Chapter 5

Context-Free Languages

In 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

$$A \to x,$$

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

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

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

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

This, and similar derivations, make 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

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

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 $n = m$ is 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

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

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

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

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 26 at the end of this section you are asked to find one of them.

Example 5.4

Consider the grammar with productions

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

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

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

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 a grammar that is 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. $S \to AB.$
2. $A \to aaA.$
3. $A \to \lambda.$
4. $B \to Bb.$
5. $B \to \lambda.$

This grammar generates the language $L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}$. Carry out a few derivations to convince yourself of this.

Consider now the two derivations

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

and

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

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 also 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**.

Example 5.5

Consider the grammar with productions

$$S \to aAB,$$
$$A \to bBb,$$
$$B \to A|\lambda.$$

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

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

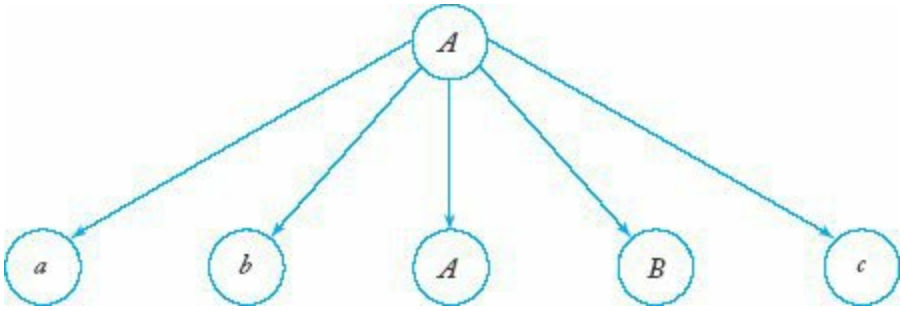
Derivation Trees

A second way of showing derivations, independent of the order in which productions are used, is by a **derivation** or **parse 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

$$A \to abABc.$$

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.

Figure 5.1



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 that 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) $a_1, a_2, \ldots, a_n$, then P must contain a production of the form

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

5. A leaf labeled λ has no siblings, that is, a vertex with a child labeled λ 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 λ'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

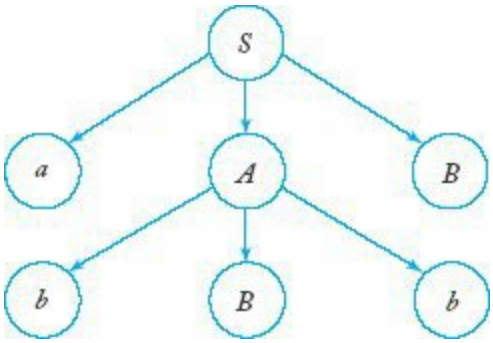
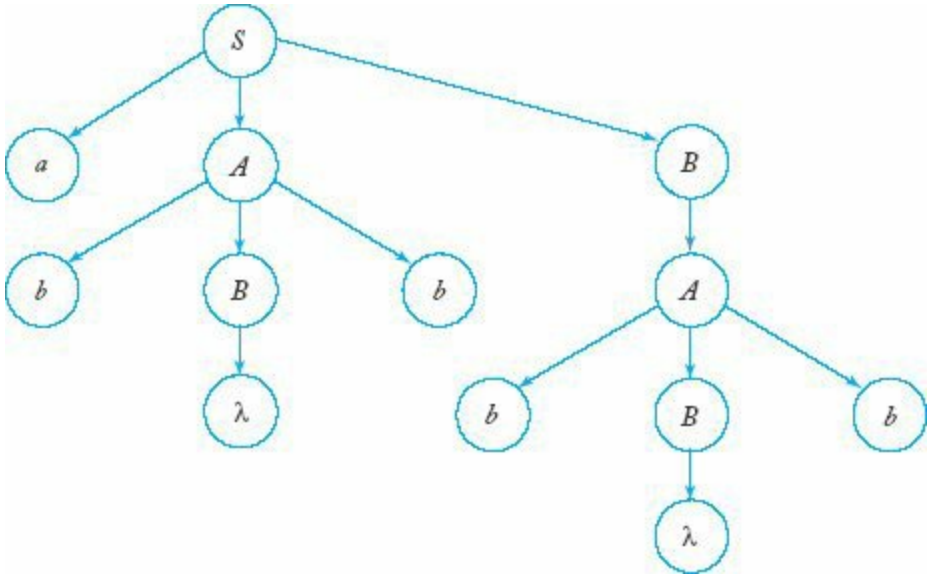


Figure 5.3



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

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

in n steps, and

$$xAy \Rightarrow xa_1a_2 \cdots a_m y = w, a_i \in V \cup T.$$

Since by the inductive assumption there is a partial derivation tree with yield xAy, and since the grammar must have production $A \rightarrow a_1a_2 \ldots a_m$, we see that by expanding the leaf labeled A, we get a partial derivation tree with yield $xa_1a_2 \ldots a_m y = w$. 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 that 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 25 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.

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) $L = \{w \in \{a, b\}^* : n_a(v) \geq n_b(v)$, where v is any prefix of $w\}$.

(g) $L = \{w \in \{a,b\}^* : n_a(w) = 2n_b(w) + 1\}$.

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

(a) $L = \{a^n b^m c^k : n = m$ or $m \leq k\}$.

(b) $L = \{a^n b^m c^k : n = m$ 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) $L = \{a^n b^m c^k : k \geq 3\}$.

9. Show that $L = \{w \in \{a,b,c\}^* : |w| = 3n_a(w)\}$ is a context-free language.

10. Find a context-free grammar for *head* (L), where L is the language in Exercise 7(a) above. For the definition of *head* see Exercise 18, Section 4.1.

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

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

13. 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 $\overline{L}$ and L^* are context-free.

14. 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.

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

$$L = \left\{uvwv^R : u, v, w \in \{a, b\}^+, |u| = |w| = 2\right\}.$$

***16.** Show that the complement of the language in Example 5.1 is context-free.

17. Show that the complement of the language in Exercise 8(c) is context-free.

18. Show that the language $L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^+, w_1 \neq w_2^R\}$, with $\Sigma = \{a,b,c\}$,is context-free.

19. Show a derivation tree for the string *aabbbb* with the grammar

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

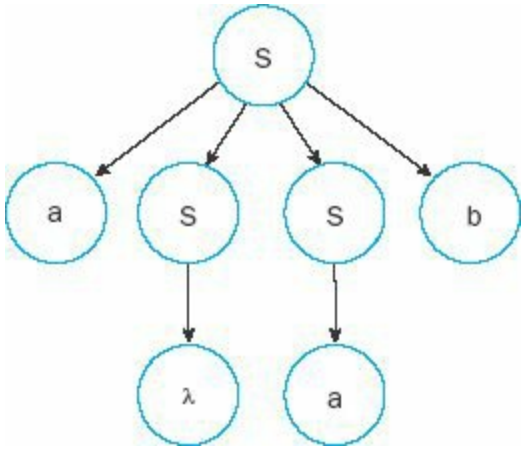
Give a verbal description of the language generated by this grammar.

20. 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.

21. Consider the derivation tree below.



Find a grammar G for which this is the derivation tree of the string aab. Then find two more sentences of $L(G)$. Find a sentence in $L(G)$ that has a derivation tree of height five or larger.

22. 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.

