step in a derivation. The notations

$$\alpha \Rightarrow^n \beta \qquad \text{and} \qquad \alpha \Rightarrow^* \beta$$

refer to a sequence of n steps and a sequence of zero or more steps, respectively, and we sometimes write

$$\alpha \Rightarrow_G \beta \qquad \text{or} \qquad \alpha \Rightarrow_G^n \beta \qquad \text{or} \qquad \alpha \Rightarrow_G^* \beta$$

to indicate explicitly that the steps involve productions in the grammar G. If $G = (V, \Sigma, S, P)$, the first statement means that there are strings α_1, α_2, and γ in $(V \cup \Sigma)^*$ and a production $A \to \gamma$ in P such that

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

In other words, β can be obtained from α in one step by applying the production $A \to \gamma$. Whenever there is no chance of confusion, we will drop the subscript G.

In the situation we have just described, in which $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$, the formula $\alpha \Rightarrow \beta$ represents a step in a derivation; if our definition of productions allowed $\alpha \to \beta$ to be a production, we might say that the variable A could be replaced by γ, depending on its *context*—i.e., depending on the values of α_1 and α_2. What makes a context-free grammar context-free is that the left side of a production is a single variable and that we may apply the production to any string containing that variable, independent of the context. In Chapter 8 we will consider grammars, and productions, that are not context-free.

Definition 4.7 The Language Generated by a CFG

If $G = (V, \Sigma, S, P)$ is 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 with $L = L(G)$.

EXAMPLE 4.8 The Language *AEqB*

Exercises 1.65 and 4.16 both allow us to find context-free grammars for the language $AEqB = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$ that use only the variable S. In this example we consider a grammar with three variables that is based on a definition involving mutual recursion.

If x is a nonnull string in $AEqB$, then either $x = ay$, where $y \in L_b = \{z \mid n_b(z) = n_a(z) + 1\}$, or $x = by$, where $y \in L_a = \{z \mid n_a(z) = n_b(z) + 1\}$. Let us use the variables A and B to represent L_a and L_b, respectively, and try to find a CFG for $AEqB$ that involves the three variables S, A, and B. So far, the appropriate productions are

$$S \to \Lambda \mid aB \mid bA$$

All we need are productions starting with A and B.

If a string x in L_a starts with a, then the remaining substring is an element of $AEqB$. What if it starts with b? Then $x = by$, where y has two more a's than b's. The crucial observation here is that every string y having two more a's than b's must be the concatenation of two strings y_1 and y_2, each with *one* more a. To see this, think about the relative numbers of a's and b's in each prefix of y; specifically, for each prefix z, let $d(z) = n_a(z) - n_b(z)$.

The shortest prefix of y is Λ, and $d(\Lambda)$ is obviously 0; the longest prefix is y itself, and $d(y) = 2$ by assumption. Each time we add a single symbol to a prefix, the d-value changes (either increases or decreases) by 1. If a quantity starts at 0, changes by 1 at each step, and ends up at 2, it must be 1 at some point! Therefore, for some string y_1 with $d(y_1) = 1$, and some other string y_2, $y = y_1 y_2$, and since $d(y) = d(y_1) + d(y_2) = 2$, $d(y_2)$ must also be 1.

The argument is exactly the same for a string in L_b that starts with a. We conclude that *AEqB* is generated by the CFG with productions

$$S \to \Lambda \mid aB \mid bA$$
$$A \to aS \mid bAA$$
$$B \to bS \mid aBB$$

One feature of this CFG is that if we call A the start variable instead of S, it also works as a grammar generating the language L_a, and similarly for B and L_b.

We can obtain many more examples of context-free languages from the following theorem, which describes three ways of starting with CFLs and constructing new ones.

Theorem 4.9
If L_1 and L_2 are context-free languages over an alphabet Σ, then $L_1 \cup L_2$, $L_1 L_2$, and L_1^* are also CFLs.

Proof
Suppose $G_1 = (V_1, \Sigma, S_1, P_1)$ generates L_1 and $G_2 = (V_2, \Sigma, S_2, P_2)$ generates L_2. We consider the three new languages one at a time, and in the first two cases we assume, by renaming variables if necessary, that G_1 and G_2 have no variables in common.

1. We construct a CFG $G_u = (V_u, \Sigma, S_u, P_u)$ generating $L_1 \cup L_2$, as follows. S_u is a new variable not in either V_1 or V_2,

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

   and

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

   For every $x \in L_1 \cup L_2$, we can derive x in the grammar G_u by starting with either $S_u \to S_1$ or $S_u \to S_2$ and continuing with the derivation in either G_1 or G_2. On the other hand, if $S_u \Rightarrow^*_{G_U} x$, the first step in any derivation must be either $S_u \Rightarrow S_1$ or $S_u \Rightarrow S_2$, because those are the only productions with left side S_u. In the first case, the remaining steps must involve productions in G_1, because no variables in V_2 can appear, and so $x \in L_1$; similarly, in the second case $x \in L_2$. Therefore, $L(G_u) = L_1 \cup L_2$.

2. To obtain $G_c = (V_c, \Sigma, S_c, P_c)$ generating L_1L_2, we add the single variable S_c to the set $V_1 \cup V_2$, just as in the union, and we let

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

For a string $x = x_1x_2 \in L_1L_2$, where $x_1 \in L_1$ and $x_2 \in L_2$, a derivation of x in G_c is

$$Sc \Rightarrow S_1S_2 \Rightarrow^* x_1S_2 \Rightarrow^* x_1x_2$$

where the second step (actually a sequence of steps) is a derivation of x_1 in G_1 and the third step is a derivation of x_2 in G_2. Conversely, since the first step in any derivation in G_c must be $S_c \Rightarrow S_1S_2$, every string x derivable from S_c must have the form $x = x_1x_2$, where for each i, x_i is derivable from S_i in G_c. But because $V_1 \cap V_2 = \emptyset$, being derivable from S_i in G_c means being derivable from S_i in G_i, so that $x \in L_1L_2$.

3. We can define a CFG $G^* = (V, \Sigma, S, P)$ generating L_1^* by letting $V = V_1 \cup \{S\}$, where S is a variable not in V_1, and adding productions that generate all possible strings S_1^k, where $k \geq 0$. Let

$$P = P_1 \cup \{S \rightarrow SS_1 \mid \Lambda\}$$

Every string in L_1^* is an element of $L(G_1)^k$ for some $k \geq 0$ and can therefore be obtained from S_1^k; therefore, $L_1^* \subseteq L(G^*)$. On the other hand, every string x in $L(G^*)$ must be derivable from S_1^k for some $k \geq 0$, and we may conclude that $x \in L(G_1)^k \subseteq L(G_1)^*$, because the only productions in P starting with S_1 are the ones in G_1.

The Language $\{a^ib^jc^k \mid j \neq i + k\}$ EXAMPLE 4.10

Let L be the language $\{a^ib^jc^k \mid j \neq i + k\} \subseteq \{a, b, c\}^*$. The form of each element of L might suggest that we try expressing L as the concatenation of three languages L_1, L_2, and L_3, which contain strings of a's, strings of b's, and strings of c's, respectively. This approach doesn't work. Both $a^2b^4c^3$ and $a^3b^4c^2$ are in L; if a^2 were in L_1, b^4 were in L_2, and c^2 were in L_3, then $a^2b^4c^2$, which isn't an element of L, would be in $L_1L_2L_3$.

