

# Chapter 5

## Context-Free Languages

**I**n the last chapter, we discovered that not all languages are regular. While regular languages are effective in describing certain simple patterns, one does not need to look very far for examples of nonregular languages. The relevance of these limitations to programming languages becomes evident if we reinterpret some of the examples. If in  $L = \{a^n b^n : n \geq 0\}$  we substitute a left parenthesis for  $a$  and a right parenthesis for  $b$ , then parentheses strings such as  $()$  and  $((()))$  are in  $L$ , but  $((()$  is not. The language therefore describes a simple kind of nested structure found in programming languages, indicating that some properties of programming languages require something beyond regular languages. In order to cover this and other more complicated features we must enlarge the family of languages. This leads us to consider **context-free** languages and grammars.

We begin this chapter by defining context-free grammars and languages, illustrating the definitions with some simple examples. Next, we consider the important membership problem; in particular we ask how we can tell if a given string is derivable from a given grammar. Explaining a sentence through its grammatical derivation is familiar to most of us from a study

of natural languages and is called **parsing**. Parsing is a way of describing sentence structure. It is important whenever we need to understand the meaning of a sentence, as we do for instance in translating from one language to another. In computer science, this is relevant in interpreters, compilers, and other translating programs.

The topic of context-free languages is perhaps the most important aspect of formal language theory as it applies to programming languages. Actual programming languages have many features that can be described elegantly by means of context-free languages. What formal language theory tells us about context-free languages has important applications in the design of programming languages as well as in the construction of efficient compilers. We touch upon this briefly in Section 5.3.

---

## 5.1 Context-Free Grammars

The productions in a regular grammar are restricted in two ways: the left side must be a single variable, while the right side has a special form. To create grammars that are more powerful, we must relax some of these restrictions. By retaining the restriction on the left side, but permitting anything on the right, we get context-free grammars.

---

### Definition 5.1

A grammar  $G = (V, T, S, P)$  is said to be **context-free** if all productions in  $P$  have the form

$$A \rightarrow x,$$

where  $A \in V$  and  $x \in (V \cup T)^*$ .

A language  $L$  is said to be context-free if and only if there is a context-free grammar  $G$  such that  $L = L(G)$ .

---

Every regular grammar is context-free, so a regular language is also a context-free one. But, as we know from simple examples such as  $\{a^n b^n\}$ , there are nonregular languages. We have already shown in Example 1.11 that this language can be generated by a context-free grammar, so we see that the family of regular languages is a proper subset of the family of context-free languages.

Context-free grammars derive their name from the fact that the substitution of the variable on the left of a production can be made any time such a variable appears in a sentential form. It does not depend on the

symbols in the rest of the sentential form (the context). This feature is the consequence of allowing only a single variable on the left side of the production.

## Examples of Context-Free Languages

### Example 5.1

The grammar  $G = (\{S\}, \{a, b\}, S, P)$ , with productions

$$\begin{aligned} S &\rightarrow aSa, \\ S &\rightarrow bSb, \\ S &\rightarrow \lambda, \end{aligned}$$

is context-free. A typical derivation in this grammar is

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbbaa.$$

This makes it clear that

$$L(G) = \{ww^R : w \in \{a, b\}^*\}.$$

The language is context-free, but as shown in Example 4.8, it is not regular. ■

### Example 5.2

The grammar  $G$ , with productions

$$\begin{aligned} S &\rightarrow abB, \\ A &\rightarrow aaBb, \\ B &\rightarrow bbAa, \\ A &\rightarrow \lambda, \end{aligned}$$

is context-free. We leave it to the reader to show that

$$L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}.$$

Both of the above examples involve grammars that are not only context-free, but linear. Regular and linear grammars are clearly context-free, but a context-free grammar is not necessarily linear.

**Example 5.3**

The language

$$L = \{a^n b^m : n \neq m\}$$

is context-free.

To show this, we need to produce a context-free grammar for the language. The case of  $n = m$  was solved in Example 1.11 and we can build on that solution. Take the case  $n > m$ . We first generate a string with an equal number of  $a$ 's and  $b$ 's, then add extra  $a$ 's on the left. This is done with

$$\begin{aligned} S &\rightarrow AS_1, \\ S_1 &\rightarrow aS_1b|\lambda, \\ A &\rightarrow aA|a. \end{aligned}$$

We can use similar reasoning for the case  $n < m$ , and we get the answer

$$\begin{aligned} S &\rightarrow AS_1|S_1B, \\ S_1 &\rightarrow aS_1b|\lambda, \\ A &\rightarrow aA|a, \\ B &\rightarrow bB|b. \end{aligned}$$

The resulting grammar is context-free, hence  $L$  is a context-free language. However, the grammar is not linear.