23. Find a context-free grammar for the set of all regular expressions on the alphabet $\{a, b\}$.

24. Find a context-free grammar that can generate all the production rules for context-free grammars with $T = \{a, b\}$ and $V = \{A, B, C\}$.

25. 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.

26. Find a linear grammar for the language in Example 5.3.

27. 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

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results 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 **exhaustive search parsing** or **brute force parsing.** 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

$$S \to SS \,|aSb|\, bSa|\lambda$$

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

$$S \Rightarrow SS \Rightarrow SSS,$$
$$S \Rightarrow SS \Rightarrow aSbS,$$
$$S \Rightarrow SS \Rightarrow bSaS,$$
$$S \Rightarrow SS \Rightarrow S,$$

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

$$S \Rightarrow aSb \Rightarrow aSSb,$$
$$S \Rightarrow aSb \Rightarrow aaSbb,$$
$$S \Rightarrow aSb \Rightarrow abSab,$$
$$S \Rightarrow aSb \Rightarrow ab.$$

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 that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B,$$

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

$$\begin{aligned} M &= |P| + |P|^2 + \cdots + |P|^{2|w|} \\ &= O(P^{2|w|+1}). \end{aligned} \tag{5.2}$$

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 parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that 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 $w = a_1a_2\ldots a_n$. 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

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

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.**

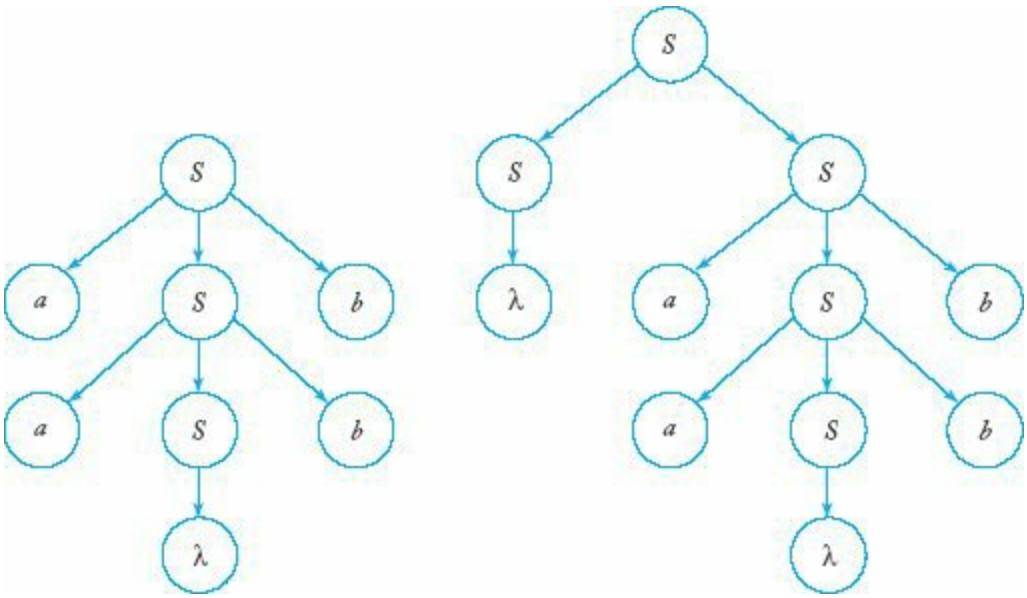
Definition 5.5

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.

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.

Example 5.11

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

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

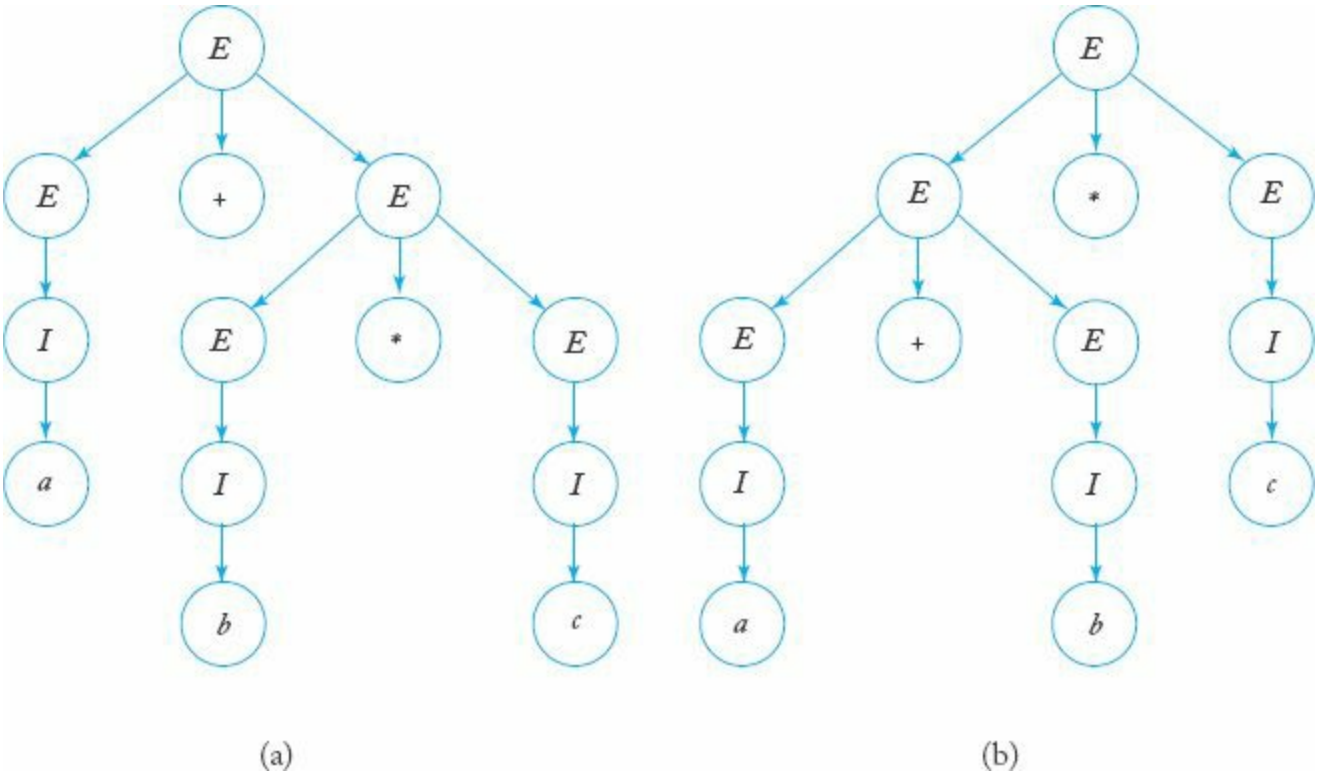
and productions

$$E \rightarrow I,$$
$$E \rightarrow E + E,$$
$$E \rightarrow E * E,$$
$$E \rightarrow (E),$$
$$I \rightarrow a \mid b \mid c.$$

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-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.

Figure 5.5

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



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.