Instead, we start by noticing that $j \neq i + k$ means that either $j > i + k$ or $j < i + k$, so that L is the union

$$L = L_1 \cup L_2 = \{a^ib^jc^k \mid j > i + k\} \cup \{a^ib^jc^k \mid j < i + k\}$$

There's still no way to express L_1 or L_2 as a threefold concatenation using the approach we tried originally (see Exercise 4.11), but L_1 can be expressed as a concatenation of a slightly different form. Observe that for any natural numbers i and k,

$$a^ib^{i+k}c^k = (a^ib^i)(b^kc^k)$$

and a string in L_1 differs from a string like this only in having at least one extra b in the middle. In other words,

$$L_1 = MNP = \{a^ib^i \mid i \geq 0\}\ \{b^m \mid m > 0\}\ \{b^kc^k \mid k \geq 0\}$$

and it will be easy to find CFGs for the three languages M, N, and P.

L_2 is slightly more complicated. For a string $a^i b^j c^k$ in L_1, knowing that $j > i + k$ tells us in particular that $j > i$; for a string $a^i b^j c^k$ in L_2, such that $j < i + k$, it is helpful to know either how j is related to i or how it is related to k. Let us also split L_2 into a union of two languages:

$$L_2 = L_3 \cup L_4 = \{a^i b^j c^k \mid j < i\} \cup \{a^i b^j c^k \mid i \leq j < i + k\}$$

Now we can use the same approach for L_3 and L_4 as we used for L_1. We can write a string in L_3 as

$$a^i b^j c^k = (a^{i-j})\ (a^j b^j)\ (c^k)$$

where $i - j > 0$, $j \geq 0$, and $k \geq 0$ (and these inequalities are the only constraints on $i - j$, j, and k), and a string in L_4 as

$$a^i b^j c^k = (a^i b^i)\ (b^{j-i} c^{j-i})\ (c^{k-j+i})$$

where i, $j - i$, and $k - j + i$ are natural numbers that are arbitrary except that $i > 0$, $j - i \geq 0$, and $k - j + i > 0$ (i.e., $k + i > j$). It follows that

$$L_3 = QRT = \{a^i \mid i > 0\}\ \{b^i c^i \mid i \geq 0\}\ \{c^i \mid i \geq 0\}$$
$$L_4 = UVW = \{a^i b^i \mid i > 0\}\ \{b^i c^i \mid i \geq 0\}\ \{c^i \mid i > 0\}$$

We have now expressed L as the union of the three languages L_1, L_3, and L_4, and each of these three can be written as the concatenation of three languages. The context-free grammar we are looking for will have productions

$$S \to S_1 \mid S_3 \mid S_4 \qquad S_1 \to S_M S_N S_P \qquad S_3 \to S_Q S_R S_T \qquad S_4 \to S_U S_V S_W$$

as well as productions that start with the nine variables S_M, S_N, ..., S_W. We present productions to take care of the first three of these and leave the last six to you.

$$S_M \to aS_M b \mid \Lambda \qquad S_N \to bS_N \mid b \qquad S_P \to bS_N c \mid \Lambda$$

4.3 REGULAR LANGUAGES AND REGULAR GRAMMARS

The three operations in Theorem 4.9 are the ones involved in the recursive definition of regular languages (Definition 3.1). The "basic" regular languages over an alphabet Σ (the languages $\emptyset$ and $\{\sigma\}$, for every $\sigma \in \Sigma$) are context-free languages. These two statements provide the ingredients that would be necessary for a proof using structural induction that every regular language is a CFL.

EXAMPLE 4.11 A CFG Corresponding to a Regular Expression

Let $L \subseteq \{a, b\}^*$ be the language corresponding to the regular expression

$$bba(ab)^* + (ab + ba^*b)^*ba$$

A literal application of the constructions in Theorem 4.9 would be tedious: the expression $ab + ba^*b$ alone involves all three operations, including three uses of concatenation. We don't really need to introduce separate variables for the languages $\{a\}$ and $\{b\}$, and there are other similar shortcuts to reduce the number of variables in the grammar. (Keep in mind that just as when we constructed an NFA for a given regular expression in Chapter 3, sometimes a literal application of the constructions helps to avoid errors.)

Here we might start by introducing the productions $S \to S_1 \mid S_2$, since the language is a union of two languages L_1 and L_2. The productions

$$S_1 \to S_1ab \mid bba$$

are sufficient to generate L_1. L_2 is complicated enough that we introduce another variable T for the language corresponding to $ab + ba^*b$, and the productions

$$S_2 \to TS_2 \mid ba$$

will then take care of L_2. Finally, the productions

$$T \to ab \mid bUb \qquad U \to aU \mid \Lambda$$

are sufficient to generate the language represented by T. The complete CFG for L has five variables and the productions

$$S \to S_1 \mid S_2 \qquad S_1 \to S_1ab \mid bba \qquad S_2 \to TS_2 \mid ba$$
$$T \to ab \mid bUb \qquad U \to aU \mid \Lambda$$

Not only can every regular language be generated by a context-free grammar, but it can be generated by a CFG of a particularly simple form. To introduce this type of grammar, we consider the finite automaton in Figure 4.12, which accepts the language $\{a, b\}^*\{ba\}$.

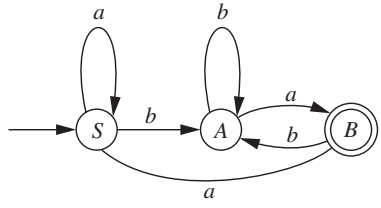
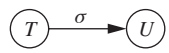


Figure 4.12
An FA accepting $\{a, b\}^*\{ba\}$.

The idea behind the corresponding CFG is that states in the FA correspond to variables in the grammar, and for each transition



the grammar will have a production $T \to \sigma U$. The six transitions in the figure result in the six productions

$$S \to aS \mid bA \qquad A \to bA \mid aB \qquad B \to bA \mid aS$$

The only other productions in the grammar will be Λ-productions (i.e., the right side will be Λ), and so the current string in an uncompleted derivation will consist of a string of terminal symbols and a single variable at the end. The sequence of transitions

$$S \overset{b}{\to} A \overset{b}{\to} A \overset{a}{\to} B \overset{a}{\to} S \overset{b}{\to} A \overset{a}{\to} B$$

corresponds to the sequence of steps

$$S \Rightarrow bA \Rightarrow bbA \Rightarrow bbaB \Rightarrow bbaaS \Rightarrow bbaabA \Rightarrow bbaabaB$$

We can see better now how the correspondence between states and variables makes sense: a state is the way an FA "remembers" how to react to the next input symbol, and the variable at the end of the current string can be thought of as the state of the derivation—the way the derivation remembers, for each possible terminal symbol, how to generate the appropriate little piece of the string that should come next.

The way the string *bbaaba* is finally accepted by the FA is that the current state B is an accepting state, and the way the corresponding derivation terminates is that the variable B at the end of the current string will be replaced by Λ.

Definition 4.13 Regular Grammars

A context-free grammar $G = (V, \Sigma, S, P)$ is *regular* if every production is of the form $A \to \sigma B$ or $A \to \Lambda$, where $A, B \in V$ and $\sigma \in \Sigma$.

Theorem 4.14

For every language $L \subset \Sigma^*$, L is regular if and only if $L = L(G)$ for some regular grammar G.

Proof

If L is a regular language, then for some finite automaton $M = (Q, \Sigma, q_0, A, \delta)$, $L = L(M)$. As in the grammar above for the FA in Figure 4.12, we define $G = (V, \Sigma, S, P)$ by letting V be Q, letting S be the initial state q_0, and letting P be the set containing a production $T \to aU$ for

every transition $\delta(T, a) = U$ in M, and a production $T \to \Lambda$ for every accepting state T of M. G is a regular grammar. It is easy to see, just as in the example, that for every $x = a_1a_2 \ldots a_n$, the transitions on these symbols that start at q_0 end at an accepting state if and only if there is a derivation of x in G; in other words, $L = L(G)$.

