3 Context-Free Languages

3.1 CONTEXT-FREE GRAMMARS

Think of yourself as a language processor. You can recognize a legal English sentence when you hear one; "the cat is in the hat" is at least syntactically correct (whether or not it says anything that happens to be the truth), but "hat the the in is cat" is gibberish. However you manage to do it, you can immediately tell when reading such sentences whether they are formed according to generally accepted rules for sentence structure. In this respect you are acting as a **language recognizer**: a device that accepts valid strings. The finite automata of the last chapter are formalized types of language recognizers.

You also, however, are capable of producing legal English sentences. Again, why you would want to do so and how you manage to do it are not our concern; but the fact is that you occasionally speak or write sentences, and in general they are syntactically correct (even when they are lies). In this respect you are acting as a **language generator**. In this section we shall study certain types of formal language generators. Such a device begins, when given some sort of "start" signal, to construct a string. Its operation is not completely determined from the beginning but is nevertheless limited by a set of rules. Eventually this process halts, and the device outputs a completed string. The language defined by the device is the set of all strings that it can produce.

Neither a recognizer nor a generator for the English language is at all easy to produce; indeed, designing such devices for large subsets of natural languages has been a challenging research front for several decades. Nevertheless the idea of a language generator has some explanatory force in attempts to discuss human language. More important for us, however, is the theory of generators of formal, "artificial" languages, such as the regular languages and the important class of "context-free" languages introduced below. This theory will neatly complement the study of automata, which recognize languages, and is also of practical value in the specification and analysis of computer languages.

Regular expressions can be viewed as language generators. For example, consider the regular expression $a(a^* \cup b^*)b$. A verbal description of how to generate a string in accordance with this expression would be the following

> First output an a. Then do one of the following two things:
> Either output a number of a's or output a number of b's.
> Finally output a b.

The language associated with this language generator —that is, the set of all strings that can be produced by the process just described —is, of course, exactly the regular language defined in the way described earlier by the regular expression $a(a^* \cup b^*)b$.

In this chapter we shall study certain more complex sorts of language generators, called **context-free grammars**, which are based on a more complete understanding of the structure of the strings belonging to the language. To take again the example of the language generated by $a(a^* \cup b^*)b$, note that any string in this language consists of a leading a, followed by a *middle part* —generated by $(a^* \cup b^*)$— followed by a trailing b. If we let S be a new symbol interpreted as "a string in the language," and M be a symbol standing for "middle part," then we can express this observation by writing

$$S \rightarrow aMb,$$

where $\rightarrow$ is read "can be." We call such an expression a **rule**. What can M, the middle part, be? The answer is: either a string of a's or a string of b's. We express this by adding the rules

$$M \rightarrow A \text{ and } M \rightarrow B,$$

where A and B are new symbols that stand for strings of a's and b's, respectively. Now, what is a string of a's? It can be the empty string

$$A \rightarrow e,$$

or it may consist of a leading a followed by a string of a's:

$$A \rightarrow aA.$$

Similarly, for B:

$$B \rightarrow e \text{ and } B \rightarrow bB.$$

The language denoted by the regular expression $a(a^* \cup b^*)b$ can then be defined alternatively by the following language generator.

Start with the string consisting of the single symbol S. Find a symbol in the current string that appears to the left of $\rightarrow$ in one of the rules above. Replace an occurrence of this symbol with the string that appears to the right of $\rightarrow$ in the same rule. Repeat this process until no such symbol can be found.

For example, to generate the string $aaab$ we start with S, as specified; we then replace S by aMb according to the first rule, $S \rightarrow aMb$. To aMb we apply the rule $M \rightarrow A$ and obtain aAb. We then twice apply the rule $A \rightarrow aA$ to get the string $aaaAb$. Finally, we apply the rule $A \rightarrow e$. In the resulting string, $aaab$, we cannot identify any symbol that appears to the left of $\rightarrow$ in some rule. Thus the operation of our language generator has ended, and $aaab$ was produced, as promised.

A **context-free grammar** is a language generator that operates like the one above, with some such set of rules. Let us pause to explain at this point why such a language generator is called context-free. Consider the string $aaAb$, which was an intermediate stage in the generation of $aaab$. It is natural to call the strings aa and b that surround the symbol A the **context** of A in this particular string. Now, the rule $A \rightarrow aA$ says that we can replace A by the string aA no matter what the surrounding strings are; in other words, *independently of the context of* A. In Chapter 4 we examine more general grammars, in which replacements may be conditioned on the existence of an appropriate context.