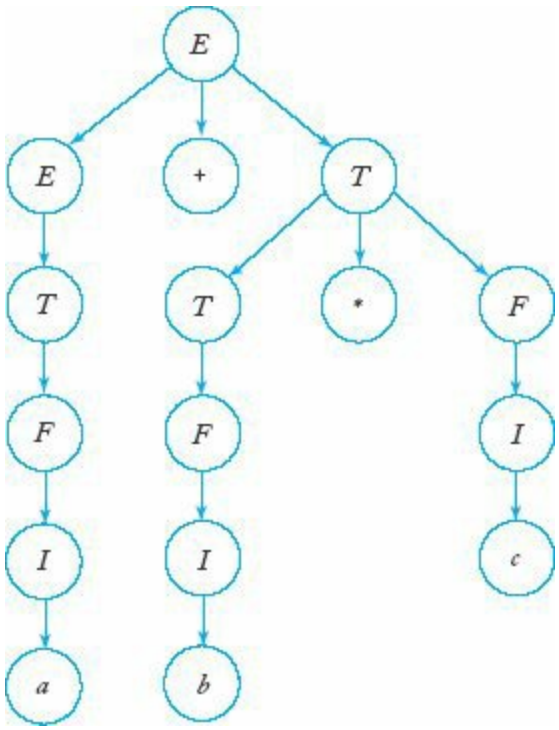
Example 5.12

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

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

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 is also 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.

Figure 5.6



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 nonnegative, 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

$$\begin{aligned} S_1 &\rightarrow S_1 c | A, \\ A &\rightarrow aAb | \lambda \end{aligned}$$

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

$$\begin{aligned} S_2 &\rightarrow aS_2 | B, \\ B &\rightarrow bBc | \lambda. \end{aligned}$$

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 not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous 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.

$$\begin{aligned} S &\rightarrow AB|aaB, \\ A &\rightarrow a|Aa, \\ B &\rightarrow b. \end{aligned}$$

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 $abbbbbb$ with the grammar in Example 5.5. In general, how many rounds will be needed to parse any string w in this language?

18. Is the string $aabbababb$ in the language generated by the grammar $S \rightarrow aSS|b$?

19. Show that the grammar in Example 1.14 is unambiguous.

20. 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.

21. Find a grammar equivalent to that in Example 5.5 that 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 that occur in programming languages, but as we argue 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 →. 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 C-like programming languages are susceptible to definition by restricted forms of context-free grammars. For example, the *while* statement in C can be defined as

$$\langle while\_statement \rangle ::= while \langle expression \rangle \langle statement \rangle .$$

Here the keyword *while* 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 $\langle while\_statement \rangle$ on the left is always associated with the terminal *while* 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 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. These grammars 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 that 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 that are syntactically correct in this sense are in fact acceptable programs. For C, the usual BNF definition allows constructs such as

$$char \qquad a, b, c;$$

followed by

$$c = 3.2;$$

This combination is not acceptable to C compilers since it violates the constraint, "a character variable cannot be assigned a real value." Context-free grammars cannot express the fact that type clashes may not be permitted. Such rules are part of programming language semantics, since they have 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 has been as universally accepted and are as successful for semantic definition as context-free languages have been for syntax.

EXERCISES

1. Consult a book on C for formal defintions of the following constructs.

   (a) literal

   (b) for statement

   (c) if-else statement

   (d) do statement

   (e) compound statement

   (f) return statement

2. Find examples of features of C that cannot be described by context-free grammars.

Chapter 6

Simplification of Context-Free Grammars and Normal Forms

Before we can study context-free languages in greater depth, we must attend to some technical matters. The definition of a context-free grammar imposes no restriction whatsoever on the right side of a production. However, complete freedom is not necessary and, in fact, is a detriment in some arguments. In Theorem 5.2, we see the convenience of certain restrictions on grammatical forms; eliminating rules of the form $A \to \lambda$ and $A \to B$ make the arguments easier. In many instances, it is desirable to place even more stringent restrictions on the grammar. Because of this, we need to look at methods for transforming an arbitrary context-free grammar into an equivalent one that satisfies certain restrictions on its form. In this chapter we study several transformations and substitutions that will be useful in subsequent discussions.

We also investigate **normal forms** for context-free grammars. A normal form is one that, although restricted, is broad enough so that any grammar has an equivalent normal-form version. We introduce two of the most useful of these, the **Chomsky normal form** and the **Greibach normal form**. Both have many practical and theoretical uses. An immediate application of the Chomsky normal form to parsing is given in Section 6.3.

The somewhat tedious nature of the material in this chapter lies in the fact that many of the arguments are manipulative and give little intuitive insight. For our purposes, this technical aspect is relatively unimportant and can be read casually. The various conclusions are significant; they will be used many times in later discussions.

6.1 Methods for Transforming Grammars

We first raise an issue that is somewhat of a nuisance with grammars and languages in general: the presence of the empty string. The empty string plays a rather singular role in many theorems and proofs, and it is often necessary to give it special attention. We prefer to remove it from consideration altogether, looking only at languages that do not contain λ. In doing so, we do not lose generality, as we see from the following considerations. Let L be any context-free language, and let $G = (V, T, S, P)$ be a context-free grammar for $L - \{\lambda\}$. Then the grammar we obtain by adding to V the new variable S_0, making S_0 the start variable, and adding to P the productions

$$S_0 \to S|\lambda$$

generates L. Therefore, any nontrivial conclusion we can make for $L - \{\lambda\}$ will almost certainly transfer to L. Also, given any context-free grammar G, there is a method for obtaining $\hat{G}$ such that $L(\hat{G}) = L(G) - \{\lambda\}$ (see Exercises 13 and 14 at the end of this section). Consequently, for all practical purposes, there is no difference between context-free languages that include λ and those that do not. For the rest of this chapter, unless otherwise stated, we will restrict our discussion to λ-free languages.

A Useful Substitution Rule

Many rules govern generating equivalent grammars by means of substitutions. Here we give one that is very useful for simplifying grammars in various ways. We will not define the term *simplification* precisely, but we will use it nevertheless. What we mean by it is the removal of certain types of undesirable productions; the process does not necessarily result in an actual reduction of the number of rules.

Theorem 6.1

Let $G = (V, T, S, P)$ be a context-free grammar. Suppose that P contains a production of the form

$$A \rightarrow x_1 B x_2.$$

Assume that A and B are different variables and that

$$B \rightarrow y_1 \mid y_2 \mid \ldots \mid y_n$$

is the set of all productions in P that have B as the left side. Let $\hat{G} = (V, T, S, \hat{P})$ be the grammar in which $\hat{P}$ is constructed by deleting

$$A \rightarrow x_1 B x_2 \tag{6.1}$$

from P, and adding to it

$$A \rightarrow x_1 y_1 x_2 \mid x_1 y_2 x_2 \mid \cdots \mid x_1 y_n x_2.$$

Then

$$L(\hat{G}) = L(G).$$

Proof: Suppose that $w \in L(G)$, so that

$$S \overset{*}{\Rightarrow}_G w.$$

The subscript on the derivation sign ⇒ is used here to distinguish between derivations with different

grammars. If this derivation does not involve the production (6.1), then obviously

$$S \overset{*}{\Rightarrow}_{\widehat{G}} w.$$

If it does, then look at the derivation the first time (6.1) is used. The B so introduced eventually has to be replaced; we lose nothing by assuming that this is done immediately (see Exercise 18 at the end of this section). Thus

$$S \overset{*}{\Rightarrow}_{G} u_1 A u_2 \Rightarrow_{G} u_1 x_1 B x_2 u_2 \Rightarrow_{G} u_1 x_1 y_j x_2 u_2.$$

But with grammar $\widehat{G}$ we can get

$$S \overset{*}{\Rightarrow}_{\widehat{G}} u_1 A u_2 \Rightarrow_{\widehat{G}} u_1 x_1 y_j x_2 u_2.$$

Thus we can reach the same sentential form with G and $\widehat{G}$. If (6.1) is used again later, we can repeat the argument. It follows then, by induction on the number of times the production is applied, that

$$S \overset{*}{\Rightarrow}_{\widehat{G}} w.$$

Therefore, if $w \in L(G)$, then $w \in L\left(\widehat{G}\right)$.

