CHAPTER

6 PROPERTIES OF CONTEXT-FREE LANGUAGES

To a large extent this chapter parallels Chapter 3. We shall first give a pumping lemma for context-free languages and use it to show that certain languages are not context free. We then consider closure properties of CFL's and finally we give algorithms to answer certain questions about CFL's.

6.1 THE PUMPING LEMMA FOR CFL's

The pumping lemma for regular sets states that every sufficiently long string in a regular set contains a short substring that can be pumped. That is, inserting as many copies of the substring as we like always yields a string in the regular set. The pumping lemma for CFL's states that there are always two short substrings close together that can be repeated, both the same number of times, as often as we like. The formal statement of the pumping lemma is as follows.

Lemma 6.1 (The pumping lemma for context-free languages). Let L be any CFL. Then there is a constant n, depending only on L, such that if z is in L and $|z| \geq n$, then we may write $z = uvwxy$ such that

1) $|vx| \geq 1$,
2) $|vwx| \leq n$, and
3) for all $i \geq 0$ uv^iwx^iy is in L.

Proof Let G be a Chomsky normal-form grammar generating $L - \{\epsilon\}$. Observe that if z is in $L(G)$ and z is long, then any parse tree for z must contain a long path. More precisely, we show by induction on i that if the parse tree of a word generated by a Chomsky normal-form grammar has no path of length greater than i, then the word is of length no greater than 2^{i-1}. The basis, $i = 1$, is trivial, since the tree must be of the form shown in Fig. 6.1(a). For the induction step, let $i > 1$. Let the root and its sons be as shown in Fig. 6.1(b). If there are no paths of length greater than $i - 1$ in trees T_1 and T_2, then the trees generate words of 2^{i-2} or fewer symbols. Thus the entire tree generates a word no longer than 2^{i-1}

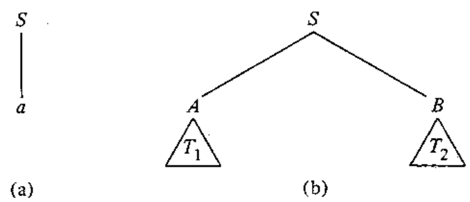


Fig. 6.1 Parse trees.

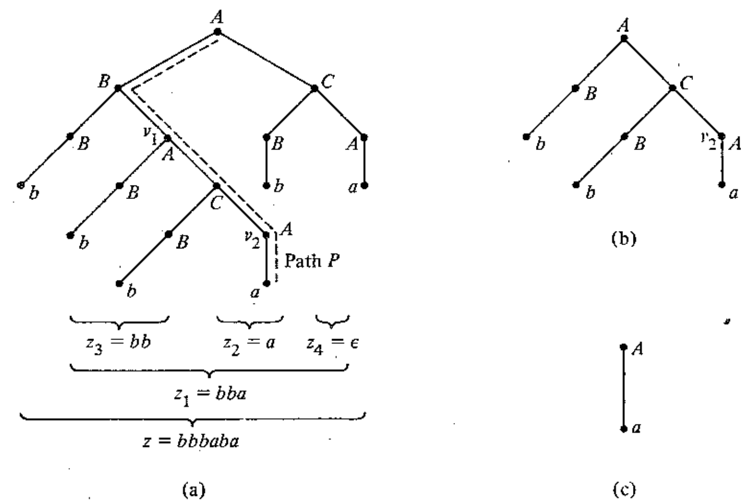
Let G have k variables and let $n = 2^k$. If z is in $L(G)$ and $|z| \geq n$, then since $|z| > 2^{k-1}$, any parse tree for z must have a path of length at least $k + 1$. But such a path has at least $k + 2$ vertices, all but the last of which are labeled by variables. Thus there must be some variable that appears twice on the path.

We can in fact say more. Some variable must appear twice near the bottom of the path. In particular, let P be a path that is as long or longer than any path in the tree. Then there must be two vertices v_1 and v_2 on the path satisfying the following conditions.

1) The vertices v_1 and v_2 both have the same label, say A.
2) Vertex v_1 is closer to the root than vertex v_2.
3) The portion of the path from v_1 to the leaf is of length at most $k + 1$.

To see that v_1 and v_2 can always be found, just proceed up path P from the leaf, keeping track of the labels encountered. Of the first $k + 2$ vertices, only the leaf has a terminal label. The remaining $k + 1$ vertices cannot have distinct variable labels.