In a context-free grammar, some symbols appear to the left of $\rightarrow$ in rules —S, M, A, and B in our example— and some —a and b— do not. Symbols of the latter kind are called **terminals**, since the production of a string consisting solely of such symbols signals the termination of the generation process. All these ideas are stated formally in the next definition.

Definition 3.1.1: A **context-free grammar** G is a quadruple (V, Σ, R, S), where

V is an alphabet,
Σ (the set of **terminals**) is a subset of V,
R (the set of **rules**) is a finite subset of $(V - \Sigma) \times V^*$, and
S (the **start symbol**) is an element of $V - \Sigma$.

The members of $V - \Sigma$ are called **nonterminals**. For any $A \in V - \Sigma$ and $u \in V^*$, we write $A \rightarrow_G u$ whenever $(A, u) \in R$. For any strings $u, v \in V^*$, we write $u \Rightarrow_G v$ if and only if there are strings $x, y \in V^*$ and $A \in V - \Sigma$ such that $u = xAy$, $v = xv'y$, and $A \rightarrow_G v'$. The relation $\Rightarrow_G^*$ is the reflexive, transitive closure of $\Rightarrow_G$. Finally, $L(G)$, the **language generated** by G, is $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$; we also say that G **generates** each string in $L(G)$. A language L is said to be a **context-free language** if $L = L(G)$ for some context-free grammar G.

When the grammar to which we refer is obvious, we write $A \rightarrow w$ and $u \Rightarrow v$ instead of $A \rightarrow_G w$ and $u \Rightarrow_G v$.

We call any sequence of the form

$$w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$$

a **derivation** in G of w_n from w_0. Here $w_0, \cdots, w_n$ may be any strings in V^*, and n, the **length** of the derivation, may be any natural number, including zero. We also say that the derivation has n **steps**.

Example 3.1.1: Consider the context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S, a, b\}$, $\Sigma = \{a, b\}$, and R consists of the rules $S \rightarrow aSb$ and $S \rightarrow e$. A possible derivation is

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

Here the first two steps used the rule $S \rightarrow aSb$, and the last used the rule $S \rightarrow e$. In fact, it is not hard to see that $L(G) = \{a^n b^n : n \geq 0\}$. Hence some context-free languages are not regular.◇

We shall soon see, however, that all regular languages are context-free.

Example 3.1.2: Let G be the grammar $(W, \Sigma, R, S,)$, where

$$\begin{aligned}
W &= \{S, A, N, V, P\} \cup \Sigma,\\
\Sigma &= \{\text{Jim, big, green, cheese, ate}\},\\
R &= \{P \rightarrow N,\\
&\quad P \rightarrow AP,\\
&\quad S \rightarrow PVP,\\
&\quad A \rightarrow \text{big},\\
&\quad A \rightarrow \text{green},\\
&\quad N \rightarrow \text{cheese},\\
&\quad N \rightarrow \text{Jim},\\
&\quad V \rightarrow \text{ate}\}
\end{aligned}$$

Here G is designed to be a grammar for a part of English; S stands for *sentence*, A for *adjective*, N for *noun*, V for *verb*, and P for *phrase*. The following are some strings in $L(G)$.

Jim ate cheese
big Jim ate green cheese
big cheese ate Jim

Unfortunately, the following are also strings in $L(G)$:

big cheese ate green green big green big cheese
green Jim ate green big Jim

Example 3.1.3: Computer programs written in any programming language must satisfy some rigid criteria in order to be syntactically correct and therefore amenable to mechanical interpretation. Fortunately, the syntax of most programming languages can, unlike that of human languages, be captured by context-free grammars. We shall see in Section 3.7 that being context-free is extremely helpful when it comes to *parsing* a program, that is, analyzing it to understand its syntax. Here, we give a grammar that generates a fragment of many common programming languages. This language consists of all strings over the alphabet $\{(,),+,*,\mathsf{id}\}$ that represent syntactically correct arithmetic expressions involving $+$ and $*$. id stands for any *identifier*, that is to say, variable name.† Examples of such strings are id and $\mathsf{id}*(\mathsf{id}*\mathsf{id}+\mathsf{id})$, but not $*\mathsf{id}+($ or $+*\mathsf{id}$.