.

By similar reasoning, we can show that if $w \in L(\widehat{G})$ then $w \in L(G)$, completing the proof. ■

Theorem 6.1 is a simple and quite intuitive substitution rule: A production $A \rightarrow x_1 B x_2$ can be eliminated from a grammar if we put in its place the set of productions in which B is replaced by all strings it derives in one step. In this result, it is necessary that A and B be different variables. The case when $A = B$ is partially addressed in Exercises 23 and 24 at the end of this section.

Example 6.1

Consider $G = (\{A, B\}, \{a,b,c\}, A, P)$ with productions

$$A \rightarrow a \,|aaA|\, abBc,$$
$$B \rightarrow abbA|b.$$

Using the suggested substitution for the variable B, we get the grammar $\widehat{G}$ with productions

$$A \rightarrow a \,|aaA|\, ababbAc|abbc,$$
$$B \rightarrow abbA|b.$$

The new grammar $\widehat{G}$ is equivalent to G. The string *aaabbc* has the derivation

$$A \Rightarrow aaA \Rightarrow aaabBc \Rightarrow aaabbc$$

in G, and the corresponding derivation

$$A \Rightarrow aaA \Rightarrow aaabbc$$

in $\hat{G}$.

Notice that, in this case, the variable *B* and its associated productions are still in the grammar even though they can no longer play a part in any derivation. We will next show how such unnecessary productions can be removed from a grammar.

Removing Useless Productions

One invariably wants to remove productions from a grammar that can never take part in any derivation. For example, in the grammar whose entire production set is

$$S \rightarrow aSb \,|\lambda|\, A,$$
$$A \rightarrow aA,$$

the production $S \rightarrow A$ clearly plays no role, as *A* cannot be transformed into a terminal string. While *A* can occur in a string derived from *S*, this can never lead to a sentence. Removing this production leaves the language unaffected and is a simplification by any definition.

Definition 6.1

Let $G = (V, T, S, P)$ be a context-free grammar. A variable $A \in V$ is said to be **useful** if and only if there is at least one $\omega \in L(G)$ such that

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w, \tag{6.2}$$

with *x*, *y* in $(V \cup T)^*$. In words, a variable is useful if and only if it occurs in at least one derivation. A variable that is not useful is called **useless**. A production is useless if it involves any useless variable.

Example 6.2

A variable may be useless because there is no way of getting a terminal string from it. The case just mentioned is of this kind. Another reason a variable may be useless is shown in the next grammar. In a grammar with start symbol *S* and productions

$$S \to A,$$
$$A \to aA|\lambda,$$
$$B \to bA,$$

the variable B is useless and so is the production $B \to bA$. Although B can derive a terminal string, there is no way we can achieve $S \overset{*}{\Rightarrow} xBy$.

This example illustrates the two reasons why a variable is useless: either because it cannot be reached from the start symbol or because it cannot derive a terminal string. A procedure for removing useless variables and productions is based on recognizing these two situations. Before we present the general case and the corresponding theorem, let us look at another example.

Example 6.3

Eliminate useless symbols and productions from $G = (V,T,S,P)$, where $V = \{S, A, B, C\}$ and $T = \{a, b\}$, with P consisting of

$$S \to aS\,|A|\,C,$$
$$A \to a,$$
$$B \to aa,$$
$$C \to aCb.$$

First, we identify the set of variables that can lead to a terminal string. Because $A \to a$ and $B \to aa$, the variables A and B belong to this set. So does S, because $S \Rightarrow A \Rightarrow a$. However, this argument cannot be made for C, thus identifying it as useless. Removing C and its corresponding productions, we are led to the grammar G_1 with variables $V_1 = \{S, A, B\}$, terminals $T = \{a\}$, and productions

$$S \to aS|A,$$
$$A \to a,$$
$$B \to aa.$$

Next we want to eliminate the variables that cannot be reached from the start variable. For this, we can draw a **dependency graph** for the variables. Dependency graphs are a way of visualizing complex relationships and are found in many applications. For context-free grammars, a dependency graph has its vertices labeled with variables, with an edge between vertices C and D if and only if there is a production of the form

$$C \to xD_y.$$

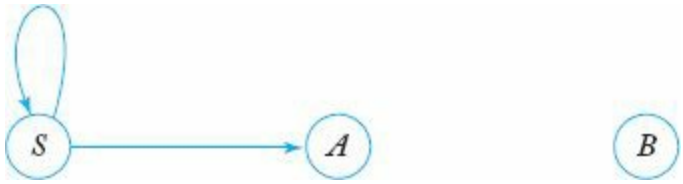
A dependency graph for V_1 is shown in Figure 6.1. A variable is useful only if there is a path from the vertex labeled S to the vertex labeled with that variable. In our case, Figure 6.1 shows that B is useless. Removing it and the affected productions and terminals, we are led to the final answer

$\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ with $\widehat{V} = \{S, A\}, \widehat{T} = \{a\}$, and productions

$$S \to aS|A,$$
$$A \to a.$$

The formalization of this process leads to a general construction and the corresponding theorem.

Figure 6.1



Theorem 6.2

Let $G = (V, T, S, P)$ be a context-free grammar. Then there exists an equivalent grammar $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ that does not contain any useless variables or productions.

Proof: The grammar $\widehat{G}$ can be generated from G by an algorithm consisting of two parts. In the first part we construct an intermediate grammar $G_1 = (V_1, T_2, S, P_1)$ such that V_1 contains only variables A for which

$$A \stackrel{*}{\Rightarrow} w \in T^*$$

is possible. The steps in the algorithm are

1. Set V_1 to ø.

2. Repeat the following step until no more variables are added to V_1. For every $A \in V$ for which P has a production of the form

$$A \to x_1 x_2 \cdots x_n, \text{ with all } x_i \text{ in } V_1 \cup T,$$

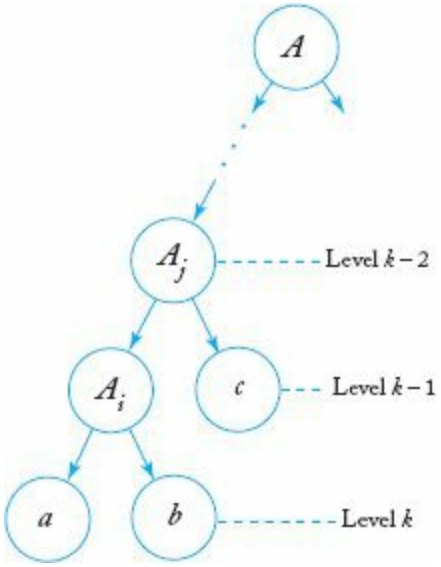
add A to V_1.