Now the subtree T_1 with root v_1 represents the derivation of a subword of length at most 2^k. This is true because there can be no path in T_1 of length greater than $k + 1$, since P was a path of longest length in the entire tree. Let z_1 be the yield of the subtree T_1. If T_2 is the subtree generated by vertex v_2, and z_2 is the yield of the subtree T_2, then we can write z_1 as $z_3 z_2 z_4$. Furthermore, z_3 and z_4 cannot both be ϵ, since the first production used in the derivation of z_1 must be of the form $A \to BC$ for some variables B and C. The subtree T_2 must be completely within either the subtree generated by B or the subtree generated by C. The above is illustrated in Fig. 6.2.



$z_1 = z_3 z_2 z_4$, where $z_3 = bb$ and $z_4 = \epsilon$

$G = (\{A, B, C\}, \{a, b\}, \{A \to BC, B \to BA, C \to BA, A \to a, B \to b\}, A)$

Fig. 6.2 Illustration of subtrees T_1 and T_2 of Lemma 6.1. (a) Tree. (b) Subtree T_1. (c) Subtree T_2.

We now know that

$$A \overset{*}{\underset{G}{\Rightarrow}} z_3 A z_4 \quad \text{and} \quad A \overset{*}{\underset{G}{\Rightarrow}} z_2, \quad \text{where} \quad |z_3 z_2 z_4| \leq 2^k = n.$$

But it follows that $A \overset{*}{\underset{G}{\Rightarrow}} z_3^i z_2 z_4^i$ for each $i \geq 0$. (See Fig. 6.3.) The string z can clearly be written as $u z_3 z_2 z_4 y$, for some u and y. We let $z_3 = v$, $z_2 = w$, and $z_4 = x$, to complete the proof. $\square$

Applications of the pumping lemma

The pumping lemma can be used to prove a variety of languages not to be context free, using the same "adversary" argument as for the regular set pumping lemma.

Example 6.1 Consider the language $L_1 = \{a^i b^i c^i \mid i \geq 1\}$. Suppose L were context free and let n be the constant of Lemma 6.1. Consider $z = a^n b^n c^n$. Write $z = uvwxy$ so as to satisfy the conditions of the pumping lemma. We must ask ourselves where v and x, the strings that get pumped, could lie in $a^n b^n c^n$. Since $|vwx| \leq n$, it is not possible for vx to contain instances of a's and c's, because the rightmost a is $n + 1$ positions away from the leftmost c. If v and x consist of a's only, then uwy (the string $uv^i wx^i y$ with $i = 0$) has n b's and n c's but fewer than n a's, since $|vx| \geq 1$.

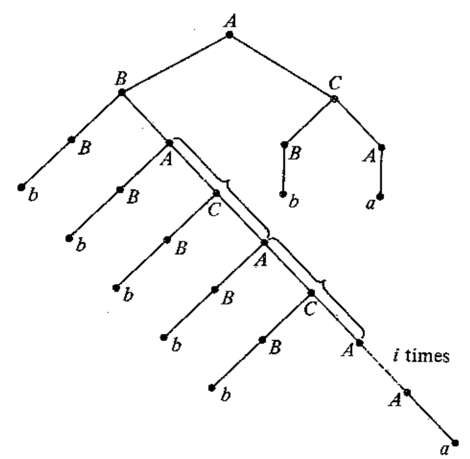


Fig. 6.3 The derivation of uv^iwx^iy, where $u = b$, $v = bb$, $w = a$, $x = \epsilon$, $y = ba$.

Thus, uwy is not of the form $a^jb^jc^j$. But by the pumping lemma vwy is in L_1, a contradiction.

The cases where v and x consist only of b's or only of c's are disposed of similarly. If vx has a's and b's, then uwy has more c's than a's or b's, and again it is not in L_1. If vx contains b's and c's, a similar contradiction results. We conclude that L_1 is not a context-free language.

The pumping lemma can also be used to show that certain languages similar to L_1 are not context free. Some examples are

$$\{a^ib^ic^j \mid j \geq i\} \quad \text{and} \quad \{a^ib^jc^k \mid i \leq j \leq k\}.$$

Another type of relationship that CFG's cannot enforce is illustrated in the next example.