Let $G=(V,\Sigma,R,E)$ where V, Σ, and R are as follows.

$$\begin{aligned}
V &= \{+,*,(,),\mathsf{id},T,F,E\},\\
\Sigma &= \{+,*,(,),\mathsf{id}\},\\
R &= \{E \to E+T, && (R1)\\
&\quad E \to T, && (R2)\\
&\quad T \to T*F, && (R3)\\
&\quad T \to F, && (R4)\\
&\quad F \to (E), && (R5)\\
&\quad F \to \mathsf{id}\}. && (R6)
\end{aligned}$$

The symbols E, T, and F are abbreviations for *expression*, *term*, and *factor*, respectively.

The grammar G generates the string $(\mathsf{id}*\mathsf{id}+\mathsf{id})*(\mathsf{id}+\mathsf{id})$ by the following derivation.

$$\begin{aligned}
E &\Rightarrow T && \text{by Rule R2}\\
&\Rightarrow T*F && \text{by Rule R3}\\
&\Rightarrow T*(E) && \text{by Rule R5}\\
&\Rightarrow T*(E+T) && \text{by Rule R1}
\end{aligned}$$

† Incidentally, discovering such identifiers (or reserved words of the language, or numerical constants) in the program is accomplished at the earlier stage of *lexical analysis*, by algorithms based on regular expressions and finite automata.

$\Rightarrow T * (T + T)$	by Rule R2
$\Rightarrow T * (F + T)$	by Rule R4
$\Rightarrow T * (\mathsf{id} + T)$	by Rule R6
$\Rightarrow T * (\mathsf{id} + F)$	by Rule R4
$\Rightarrow T * (\mathsf{id} + \mathsf{id})$	by Rule R6
$\Rightarrow F * (\mathsf{id} + \mathsf{id})$	by Rule R4
$\Rightarrow (E) * (\mathsf{id} + \mathsf{id})$	by Rule R5
$\Rightarrow (E + T) * (\mathsf{id} + \mathsf{id})$	by Rule R1
$\Rightarrow (E + F) * (\mathsf{id} + \mathsf{id})$	by Rule R4
$\Rightarrow (E + \mathsf{id}) * (\mathsf{id} + \mathsf{id})$	by Rule R6
$\Rightarrow (T + \mathsf{id}) * (\mathsf{id} + \mathsf{id})$	by Rule R2
$\Rightarrow (T * F + \mathsf{id}) * (\mathsf{id} + \mathsf{id})$	by Rule R3
$\Rightarrow (F * F + \mathsf{id}) * (\mathsf{id} + \mathsf{id})$	by Rule R4
$\Rightarrow (F * \mathsf{id} + \mathsf{id}) * (\mathsf{id} + \mathsf{id})$	by Rule R6
$\Rightarrow (\mathsf{id} * \mathsf{id} + \mathsf{id}) * (\mathsf{id} + \mathsf{id})$	by Rule R6

See Problem 3.1.8 for context-free grammars that generate larger subsets of programming languages.◊

Example 3.1.4: The following grammar generates all strings of properly balanced left and right parentheses: every left parenthesis can be paired with a unique subsequent right parenthesis, and every right parenthesis can be paired with a unique preceding left parenthesis. Moreover, the string between any such pair has the same property. We let $G = (V, \Sigma, R, S)$, where

$$\begin{aligned}
V &= \{S, (,)\},\\
\Sigma &= \{(,)\},\\
R &= \{S \to e,\\
&\qquad S \to SS,\\
&\qquad S \to (S)\}.
\end{aligned}$$

Two derivations in this grammar are

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow ()(())$$

and

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()(())$$

Thus the same string may have several derivations in a context-free grammar; in the next subsection we discuss the intricate ways in which such derivations may be related.

Incidentally, $L(G)$ is another context-free language that is not regular (that it is not regular was the object of Problem 2.4.6).◊

Example 3.1.5: Obviously, there are context-free languages that are not regular (we have already seen two examples). However, *all regular languages are context-free.* In the course of this chapter we shall encounter several proofs of this fact. For example, we shall see in Section 3.3 that context-free languages are precisely the languages accepted by certain language acceptors called ***pushdown automata.*** Now we shall also point out that the pushdown acceptor is a generalization of the finite automaton, in the sense that any finite automaton can be trivially considered as a pushdown automaton. Hence all regular languages are context-free.