3. Take P_1 as all the productions in P whose symbols are all in $(V_1 \cup T)$.

Clearly this procedure terminates. It is equally clear that if $A \in V_1$, then $A \stackrel{*}{\Rightarrow} w \in T^*$ is a possible derivation with G_1. The remaining issue is whether every A for which $A \stackrel{*}{\Rightarrow} w = ab\cdots$ is added to V_1 before the procedure terminates. To see this, consider any such A and look at the partial derivation tree corresponding to that derivation (Figure 6.2). At level k, there are only terminals, so

every variable A_i at level $k-1$ will be added to V_1 on the first pass through Step 2 of the algorithm. Any variable at level $k-2$ will then be added to V_1 on the second pass through Step 2. The third time through Step 2, all variables at level $k-3$ will be added, and so on. The algorithm cannot terminate while there are variables in the tree that are not yet in V_1. Hence A will eventually be added to V_1.

Figure 6.2



In the second part of the construction, we get the final answer $\hat{G}$ from G_1. We draw the variable dependency graph for G_1 and from it find all variables that can not be reached from S. These are removed from the variable set, as are the productions involving them. We can also eliminate any terminal that does not occur in some useful production. The result is the grammar $\hat{G} = \left(\hat{V}, \hat{T}, S, \hat{P}\right)$.

Because of the construction, $\hat{G}$ does not contain any useless symbols or productions. Also, for each $\omega \in L(G)$ we have a derivation

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w.$$

Since the construction of $\hat{G}$ retains A and all associated productions, we have everything needed to make the derivation

$$S \overset{*}{\Rightarrow}_{\hat{G}} xAy \overset{*}{\Rightarrow}_{\hat{G}} w.$$

The grammar $\hat{G}$ is constructed from G by the removal of productions, so that $\hat{P} \subseteq P$. Consequently $L\left(\hat{G}\right) \subseteq L(G)$. Putting the two results together, we see that G and $\hat{G}$ are equivalent. ■

Removing λ-Productions

One kind of production that is sometimes undesirable is one in which the right side is the empty string.

Definition 6.2

Any production of a context-free grammar of the form

$$A \rightarrow \lambda$$

is called a λ-**production**. Any variable A for which the derivation

$$A \overset{*}{\Rightarrow} \lambda \tag{6.3}$$

is possible is called **nullable**.

A grammar may generate a language not containing λ, yet have some λ-productions or nullable variables. In such cases, the λ-productions can be removed.

Example 6.4

Consider the grammar

$$\begin{aligned} S &\rightarrow aS_1b, \\ S_1 &\rightarrow aS_1b|\lambda, \end{aligned}$$

with start variable S. This grammar generates the λ-free language $\{a^nb^n : n \geq 1\}$. The λ-production $S_1 \rightarrow \lambda$ can be removed after adding new productions obtained by substituting λ for S_1 where it occurs on the right. Doing this we get the grammar

$$\begin{aligned} S &\rightarrow aS_1b|ab, \\ S_1 &\rightarrow aS_1b|ab. \end{aligned}$$

We can easily show that this new grammar generates the same language as the original one.

In more general situations, substitutions for λ-productions can be made in a similar, although more complicated, manner.

Theorem 6.3

Let G be any context-free grammar with λ not in $L(G)$. Then there exists an equivalent grammar $\widehat{G}$ having no λ-productions.

Proof: We first find the set V_N of all nullable variables of G, using the following steps.

1. For all productions $A \to \lambda$, put A into V_N.

2. Repeat the following step until no further variables are added to V_N.

For all productions

$$B \to A_1 A_2 \ldots A_n,$$

where $A_1, A_2, \ldots, A_n$ are in V_N, put B into V_N.

Once the set V_N has been found, we are ready to construct $\widehat{P}$. To do so, we look at all productions in P of the form

$$A \to x_1 x_2 \cdots x_m, m \geq 1,$$

where each $x_i \in V \cup T$. For each such production of P, we put into $\widehat{P}$ that production as well as all those generated by replacing nullable variables with λ in all possible combinations. For example, if x_i and x_j are both nullable, there will be one production in $\widehat{P}$ with x_i replaced with λ, one in which x_j is replaced with λ, and one in which both x_i and x_j are replaced with λ. There is one exception: If all x_i are nullable, the production $A \to \lambda$ is not put into $\widehat{P}$.

The argument that this grammar $\widehat{G}$ is equivalent to G is straightforward and will be left to the reader. ■

Example 6.5

Find a context-free grammar without λ-productions equivalent to the grammar defined by

$$\begin{aligned} S &\to ABaC, \\ A &\to BC, \\ B &\to b|\lambda, \\ C &\to D|\lambda, \\ D &\to d. \end{aligned}$$

From the first step of the construction in Theorem 6.3, we find that the nullable variables are A, B, C. Then, following the second step of the construction, we get

$$\begin{aligned} S &\to ABaC\,|BaC|\,AaC\,|ABa|\,aC\,|Aa|\,Ba|a, \\ A &\to B\,|C|\,BC, \\ B &\to b, \\ C &\to D, \\ D &\to d. \end{aligned}$$

Removing Unit-Productions

As we have seen in Theorem 5.2, productions in which both sides are a single variable are at times undesirable.

Definition 6.3

Any production of a context-free grammar of the form

$$A \to B,$$

where $A, B \in V$, is called a **unit-production.**

To remove unit-productions, we use the substitution rule discussed in Theorem 6.1. As the construction in the next theorem shows, this can be done if we proceed with some care.

Theorem 6.4

Let $G = (V, T, S, P)$ be any context-free grammar without λ-productions. Then there exists a context-free grammar $\hat{G} = \left(\hat{V}, \hat{T}, S, \hat{P}\right)$ that does not have any unit-productions and that is equivalent to G.

Proof: Obviously, any unit-production of the form $A \to A$ can be removed from the grammar without effect, and we need only consider $A \to B$, where A and B are different variables. At first sight, it may seem that we can use Theorem 6.1 directly with $x_1 = x_2 = \lambda$ to replace

$$A \to B$$

with

$$A \to y_1 \mid y_2 \mid \ldots \mid y_n.$$

But this will not always work; in the special case

$$A \to B,$$
$$B \to A,$$

the unit-productions are not removed. To get around this, we first find, for each A, all variables B such that

$$A \overset{*}{\Rightarrow} B. \tag{6.4}$$

We can do this by drawing a dependency graph with an edge (C, D) when-ever the grammar has a unit-production $C \to D$; then (6.4) holds whenever there is a walk between A and B. The new grammar $\widehat{G}$ is generated by first putting into $\widehat{P}$ all non-unit productions of P. Next, for all A and B satisfying (6.4), we add to $\widehat{P}$

$$A \to y_1 \,|y_2| \ldots |y_n,$$

where $B \to y_1 \,|y_2| \ldots |y_n$ is the set of all rules in $\widehat{P}$ with B on the left. Note that since $B \to y_1 \,|y_2| \ldots |y_n$ is taken from $\widehat{P}$, none of the y_i can be a single variable, so that no unit-productions are created by the last step.