In the other direction, if G is a regular grammar with $L = L(G)$, we can reverse the construction to produce $M = (Q, \Sigma, q_0, A, \delta)$. Q is the set of variables of G, q_0 is the start variable, A is the set of states (variables) for which there are Λ-productions in G, and for every production $T \to \sigma U$ there is a transition $T \xrightarrow{\sigma} U$ in M. We cannot expect that M is an ordinary finite automaton, because for some combinations of T and a, there may be either more than one or fewer than one U for which $T \to \sigma U$ is a production. But M is an NFA, and the argument in the first part of the proof is still sufficient; for every string x, there is a sequence of transitions involving the symbols of x that starts at q_0 and ends in an accepting state if and only if there is a derivation of x in G. Therefore, L is regular, because it is the language accepted by the NFA M.

The word *regular* is sometimes used to describe grammars that are slightly different from the ones in Definition 4.13. Grammars in which every production has one of the forms $A \to \sigma B$, $A \to \sigma$, or $A \to \Lambda$ are equivalent to finite automata in the same way that our regular grammars are. Grammars with only the first two of these types of productions generate regular languages, and for every language L, $L - \{\Lambda\}$ can be generated by such a grammar. Similarly, a language L is regular if and only if the set of nonnull strings in L can be generated by a grammar in which all productions have the form $A \to xB$ or $A \to x$, where A and B are variables and x is a nonnull string of terminals. Grammars of this last type are also called *linear*. Some of these variations are discussed in the exercises.

4.4 | DERIVATION TREES AND AMBIGUITY

In most of our examples so far, we have been interested in what strings a context-free grammar generates. As Example 4.2 suggests, it is also useful to consider *how* a string is generated by a CFG. A derivation may provide information about the structure of the string, and if a string has several possible derivations, one may be more appropriate than another.

Just as diagramming a sentence might help to exhibit its grammatical structure, drawing a *tree* to represent a derivation of a string helps to visualize the steps of the derivation and the corresponding structure of the string. In a derivation tree, the root node represents the start variable S. For each interior node N, the portion of the tree consisting of N and its children represents a production $A \to \alpha$ used in the derivation. N represents the variable A, and the children, from left to right, represent the symbols of α. (If the production is $A \to \Lambda$, the node N has a single child representing Λ.) Each leaf node in the tree represents either a terminal symbol

or Λ, and the string being derived is the one obtained by reading the leaf nodes from left to right.

A derivation of a string x in a CFG is a sequence of steps, and a single step is described by specifying the current string, a variable appearing in the string, a particular occurrence of that variable, and the production starting with that variable. We will adopt the phrase *variable-occurrence* to mean a particular occurrence of a particular variable. (For example, if S and A are variables, the string $S + A * S$ contains three variable-occurrences.) Using this terminology, we may say that a step in a derivation is determined by the current string, a particular variable-occurrence in the string, and the production to be applied to that variable-occurrence.

It is easy to see that for each derivation in a CFG, there is exactly one derivation tree. The derivation begins with the start symbol S, which corresponds to the root node of the tree, and the children of that node are determined by the first production in the derivation. At each subsequent step, a production is applied, involving a variable-occurrence corresponding to a node N in the tree; that production determines the portion of the tree consisting of N and its children. For example, the derivation

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + (S) \Rightarrow a + (S * S) \Rightarrow a + (a * S) \Rightarrow a + (a * a)$$

in the CFG for *Expr* in Example 4.2 corresponds to the derivation tree in Figure 4.15.

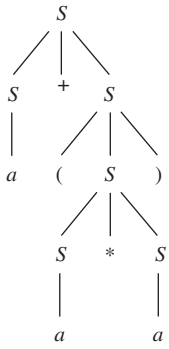


Figure 4.15 A derivation tree for $a + (a * a)$ in Example 4.2.

There are other derivations that also correspond to this tree. Every derivation of the string $a + (a * a)$ must begin

$$S \Rightarrow S + S$$

but now there are two possible ways to proceed, since the next step could involve either of the two occurrences of S. If we chose the rightmost one, so as to obtain $S + (S)$, we would still have two choices for the step after that.

When we said above, "if a string has several possible derivations, one may be more appropriate than another", we were referring, not to derivations that differed in this way, but to derivations corresponding to different derivation trees. Among all the derivations corresponding to the derivation tree in Figure 4.15 (see Exercise 4.30), there are no *essential* differences, but only differences having to do with which occurrence of S we choose for the next step. We could eliminate these choices by agreeing, at every step where the current string has more than one variable-occurrence, to use the *leftmost* one.

Definition 4.16 Leftmost and Rightmost Derivations

A derivation in a context-free grammar is a *leftmost* derivation (LMD) if, at each step, a production is applied to the leftmost variable-occurrence in the current string. A rightmost derivation (RMD) is defined similarly.

Theorem 4.17

If G is a context-free grammar, then for every $x \in L(G)$, these three statements are equivalent:

1. x has more than one derivation tree.
2. x has more than one leftmost derivation.
3. x has more than one rightmost derivation.

Proof

We will show that x has more than one derivation tree if and only if x has more than one LMD, and the equivalence involving rightmost derivations will follow similarly.

If there are two different derivation trees for the string x, each of them has a corresponding leftmost derivation. The two LMDs must be different—otherwise that derivation would correspond to two different derivation trees, and this is impossible for any derivation, leftmost or otherwise.

If there are two different leftmost derivations of x, let the corresponding derivation trees be T_1 and T_2. Suppose that in the first step where the two derivations differ, 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, because the derivations are leftmost; A is a variable; and $\alpha_1 \neq \alpha_2$. In both T_1 and T_2 there is a node corresponding to the variable-occurrence 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, and so $T_1 \neq T_2$.

Definition 4.18 Ambiguity in a CFG

A context-free grammar G is *ambiguous* if for at least one $x \in L(G)$, x has more than one derivation tree (or, equivalently, more than one leftmost derivation).

We will return to the algebraic-expression grammar of Example 4.2 shortly, but first we consider an example of ambiguity arising from the definition of if-statements in Example 4.4.

EXAMPLE 4.19 The Dangling *else*

In the C programming language, an if-statement can be defined by these grammar rules:

$$\begin{aligned} S \rightarrow\ & \text{if } (\ E\ )\ S\ \mid \\ & \text{if } (\ E\ )\ S\ \text{ else } S\ \mid \\ & OS \end{aligned}$$

(In our notation, E is short for <expression>, S for <statement>, and OS for <otherstatement>.) A statement in C that illustrates the ambiguity of these rules is

```
if (e1) if (e2) f(); else g();
```

The problem is that although in C the *else* should be associated with the second *if*, as in

```
if (e1) { if (e2) f(); else g(); }
```

there is nothing in these grammar rules to rule out the other interpretation:

```
if (e1) { if (e2) f(); } else g();
```

The two derivation trees in Figure 4.21 show the two possible interpretations of the statement; the correct one is in Figure 4.21a.

There are equivalent grammar rules that allow only the correct interpretation. One possibility is

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow \text{if } (\ E\ )\ S_1\ \text{ else } S_1 \mid OS \\ S_2 &\rightarrow \text{if } (\ E\ )\ S\ \mid \\ &\qquad \text{if } (\ E\ )\ S_1\ \text{ else } S_2 \end{aligned}$$

These rules generate the same strings as the original ones and are unambiguous. We will not prove either fact, but you can see how the second might be true. The variable S_1 represents a statement in which every *if* is matched by a corresponding *else*, and every statement derived from S_2 contains at least one unmatched *if*. The only variable appearing before *else* in these rules is S_1; because the *else* cannot match any of the *if* s in the statement derived from S_1, it must match the *if* that appeared at the same time it did.

EXAMPLE 4.20 Ambiguity in the CFG for *Expr* in Example 4.2

In the grammar G in Example 4.2, with productions

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

the string $a + a * a$ has the two leftmost derivations

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

which correspond to the derivation trees in Figure 4.22.