The particular form of the grammar given here was chosen for the purpose of illustration; there are many other equivalent context-free grammars. In fact, there are some simple linear ones for this language. In Exercise 25 at the end of this section you are asked to find one of them. ■

**Example 5.4**

Consider the grammar with productions

$$S \rightarrow aSb|SS|\lambda.$$

This is another grammar that is context-free, but not linear. Some strings in  $L(G)$  are  $abaabb$ ,  $aababb$ , and  $ababab$ . It is not difficult to conjecture and prove that

$$\begin{aligned} L = \{w \in \{a, b\}^* : n_a(w) = n_b(w) \text{ and } n_a(v) \geq n_b(v), \\ \text{where } v \text{ is any prefix of } w\}. \end{aligned} \tag{5.1}$$

We can see the connection with programming languages clearly if we replace  $a$  and  $b$  with left and right parentheses, respectively. The language  $L$

includes such strings as  $(( ))$  and  $()()()$  and is in fact the set of all properly nested parenthesis structures for the common programming languages.

Here again there are many other equivalent grammars. But, in contrast to Example 5.3, it is not so easy to see if there are any linear ones. We will have to wait until Chapter 8 before we can answer this question. ■

## Leftmost and Rightmost Derivations

In context-free grammars that are not linear, a derivation may involve sentential forms with more than one variable. In such cases, we have a choice in the order in which variables are replaced. Take for example the grammar  $G = (\{A, B, S\}, \{a, b\}, S, P)$  with productions

1.  $S \rightarrow AB$ .
2.  $A \rightarrow aaA$ .
3.  $A \rightarrow \lambda$ .
4.  $B \rightarrow Bb$ .
5.  $B \rightarrow \lambda$ .

It is easy to see that this grammar generates the language  $L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}$ .

Consider now the two derivations

$$S \xRightarrow{1} AB \xRightarrow{2} aaAB \xRightarrow{3} aaB \xRightarrow{4} aaBb \xRightarrow{5} aab$$

and

$$S \xRightarrow{1} AB \xRightarrow{4} ABb \xRightarrow{2} aaABb \xRightarrow{5} aaAb \xRightarrow{3} aab.$$

In order to show which production is applied, we have numbered the productions and written the appropriate number on the  $\Rightarrow$  symbol. From this we see that the two derivations not only yield the same sentence but use exactly the same productions. The difference is entirely in the order in which the productions are applied. To remove such irrelevant factors, we often require that the variables be replaced in a specific order.

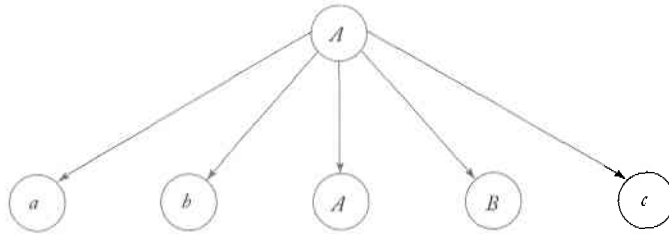
---

### Definition 5.2

A derivation is said to be **leftmost** if in each step the leftmost variable in the sentential form is replaced. If in each step the rightmost variable is replaced, we call the derivation **rightmost**.

---

Figure 5.1

**Example 5.5**

Consider the grammar with productions

$$S \rightarrow aAB,$$

$$A \rightarrow bBb,$$

$$B \rightarrow A|\lambda.$$

Then

$$S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$$

is a leftmost derivation of the string  $abbbb$ . A rightmost derivation of the same string is

$$S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb.$$

## Derivation Trees

A second way of showing derivations, independent of the order in which productions are used, is by a **derivation tree**. A derivation tree is an ordered tree in which nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right sides. For example, Figure 5.1 shows part of a derivation tree representing the production

$$A \rightarrow abABc.$$

In a derivation tree, a node labeled with a variable occurring on the left side of a production has children consisting of the symbols on the right side of that production. Beginning with the root, labeled with the start symbol and ending in leaves that are terminals, a derivation tree shows how each variable is replaced in the derivation. The following definition makes this notion precise.

**Definition 5.3**

Let  $G = (V, T, S, P)$  be a context-free grammar. An ordered tree is a derivation tree for  $G$  if and only if it has the following properties.

1. The root is labeled  $S$ .
2. Every leaf has a label from  $T \cup \{\lambda\}$ .
3. Every interior vertex (a vertex which is not a leaf) has a label from  $V$ .
4. If a vertex has label  $A \in V$ , and its children are labeled (from left to right)  $a_1, a_2, \dots, a_n$ , then  $P$  must contain a production of the form

$$A \rightarrow a_1 a_2 \cdots a_n.$$

5. A leaf labeled  $\lambda$  has no siblings, that is, a vertex with a child labeled  $\lambda$  can have no other children.

A tree that has properties 3, 4 and 5, but in which 1 does not necessarily hold and in which property 2 is replaced by:

- 2a. Every leaf has a label from  $V \cup T \cup \{\lambda\}$

is said to be a **partial derivation tree**.

The string of symbols obtained by reading the leaves of the tree from left to right, omitting any  $\lambda$ 's encountered, is said to be the **yield** of the tree. The descriptive term *left to right* can be given a precise meaning. The yield is the string of terminals in the order they are encountered when the tree is traversed in a depth-first manner, always taking the leftmost unexplored branch.