To show that the resulting grammar is equivalent to the original one, we can follow the same line of reasoning as in Theorem 6.1. ■

Example 6.6

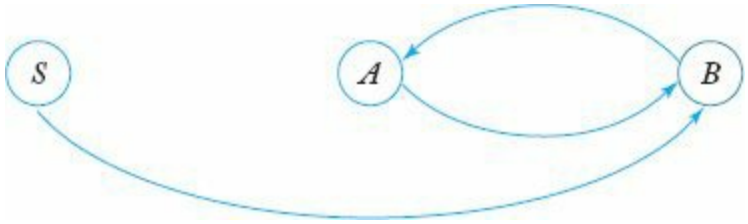
Remove all unit-productions from

$$\begin{aligned} S &\to Aa|B, \\ B &\to A|bb, \\ A &\to a\,|bc|\,B. \end{aligned}$$

The dependency graph for the unit-productions is given in Figure 6.3; we see from it that $S \overset{*}{\Rightarrow} A, S \overset{*}{\Rightarrow} B, B \overset{*}{\Rightarrow} A$, and $A \overset{*}{\Rightarrow} B$. Hence, we add to the original non-unit productions

$$\begin{aligned} S &\to Aa, \\ A &\to a|bc, \\ B &\to bb, \end{aligned}$$

Figure 6.3



the new rules

$$S \rightarrow a\,|bc|\,bb,$$
$$A \rightarrow bb,$$
$$B \rightarrow a|bc,$$

to obtain the equivalent grammar

$$S \rightarrow a\,|bc|\,bb|Aa,$$
$$A \rightarrow a\,|bb|\,bc,$$
$$B \rightarrow a\,|bb|\,bc.$$

Note that the removal of the unit-productions has made B and the associated productions useless.

We can put all these results together to show that grammars for context-free languages can be made free of useless productions, λ-productions, and unit-productions.

Theorem 6.5

Let L be a context-free language that does not contain λ. Then there exists a context-free grammar that generates L and that does not have any useless productions, λ-productions, or unit-productions.

Proof: The procedures given in Theorems 6.2, 6.3, and 6.4 remove these kinds of productions in turn. The only point that needs consideration is that the removal of one type of production may introduce productions of another type; for example, the procedure for removing λ-productions can create new unit-productions. Also, Theorem 6.4 requires that the grammar have no λ-productions. But note that the removal of unit-productions does not create λ-productions (Exercise 16 at the end of this section), and the removal of useless productions does not create λ-productions or unit-productions (Exercise 17 at the end of this section). Therefore, we can remove all undesirable productions using the following sequence of steps:

1. Remove λ-productions.
2. Remove unit-productions.
3. Remove useless productions.

The result will then have none of these productions, and the theorem is proved. ■

EXERCISES

1. Complete the proof of Theorem 6.1 by showing that

$$S \overset{*}{\Rightarrow}_{\hat{G}} w$$

implies

$$S \stackrel{*}{\Rightarrow}_G w.$$

2. In Example 6.1, show a derivation tree for the string *ababbac*, using both the original and the modified grammar.

3. Show that the two grammars

$$\begin{aligned} S &\to abAB|ba, \\ A &\to aaa, \\ B &\to aA|bb \end{aligned}$$

and

$$\begin{aligned} S &\to abAaA\,|abAbb|\,ba, \\ A &\to aaa \end{aligned}$$

are equivalent.

4. In Theorem 6.1, why is it necessary to assume that A and B are different variables?

5. Eliminate all useless productions from the grammar

$$\begin{aligned} S &\to aS|AB, \\ A &\to bA, \\ B &\to AA. \end{aligned}$$

What language does this grammar generate?

6. Eliminate useless productions from

$$\begin{aligned} S &\to a\,|aA|\,B|C, \\ A &\to aB|\lambda, \\ B &\to Aa, \\ C &\to cCD, \\ D &\to ddd. \end{aligned}$$

7. Eliminate all λ-productions from

$$\begin{aligned} S &\to AaB|aaB, \\ A &\to \lambda, \\ B &\to bbA|\lambda. \end{aligned}$$

8. Remove all unit-productions, all useless productions, and all λ-productions from the grammar

$$S \to aA|aBB,$$
$$A \to aaA|\lambda,$$
$$B \to bB|bbC,$$
$$C \to B.$$

What language does this grammar generate?

9. Eliminate all unit-productions from the grammar in Exercise 6.

10. Complete the proof of Theorem 6.3.

11. Complete the proof of Theorem 6.4.

12. Use the construction in Theorem 6.3 to remove λ-productions from the grammar in Example 5.4. What language does the resulting grammar generate?

13. Consider the grammar G with productions

$$S \to A|B,$$
$$A \to \lambda,$$
$$B \to aBb,$$
$$B \to b.$$

Construct a grammar $\widehat{G}$ by applying the algorithm in Theorem 6.3 to G. What is the difference between $L(G)$ and $L(\widehat{G})$?

14. Suppose that G is a context-free grammar for which $\lambda \in L(G)$. Show that if we apply the construction in Theorem 6.3, we obtain a new grammar $\widehat{G}$ such that $L(\widehat{G}) = L(G) - \{\lambda\}$.

15. Give an example of a situation in which the removal of λ-productions introduces previously nonexistent unit-productions.

16. Let G be a grammar without λ-productions, but possibly with some unit-productions. Show that the construction of Theorem 6.4 does not then introduce any λ-productions.

17. Show that if a grammar has no λ-productions and no unit-productions, then the removal of useless productions by the construction of Theorem 6.2 does not introduce any such productions.

18. Justify the claim made in the proof of Theorem 6.1 that the variable B can be replaced as soon as it appears.

19. Suppose that a context-free grammar $G = (V, T, S, P)$ has a production of the form

$$A \to xy,$$

where $x, y \in (V \cup T)^+$. Prove that if this rule is replaced by

$$A \to By,$$
$$B \to x,$$

where $B \notin V$, then the resulting grammar is equivalent to the original one.

20. Consider the procedure suggested in Theorem 6.2 for the removal of useless productions. Reverse the order of the two parts, first eliminating variables that cannot be reached from S, then removing those that do not yield a terminal string. Does the new procedure still work correctly? If so, prove it. If not, give a counterexample.

21. It is possible to define the term *simplification* precisely by introducing the concept of **complexity** of a grammar. This can be done in many ways; one of them is through the length of all the strings giving the production rules. For example, we might use

$$complexity\,(G) = \sum_{A \to v \in P} \{1 + |v|\}.$$

Show that the removal of useless productions always reduces the complexity in this sense. What can you say about the removal of λ-productions and unit-productions?

22. A context-free grammar G is said to be minimal for a given language L if $complexity\ (G) \leq complexity\ (\widehat{G})$ for any $\widehat{G}$ generating L. Show by example that the removal of useless productions does not necessarily produce a minimal grammar.

***23.** Prove the following result. Let $G = (V, T, S, P)$ be a context-free grammar. Divide the set of productions whose left sides are some given variable (say, A), into two disjoint subsets

$$A \to Ax_1 \,|Ax_2| \cdots |Ax_n,$$

$$A \to y_1 \,|y_2| \cdots |y_m,$$

where x_i, y_i are in $(V \cup T)^*$, but A is not a prefix of any y_i. Consider the grammar $\widehat{G} = \left(V \cup \{Z\}, T, S, \widehat{P}\right)$, where $Z \notin V$ and $\widehat{P}$ is obtained by replacing all productions that have A on the left by

$$\begin{aligned} A &\to y_i | y_i Z, \quad i = 1, 2, \ldots, m, \\ Z &\to x_i | x_i Z, \quad i = 1, 2, \ldots, n. \end{aligned}$$

Then $L\,(G) = L\,(\widehat{G})$.

24. Use the result of the preceding exercise to rewrite the grammar

$$\begin{aligned} A &\to Aa \,|aBc|\, \lambda, \\ B &\to Bb|bc \end{aligned}$$

so that it no longer has productions of the form $A \to Ax$ or $B \to Bx$.