Example 6.2 Let $L_2 = \{a^ib^jc^id^j \mid i \geq 1 \text{ and } j \geq 1\}$. Suppose L_2 is a CFL, and let n be the constant in Lemma 6.1. Consider the string $z = a^nb^nc^nd^n$. Let $z = uvwxy$ satisfy the conditions of the pumping lemma. Then as $|vwx| \leq n$, vx can contain at most two different symbols. Furthermore, if vx contains two different symbols, they must be consecutive, for example, a and b. If vx has only a's, then uwy has fewer a's than c's and is not in L_2, a contradiction. We proceed similarly if vx consists of only b's, only c's, or only d's. Now suppose vx has a's and b's. Then vwy still has fewer a's than c's. A similar contradiction occurs if vx consists of b's and c's or c's and d's. Since these are the only possibilities, we conclude that L_2 is not context free.

Ogden's lemma

There are certain non-CFL's for which the pumping lemma is of no help. For example,

$$L_3 = \{a^ib^jc^kd^\ell \mid \text{either } i = 0 \text{ or } j = k = \ell\}$$

is not context free. However, if we choose $z = b^jc^kd^\ell$, and write $z = uvwxy$, then it is always possible to choose u, v, w, x, and y so that uv^mwx^my is in L_3 for all m. For example, choose vwx to have only b's. If we choose $z = a^ib^jc^jd^j$, then v and x might consist only of a's, in which case uv^mwx^my is again in L_3 for all m.

What we need is a stronger version of the pumping lemma that allows us to focus on some small number of positions in the string and pump them. Such an extension is easy for regular sets, as any sequence of $n + 1$ states of an n-state FA must contain some state twice, and the intervening string can be pumped. The result for CFL's is much harder to obtain but can be shown. Here we state and prove a weak version of what is known as Ogden's lemma.

Lemma 6.2 (Ogden's lemma). Let L be a CFL. Then there is a constant n (which may in fact be the same as for the pumping lemma) such that if z is any word in L, and we mark any n or more positions of z "distinguished," then we can write $z = uvwxy$, such that:

1) v and x together have at least one distinguished position,
2) vwx has at most n distinguished positions, and
3) for all $i \geq 0$, uv^iwx^iy is in L.

Proof Let G be a Chomsky normal-form grammar generating $L - \{\epsilon\}$. Let G have k variables and choose $n = 2^k + 1$. We must construct a path P in the tree analogous to path P in the proof of the pumping lemma. However, since we worry only about distinguished positions here, we cannot concern ourselves with every vertex along P, but only with *branch points*, which are vertices both of whose sons have distinguished descendants.

Construct P as follows. Begin by putting the root on path P. Suppose r is the last vertex placed on P. If r is a leaf, we end. If r has only one son with distinguished descendants, add that son to P and repeat the process there. If both sons of r have distinguished descendants, call r a branch point and add the son with the larger number of distinguished descendants to P (break a tie arbitrarily). This process is illustrated in Fig. 6.4.

It follows that each branch point on P has at least half as many distinguished descendants as the previous branch point. Since there are at least n distinguished positions in z, and all of these are descendants of the root, it follows that there are at least $k + 1$ branch points on P. Thus among the last $k + 1$ branch points are two with the same label. We may select v_1 and v_2 to be two of these branch points with the same label and with v_1 closer to the root than v_2. The proof then proceeds exactly as for the pumping lemma. □

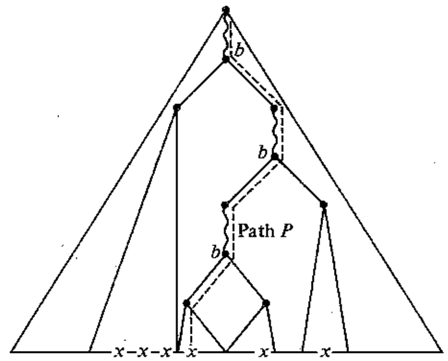


Fig. 6.4 The path P. Distinguished positions are marked x. Branch points are marked b