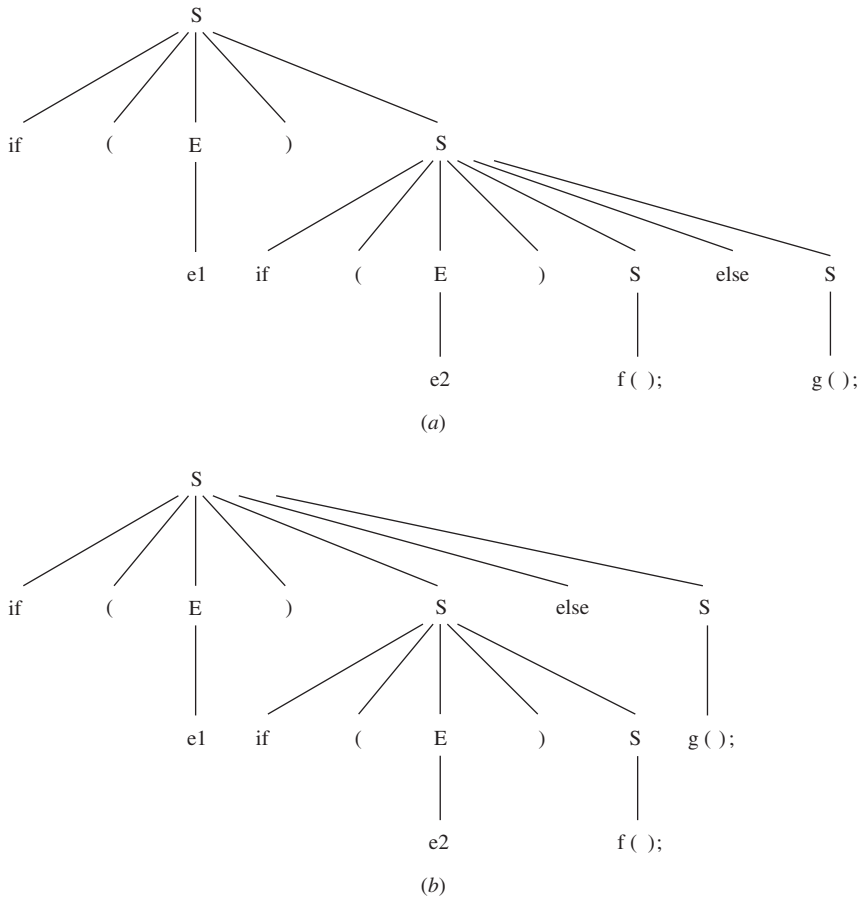


Figure 4.21 |
Two possible interpretations of a dangling *else*.

We observed in Example 4.2 that the first of these two LMDs (or the first of the two derivation trees) matches the interpretation of $a + a * a$ as a sum, rather than as a product; one reason for the ambiguity is that the grammar allows either of the two operations $+$ and $*$ to be given higher precedence. Just as adding braces to the C statements in Example 4.19 allowed only the correct interpretation, the addition of parentheses in this expression, to obtain $a + (a * a)$, has the same effect; the only leftmost derivation of this string is

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + (S) \Rightarrow a + (S * S) \Rightarrow a + (a * S) \Rightarrow a + (a * a)$$

Another rule in algebra that is not enforced by this grammar is the rule that says operations of the same precedence are performed left-to-right. For this reason, both the expressions $a + a + a$ and $a * a * a$ also illustrate the ambiguity of the grammar. The first expression has the (correct) LMD

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

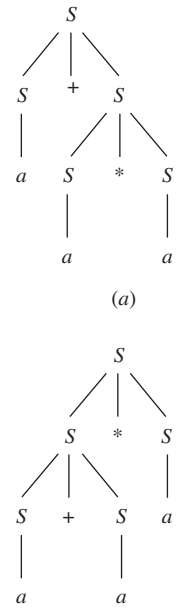


Figure 4.22 |
Two derivation trees for $a + a * a$ in Example 4.2.

corresponding to the interpretation $(a + a) + a$, as well as the LMD

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

corresponding to the other way of grouping the terms.

In order to find an unambiguous CFG G_1 generating this language, we look for productions that do not allow any choice regarding either the precedence of the two operators or the order in which operators of the same precedence are performed. To some extent, we can ignore the parentheses at least temporarily. Let S_1 be the start symbol of G_1.

Saying that multiplication should have higher precedence than addition means that when there is any doubt, an expression should be treated as a sum rather than a product. For this reason, we concentrate first on productions that will give us sums of terms. The problem with $S_1 \to S_1 + S_1$ is that if we are attempting to derive a sum of three or more terms, there are too many choices as to how many terms will come from the first S_1 and how many from the second. Saying that additions are performed left-to-right, or that $+$ "associates to the left", suggests that we think of a sum of n terms as another sum plus *one* additional term. We try

$$S_1 \to S_1 + T \mid T$$

where T represents a term—that is, an expression that may be part of a sum but is not itself a sum.

It is probably clear already that we would no longer want $S_1 \to S_1 * S_1$, even if it didn't cause the same problems as $S_1 \to S_1 + S_1$; we don't want to say that an expression can be either a sum or a product, because that was one of the sources of ambiguity. Instead, we say that *terms* can be products. Products of what? *Factors*. This suggests

$$T \to T * F \mid F$$

where, in the same way that $S_1 + T$ is preferable to $T + S_1$, $T * F$ is preferable to $F * T$. We now have a hierarchy with three levels: expressions are sums of terms, and terms are products of factors. Furthermore, we have incorporated the rule that each operator associates to the left.

What remains is to deal with a and expressions with parentheses. The productions $S_1 \to T$ and $T \to F$ allow us to have an expression with a single term and a term with a single factor; thus, although a by itself is a valid expression, it is best to call it a factor, because it is neither a product nor a sum. Similarly, an expression in parentheses should also be considered a factor. What is in the parentheses should be an expression, because once we introduce a pair of parentheses we can start all over with what's inside.

The CFG that we have now obtained is $G_1 = (V, \Sigma, S_1, P)$, where $V = \{S_1, T, F\}$, $\Sigma = \{a, +, *, (,)\}$, and P contains the productions

$$\begin{aligned} S_1 &\to S_1 + T \mid T \\ T &\to T * F \mid F \\ F &\to a \mid (S_1) \end{aligned}$$

Both halves of the statement $L(G) = L(G_1)$ can be proved by mathematical induction on the length of a string. The details, particularly for the statement $L(G) \subseteq L(G_1)$, are somewhat involved and are left to the exercises.

The rest of this section is devoted to proving that G_1 is unambiguous. In the proof, for a string $x \in L(G_1)$, it is helpful to talk about a symbol in x that is *within parentheses*, and we want this to mean that it lies between the left and right parentheses that are introduced in a single step of a derivation of x. One might ask, however, whether there can be two derivations of x, and a symbol that has this property with respect to one of the two but not the other. The purpose of Definition 4.23 and Theorem 4.24 is to establish that this formulation is a satisfactory one and that "within parentheses" doesn't depend on which derivation we are using. In the proof of Theorem 4.24, we will use the result obtained in Example 1.25, that balanced strings of parentheses are precisely the strings with equal numbers of left and right parentheses and no prefix having more right than left parentheses.

Definition 4.23 The Mate of a Left Parenthesis in a Balanced String

The *mate* of a left parenthesis in a balanced string is the first right parenthesis following it for which the string starting with the left and ending with the right has equal numbers of left and right parentheses. (Every left parenthesis in a balanced string has a mate—see Exercise 4.41.)

Theorem 4.24

For every $x \in L(G_1)$, every derivation of x in G_1, and every step of this derivation in which two parentheses are introduced, the right parenthesis is the mate of the left.

Proof

Suppose $x = x_1({}_0z)_0x_2$, where ${}_0$ and $)_0$ are two occurrences of parentheses that are introduced in the same step in some derivation of x. Then $F \Rightarrow ({}_0S_1)_0 \Rightarrow^* ({}_0z)_0$. It follows that z is a balanced string, and by the definition of "mate", the mate of ${}_0$ cannot appear after $)_0$.