*25. Prove the following counterpart of Exercise 23. Let the set of productions involving the variable A on the left be divided into two disjoint subsets

$$A \to x_1 A \,|x_2 A| \cdots |x_n A$$

and

$$A \to y_1 \,|y_2| \cdots |y_m,$$

where A is not a suffix of any y_i. Show that the grammar obtained by replacing these productions with

$$\begin{aligned} A &\to y_i | Z y_i, \quad i = 1, 2, \ldots, m, \\ Z &\to x_i | Z x_i, \quad i = 1, 2, \ldots, n, \end{aligned}$$

is equivalent to the original grammar.

6.2 Two Important Normal Forms

There are many kinds of normal forms we can establish for context-free grammars. Some of these, because of their wide usefulness, have been studied extensively. We consider two of them briefly.

Chomsky Normal Form

One kind of normal form we can look for is one in which the number of symbols on the right of a production is strictly limited. In particular, we can ask that the string on the right of a production consist of no more than two symbols. One instance of this is the **Chomsky normal form**.

Definition 6.4

A context-free grammar is in Chomsky normal form if all productions are of the form

$$A \to BC$$

or

$$A \to a,$$

where A, B, C are in V, and a is in T.

Example 6.7

The grammar

$$S \rightarrow AS|a,$$
$$A \rightarrow SA|b$$

is in Chomsky normal form. The grammar

$$S \rightarrow AS|AAS,$$
$$A \rightarrow SA|aa$$

is not; both productions $S \rightarrow AAS$ and $A \rightarrow aa$ violate the conditions of Definition 6.4.

Theorem 6.6

Any context-free grammar $G = (V, T, S, P)$ with $\lambda \notin L(G)$ has an equivalent grammar $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ in Chomsky normal form.

Proof: Because of Theorem 6.5, we can assume without loss of generality that G has no λ-productions and no unit-productions. The construction of $\widehat{G}$ will be done in two steps.

Step 1: Construct a grammar $G_1 = (V_1, T, S, P_1)$ from G by considering all productions in P in the form

$$A \rightarrow x_1 x_2 \cdots x_n, \tag{6.5}$$

where each x_i is a symbol either in V or in T. If $n = 1$, then x_1 must be a terminal since we have no unit-productions. In this case, put the production into P_1. If $n \geq 2$, introduce new variables Ba for each $a \in T$. For each production of P in the form (6.5) we put into P_1 the production

$$A \rightarrow C_1 C_2 \ldots C_n,$$

where

$$C_i = x_i \text{ if } x_i \text{ is in } V,$$

and

$$C_i = B_a \text{ if } x_i = a.$$

For every B_a we also put into P_1 the production

$$B_a \rightarrow a.$$

This part of the algorithm removes all terminals from productions whose right side has length greater than one, replacing them with newly introduced variables. At the end of this step we have a grammar G_1 all of whose productions have the form

$$A \to a, \tag{6.6}$$

or

$$A \to C_1 C_2 \cdots C_n, \tag{6.7}$$

where $C_i \in V_1$.

It is an easy consequence of Theorem 6.1 that

$$L(G_1) = L(G).$$

Step 2: In the second step, we introduce additional variables to reduce the length of the right sides of the productions where necessary. First we put all productions of the form (6.6) as well as all the productions of the form (6.7) with $n = 2$ into $\hat{P}$. For $n \geq 2$, we introduce new variables D_1, D_2,…and put into $\hat{P}$ the productions

$$\begin{aligned} A &\to C_1 D_1, \\ D_1 &\to C_2 D_2, \\ &\vdots \\ D_{n-2} &\to C_{n-1} C_n. \end{aligned}$$

Obviously, the resulting grammar $\hat{G}$ is in Chomsky normal form. Repeated applications of Theorem 6.1 will show that $L(G_1) = L(\hat{G})$, so that

$$L(\hat{G}) = L(G).$$

This somewhat informal argument can easily be made more precise. We will leave this to the reader.■

Example 6.8

Convert the grammar with productions

$$
\begin{aligned}
S &\to ABa,\\
A &\to aab,\\
B &\to Ac
\end{aligned}
$$

to Chomsky normal form.

As required by the construction of Theorem 6.6, the grammar does not have any λ-productions or any unit-productions.

In Step 1, we introduce new variables B_a, B_b, B_c and use the algorithm to get

$$
\begin{aligned}
S &\to ABB_a,\\
A &\to B_aB_aB_b,\\
B &\to AB_c,\\
B_a &\to a,\\
B_b &\to b,\\
B_c &\to c.
\end{aligned}
$$

In the second step, we introduce additional variables to get the first two productions into normal form and we get the final result

$$
\begin{aligned}
S &\to AD_1,\\
D_1 &\to BB_a,\\
A &\to B_aD_2,\\
D_2 &\to B_aB_b,\\
B &\to AB_c,\\
B_a &\to a,\\
B_b &\to b,\\
B_c &\to c.
\end{aligned}
$$

Greibach Normal Form

Another useful grammatical form is the **Greibach normal form.** Here we put restrictions not on the length of the right sides of a production, but on the positions in which terminals and variables can appear. Arguments justifying Greibach normal form are a little complicated and not very transparent. Similarly, constructing a grammar in Greibach normal form equivalent to a given context-free grammar is tedious. We therefore deal with this matter very briefly. Nevertheless, Greibach normal form has many theoretical and practical consequences.

Definition 6.5

A context-free grammar is said to be in Greibach normal form if all productions have the form

$$A \to ax,$$

where $a \in T$ and $x \in V^*$

If we compare this with Definition 5.4, we see that the form $A \to ax$ is common to both Greibach normal form and s-grammars, but Greibach normal form does not carry the restriction that the pair *(A, a)* occur at most once. This additional freedom gives Greibach normal form a generality not possessed by s-grammars.

If a grammar is not in Greibach normal form, we may be able to rewrite it in this form with some of the techniques encountered above. Here are two simple examples.

Example 6.9

The grammar

$$\begin{aligned} S &\to AB, \\ A &\to aA \,|bB|\, b, \\ B &\to b \end{aligned}$$

is not in Greibach normal form. However, using the substitution given by Theorem 6.1, we immediately get the equivalent grammar

$$\begin{aligned} S &\to aAB \,|bBB|\, bB, \\ A &\to aA \,|bB|\, b, \\ B &\to b, \end{aligned}$$

which is in Greibach normal form.

Example 6.10

Convert the grammar

$$S \to abSb|aa$$

into Greibach normal form.

Here we can use a device similar to the one introduced in the construction of Chomsky normal form. We introduce new variables A and B that are essentially synonyms for a and b, respectively. Substituting for the terminals with their associated variables leads to the equivalent grammar

$$\begin{aligned} S &\to aBSB|aA, \\ A &\to a, \\ B &\to b, \end{aligned}$$

which is in Greibach normal form.

In general, though, neither the conversion of a given grammar to Greibach normal form nor the proof that this can always be done is a simple matter. We introduce Greibach normal form here because it will simplify the technical discussion of an important result in the next chapter. However, from a conceptual viewpoint, Greibach normal form plays no further role in our discussion, so we only quote the following general result without proof.

Theorem 6.7

For every context-free grammar G with $\lambda \notin L(G)$, there exists an equivalent grammar $\hat{G}$ in Greibach normal form.