---

Example 6.3 Let $L_4 = \{a^ib^jc^k \mid i \neq j, j \neq k \text{ and } i \neq k\}$. Suppose L_4 were a context-free language. Let n be the constant in Ogden's lemma and consider the string $z = a^nb^{n+n!}c^{n+2n!}$. Let the positions of the a's be distinguished and let $z = uvwxy$ satisfy the conditions of Ogden's lemma. If either v or x contains two distinct symbols, then uv^2wx^2y is not in L_4. (For example, if v is in a^+b^+, then uv^2wx^2y has a b preceding an a.) Now at least one of v and x must contain a's since only a's are in distinguished positions. Thus, if x is in b^* or c^*, v must be in a^+. If x is in a^+, then v must be in a^*, otherwise a b or c would precede an a. We consider in detail the situation where x is in b^*. The other cases are handled similarly. Suppose x is in b^* and v in a^+. Let $p = |v|$. Then $1 \leq p \leq n$, so p divides $n!$ Let q be the integer such that $pq = n!$ Then

$$z' = uv^{2q+1}wx^{2q+1}y$$

is in L_4. But $v^{2q+1} = a^{2pq+p} = a^{2n!+p}$. Since uwy contains exactly $(n-p)$ a's, z' has $(2n! + n)$ a's. However, since v and x have no c's, z' also has $(2n! + n)$ c's and hence is not in L_4, a contradiction. A similar contradiction occurs if x is in a^+ or c^*. Thus L_4 is not a context-free language.

Note that Lemma 6.1 is a special case of Ogden's lemma in which all positions are distinguished.

6.2 CLOSURE PROPERTIES OF CFL's

We now consider some operations that preserve context-free languages. The operations are useful not only in constructing or proving that certain languages are context free, but also in proving certain languages not to be context free. A given language L can be shown not to be context free by constructing from L a language that is not context free using only operations preserving CFL's.

Theorem 6.1 Context-free languages are closed under union, concatenation and Kleene closure.

Proof Let L_1 and L_2 be CFL's generated by the CFG's

$$G_1 = (V_1, T_1, P_1, S_1) \quad \text{and} \quad G_2 = (V_2, T_2, P_2, S_2),$$

respectively. Since we may rename variables at will without changing the language generated, we assume that V_1 and V_2 are disjoint. Assume also that S_3, S_4, and S_5 are not in V_1 or V_2.

For $L_1 \cup L_2$ construct grammar $G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_3, S_3)$, where P_3 is $P_1 \cup P_2$ plus the productions $S_3 \rightarrow S_1 \mid S_2$. If w is in L_1, then the derivation $S_3 \underset{G_3}{\Rightarrow} S_1 \underset{G_1}{\overset{*}{\Rightarrow}} w$ is a derivation in G_3, as every production of G_1 is a production of G_3. Similarly, every word in L_2 has a derivation in G_3 beginning with $S_3 \Rightarrow S_2$. Thus $L_1 \cup L_2 \subseteq L(G_3)$. For the converse, let w be in $L(G_3)$. Then the derivation $S_3 \underset{G_3}{\overset{*}{\Rightarrow}} w$ begins with either $S_3 \underset{G_3}{\Rightarrow} S_1 \underset{G_3}{\overset{*}{\Rightarrow}} w$ or $S_3 \underset{G_3}{\Rightarrow} S_2 \underset{G_3}{\overset{*}{\Rightarrow}} w$. In the former case, as V_1 and V_2 are disjoint, only symbols of G_1 may appear in the derivation $S_1 \underset{G_3}{\overset{*}{\Rightarrow}} w$. As the only productions of P_3 that involve only symbols of G_1 are those from P_1, we conclude that only productions of P_1 are used in the derivation $S_1 \underset{G_3}{\overset{*}{\Rightarrow}} w$. Thus $S_1 \underset{G_1}{\overset{*}{\Rightarrow}} w$, and w is in L_1. Analogously, if the derivation starts $S_3 \underset{G_3}{\Rightarrow} S_2$, we may conclude w is in L_2. Hence $L(G_3) \subseteq L_1 \cup L_2$, so $L(G_3) = L_1 \cup L_2$, as desired.

For concatenation, let $G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_4, S_4)$, where P_4 is $P_1 \cup P_2$ plus the production $S_4 \rightarrow S_1S_2$. A proof that $L(G_4) = L(G_1)L(G_2)$ is similar to the proof for union and is omitted.

For closure, let $G_5 = (V_1 \cup \{S_5\}, T_1, P_5, S_5)$, where P_5 is P_1 plus the productions $S_5 \rightarrow S_1S_5 \mid \epsilon$. We again leave the proof that $L(G_5) = L(G_1)^*$ to the reader. □

Substitution and homomorphisms

Theorem 6.2 The context-free languages are closed under substitution.