For another proof, we shall see in Section 3.5 that the class of context-free languages is closed under union, concatenation, and Kleene star (Theorem 3.5.1); furthermore, the trivial languages $\emptyset$ and $\{a\}$ are definitely context-free (generated by the context-free grammars with no rules, or with only the rule $S \to a$, respectively). Hence the class of context-free languages must contain all regular languages, the closure of the trivial languages under these operations.

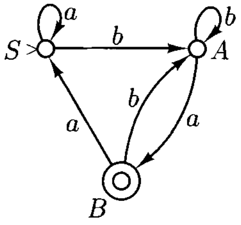


Figure 3-1

But let us now show that all regular languages are context-free *by a direct construction.* Consider the regular language accepted by the deterministic finite automaton $M = (K, \Sigma, \delta, s, F)$. The same language is generated by the grammar $G(M) = (V, \Sigma, R, S)$, where $V = K \cup \Sigma$, $S = s$, and R consists of these rules:

$$R = \{q \to ap : \delta(q, a) = p\} \cup \{q \to e : q \in F\}.$$

That is, the nonterminals are the states of the automaton; as for rules, for each transition from q to p on input a we have in R the rule $q \to ap$. For example, for the automaton in Figure 3-1 we would construct this grammar:

$$S \to aS, S \to bA, A \to aB, A \to bA, B \to aS, B \to bA, B \to e.$$

It is left as an exercise to show that the resulting context-free grammar generates precisely the language accepted by the automaton (see Problem 3.1.10 for a general treatment of context-free grammars such as $G(M)$ above, and their relationship with finite automata).◇

Problems for Section 3.1

3.1.1. Consider the grammar $G = (V, \Sigma, R, S)$, where

$$\begin{aligned} V &= \{a, b, S, A\}, \\ \Sigma &= \{a, b\}, \\ R &= \{S \to AA, \\ &\quad A \to AAA,. \\ &\quad A \to a, \\ &\quad A \to bA, \\ &\quad A \to Ab\}. \end{aligned}$$

(a) Which strings of $L(G)$ can be produced by derivations of four or fewer steps?
(b) Give at least four distinct derivations for the string *babbab*.
(c) For any $m, n, p > 0$, describe a derivation in G of the string $b^m a b^n a b^p$.

3.1.2. Consider the grammar (V, Σ, R, S), where V, Σ, and R are defined as follows:

$$\begin{aligned} V &= \{a, b, S, A\}, \\ \Sigma &= \{a, b\}, \\ R &= \{S \to aAa, \\ &\quad S \to bAb, \\ &\quad S \to e, \\ &\quad A \to SS\}. \end{aligned}$$

Give a derivation of the string *baabbb* in G. (Notice that, unlike all other context-free languages we have seen so far, this one is very difficult to describe in English.)

3.1.3. Construct context-free grammars that generate each of these languages.
(a) $\{wcw^R : w \in \{a, b\}^*\}$
(b) $\{ww^R : w \in \{a, b\}^*\}$
(c) $\{w \in \{a, b\}^* : w = w^R\}$

3.1.4. Consider the alphabet $\Sigma = \{a, b, (,), \cup, ^*, \emptyset\}$. Construct a context-free grammar that generates all strings in Σ^* that are regular expressions over $\{a, b\}$.

3.1.5. Consider the context-free grammar $G = (V, \Sigma, R, S)$, where

$$\begin{aligned}
V &= \{a, b, S, A, B\},\\
\Sigma &= \{a, b\},\\
R &= \{S \to aB,\\
&\quad S \to bA,\\
&\quad A \to a,\\
&\quad A \to aS,\\
&\quad A \to BAA,\\
&\quad B \to b,\\
&\quad B \to bS,\\
&\quad B \to ABB\}.
\end{aligned}$$

(a) Show that $ababba \in L(G)$.
(b) Prove that $L(G)$ is the set of all nonempty strings in $\{a, b\}$ that have equal numbers of occurrences of a and b.

3.1.6. Let G be a context-free grammar and let $k > 0$. We let $L_k(G) \subseteq L(G)$ be the set of all strings that have a derivation in G with k or fewer steps.
(a) What is $L_5(G)$, where G is the grammar of Example 3.1.4
(b) Show that, for all context-free grammars G and all $k > 0$, $L_k(G)$ is finite.

3.1.7. Let $G = (V, \Sigma, R, S)$, where $V = \{a, b, S\}$, $\Sigma = \{a, b\}$, and $R = \{S \to aSb, S \to aSa, S \to bSa, S \to bSb, S \to e\}$. Show that $L(G)$ is regular.