EXERCISES

1. Provide the details of the proof of Theorem 6.6.
2. Convert the grammar $S \to aSb|ab$ into Chomsky normal form.
3. Transform the grammar $S \to aSaA|A, A \to abA|b$ into Chomsky normal form.
4. Transform the grammar with productions

$$S \rightarrow abAB,$$
$$A \rightarrow bAB|\lambda,$$
$$B \rightarrow BAa\,|A|\,\lambda$$

into Chomsky normal form.

5. Convert the grammar

$$S \rightarrow AB|aB,$$
$$A \rightarrow aab|\lambda,$$
$$B \rightarrow bbA$$

into Chomsky normal form.

6. Let $G = (V, T, S, P)$ be any context-free grammar without any λ-productions or unit-productions. Let k be the maximum number of symbols on the right of any production in P. Show that there is an equivalent grammar in Chomsky normal form with no more than $(k-1)\,|P| + |T|$ production rules.

7. Draw the dependency graph for the grammar in Exercise 4.

8. A linear language is one for which there exists a linear grammar (for a definition, see Example 3.14). Let L be any linear language not containing λ. Show that there exists a grammar $G = (V, T, S, P)$ all of whose productions have one of the forms

$$A \rightarrow aB,$$
$$A \rightarrow Ba,$$
$$A \rightarrow a,$$

where $a \in T$, $A, B \in V$, such that $L = L(G)$.

9. Show that for every context-free grammar $G = (V, T, S, P)$ there is an equivalent one in which all productions have the form

$$A \rightarrow aBC,$$

or

$$A \rightarrow \lambda,$$

where $a \in \Sigma \cup \{\lambda\}, A, B, C \in V$.

10. Convert the grammar

$$S \to aSb|bSa|a|b$$

into Greibach normal form.

11. Convert the following grammar into Greibach normal form.

$$S \to aSb|ab.$$

12. Convert the grammar

$$S \to ab\,|aS|\,aaS$$

into Greibach normal form.

13. Convert the grammar

$$\begin{aligned} S &\to ABb|a, \\ A &\to aaA|B, \\ B &\to bAb \end{aligned}$$

into Greibach normal form.

14. Can every linear grammar be converted to a form in which all productions look like $A \to ax$, where $a \in T$ and $x \in V \cup \{\lambda\}$?

15. A context-free grammar is said to be in two-standard form if all production rules satisfy the following pattern

$$\begin{aligned} A &\to aBC, \\ A &\to aB, \\ A &\to a, \end{aligned}$$

where $A, B, C \in V$ and $a \in T$.

Convert the grammar $G = (\{S, A, B, C\}, \{a, b\}, S, P)$ with P given as

$$\begin{aligned} S &\to aSA, \\ A &\to bABC, \\ B &\to b, \\ C &\to aBC \end{aligned}$$

into two-standard form.

*16. Two-standard form is general; for any context-free grammar G with $\lambda \in L(G)$, there exists an equivalent grammar in two-standard form. Prove this.

6.3 A Membership Algorithm for Context-Free Grammars*

In Section 5.2, we claim, without any elaboration, that membership and parsing algorithms for context-free grammars exist that require approximately $|w|^3$ steps to parse a string w. We are now in a position to justify this claim. The algorithm we will describe here is called the CYK algorithm, after its originators J. Cocke, D. H. Younger, and T. Kasami. The algorithm works only if the grammar is in Chomsky normal form and succeeds by breaking one problem into a sequence of smaller ones in the following way. Assume that we have a grammar $G = (V, T, S, P)$ in Chomsky normal form and a string

$$w = a_1 a_2 \ldots a_n.$$

We define substrings

$$w_{ij} = a_i \ldots a_j.$$

and subsets of V

$$V_{ij} = \left\{ A \in V : A \overset{*}{\Rightarrow} w_{ij} \right\}$$

Clearly, $w \in L(G)$ if and only if $S \in V_{1n}$.

To compute V_{ij}, observe that $A \in V_{ii}$ if and only if G contains a production $A \rightarrow a_i$. Therefore, V_{ii} can be computed for all $1 \leq i \leq n$ by inspection of w and the productions of the grammar. To continue, notice that for $j > i$, A derives w_{ij} if and only if there is a production $A \rightarrow BC$, with $B \overset{*}{\Rightarrow} w_{ik}$ and $C \overset{*}{\Rightarrow} w_{k+1j}$ for some k with $i \leq k$, $k < j$. In other words,

$$V_{ij} = \bigcup_{k \in \{i, i+1, \ldots, j-1\}} \{A : A \rightarrow BC, \text{ with } B \in V_{ik}, C \in V_{k+1,j}\}. \quad (6.8)$$

An inspection of the indices in (6.8) shows that it can be used to compute all the V_{ij} if we proceed in the sequence

1. Compute $V_{11}, V_{22}, \ldots, V_{nn}$;
2. Compute $V_{12}, V_{23}, \ldots, V_{n-i,n}$,
3. Compute $V_{13}, V_{24}, V_{n-2,n}$,

and so on.

Example 6.11

Determine whether the string $w = aabbb$ is in the language generated by the grammar

$$\begin{aligned} S &\to AB, \\ A &\to BB|a, \\ B &\to AB|b. \end{aligned}$$

First note that $w_{11} = a$, so V_{11} is the set of all variables that immediately derive a, that is, $V_{11} = \{A\}$. Since $w_{22} = a$, we also have $V_{22} = \{A\}$ and, similarly,

$$V_{11} = \{A\}, V_{22} = \{A\}, V_{33} = \{B\}, V_{44} = \{B\}, V_{55} = \{B\}.$$

Now we use (6.8) to get

$$V_{12} = \{A : A \to BC, B \in V_{11}, C \in V_{22}\}.$$

Since $V_{11} = \{A\}$ and $V_{22} = \{A\}$, the set consists of all variables that occur on the left side of a production whose right side is AA. Since there are none, $V12$ is empty. Next,

$$V_{23} = \{A : A \to BC, B \in V_{22}, C \in V_{33}\},$$

so the required right side is AB, and we have $V_{23} = \{S, B\}$. A straightforward argument along these lines then gives

$$\begin{aligned} V_{12} &= \varnothing, V_{23} = \{S, B\}, V_{34} = \{A\}, V_{45} = \{A\}, \\ V_{13} &= \{S, B\}, V_{24} = \{A\}, V_{35} = \{S, B\}, \\ V_{14} &= \{A\}, V_{25} = \{S, B\}, \\ V_{15} &= \{S, B\}, \end{aligned}$$

so that $w \in L(G)$.

The CYK algorithm, as described here, determines membership for any language generated by a grammar in Chomsky normal form. With some additions to keep track of how the elements of V_{ij} are derived, itcan be converted into a parsing method. To see that the CYK membership algorithm requires O ($n3$) steps, notice that exactly n (n +1) /2 sets of V_{ij} have to be computed. Each involves the evaluation of at most n terms in (6.8), so the claimed result follows.

EXERCISES

1. Use the CYK algorithm to determine whether the strings *aabb*, *aabba*, and *abbbb* are in the language generated by the grammar in Example 6.11.
2. Use the CYK algorithm to find a parsing of the string *aab*, using the grammar of Example 6.11.
3. Use the approach employed in Exercise 2 to show how the CYK membership algorithm can be made into a parsing method.
4. Use the CYK method to determine if the string $w = aaabbbbab$ is in the language generated by the grammar $S \rightarrow aSb|b$.