Proof Let L be a CFL, $L \subseteq \Sigma^*$, and for each a in Σ let L_a be a CFL. Let L be $L(G)$ and for each a in Σ let L_a be $L(G_a)$. Without loss of generality assume that the variables of G and the G_a's are disjoint. Construct a grammar G' as follows. The variables of G' are all the variables of G and the G_a's; the terminals of G' are the terminals of the G_a's. The start symbol of G' is the start symbol of G. The productions of G' are all the productions of the G_a's together with those productions formed by taking a production $A \rightarrow \alpha$ of G and substituting S_a, the start symbol of G_a, for each instance of an a in Σ appearing in α. □

Example 6.4 Let L be the set of words with an equal number of a's and b's, $L_a = \{0^n1^n \mid n \geq 1\}$ and $L_b = \{ww^R \mid w \text{ is in } (\mathbf{0} + \mathbf{2})^*\}$. For G we may choose

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

For G_a take

$$S_a \to 0S_a1 \mid 01$$

For G_b take

$$S_b \to 0S_b0 \mid 2S_b2 \mid \epsilon$$

If f is the substitution $f(a) = L_a$ and $f(b) = L_b$, then $f(L)$ is generated by the grammar

$$S \to S_aSS_bS \mid S_bSS_aS \mid \epsilon$$
$$S_a \to 0S_a1 \mid 01$$
$$S_b \to 0S_b0 \mid 2S_b2 \mid \epsilon$$

One should observe that since $\{a, b\}$, $\{ab\}$, and $\mathbf{a}^*$ are CFL's, the closure of CFL's under substitution implies closure under union, concatenation, and $*$. The union of L_a and L_b is simply the substitution of L_a and L_b into $\{a, b\}$ and similarly L_aL_b and L_a^* are the substitutions into $\{ab\}$ and $\mathbf{a}^*$, respectively. Thus Theorem 6.1 could be presented as a corollary of Theorem 6.2.

Since a homomorphism is a special type of substitution we state the following corollary.

Corollary The CFL's are closed under homomorphism.

Theorem 6.3 The context-free languages are closed under inverse homomorphism.

Proof As with regular sets, a machine-based proof for closure under inverse homomorphism is easiest to understand. Let $h: \Sigma \to \Delta$ be a homomorphism and L be a CFL. Let $L = L(M)$, where M is the PDA $(Q, \Delta, \Gamma, \delta, q_0, Z_0, F)$. In analogy with the finite-automaton construction of Theorem 3.5, we construct PDA M' accepting $h^{-1}(L)$ as follows. On input a, M' generates the string $h(a)$ and simulates M on $h(a)$. If M' were a finite automaton, all it could do on a string $h(a)$ would be to change state; so M' could simulate such a composite move in one of its moves. However, in the PDA case, M could pop many symbols on a string, or, since it is nondeterministic, make moves that push an arbitrary number of symbols on the stack. Thus M' cannot necessarily simulate M's moves on $h(a)$ with one (or any finite number of) moves of its own.

What we do is give M' a buffer, in which it may store $h(a)$. Then M' may simulate any ϵ-moves of M it likes and consume the symbols of $h(a)$ one at a time, as if they were M's input. As the buffer is part of M''s finite control, it cannot be allowed to grow arbitrarily long. We ensure that it does not, by permitting M' to read an input symbol only when the buffer is empty. Thus the buffer holds a suffix of $h(a)$ for some a at all times. M' accepts its input w if the buffer is empty and M is in a final state. That is, M has accepted $h(w)$. Thus $L(M') = \{w \mid h(w) \text{ is in } L\}$, that is

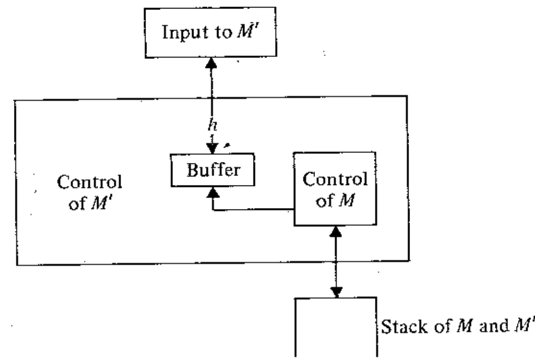


Fig. 6.5 Construction of a PDA accepting $h^{-1}(L)$.