3.1.8. A program in a real programming language, such as C or Pascal, consists of statements, where each statement is one of several types:

(1) assignment statement, of the form id $:= E$, where E is any arithmetic expression (generated by the grammar of Example 3.1.3).
(2) conditional statement, of the form, say, if $E < E$ then statement, or a while statement of the form while $E < E$ do statement.
(3) goto statement; furthermore, each statement could be preceded by a label.
(4) compound statement, that is, many statements preceded by a begin, followed by an end, and separated by a ";".

Give a context-free grammar that generates all possible statements in the simplified programming language described above.

3.1.9. Show that the following languages are context-free by exhibiting context-free grammars generating each.

(a) $\{a^m b^n : m \geq n\}$

(b) $\{a^m b^n c^p d^q : m + n = p + q\}$

(c) $\{w \in \{a, b\}^* :$ w has twice as many b's as a's$\}$

(d) $\{uawb : u, w \in \{a, b\}^*, |u| = |w|\}$

(e) $w_1 c w_2 c \ldots c w_k c c w_j^R : k \geq 1, 1 \leq j \leq k, w_i \in \{a, b\}^+$ for $i = 1, \ldots, k\}$

(f) $\{a^m b^n : m \leq 2n\}$

3.1.10. Call a context-free grammar $G = (V, \Sigma, R, S)$ **regular** (or **right-linear**) if $R \subseteq (V - \Sigma) \times \Sigma^*(V - \Sigma \cup \{e\})$; that is, if each transition has a right-hand side that consists of a string of terminals followed by at most one nonterminal.

(a) Consider the regular grammar $G = (V, \Sigma, R, S)$, where

$$\begin{aligned} V &= \{a, b, A, B, S\} \\ \Sigma &= \{a, b\} \\ R &= \{S \to abA, S \to B, S \to baB, S \to e, \\ &\qquad A \to bS, B \to aS, A \to b\}. \end{aligned}$$

Construct a nondeterministic finite automaton M such that $L(M) = L(G)$. Trace the transitions of M that lead to the acceptance of the string $abba$, and compare with a derivation of the same string in G.

(b) Prove that a language is regular if and only if there is a regular grammar that generates it. (*Hint:* Recall Example 3.1.5.)

(c) Call a context-free grammar $G = (V, \Sigma, R, S)$ **left-linear** if and only if $R \subseteq (V - \Sigma) \times (V - \Sigma) \cup \{e\})\Sigma^*$. Show that a language is regular if and only if it is the language generated by some left-linear grammar.

(d) Suppose that $G = (V, \Sigma, R, S)$ is a context-free grammar such that each rule in R is *either* of the form $A \to wB$ or of the form $A \to Bw$ or of the form $A \to w$, where in each case $A, B \in V - \Sigma$ and $w \in \Sigma^*$. Is $L(G)$ necessarily regular? Prove it or give a counter-example.

3.2 PARSE TREES

Let G be a context-free grammar. A string $w \in L(G)$ may have many derivations in G. For example, if G is the context-free grammar that generates the language of balanced parentheses (recall Example 3.1.4), then the string ()() can be derived from S by at least two distinct derivations, namely,

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

and

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ()()$$

However, these two derivations are in a sense "the same." The rules used are the same, and they are applied at the same places in the intermediate string. The only difference is in the *order* in which the rules are applied. Intuitively, both derivations can be pictured as in Figure 3-2.

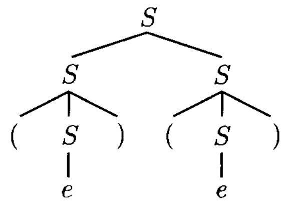


Figure 3-2

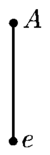
We call such a picture a **parse tree**. The points are called **nodes**; each node carries a **label** that is a symbol in V. The topmost node is called the **root**, and the nodes along the bottom are called **leaves**. All leaves are labeled by *terminals*, or possibly the empty string e. By concatenating the labels of the leaves from left to right, we obtain the derived string of terminals, which is called the **yield** of the parse tree.

More formally, for an arbitrary context-free grammar $G = (V, \Sigma, R, S)$, we define its parse trees and their roots, leaves, and yields, as follows.