If it appears before $)_0$, however, then $z = z_1)_1z_2$, for some strings z_1 and z_2, where $)_1$ is the mate of ${}_0$ and z_1 has equal numbers of left and right parentheses. This implies that the prefix $z_1)_1$ of z has more right parentheses than left, which is impossible if z is balanced. Therefore, $)_0$ is the mate of ${}_0$.

Definition 4.23 and Theorem 4.24 allow us to say that "between the two parentheses produced in a single step of a derivation" is equivalent to "between some left parenthesis and its mate", so that either of these can be taken as the definition of "within parentheses".

Theorem 4.25

The context-free grammar G_1 with productions

$$S_1 \rightarrow S_1 + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow a \mid (S_1)$$

is unambiguous.

Proof

We prove the following statement, which implies the result: For every x derivable from one of the variables S_1, T, or F in G_1, x has only one leftmost derivation from that variable. The proof is by mathematical induction on $|x|$. The basis step, for $x = a$, is easy, because for each of the three variables, a has exactly one derivation from that variable.

The induction hypothesis is that $k \geq 1$ and that for every y derivable from S_1, T, or F for which $|y| \leq k$, y has only one leftmost derivation from that variable. We wish to show that the same is true for a string $x \in L(G_1)$ with $|x| = k + 1$.

We start with the case in which x contains at least one occurrence of $+$ that is not within parentheses. Because the only occurrences of $+$ in strings derivable from T or F *are* within parentheses, every derivation of x must begin $S_1 \Rightarrow S_1 + T$, and the occurrence of $+$ in this step is the last one in x that is not within parentheses. Every leftmost derivation of x from S_1 must then have the form

$$S_1 \Rightarrow S_1 + T \Rightarrow^* y + T \Rightarrow^* y + z$$

where the last two steps represent leftmost derivations of y from S_1 and z from T, respectively, and the $+$ is still the last one not within parentheses. According to the induction hypothesis, y has only one leftmost derivation from S_1, and z has only one from T; therefore, x has only one LMD from S_1.

Next we consider the case in which every occurrence of $+$ in x is within parentheses but there is at least one occurrence of $*$ not within parentheses. Because x cannot be derived from F, every derivation of x from S_1 must begin $S_1 \Rightarrow T \Rightarrow T * F$, and every derivation from T must begin $T \Rightarrow T * F$. In either case, this occurrence of $*$ must be the last one in x not within parentheses. As before, the subsequent steps of every LMD must be

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

in which the derivations of y from T and z from F are both leftmost. The induction hypothesis tells us that there is only one way for each of these derivations to proceed, and that there is only one leftmost derivation of x from S_1 or T.

Finally, suppose that every occurrence of $+$ or $*$ in x is within parentheses. Then x can be derived from any of the three variables, but every derivation from S_1 begins

$$S_1 \Rightarrow T \Rightarrow F \Rightarrow (S_1)$$

and every derivation from T or F begins the same way but with the first one or two steps omitted. Therefore, $x = (y)$, where $S_1 \Rightarrow^* y$. By the induction hypothesis, y has only one LMD from S_1, and it follows that x has only one from each of the three variables.

4.5 SIMPLIFIED FORMS AND NORMAL FORMS

Questions about the strings generated by a context-free grammar G are sometimes easier to answer if we know something about the form of the productions. Sometimes this means knowing that certain types of productions never occur, and sometimes it means knowing that every production has a certain simple form. For example, suppose we want to know whether a string x is generated by G, and we look for an answer by trying all derivations with one step, then all derivations with two steps, and so on. If we don't find a derivation that produces x, how long do we have to keep trying?

The number t of terminals in the current string of a derivation cannot decrease at any step. If G has no Λ-productions (of the form $A \rightarrow \Lambda$), then the length l of the current string can't decrease either, which means that the sum $t + l$ can't decrease. If we also know that G has no "unit productions" (of the form $A \rightarrow B$, where A and B are variables), then the sum must increase, because every step either adds a terminal or increases the number of variables. A derivation of x starts with S, for which $l + t = 1$, and ends with x, for which $l + t = 2|x|$; therefore, no derivation has more than $2|x| - 1$ steps. If we try all derivations with this many steps or fewer and don't find one that generates x, we may conclude that $x \notin L(G)$.

In this section we show how to modify an arbitrary CFG G so that the modified grammar has no productions of either of these types but still generates $L(G)$, except possibly for the string Λ. We conclude by showing how to modify G further (eliminating both these types of productions is the first step) so as to obtain a CFG that is still essentially equivalent to G but is in *Chomsky normal form*, so that every production has one of two very simple forms.

A simple example will suggest the idea of the algorithm to eliminate Λ-productions. Suppose one of the productions in G is

$$A \rightarrow BCDCB$$

and that from both the variables B and C, Λ can be derived (in one or more steps), as well as other nonnull strings of terminals. Once the algorithm has been applied, the steps that replace B and C by Λ will no longer be possible, but we must still be able to get all the nonnull strings from A that we could have gotten using these

steps. This means that we should retain the production $A \to BCDCB$, but we should add $A \to CDCB$ (because we could have replaced the first B by Λ and the other occurrences of B and C by nonnull strings), $A \to DCB$, $A \to CDB$, and so on—every one of the fifteen productions that result from leaving out one or more of the four occurrences of B and C in the right-hand string.

The necessary first step, of course, is to identify the variables like B and C from which Λ can be derived. We will refer to these as *nullable* variables. It is easy to see that the set of nullable variables can be obtained using the following recursive definition.

Definition 4.26 A Recursive Definition of the Set of Nullable Variables of *G*

1. Every variable A for which there is a production $A \to \Lambda$ is nullable.
2. If $A_1, A_2, \ldots, A_k$ are nullable variables (not necessarily distinct), and
$$B \to A_1 A_2 \ldots A_k$$
is a production, then B is nullable.

This definition leads immediately to an algorithm for identifying the nullable variables (see Example 1.21).

Theorem 4.27

For every context-free grammar $G = (V, \Sigma, S, P)$, the following algorithm produces a CFG $G_1 = (V, \Sigma, S, P_1)$ having no Λ-productions and satisfying $L(G_1) = L(G) - \{\Lambda\}$.

1. Identify the nullable variables in V, and initialize P_1 to P.
2. For every production $A \to \alpha$ in P, add to P_1 every production obtained from this one by deleting from α one or more variable-occurrences involving a nullable variable.
3. Delete every Λ-production from P_1, as well as every production of the form $A \to A$.

Proof

It is obvious that G_1 has no Λ-productions. We show that for every $A \in V$ and every nonnull $x \in \Sigma^*$, $A \Rightarrow_G^* x$ if and only if $A \Rightarrow_{G_1}^* x$.

First we show, using mathematical induction on n, that for every $n \geq 1$, every $A \in V$, and every $x \in \Sigma^*$ with $x \neq \Lambda$, if $A \Rightarrow_G^n x$, then $A \Rightarrow_{G_1}^* x$. For the basis step, suppose $A \Rightarrow_G^1 x$. Then $A \to x$ is a production in P, and since $x \neq \Lambda$, this production is also in P_1.