$L(M') = h^{-1}(L(M))$. The arrangement is depicted in Fig. 6.5; the formal construction follows.

Let $M' = (Q', \Sigma, \Gamma, \delta', [q_0, \epsilon], Z_0, F \times \{\epsilon\})$, where Q' consists of pairs $[q, x]$ such that q is in Q and x is a (not necessarily proper) suffix of some $h(a)$ for a in Σ. δ' is defined as follows:

1) $\delta'([q, x], \epsilon, Y)$ contains all $([p, x], \gamma)$ such that $\delta(q, \epsilon, Y)$ contains (p, γ). Simulate ϵ-moves of M independent of the buffer contents.
2) $\delta'([q, ax], \epsilon, Y)$ contains all $([p, x], \gamma)$ such that $\delta(q, a, Y)$ contains (p, γ). Simulate moves of M on input a in Δ, removing a from the front of the buffer.
3) $\delta'([q, \epsilon], a, Y)$ contains $([q, h(a)], Y)$ for all a in Σ and Y in Γ. Load the buffer with $h(a)$, reading a from M''s input; the state and stack of M remain unchanged.

To show that $L(M') = h^{-1}(L(M))$ first observe that by one application of rule (3), followed by applications of rules (1) and (2), if $(q, h(a), \alpha) \overset{*}{\underset{M}{\vdash}} (p, \epsilon, \beta)$, then

$$([q, \epsilon], a, \alpha) \underset{M'}{\vdash} ([q, h(a)], \epsilon, \alpha) \overset{*}{\underset{M'}{\vdash}} ([p, \epsilon], \epsilon, \beta).$$

Thus if M accepts $h(w)$, that is,

$$(q_0, h(w), Z_0) \overset{*}{\underset{M}{\vdash}} (p, \epsilon, \beta)$$

for some p in F and β in Γ^*, it follows that

$$([q_0, \epsilon], w, Z_0) \overset{*}{\underset{M'}{\vdash}} ([p, \epsilon], \epsilon, \beta),$$

so M' accepts w. Thus $L(M') \supseteq h^{-1}(L(M))$.

Conversely, suppose M' accepts $w = a_1 a_2 \cdots a_n$. Then since rule (3) can be applied only with the buffer (second component of M''s state) empty, the sequence of the moves of M' leading to acceptance can be written

$$\begin{aligned}([q_0, \epsilon], a_1 a_2 \cdots a_n, Z_0) &\overset{*}{\underset{M'}{\vdash}} ([p_1, \epsilon], a_1 a_2 \cdots a_n, \alpha_1),\\ &\underset{M'}{\vdash} ([p_1, h(a_1)], a_2 a_3 \cdots a_n, \alpha_1),\\ &\overset{*}{\underset{M'}{\vdash}} ([p_2, \epsilon], a_2 a_3 \cdots a_n, \alpha_2),\\ &\underset{M'}{\vdash} ([p_2, h(a_2)], a_3 a_4 \cdots a_n, \alpha_2),\\ &\vdots\\ &\overset{*}{\underset{M'}{\vdash}} ([p_{n-1}, \epsilon], a_n, \alpha_n),\\ &\underset{M'}{\vdash} ([p_{n-1}, h(a_n)], \epsilon, \alpha_n),\\ &\overset{*}{\underset{M'}{\vdash}} ([p_n, \epsilon], \epsilon, \alpha_{n+1}),\end{aligned}$$

where p_n is in F. The transitions from state $[p_i, \epsilon]$ to $[p_i, h(a_i)]$ are by rule (3), the other transitions are by rules (1) and (2). Thus, $(q_0, \epsilon, Z_0) \overset{*}{\underset{M}{\vdash}} (p_1 \epsilon, \alpha_1)$, and for all i,

$$(p_i, h(a_i), \alpha_i) \overset{*}{\underset{M}{\vdash}} (p_{i+1}, \epsilon, \alpha_{i+1}).$$

Putting these moves together, we have

$$(q_0, h(a_1 a_2 \cdots a_n), Z_0) \overset{*}{\underset{M}{\vdash}} (p_n, \epsilon, \alpha_{n+1}),$$

so $h(a_1 a_2 \cdots a_n)$ is in $L(M)$. Hence $L(M') \subseteq h^{-1}(L(M))$, whereupon we conclude $L(M') = h^{-1}(L(M))$. □

Boolean operations

There are several closure properties of regular sets that are not possessed by the context-free languages. Notable among these are closure under intersection and complementation.