### Example 5.6

Consider the grammar  $G$ , with productions

$$S \rightarrow aAB,$$

$$A \rightarrow bBb,$$

$$B \rightarrow A|\lambda.$$

The tree in Figure 5.2 is a partial derivation tree for  $G$ , while the tree in Figure 5.3 is a derivation tree. The string  $abBbB$ , which is the yield of the first tree, is a sentential form of  $G$ . The yield of the second tree,  $abbbb$  is a sentence of  $L(G)$ .

Figure 5.2

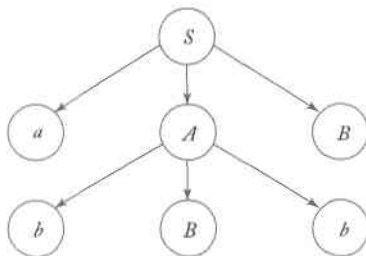
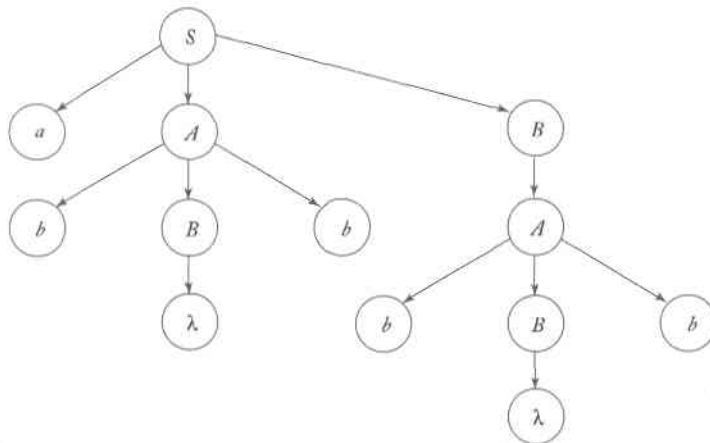


Figure 5.3



### Relation Between Sentential Forms and Derivation Trees

Derivation trees give a very explicit and easily comprehended description of a derivation. Like transition graphs for finite automata, this explicitness is a great help in making arguments. First, though, we must establish the connection between derivations and derivation trees.

#### Theorem 5.1

Let  $G = (V, T, S, P)$  be a context-free grammar. Then for every  $w \in L(G)$ , there exists a derivation tree of  $G$  whose yield is  $w$ . Conversely, the yield of any derivation tree is in  $L(G)$ . Also, if  $t_G$  is any partial derivation tree for  $G$  whose root is labeled  $S$ , then the yield of  $t_G$  is a sentential form of  $G$ .

**Proof:** First we show that for every sentential form of  $L(G)$  there is a corresponding partial derivation tree. We do this by induction on the number of steps in the derivation. As a basis, we note that the claimed result is true for every sentential form derivable in one step. Since  $S \Rightarrow u$  implies that there is a production  $S \rightarrow u$ , this follows immediately from Definition 5.3.

Assume that for every sentential form derivable in  $n$  steps, there is a corresponding partial derivation tree. Now any  $w$  derivable in  $n + 1$  steps



must be such that

$$S \xRightarrow{*} xAy, \quad x, y \in (V \cup T)^*, \quad A \in V,$$

in  $n$  steps, and

$$xAy \Rightarrow xa_1a_2 \cdots a_my = w, a_1 \in V \cup T.$$

Since by the inductive assumption there is a partial derivation tree with yield  $xAy$ , and since the grammar must have production  $A \rightarrow a_1a_2 \cdots a_m$ , we see that by expanding the leaf labeled  $A$ , we get a partial derivation tree with yield  $xa_1a_2 \cdots a_my = w$ . By induction, we therefore claim that the result is true for all sentential forms.

In a similar vein, we can show that every partial derivation tree represents some sentential form. We will leave this as an exercise.

Since a derivation tree is also a partial derivation tree whose leaves are terminals, it follows that every sentence in  $L(G)$  is the yield of some derivation tree of  $G$  and that the yield of every derivation tree is in  $L(G)$ . ■

Derivation trees show which productions are used in obtaining a sentence, but do not give the order of their application. Derivation trees are able to represent any derivation, reflecting the fact that this order is irrelevant, an observation which allows us to close a gap in the preceding discussion. By definition, any  $w \in L(G)$  has a derivation, but we have not claimed that it also had a leftmost or rightmost derivation. However, once we have a derivation tree, we can always get a leftmost derivation by thinking of the tree as having been built in such a way that the leftmost variable in the tree was always expanded first. Filling in a few details, we are led to the not surprising result that any  $w \in L(G)$  has a leftmost and a rightmost derivation (for details, see Exercise 24 at the end of this section).

## EXERCISES

1. Complete the arguments in Example 5.2, showing that the language given is generated by the grammar.
2. Draw the derivation tree corresponding to the derivation in Example 5.1.
3. Give a derivation tree for  $w = abbbaabbaba$  for the grammar in Example 5.2. Use the derivation tree to find a leftmost derivation.
4. Show that the grammar in Example 5.4 does in fact generate the language described in Equation 5.1. ●
5. Is the language in Example 5.2 regular?
6. Complete the proof in Theorem 5.1 by showing that the yield of every partial derivation tree with root  $S$  is a sentential form of  $G$ .

