

FIGURE 2.6 The two parse trees for the string $a+a*x$ 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 $\{a^i b^j c^k \mid i = j \text{ or } j = k\}$ 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

$$A \rightarrow BC$$

$$A \rightarrow a$$

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 $S \rightarrow \epsilon$, 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 $A \rightarrow \epsilon$. We also eliminate all *unit rules* of the form $A \rightarrow B$. 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 $S_0 \rightarrow S$, 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 $A \rightarrow \epsilon$, 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 $R \rightarrow uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \rightarrow uv$. We do so for each occurrence of an A , so the rule $R \rightarrow uAvAw$ causes us to add $R \rightarrow uvAw$, $R \rightarrow uAvw$, and $R \rightarrow uvw$. If we have the rule $R \rightarrow A$, we add $R \rightarrow \epsilon$ unless we had previously removed the rule $R \rightarrow \epsilon$. 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 $A \rightarrow B$. Then, whenever a rule $B \rightarrow u$ appears, we add the rule $A \rightarrow u$ 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 $A \rightarrow u_1 u_2 \dots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol,

with the rules $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2$, $A_2 \rightarrow u_3A_3, \dots$, and $A_{k-2} \rightarrow u_{k-1}u_k$. 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 $U_i \rightarrow u_i$.

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.

$$\begin{array}{ll}
 S \rightarrow ASA | aB & S_0 \rightarrow S \\
 A \rightarrow B | S & S \rightarrow ASA | aB \\
 B \rightarrow b | \epsilon & A \rightarrow B | S \\
 & B \rightarrow b | \epsilon
 \end{array}$$

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

$$\begin{array}{ll}
 S_0 \rightarrow S & S_0 \rightarrow S \\
 S \rightarrow ASA | aB | a & S \rightarrow ASA | aB | a | SA | AS | S \\
 A \rightarrow B | S | \epsilon & A \rightarrow B | S \\
 B \rightarrow b | \epsilon & B \rightarrow b
 \end{array}$$

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

$$\begin{array}{ll}
 S_0 \rightarrow S & S_0 \rightarrow S | ASA | aB | a | SA | AS \\
 S \rightarrow ASA | aB | a | SA | AS | S & S \rightarrow ASA | aB | a | SA | AS \\
 A \rightarrow B | S & A \rightarrow B | S \\
 B \rightarrow b & B \rightarrow b
 \end{array}$$

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

$$\begin{array}{ll}
 S_0 \rightarrow ASA | aB | a | SA | AS & S_0 \rightarrow ASA | aB | a | SA | AS \\
 S \rightarrow ASA | aB | a | SA | AS & S \rightarrow ASA | aB | a | SA | AS \\
 A \rightarrow B | S | b & A \rightarrow S | b | ASA | aB | a | SA | AS \\
 B \rightarrow b & B \rightarrow b
 \end{array}$$

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 $U_i \rightarrow a$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow a$.)

$$\begin{array}{l}
 S_0 \rightarrow AA_1 | UB | a | SA | AS \\
 S \rightarrow AA_1 | UB | a | SA | AS \\
 A \rightarrow b | AA_1 | UB | a | SA | AS \\
 A_1 \rightarrow SA \\
 U \rightarrow a \\
 B \rightarrow b
 \end{array}$$

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.

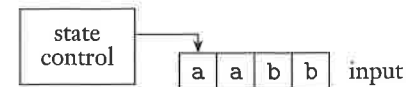


FIGURE 2.11
Schematic of a finite automaton