Suppose that $k \geq 1$ and that for every $n \leq k$, every variable A, and every nonnull $x \in \Sigma^*$ for which $A \Rightarrow_G^n x$, $A \Rightarrow_{G_1}^* x$. Now suppose that

$A \Rightarrow_G^{k+1} x$, for some $x \in \Sigma^* - \{\Lambda\}$. Let the first step in some $(k+1)$-step derivation of x from A in G be

$$A \Rightarrow X_1 X_2 \ldots X_m$$

where each X_i is either a variable or a terminal. Then $x = x_1 x_2 \ldots x_m$, where for each i, either $x_i = X_i$ (a terminal), or x_i is a string derivable from the variable X_i in k or fewer steps. When all the (nullable) variables X_i for which the corresponding x_i is Λ are deleted from the string $X_1 X_2 \ldots X_m$, there are still some X_i's left, because $x \neq \Lambda$, and so the resulting production is an element of P_1. Furthermore, the induction hypothesis tells us that for each variable X_i remaining, $X_i \Rightarrow_{G_1}^* x_i$. Therefore, $A \Rightarrow_{G_1}^* x$.

For the converse, we show, using mathematical induction on n, that for every $n \geq 1$, every variable A, and every $x \in \Sigma^*$, if $A \Rightarrow_{G_1}^n x$, then $A \Rightarrow_G^* x$. If $A \Rightarrow_{G_1}^1 x$, then $A \rightarrow x$ is a production in P_1. It follows that $A \rightarrow \alpha$ is a production in P, where α is a string from which x can be obtained by deleting zero or more occurrences of nullable variables. Therefore, $A \Rightarrow_G^* x$, because we can begin a derivation with the production $A \rightarrow \alpha$ and continue by deriving Λ from each of the nullable variables that was deleted from α to obtain x.

Suppose that $k \geq 1$, that for every $n \leq k$, every A, and every string x with $A \Rightarrow_{G_1}^n x$, $A \Rightarrow_G^* x$, and that $A \Rightarrow_{G_1}^{k+1} x$. Again, let the first step of some $(k+1)$-step derivation of x from A in G_1 be

$$A \Rightarrow X_1 X_2 \ldots X_m$$

Then $x = x_1 x_2 \ldots x_m$, where for each i, either $x_i = X_i$ or x_i is a string derivable from the variable X_i in the grammar G_1 in k or fewer steps. By the induction hypothesis, $X_i \Rightarrow_G^* x_i$ for each i. By definition of G_1, there is a production $A \rightarrow \alpha$ in P so that $X_1 X_2 \ldots X_m$ can be obtained from α by deleting some variable-occurrences involving nullable variables. This implies that $A \Rightarrow_G^* X_1 X_2 \ldots X_m$, and therefore that we can derive x from A in G, by first deriving $X_1 \ldots X_m$ and then deriving each x_i from the corresponding X_i.

The procedure we use to eliminate unit productions from a CFG is rather similar. We first identify pairs of variables (A, B) for which $A \Rightarrow^* B$ (not only those for which $A \rightarrow B$ is a production); then, for each such pair (A, B), and each nonunit production $B \rightarrow \alpha$, we add the production $A \rightarrow \alpha$.

If we make the simplifying assumption that we have already eliminated Λ-productions from the grammar, then a sequence of steps by which $A \Rightarrow^* B$ involves only unit productions. This allows us, for a variable A, to formulate the following recursive definition of an "A-derivable" variable (a variable B different from A for which $A \Rightarrow^* B$):

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

Theorem 4.28

For every context-free grammar $G = (V, \Sigma, S, P)$ without Λ-productions, the CFG $G_1 = (V, \Sigma, S, P_1)$ produced by the following algorithm generates the same language as G and has no unit productions.

1. Initialize P_1 to be P, and for each $A \in V$, identify the A-derivable variables.
2. For every pair (A, B) of variables for which B is A-derivable, and every nonunit production $B \to \alpha$, add the production $A \to \alpha$ to P_1.
3. Delete all unit productions from P_1.

Proof

The proof that $L(G_1) = L(G)$ is a straightforward induction proof and is omitted.

Definition 4.29 Chomsky Normal Form

A context-free grammar is said to be in *Chomsky normal form* if every production is of one of these two types:

$$A \to BC \quad \text{(where } B \text{ and } C \text{ are variables)}$$
$$A \to \sigma \quad \text{(where } \sigma \text{ is a terminal symbol)}$$

Theorem 4.30

For every context-free grammar G, there is another CFG G_1 in Chomsky normal form such that $L(G_1) = L(G) - \{\Lambda\}$.

Proof

We describe briefly the algorithm that can be used to construct the grammar G_1, and it is not hard to see that it generates the same language as G, except possibly for the string Λ.

The first step is to apply the algorithms presented in Theorems 4.6 and 4.7 to eliminate Λ-productions and unit productions. The second step is to introduce for every terminal symbol σ a variable X_σ and a production $X_\sigma \to \sigma$, and in every production whose right side has at least two symbols, to replace every occurrence of a terminal by the corresponding

variable. At this point, every production looks like either $A \to \sigma$ or

$$A \to B_1 B_2 \ldots B_k$$

where the B_i's are variables and $k \geq 2$.

The last step is to replace each production having more than two variable-occurrences on the right by an equivalent set of productions, each of which has exactly two variable-occurrences. This step is described best by an example. The production

$$A \to BACBDCBA$$

would be replaced by

$$A \to BY_1 \quad Y_1 \to AY_2 \quad Y_2 \to CY_3 \quad Y_3 \to BY_4 \quad Y_4 \to DY_5$$
$$Y_5 \to CY_6 \quad Y_6 \to BA$$

where the new variables $Y_1, \ldots, Y_6$ are specific to this production and are not used anywhere else.

Converting a CFG to Chomsky Normal Form

EXAMPLE 4.31

This example will illustrate all the algorithms described in this section: to identify nullable variables, to eliminate Λ-productions, to identify A-derivable variables for each A, to eliminate unit productions, and to convert to Chomsky normal form. Let G be the context-free grammar with productions

$$S \to TU \mid V$$
$$T \to aTb \mid \Lambda$$
$$U \to cU \mid \Lambda$$
$$V \to aVc \mid W$$
$$W \to bW \mid \Lambda$$

which can be seen to generate the language $\{a^i b^j c^k \mid i = j \text{ or } i = k\}$.

1. (Identifying nullable variables) The variables T, U, and W are nullable because they are involved in Λ-productions; V is nullable because of the production $V \to W$; and S is also, either because of the production $S \to TU$ or because of $S \to V$. So all the variables are!
2. (Eliminating Λ-productions) Before the Λ-productions are eliminated, the following productions are added:

   $$S \to T \qquad S \to U \qquad T \to ab \qquad U \to c \qquad V \to ac \qquad W \to b$$

   After eliminating Λ-productions, we are left with

   $$S \to TU \mid T \mid U \mid V \qquad T \to aTb \mid ab \qquad U \to cU \mid c$$
   $$V \to aVc \mid ac \mid W \qquad W \to bW \mid b$$

3. (Identifying A-derivable variables, for each A) The S-derivable variables obviously include T, U, and V, and they also include W because of the production $V \to W$. The V-derivable variable is W.

4. (Eliminating unit productions) We add the productions

$$S \to aTb \mid ab \mid cU \mid c \mid aVc \mid ac \mid bW \mid b \qquad V \to bW \mid b$$

before eliminating unit productions. At this stage, we have

$$\begin{aligned} S &\to TU \mid aTb \mid ab \mid cU \mid c \mid aVc \mid ac \mid bW \mid b \\ T &\to aTb \mid ab \\ U &\to cU \mid c \\ V &\to aVc \mid ac \mid bW \mid b \\ W &\to bW \mid b \end{aligned}$$

5. (Converting to Chomsky normal form) We replace a, b, and c by X_a, X_b, and X_c, respectively, in productions whose right sides are not single terminals, obtaining

$$\begin{aligned} S &\to TU \mid X_aTX_b \mid X_aX_b \mid X_cU \mid c \mid X_aVX_c \mid X_aX_c \mid X_bW \mid b \\ T &\to X_aTX_b \mid X_aX_b \\ U &\to X_cU \mid c \\ V &\to X_aVX_c \mid X_aX_c \mid X_bW \mid b \\ W &\to X_bW \mid b \end{aligned}$$

This grammar fails to be in Chomsky normal form only because of the productions $S \to X_aTX_b$, $S \to X_aVX_c$, $T \to X_aTX_b$, and $V \to X_aVX_c$. When we take care of these as described above, we obtain the final CFG G_1 with productions

$$\begin{aligned} S &\to TU \mid X_aY_1 \mid X_aX_b \mid X_cU \mid c \mid X_aY_2 \mid X_aX_c \mid X_bW \mid b \\ Y_1 &\to TX_b \\ Y_2 &\to VX_c \\ T &\to X_aY_3 \mid X_aX_b \\ Y_3 &\to TX_b \\ U &\to X_cU \mid c \\ V &\to X_aY_4 \mid X_aX_c \mid X_bW \mid b \\ Y_4 &\to VX_c \\ W &\to X_bW \mid b \end{aligned}$$

(We obviously don't need both Y_1 and Y_3, and we don't need both Y_2 and Y_4, so we could simplify G_1 slightly.)