6 (7). Find context-free grammars for the following languages (with  $n \geq 0, m \geq 0$ ).

- (a)  $L = \{a^n b^m : n \leq m + 3\}$  ●
- (b)  $L = \{a^n b^m : n \neq m - 1\}$
- (c)  $L = \{a^n b^m : n \neq 2m\}$
- (d)  $L = \{a^n b^m : 2n \leq m \leq 3n\}$  ●
- (e)  $L = \{w \in \{a, b\}^* : n_a(w) \neq n_b(w)\}$
- (f)  $L = \{w \in \{a, b\}^* : n_a(v) \geq n_b(v), \text{ where } v \text{ is any prefix of } w\}$
- (g)  $L = \{w \in \{a, b\}^* : n_a(w) = 2n_b(w) + 1\}$ .

7 (8). Find context-free grammars for the following languages (with  $n \geq 0, m \geq 0, k \geq 0$ ).

- (a)  $L = \{a^n b^m c^k : n = m \text{ or } m \leq k\}$  ●
- (b)  $L = \{a^n b^m c^k : n = m \text{ or } m \neq k\}$
- (c)  $L = \{a^n b^m c^k : k = n + m\}$
- (d)  $L = \{a^n b^m c^k : n + 2m = k\}$
- (e)  $L = \{a^n b^m c^k : k = |n - m|\}$  ●
- (f)  $L = \{w \in \{a, b, c\}^* : n_a(w) + n_b(w) \neq n_c(w)\}$
- (g)  $L = \{a^n b^m c^k, k \neq n + m\}$
- (h)  $L = \{a^n b^n c^k : k \geq 3\}$ .

9. Find a context-free grammar for  $\text{head}(L)$ , where  $L$  is the language in Exercise 7(a) above. For the definition of  $\text{head}$  see Exercise 18, Section 4.1.

10. Find a context-free grammar for  $\Sigma = \{a, b\}$  for the language  $L = \{a^n w w^R b^n : w \in \Sigma^*, n \geq 1\}$ .

★11. Given a context-free grammar  $G$  for a language  $L$ , show how one can create from  $G$  a grammar  $\hat{G}$  so that  $L(\hat{G}) = \text{head}(L)$ .

12. Let  $L = \{a^n b^n : n \geq 0\}$ .

- (a) Show that  $L^2$  is context-free. ●
- (b) Show that  $L^k$  is context-free for any given  $k \geq 1$ .
- (c) Show that  $\bar{L}$  and  $L^*$  are context-free.

13. Let  $L_1$  be the language in Exercise 8(a) and  $L_2$  the language in Exercise 8(d). Show that  $L_1 \cup L_2$  is a context-free language.

14. Show that the following language is context-free.

$$L = \{uvww^R : u, v, w \in \{a, b\}^+, |u| = |w| = 2\}$$

★15. Show that the complement of the language in Example 5.1 is context-free. ●

16. Show that the complement of the language in Exercise 8(b) is context-free.
17. Show that the language  $L = \{w_1cw_2 : w_1, w_2 \in \{a, b\}^+, w_1 \neq w_2^R\}$ , with  $\Sigma = \{a, b, c\}$ , is context-free.
18. Show a derivation tree for the string  $aabbbb$  with the grammar

$$\begin{aligned} S &\rightarrow AB|\lambda, \\ A &\rightarrow aB, \\ B &\rightarrow Sb. \end{aligned}$$

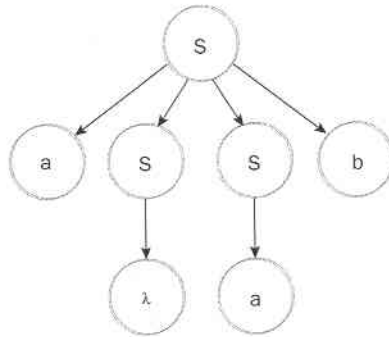
Give a verbal description of the language generated by this grammar.

19. Consider the grammar with productions

$$\begin{aligned} S &\rightarrow aaB, \\ A &\rightarrow bBb|\lambda, \\ B &\rightarrow Aa. \end{aligned}$$

Show that the string  $aabbabba$  is not in the language generated by this grammar.

20. Consider the derivation tree below.



Find a simple grammar  $G$  for which this is the derivation tree of the string  $aab$ . Then find two more sentences of  $L(G)$ .

21. Define what one might mean by properly nested parenthesis structures involving two kinds of parentheses, say  $()$  and  $[\ ]$ . Intuitively, properly nested strings in this situation are  $([\ ])$ ,  $([[\ ]])$ ,  $([\ ])([\ ])$ , but not  $([\ ])$  or  $([\ ])$ . Using your definition, give a context-free grammar for generating all properly nested parentheses.
22. Find a context-free grammar for the set of all regular expressions on the alphabet  $\{a, b\}$ .
23. Find a context-free grammar that can generate all the production rules for context-free grammars with  $T = \{a, b\}$  and  $V = \{A, B, C\}$ .
24. Prove that if  $G$  is a context-free grammar, then every  $w \in L(G)$  has a leftmost and rightmost derivation. Give an algorithm for finding such derivations from a derivation tree.