1. $\circ\ a$

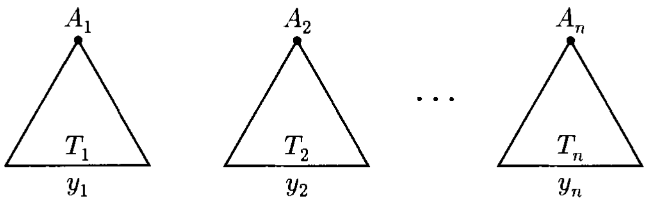
This is a parse tree for each $a \in \Sigma$. The single node of this parse tree is both the root and a leaf. The yield of this parse tree is a.

2. If $A \to e$ is a rule in R, then

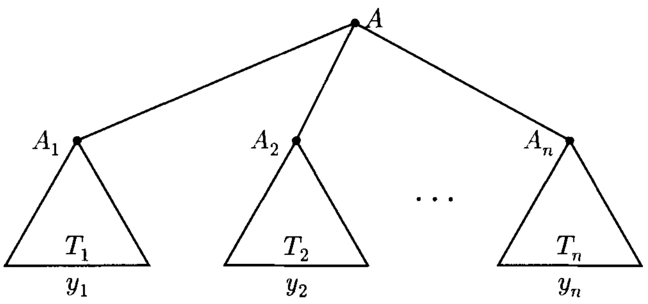


is a parse tree; its root is the node labeled A, its sole leaf is the node labeled e, and its yield is e.

3. If



Are parse trees, where $n \geq 1$, with roots labeled $A_1, \ldots, A_n$ respectively, and with yields $y_1, \ldots, y_n$, and $A \to A_1 \ldots A_n$ is a rule in R, then



is a parse tree. Its root is the new node labeled A, its leaves are the leaves of its constituent parse trees, and its yield is $y_1 \ldots y_n$.

4. Nothing else is a parse tree.

Example 3.2.1: Recall the grammar G that generates all arithmetic expressions over id (Example 3.1.3). A parse tree with yield id $*$ (id + id) is shown in Figure 3-3.◇

Intuitively, parse trees are ways of representing derivations of strings in $L(G)$ so that the superficial differences between derivations, owing to the order of application of rules, are suppressed. To put it otherwise, parse trees represent *equivalence classes of derivations.* We make this intuition precise below.

Let $G = (V, \Sigma, R, S)$ be a context-free grammar, and let $D = x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n$ and $D' = x'_1 \Rightarrow x'_2 \Rightarrow \cdots \Rightarrow x'_n$ be two derivations in G, where $x_i, x'_i \in V^*$ for $i = 1, \ldots, n$, $x_1, x'_1 \in V - \Sigma$, and $x_n, x'_n \in \Sigma^*$. That is, they are both derivations of terminal strings from a single nonterminal. We say that D **precedes** D', written $D \prec D'$, if $n > 2$ and there is an integer k, $1 < k < n$ such that

(1) for all $i \neq k$ we have $x_i = x'_i$;
(2) $x_{k-1} = x'_{k-1} = uAvBw$, where $u, v, w \in V^*$, and $A, B, \in V - \Sigma$;

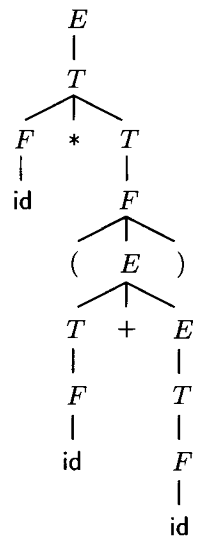


Figure 3-3

(3) $x_k = uyvBw$, where $A \rightarrow y \in R$;
(4) $x'_k = uAvzw$ where $B \rightarrow z \in R$;
(5) $x_{k+1} = x'_{k+1} = uyvzw$.

In other words, the two derivations are identical except for two consecutive steps, during which the same two nonterminals are replaced by the same two strings *but in opposite orders in the two derivations.* The derivation in which the leftmost of the two nonterminals is replaced first is said to precede the other.

