Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \to uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.4. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

AMBIGUITY

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings. This result may be undesirable for certain applications, such as programming languages, where a program should have a unique interpretation.

If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar. If a grammar generates some string ambiguously, we say that the grammar is *ambiguous*.

For example, consider grammar G_5:

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle \mathtt{+} \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \mathtt{\times} \langle \text{EXPR} \rangle \mid \mathtt{(} \langle \text{EXPR} \rangle \mathtt{)} \mid \mathtt{a}$$

This grammar generates the string a+a×a ambiguously. The following figure shows the two different parse trees.

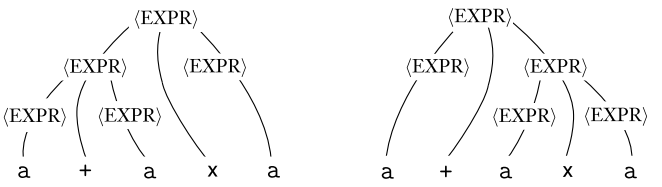

FIGURE 2.6
The two parse trees for the string a+axa in grammar G_5

This grammar doesn't capture the usual precedence relations and so may group the $+$ before the $\times$ or vice versa. In contrast, grammar G_4 generates exactly the same language, but every generated string has a unique parse tree. Hence G_4 is unambiguous, whereas G_5 is ambiguous.

Grammar G_2 (page 103) is another example of an ambiguous grammar. The sentence `the girl touches the boy with the flower` has two different derivations. In Exercise 2.8 you are asked to give the two parse trees and observe their correspondence with the two different ways to read that sentence.

Now we formalize the notion of ambiguity. When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees, not two different derivations. Two derivations may differ merely in the order in which they replace variables yet not in their overall structure. To concentrate on structure, we define a type of derivation that replaces variables in a fixed order. A derivation of a string w in a grammar G is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced. The derivation preceding Definition 2.2 (page 104) is a leftmost derivation.

DEFINITION 2.7

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called ***inherently ambiguous***. Problem 2.29 asks you to prove that the language $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.

CHOMSKY NORMAL FORM

When working with context-free grammars, it is often convenient to have them in simplified form. One of the simplest and most useful forms is called the

Chomsky normal form. Chomsky normal form is useful in giving algorithms for working with context-free grammars, as we do in Chapters 4 and 7.

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

PROOF IDEA We can convert any grammar G into Chomsky normal form. The conversion has several stages wherein rules that violate the conditions are replaced with equivalent ones that are satisfactory. First, we add a new start variable. Then, we eliminate all ***ε-rules*** of the form $A \to \varepsilon$. We also eliminate all ***unit rules*** of the form $A \to B$. In both cases we patch up the grammar to be sure that it still generates the same language. Finally, we convert the remaining rules into the proper form.

PROOF First, we add a new start variable S_0 and the rule $S_0 \to S$, where S was the original start variable. This change guarantees that the start variable doesn't occur on the right-hand side of a rule.

Second, we take care of all ε-rules. We remove an ε-rule $A \to \varepsilon$, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \to uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \to uv$. We do so for each *occurrence* of an A, so the rule $R \to uAvAw$ causes us to add $R \to uvAw$, $R \to uAvw$, and $R \to uvw$. If we have the rule $R \to A$, we add $R \to \varepsilon$ unless we had previously removed the rule $R \to \varepsilon$. We repeat these steps until we eliminate all ε-rules not involving the start variable.

Third, we handle all unit rules. We remove a unit rule $A \to B$. Then, whenever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed. As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.

Finally, we convert all remaining rules into the proper form. We replace each rule $A \to u_1 u_2 \cdots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol,

with the rules $A \to u_1A_1$, $A_1 \to u_2A_2$, $A_2 \to u_3A_3$, ..., and $A_{k-2} \to u_{k-1}u_k$. The A_i's are new variables. We replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rule $U_i \to u_i$.

EXAMPLE 2.10

Let G_6 be the following CFG and convert it to Chomsky normal form by using the conversion procedure just given. The series of grammars presented illustrates the steps in the conversion. Rules shown in bold have just been added. Rules shown in gray have just been removed.

1. The original CFG G_6 is shown on the left. The result of applying the first step to make a new start variable appears on the right.

$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

$$\boldsymbol{S_0 \to S}$$
$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

2. Remove ε-rules $B \to \varepsilon$, shown on the left, and $A \to \varepsilon$, shown on the right.

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a}$$
$$A \to B \mid S \mid \boldsymbol{\varepsilon}$$
$$B \to \mathtt{b} \mid \boldsymbol{\varepsilon}$$

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S}$$
$$A \to B \mid S \mid \boldsymbol{\varepsilon}$$
$$B \to \mathtt{b}$$

3a. Remove unit rules $S \to S$, shown on the left, and $S_0 \to S$, shown on the right.

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid \boldsymbol{S}$$
$$A \to B \mid S$$
$$B \to \mathtt{b}$$

$$S_0 \to \boldsymbol{S} \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to B \mid S$$
$$B \to \mathtt{b}$$

3b. Remove unit rules $A \to B$ and $A \to S$.

$$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to \boldsymbol{B} \mid S \mid \mathbf{b}$$
$$B \to \mathtt{b}$$

$$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to \boldsymbol{S} \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$$
$$B \to \mathtt{b}$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6. (Actually the procedure given in Theorem 2.9 produces several variables U_i and several rules $U_i \rightarrow \mathtt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow \mathtt{a}$.)

$$
\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow \mathtt{a} \\
B &\rightarrow \mathtt{b}
\end{aligned}
$$

2.2 PUSHDOWN AUTOMATA

In this section we introduce a new type of computational model called ***pushdown automata***. These automata are like nondeterministic finite automata but have an extra component called a ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it. Certain languages are more easily described in terms of generators, whereas others are more easily described by recognizers.

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read.

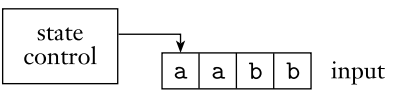

FIGURE 2.11
Schematic of a finite automaton