25. Find a linear grammar for the language in Example 5.3.
26. Let  $G = (V, T, S, P)$  be a context-free grammar such that every one of its productions is of the form  $A \rightarrow v$ , with  $|v| = k > 1$ . Show that the derivation tree for any  $w \in L(G)$  has a height  $h$  such that

$$\log_k |w| \leq h \leq \frac{(|w| - 1)}{k - 1}.$$

## 5.2 Parsing and Ambiguity

We have so far concentrated on the generative aspects of grammars. Given a grammar  $G$ , we studied the set of strings that can be derived using  $G$ . In cases of practical applications, we are also concerned with the analytical side of the grammar: given a string  $w$  of terminals, we want to know whether or not  $w$  is in  $L(G)$ . If so, we may want to find a derivation of  $w$ . An algorithm that can tell us whether  $w$  is in  $L(G)$  is a membership algorithm. The term **parsing** describes finding a sequence of productions by which a  $w \in L(G)$  is derived.

### Parsing and Membership

Given a string  $w$  in  $L(G)$ , we can parse it in a rather obvious fashion: we systematically construct all possible (say, leftmost) derivations and see whether any of them match  $w$ . Specifically, we start at round one by looking at all productions of the form

$$S \rightarrow x,$$

finding all  $x$  that can be derived from  $S$  in one step. If none of these result in a match with  $w$ , we go to the next round, in which we apply all applicable productions to the leftmost variable of every  $x$ . This gives us a set of sentential forms, some of them possibly leading to  $w$ . On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that  $w$  can never be derived from them, but in general, we will have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If  $w \in L(G)$ , then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of  $w$ .

For reference below, we will call this the **exhaustive search parsing** method. It is a form of **top-down parsing**, which we can view as the construction of a derivation tree from the root down.

**Example 5.7** Consider the grammar

$$S \rightarrow SS|aSb|bSa|\lambda$$

and the string  $w = aabb$ . Round one gives us

1.  $S \Rightarrow SS$ ,
2.  $S \Rightarrow aSb$ ,
3.  $S \Rightarrow bSa$ ,
4.  $S \Rightarrow \lambda$ .

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$\begin{aligned} S &\Rightarrow SS \Rightarrow SSS, \\ S &\Rightarrow SS \Rightarrow aSbS, \\ S &\Rightarrow SS \Rightarrow bSaS, \\ S &\Rightarrow SS \Rightarrow S, \end{aligned}$$

which are obtained by replacing the leftmost  $S$  in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$\begin{aligned} S &\Rightarrow aSb \Rightarrow aSSb, \\ S &\Rightarrow aSb \Rightarrow aaSbb, \\ S &\Rightarrow aSb \Rightarrow abSab, \\ S &\Rightarrow aSb \Rightarrow ab. \end{aligned}$$

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore  $aabb$  is in the language generated by the grammar under consideration.

---

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a  $w \in L(G)$ , it is possible that it never terminates for strings not in  $L(G)$ . This is certainly the case in

the previous example; with  $w = abb$ , the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions  $S \rightarrow \lambda$ ; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form  $A \rightarrow \lambda$  as well as those of the form  $A \rightarrow B$ . As we will see in the next chapter, this restriction does not affect the power of the resulting grammars in any significant way.

### Example 5.8

The grammar

$$S \rightarrow SS | aSb | bSa | ab | ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any  $w \in \{a, b\}^+$ , the exhaustive search parsing method will always terminate in no more than  $|w|$  rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After  $|w|$  rounds we have either produced a parsing or we know that  $w \notin L(G)$ . ■

The idea in this example can be generalized and made into a theorem for context-free languages in general.

### Theorem 5.2

Suppose that  $G = (V, T, S, P)$  is a context-free grammar which does not have any rules of the form

$$A \rightarrow \lambda,$$

or

$$A \rightarrow B,$$

where  $A, B \in V$ . Then the exhaustive search parsing method can be made into an algorithm which, for any  $w \in \Sigma^*$ , either produces a parsing of  $w$ , or tells us that no parsing is possible.

**Proof:** For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of

terminal symbols can exceed  $|w|$ , a derivation cannot involve more than  $2|w|$  rounds, at which time we either have a successful parsing or  $w$  cannot be generated by the grammar. ■

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than  $|P|$  sentential forms after one round, no more than  $|P|^2$  sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than  $2|w|$  rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|}. \quad (5.2)$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (5.2) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We will not pursue it here except for some isolated results.

### Theorem 5.3

For every context-free grammar there exists an algorithm that parses any  $w \in L(G)$  in a number of steps proportional to  $|w|^3$ .