EXERCISES

4.1. In each case below, say what language (a subset of $\{a, b\}^*$) is generated by the context-free grammar with the indicated productions.

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

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

c. $S \to SaS \mid b$

d. $S \to SaS \mid b \mid \Lambda$

e. $S \to TT \qquad T \to aT \mid Ta \mid b$

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

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

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

4.2. Find a context-free grammar corresponding to the "syntax diagram" in Figure 4.32.

4.3. In each case below, 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.

4.4. In both parts below, the productions in a CFG G are given. In each part, show first that for every string $x \in L(G)$, $n_a(x) = n_b(x)$; then find a string $x \in \{a, b\}^*$ with $n_a(x) = n_b(x)$ that is not in $L(G)$.

a. $S \to SabS \mid SbaS \mid \Lambda$

b. $S \to aSb \mid bSa \mid abS \mid baS \mid Sab \mid Sba \mid \Lambda$

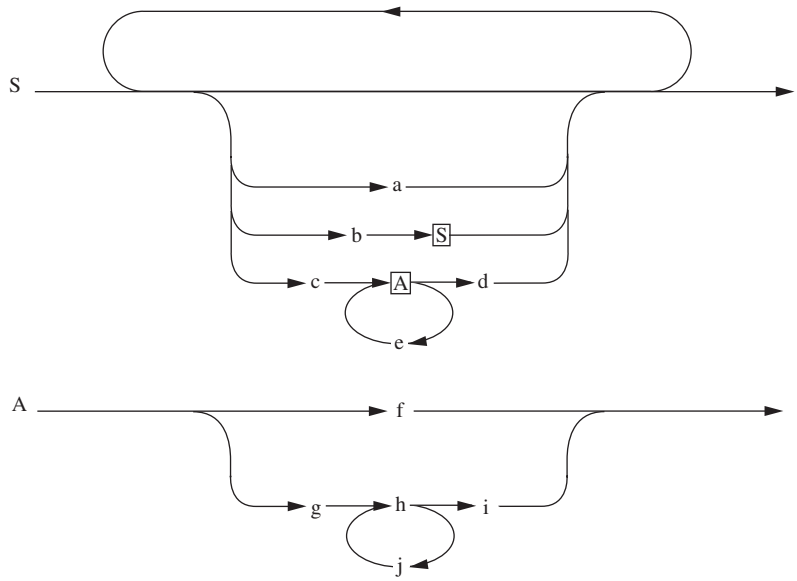


Figure 4.32

4.5. 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.

4.6. Show that the language of all nonpalindromes over $\{a, b\}$ (see Example 4.3) cannot be generated by any CFG in which $S \to aSa \mid bSb$ are the only productions with variables on the right side.

4.7. Describe the language generated by the CFG with productions

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

Give an induction proof that your answer is correct.

4.8. What language over $\{a, b\}$ does the CFG with productions

$$S \to aaS \mid bbS \mid Saa \mid Sbb \mid abSab \mid abSba \mid baSba \mid baSab \mid \Lambda$$

generate? Prove your answer.

4.9. Suppose that $G_1 = (V_1, \{a, b\}, S_1, P_1)$ and $G_2 = (V_2, \{a, b\}, S_2, P_2)$ are CFGs and that $V_1 \cap V_2 = \emptyset$.

a. It is easy to see that no matter what G_1 and G_2 are, the CFG $G_u = (V_u, \{a, b\}, S_u, P_u)$ defined by $V_u = V_1 \cup V_2$, $S_u = S_1$, and $P_u = P_1 \cup P_2 \cup \{S_1 \to S_2\}$ generates every string in $L(G_1) \cup L(G_2)$. Find grammars G_1 and G_2 (you can use $V_1 = \{S_1\}$ and $V_2 = \{S_2\}$) and a string $x \in L(G_u)$ such that $x \notin L(G_1) \cup L(G_2)$.

b. As in part (a), the CFG $G_c = (V_c, \{a, b\}, S_c, P_c)$ defined by $V_c = V_1 \cup V_2$, $S_c = S_1$, and $P_c = P_1 \cup P_2 \cup \{S_1 \to S_1S_2\}$ generates every string in $L(G_1)L(G_2)$. Find grammars G_1 and G_2 (again with $V_1 = \{S_1\}$ and $V_2 = \{S_2\}$) and a string $x \in L(G_c)$ such that $x \notin L(G_1)L(G_2)$.

c. The CFG $G^* = (V, \{a, b\}, S, P)$ defined by $V = V_1$, $S = S_1$, and $P = P_1 \cup \{S_1 \to S_1S_1 \mid \Lambda\}$ generates every string in $L(G_1)^*$. Find a grammar G_1 with $V_1 = \{S_1\}$ and a string $x \in L(G^*)$ such that $x \notin L(G)^*$.

4.10. Find context-free grammars generating each of the languages below.

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

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

c. $\{a^i b^j \mid j = 2i\}$

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

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

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

4.11. a. Show that the language $L = \{a^i b^j c^k \mid j > i + k\}$ cannot be written in the form $L = L_1L_2L_3$, where L_1, L_2, and L_3 are subsets of $\{a\}^*$, $\{b\}^*$, and $\{c\}^*$, respectively.

b. Show the same thing for the language $L = \{a^i b^j c^k \mid j < i + k\}$.

4.12. Find a context-free grammar generating the language $\{a^i b^j c^k \mid i \neq j + k\}$.

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

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

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

Show that no string in L begins with abb.

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

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

has more a's than b's.

4.16. Prove that the CFG with productions $S \to aSbS \mid aSbS \mid \Lambda$ generates the language $L = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$.

4.17. †Show that the CFG with productions

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

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

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

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

4.19. Let G be the CFG with productions $S \to a \mid aS \mid bSS \mid SSb \mid SbS$. Show that every x in $\{a, b\}^*$ with $n_a(x) > n_b(x)$ is an element of $L(G)$.

4.20. Let G be the context-free grammar with productions $S \to SaT \mid \Lambda$ $T \to TbS \mid \Lambda$. Show using mathematical induction that $L(G)$ is the language of all strings in $\{a, b\}^*$ that don't start with b. One direction is easy. For the other direction, it might be easiest to prove two statements simultaneously: (i) every string that doesn't start with b can be derived from S; (ii) every string that doesn't start with a can be derived from T.

4.21. Definition 3.1 and Theorem 4.9 provide the ingredients for a structural-induction proof that every regular language is a CFL. Give the proof.

4.22. Show that if G is a context-free grammar in which every production has one of the forms $A \to aB$, $A \to a$, and $A \to \Lambda$ (where A and B are variables and a is a terminal), then $L(G)$ is regular. Suggestion: construct an NFA accepting $L(G)$, in which there is a state for each variable in G and one additional state F, the only accepting state.