Example 3.2.2: Consider the following three derivations D_1, D_2, and D_3 in the grammar G generating all strings of balanced parentheses:

$$D_1 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$$
$$D_2 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$$
$$D_3 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$$

We have that $D_1 \prec D_2$ and $D_2 \prec D_3$. However, it is *not* the case that $D_1 \prec D_3$, since the two latter derivations differ in more than one intermediate string. Notice that all three derivations have the same parse tree, the one shown in Figure 3-4.◊

We say that two derivations D and D' are **similar** if the pair (D, D') belongs in the *reflexive, symmetric, transitive closure* of $\prec$. Since the reflexive,

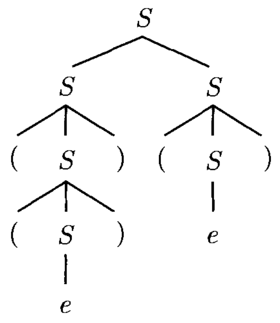


Figure 3-4

symmetric, transitive closure of any relation is by definition reflexive, symmetric, and transitive, similarity is an equivalence relation. To put it otherwise, two derivations are similar if they can be transformed into another via a sequence of "switchings" in the order in which rules are applied. Such a "switching" can replace a derivation either by one that precedes it, or by one that it precedes.

Example 3.2.2 (continued): Parse trees capture exactly, via a natural isomorphism, the equivalence classes of the "similarity" equivalence relation between derivations of a string defined above. The equivalence class of the derivations of $(())()$ corresponding to the tree in Figure 3-4 contains the derivations D_1, D_2, D_3 shown above, and also these seven:

$$\begin{aligned}
D_4 &= S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())() \\
D_5 &= S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())() \\
D_6 &= S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())() \\
D_7 &= S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())() \\
D_8 &= S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())() \\
D_9 &= S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())() \\
D_{10} &= S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()
\end{aligned}$$

These ten derivations are related by $\prec$ as shown in Figure 3-5.

$$\begin{tikzcd}
 & & D_3 \arrow[rd, phantom, "\prec" sloped] & & & & \\
D_1 \arrow[r, phantom, "\prec"] & D_2 \arrow[ru, phantom, "\prec" sloped] \arrow[rd, phantom, "\prec" sloped] & & D_5 \arrow[r, phantom, "\prec"] & D_6 \arrow[rd, phantom, "\prec" sloped] & & \\
 & & D_4 \arrow[ru, phantom, "\prec" sloped] \arrow[rd, phantom, "\prec" sloped] & & & D_9 \arrow[r, phantom, "\prec"] & D_{10} \\
 & & & D_7 \arrow[r, phantom, "\prec"] & D_8 \arrow[ru, phantom, "\prec" sloped] & &
\end{tikzcd}$$

Figure 3-5

All these ten derivations are similar, because, informally, they represent applications of the same rules at the same positions in the strings, only differing in the relative order of these applications; equivalently, one can go from any one of them to any other by repeatedly following either a $\prec$, or an inverted $\prec$. There are no other derivations similar to these.

There are, however, other derivations of $(())()$ that are not similar to the ones above —and thus are not captured by the parse tree shown in Figure 3-4. An example is the following derivation: $S \Rightarrow SS \Rightarrow SSS \Rightarrow S(S)S \Rightarrow S((S))S \Rightarrow S(())S \Rightarrow S(())(S) \Rightarrow S(())() \Rightarrow (())()$. Its parse tree is shown in Figure 3-6 (compare with Figure 3-4).◇

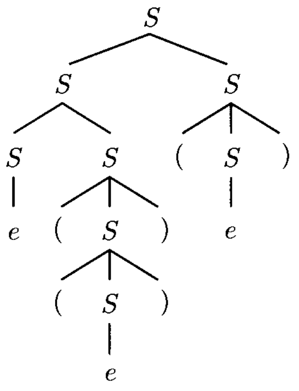


Figure 3-6

Each equivalence class of derivations under similarity, that is to say, each parse tree, contains a derivation that is *maximal* under $\prec$; that is, it is not preceded by any other derivation. This derivation is called a **leftmost derivation**. A leftmost derivation exists in every parse tree, and it can be obtained as follows. Starting from the label of the root A, repeatedly replace the *leftmost* nonterminal in the current string according to the rule suggested by the parse tree. Similarly, a **rightmost derivation** is one that does not precede any other derivation; it is obtained from the parse tree by always expanding the rightmost nonterminal in the current string. Each parse tree has exactly one leftmost and exactly one rightmost derivation. This is so because the leftmost derivation of a parse tree is uniquely determined, since at each step there is one nonterminal to replace: the leftmost one. Similarly for the rightmost derivation. In the example above, D_1 is a leftmost derivation, and D_{10} is a rightmost one.

It is easy to tell when a step of a derivation can be a part of a leftmost derivation: the leftmost nonterminal must be replaced. We write $x \overset{L}{\Rightarrow} y$ if and