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In Section 6.3 we will take this question up again briefly. More details can be found in Harrison 1978 and Hopcroft and Ullman 1979. One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a compiler based on it would need an excessive amount of time to parse even a moderately long program. What we would like to have is a parsing method which takes time proportional to the length of the string. We refer to such a method as a **linear time** parsing

algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

---

**Definition 5.4**

A context-free grammar  $G = (V, T, S, P)$  is said to be a **simple grammar** or **s-grammar** if all its productions are of the form

$$A \rightarrow ax,$$

where  $A \in V$ ,  $a \in T$ ,  $x \in V^*$ , and any pair  $(A, a)$  occurs at most once in  $P$ .

---

**Example 5.9**

The grammar

$$S \rightarrow aS | bSS | c$$

is an s-grammar. The grammar

$$S \rightarrow aS | bSS | aSS | c$$

is not an s-grammar because the pair  $(S, a)$  occurs in the two productions  $S \rightarrow aS$  and  $S \rightarrow aSS$ . ■

---

While s-grammars are quite restrictive, they are of some interest. As we will see in the next section, many features of common programming languages can be described by s-grammars.

If  $G$  is an s-grammar, then any string  $w$  in  $L(G)$  can be parsed with an effort proportional to  $|w|$ . To see this, look at the exhaustive search method and the string  $w = a_1 a_2 \cdots a_n$ . Since there can be at most one rule with  $S$  on the left, and starting with  $a_1$  on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable  $A_1$ , but since again there is at most one choice, we must have

$$S \xRightarrow{*} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than  $|w|$  steps.



## Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any  $w \in L(G)$ , exhaustive search parsing will produce a derivation tree for  $w$ . We say “a” derivation tree rather than “the” derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as **ambiguity**.

### Definition 5.5

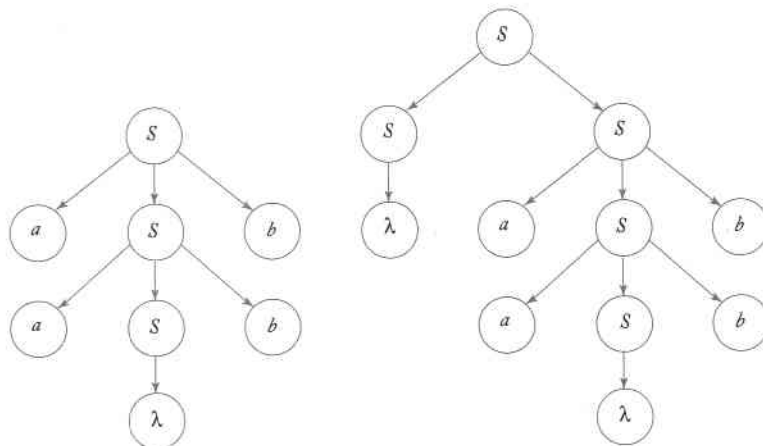
A context-free grammar  $G$  is said to be **ambiguous** if there exists some  $w \in L(G)$  that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

### Example 5.10

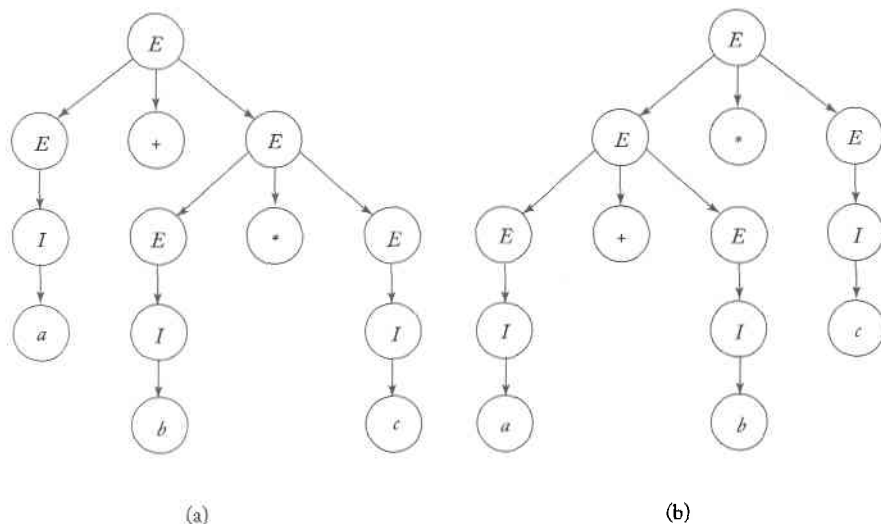
The grammar in Example 5.4, with productions  $S \rightarrow aSb | SS | \lambda$ , is ambiguous. The sentence  $aabb$  has the two derivation trees shown in Figure 5.4.

Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Figure 5.4



**Figure 5.5**  
Two derivation  
trees for  $a + b * c$ .

**Example 5.11**

Consider the grammar  $G = (V, T, E, P)$  with

$$V = \{E, I\},$$

$$T = \{a, b, c, +, *, (, )\},$$

and productions

$$E \rightarrow I,$$

$$E \rightarrow E + E,$$

$$E \rightarrow E * E,$$

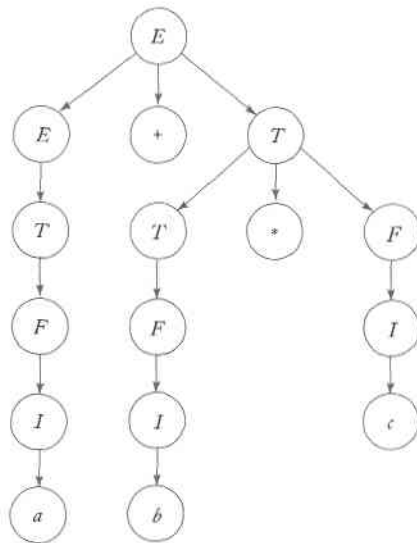
$$E \rightarrow (E),$$

$$I \rightarrow a \mid b \mid c.$$

The strings  $(a + b) * c$  and  $a * b + c$  are in  $L(G)$ . It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C and Pascal-like programming languages. The grammar is ambiguous. For instance, the string  $a + b * c$  has two different derivation trees, as shown in Figure 5.5.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators  $+$  and  $*$ . Since  $*$  normally has higher precedence than  $+$ , we would take Figure 5.5(a) as the correct parsing as it indicates that  $b * c$  is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Figure 5.6

**Example 5.12**

To rewrite the grammar in Example 5.11 we introduce new variables, taking  $V$  as  $\{E, T, F, I\}$  and replace the productions with

$$\begin{aligned}
 E &\rightarrow T, \\
 T &\rightarrow F, \\
 F &\rightarrow I, \\
 E &\rightarrow E + T, \\
 T &\rightarrow T * F, \\
 F &\rightarrow (E), \\
 I &\rightarrow a \mid b \mid c.
 \end{aligned}$$

A derivation tree of the sentence  $a + b * c$  is shown in Figure 5.6. No other derivation tree is possible for this string: the grammar is unambiguous. It also is equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we will later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

**Definition 5.6**

If  $L$  is a context-free language for which there exists an unambiguous grammar, then  $L$  is said to be unambiguous. If every grammar that generates  $L$  is ambiguous, then the language is called **inherently ambiguous**.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

**Example 5.13**

The language

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\},$$

with  $n$  and  $m$  non-negative, is an inherently ambiguous context-free language.

That  $L$  is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where  $L_1$  is generated by

$$S_1 \rightarrow S_1 c | A,$$

$$A \rightarrow aAb | \lambda,$$

and  $L_2$  is given by an analogous grammar with start symbol  $S_2$  and productions

$$S_2 \rightarrow aS_2 | B,$$

$$B \rightarrow bBc | \lambda.$$

Then  $L$  is generated by the combination of these two grammars with the additional production

$$S \rightarrow S_1 | S_2.$$

The grammar is ambiguous since the string  $a^n b^n c^n$  has two distinct derivations, one starting with  $S \Rightarrow S_1$ , the other with  $S \Rightarrow S_2$ . It does of course not follow from this that  $L$  is inherently ambiguous as there might exist some other nonambiguous grammars for it. But in some way  $L_1$  and  $L_2$  have conflicting requirements, the first putting a restriction on the number of  $a$ 's and  $b$ 's, while the second does the same for  $b$ 's and  $c$ 's. A few tries will

quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case  $n = m$  uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison 1978.

## EXERCISES

1. Find an s-grammar for  $L(aaa^*b + b)$ .
2. Find an s-grammar for  $L = \{a^n b^n : n \geq 1\}$ .
3. Find an s-grammar for  $L = \{a^n b^{n+1} : n \geq 2\}$ .
4. Show that every s-grammar is unambiguous.
5. Let  $G = (V, T, S, P)$  be an s-grammar. Give an expression for the maximum size of  $P$  in terms of  $|V|$  and  $|T|$ .
6. Show that the following grammar is ambiguous.

$$S \rightarrow AB|aaB,$$


$$A \rightarrow a|Aa,$$

$$B \rightarrow b.$$

7. Construct an unambiguous grammar equivalent to the grammar in Exercise 6.
8. Give the derivation tree for  $((a + b) * c) + a + b$ , using the grammar in Example 5.12.
9. Show that a regular language cannot be inherently ambiguous.
10. Give an unambiguous grammar that generates the set of all regular expressions on  $\Sigma = \{a, b\}$ .
11. Is it possible for a regular grammar to be ambiguous?
12. Show that the language  $L = \{ww^R : w \in \{a, b\}^*\}$  is not inherently ambiguous.
13. Show that the following grammar is ambiguous.

$$S \rightarrow aSbS|bSaS|\lambda$$

14. Show that the grammar in Example 5.4 is ambiguous, but that the language denoted by it is not.
15. Show that the grammar in Example 1.13 is ambiguous.
16. Show that the grammar in Example 5.5 is unambiguous.
17. Use the exhaustive search parsing method to parse the string  $abbbbb$  with the grammar in Example 5.5. In general, how many rounds will be needed to parse any string  $w$  in this language?

18. Show that the grammar in Example 1.14 is unambiguous.
19. Prove the following result. Let  $G = (V, T, S, P)$  be a context-free grammar in which every  $A \in V$  occurs on the left side of at most one production. Then  $G$  is unambiguous.
20. Find a grammar equivalent to that in Example 5.5 which satisfies the conditions of Theorem 5.2. 