4.23. Suppose $L \subseteq \Sigma^*$. Show that L is regular if and only if $L = L(G)$ for some context-free grammar G in which every production is either of the form $A \to Ba$ or of the form $A \to \Lambda$, where A and B are variables and a

is a terminal. (The grammars in Definition 4.13 are sometimes called *right-regular* grammars, and the ones in this exercise are sometimes called left-regular.)

4.24. Let us call a context-free grammar G a grammar of type R if every production has one of the three forms $A \to aB$, $A \to Ba$, or $A \to \Lambda$ (where A and B are variables and a is a terminal). Every regular language can be generated by a grammar of type R. Is every language that is generated by a grammar of type R regular? If so, give a proof; if not, find a counterexample.

4.25. Show that for a 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 $A \to \Lambda$ (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 $A \to \Lambda$ (where A and B are variables and $x \in \Sigma^*$).

4.26. In each part, draw an NFA (which might be an FA) accepting the language generated by the CFG having the given productions.

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

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

4.27. Find a regular grammar generating the language $L(M)$, where M is the FA shown in Figure 4.33.

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

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

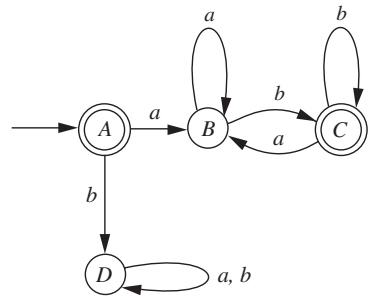


Figure 4.33 |

4.29. 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. $S \to AA \mid B \qquad A \to AAA \mid Ab \mid bA \mid a \qquad B \to bB \mid \Lambda$

4.30. a. Write the rightmost derivation of the string $a + (a * a)$ corresponding to the derivation tree in Figure 4.15.

b. How many distinct derivations (not necessarily leftmost or rightmost) does the string $a + (a * a)$ have in the CFG with productions $S \to a \mid S + S \mid S * S \mid (S)$?

4.31. Again we consider the CFG in Example 4.2, with productions $S \to a \mid S + S \mid S * S \mid (S)$.

a. How many distinct derivation trees does the string $a + (a * a)/a - a$ have in this grammar?

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

4.32. In the CFG in the previous exercise, suppose we define n_i to be the number of distinct derivation trees for the string $a + a + \ldots + a$ in which there are i a's. Then $n_1 = n_2 = 1$.

a. Find a recursive formula for n_i, by first observing that if $i > 1$ the root of a derivation tree has two children labeled S, and then considering all possibilities for the two subtrees.

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

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

4.33. 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 these statements are interpreted according to the derivation tree in Figure 4.21a and $a = 3$?

b. Same question as in (a), but when $a = 1$.

c. What is the resulting value of `x` if these statements are interpreted according to the derivation tree in Figure 4.21b and $a = 3$?

d. Same question as in (c), but when $a = 1$.

4.34. Show that the CFG with productions

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

is ambiguous.

4.35. Consider the context-free grammar with productions

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

Every 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?

4.36. For each part of Exercise 4.1, decide whether the grammar is ambiguous or not, and prove your answer.

4.37. Show that the CFG in Example 4.8 is ambiguous.

4.38. In each case below, 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. $S \to aSb \mid aaSb \mid \Lambda$

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

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

In the exercises that follow, take as the definition of "balanced string of parentheses" the criterion in Example 1.25: The string has an equal number of left and right parentheses, and no prefix has more right than left.

4.40. a. Show that for every string x of left and right parentheses, x is a prefix of a balanced string if and only if no prefix of x has more right parentheses than left.

b. Show that the language of prefixes of balanced strings of parentheses is generated by the CFG with productions $S \to (S)S \mid (S \mid \Lambda$.

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

4.42. Show that if x is a balanced string of parentheses, then either $x = (y)$ for some balanced string y or $x = x_1x_2$ for two balanced strings x_1 and x_2 that are both shorter than x.

4.43. Show that if $(_0$ is an occurrence of a left parenthesis in a balanced string, and $)_0$ is its mate, then $(_0$ is the rightmost left parentheses for which the string consisting of it and $)_0$ and everything in between is balanced.

4.44. †Show that both of the CFGs below generate the language of balanced strings of parentheses.

a. The CFG with productions $S \to S(S) \mid \Lambda$

b. The CFG with productions $S \to (S)S \mid \Lambda$

4.45. Show that both of the CFGs in the previous exercise are unambiguous.

4.46. Let x be a string of left and right parentheses. A *complete pairing* of x is a partition of the parentheses of x in to 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, and in this case the two definitions of mates coincide.

4.47. †Let G be the CFG with productions

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

and G_1 the CFG with productions

$$S_1 \to S_1 + T \mid T \qquad T \to T * F \mid F \qquad F \to (S_1) \mid a$$

(see Section 4.4).

a. Show that $L(G_1) \subseteq L(G)$. One approach is to use induction and to consider three cases in the induction step, corresponding to the three possible ways a derivation of the string in $L(G_1)$ might begin:

$$S_1 \Rightarrow S_1 + T$$
$$S_1 \Rightarrow T \;\Rightarrow\; T * F$$
$$S_1 \Rightarrow T \;\Rightarrow\; F \;\Rightarrow\; (S_1)$$

b. Show that $L(G) \subseteq L(G_1)$. Again three cases are appropriate in the induction step:

i. x has a derivation in G beginning $S \Rightarrow (S)$

ii. x has a derivation beginning $S \Rightarrow S + S$

iii. Every derivation of x in G begins $S \Rightarrow S * S$

In the second case it may be helpful to let $x = x_1 + x_2 + \ldots + x_n$, where each $x_i \in L(G_1)$ and n is as large as possible. In the third case it may be helpful to let $x = x_1 * x_2 * \ldots x_n$, where each $x_i \in L(G)$ (not $L(G_1)$) and n is as large as possible.

4.48. Show that the nullable variables defined by Definition 4.7 are precisely those variables A for which $A \Rightarrow^* \Lambda$.

4.49. In each case below, 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. $S \to AB \mid ABC$
$A \to BA \mid BC \mid \Lambda \mid a$
$B \to AC \mid CB \mid \Lambda \mid b$
$C \to BC \mid AB \mid A \mid c$

4.50. 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

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

b. G has productions

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

c. G has productions

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

4.51. 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.

4.52. 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.

4.53. †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 (see the two preceding exercises), then it is useless. The converse is not true, as the grammar with productions $S \rightarrow AB$ and $A \rightarrow a$ illustrates. (The variable A is both live and reachable but still useless.)

a. 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 $L(G2) = L(G)$.

b. Give an example to show that if the two steps are done in the opposite order, the resulting grammar may still have useless variables.

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

i. G has productions

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

ii. G has productions

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

4.54. In each case below, given the context-free grammar G, find a CFG G_1 in Chomsky normal form generating $L(G) - \{\Lambda\}$.

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

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

c. G has productions

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

4.55. For alphabets Σ_1 and Σ_2, a *homomorphism* from Σ_1^* to Σ_2^* is defined in Exercise 3.53. Show that if $f : \Sigma_1^* \rightarrow \Sigma_2^*$ is a homomorphism and $L \subseteq \Sigma_1^*$ is a context-free language, then $f(L) \subseteq \Sigma_2^*$ is also a CFG.

4.56. †Let G be the context-free grammar with productions

$$S \rightarrow aS \mid aSbS \mid c$$

and let G_1 be the one with productions

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

(G_1 is a simplified version of the second grammar in Example 4.19.)

a. Show that G is ambiguous.

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

c. Show that G_1 is unambiguous.

4.57. †Show that if a context-free grammar is unambiguous, then the grammar obtained from it by the algorithm in Theorem 4.27 is also unambiguous.

4.58. †Show that if a context-free grammar with no Λ-productions is unambiguous, then the one obtained from it by the algorithm in Theorem 4.28 is also unambiguous.