### 5.3 Context-Free Grammars and Programming Languages

One of the most important uses of the theory of formal languages is in the definition of programming languages and in the construction of interpreters and compilers for them. The basic problem here is to define a programming language precisely and to use this definition as the starting point for the writing of efficient and reliable translation programs. Both regular and context-free languages are important in achieving this. As we have seen, regular languages are used in the recognition of certain simple patterns which occur in programming languages, but as we argued in the introduction to this chapter, we need context-free languages to model more complicated aspects.

As with most other languages, we can define a programming language by a grammar. It is traditional in writing on programming languages to use a convention for specifying grammars called the *Backus-Naur* form or BNF. This form is in essence the same as the notation we have used here, but the appearance is different. In BNF, variables are enclosed in triangular brackets. Terminal symbols are written without any special marking. BNF also uses subsidiary symbols such as  $|$ , much in the way we have done. Thus, the grammar in Example 5.12 might appear in BNF as

$$\begin{aligned}\langle expression \rangle &::= \langle term \rangle \mid \langle expression \rangle + \langle term \rangle, \\ \langle term \rangle &::= \langle factor \rangle \mid \langle term \rangle * \langle factor \rangle,\end{aligned}$$

and so on. The symbols  $+$  and  $*$  are terminals. The symbol  $|$  is used as an alternator as in our notation, but  $::=$  is used instead of  $\rightarrow$ . BNF descriptions of programming languages tend to use more explicit variable identifiers to make the intent of the production explicit. But otherwise there are no significant differences between the two notations.

Many parts of a Pascal-like programming language are susceptible to definition by restricted forms of context-free grammars. For example, a Pascal **if-then-else** statement can be defined as

$$\langle if\_statement \rangle ::= if \langle expression \rangle \langle then\_clause \rangle \langle else\_clause \rangle.$$

Here the keyword *if* is a terminal symbol. All other terms are variables which still have to be defined. If we check this against Definition 5.4, we see that this looks like an s-grammar production. The variable  $\langle if\_statement \rangle$  on the left is always associated with the terminal *if* on the right. For this reason such a statement is easily and efficiently parsed. We see here a reason why we use keywords in programming languages. Keywords not only provide some visual structure that can guide the reader of a program, but also make the work of a compiler much easier.

Unfortunately, not all features of a typical programming language can be expressed by an s-grammar. The rules for  $\langle expression \rangle$  above are not of this type, so that parsing becomes less obvious. The question then arises for what grammatical rules we can permit and still parse efficiently. In compilers, extensive use has been made of what are called LL and LR grammars, which have the ability to express the less obvious features of a programming language, yet allow us to parse in linear time. This is not a simple matter, and much of it is beyond the scope of our discussion. We will briefly touch on this topic in Chapter 6, but for our purposes it suffices to realize that such grammars exist and have been widely studied.

In connection with this, the issue of ambiguity takes on added significance. The specification of a programming language must be unambiguous, otherwise a program may yield very different results when processed by different compilers or run on different systems. As Example 5.11 shows, a naive approach can easily introduce ambiguity in the grammar. To avoid such mistakes we must be able to recognize and remove ambiguities. A related question is whether a language is or is not inherently ambiguous. What we need for this purpose are algorithms for detecting and removing ambiguities in context-free grammars and for deciding whether or not a context-free language is inherently ambiguous. Unfortunately, these are very difficult tasks, impossible in the most general sense, as we will see later.

Those aspects of a programming language which can be modeled by a context-free grammar are usually referred to as its **syntax**. However, it is normally the case that not all programs which are syntactically correct in this sense are in fact acceptable programs. For Pascal, the usual BNF definition allows constructs such as

$$\begin{array}{l} \text{var } x, y : \text{real}; \\ \quad x, z : \text{integer} \end{array}$$

or

$$\begin{array}{l} \text{var } x : \text{integer}; \\ \quad x := 3.2. \end{array}$$

Neither of these two constructs is acceptable to a Pascal compiler, since they violate other constraints, such as “an integer variable cannot be assigned

a real value.” This kind of rule is part of programming language semantics, since it has to do with how we interpret the meaning of a particular construct.

Programming language semantics are a complicated matter. Nothing as elegant and concise as context-free grammars exists for the specification of programming language semantics, and consequently some semantic features may be poorly defined or ambiguous. It is an ongoing concern both in programming languages and in formal language theory to find effective methods for defining programming language semantics. Several methods have been proposed, but none of them have been as universally accepted and as successful for semantic definition as context-free languages have been for syntax.

## EXERCISES

1. Give a complete definition of *expression* for Pascal.
2. Give a BNF definition for the Pascal **while** statement (leaving the general concept *statement* undefined).
3. Give a BNF grammar that shows the relation between a Pascal program and its subprograms.
4. Give a BNF definition of a FORTRAN **do** statement.
5. Give a definition of the correct form of the **if-else** statement in C.
6. Find examples of features of C that cannot be described by context-free grammars.