Theorem 6.4 The CFL's are not closed under intersection.

Proof In Example 6.1 we showed the language $L_1 = \{a^i b^i c^i \mid i \geq 1\}$ was not a CFL. We claim that $L_2 = \{a^i b^i c^j \mid i \geq 1 \text{ and } j \geq 1\}$ and $L_3 = \{a^i b^j c^j \mid i \geq 1 \text{ and } j \geq 1\}$ are both CFL's. For example, a PDA to recognize L_2 stores the a's on its stack and cancels them against b's, then accepts its input after seeing one or more c's. Alternatively L_2 is generated by the grammar

$$S \to AB$$
$$A \to aAb \mid ab$$
$$B \to cB \mid c$$

where A generates $a^i b^i$ and B generates c^j. A similar grammar

$$S \to CD$$
$$C \to aC \mid a$$
$$D \to bDc \mid bc$$

generates L_3.

However, $L_2 \cap L_3 = L_1$. If the CFL's were closed under intersection, L_1 would thus be a CFL, contradicting Example 6.1. □

Corollary The CFL's are not closed under complementation.

Proof We know the CFL's are closed under union. If they were closed under complementation, they would, by DeMorgan's law, $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ be closed under intersection, contradicting Theorem 6.4. □

Although the class of CFL's is not closed under intersection it is closed under intersection with a regular set.

Theorem 6.5 If L is a CFL and R is a regular set, then $L \cap R$ is a CFL.

Proof Let L be $L(M)$ for PDA $M = (Q_M, \Sigma, \Gamma, \delta_M, q_0, Z_0, F_M)$, and let R be $L(A)$ for DFA $A = (Q_A, \Sigma, \delta_A, p_0, F_A)$. We construct a PDA M' for $L \cap R$ by "running M and A in parallel," as shown in Fig. 6.6. M' simulates moves of M on input ϵ without changing the state of A. When M' makes a move on input symbol a, M simulates that move and also simulates A's change of state on input a. M' accepts if and only if both A and M accept. Formally, let

$$M' = (Q_A \times Q_M, \Sigma, \Gamma, \delta, [p_0, q_0], Z_0, F_A \times F_M),$$

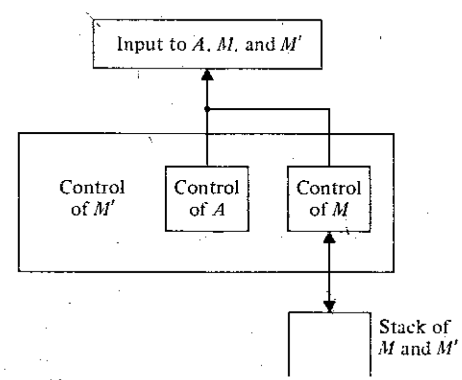


Fig. 6.6 Running an FA and a PDA in parallel.

where δ is defined by $\delta([p, q], a, X)$, contains $([p', q'], \gamma)$ if and only if $\delta_A(p, a) = p'$, and $\delta_M(q, a, X)$ contains (q', γ). Note that a may be ϵ, in which case $p' = p$.

An easy induction on i shows that

$$([p_0, q_0], w, Z_0) \overset{i}{\underset{M'}{\vdash}} ([p, q], \epsilon, \gamma)$$

if and only if

$$(q_0, w, Z_0) \overset{i}{\underset{M}{\vdash}} (q, \epsilon, \gamma) \quad \text{and} \quad \delta(p_0, w) = p.$$

The basis, $i = 0$, is trivial, since $p = p_0$, $q = q_0$, $\gamma = Z_0$, and $w = \epsilon$. For the induction, assume the statement for $i - 1$, and let

$$([p_0, q_0], xa, Z_0) \overset{i-1}{\underset{M'}{\vdash}} ([p', q'], a, \beta) \underset{M'}{\vdash} ([p, q], \epsilon, \gamma),$$

where $w = xa$, and a is ϵ or a symbol of Σ. By the inductive hypothesis,

$$\delta_A(p_0, x) = p' \quad \text{and} \quad (q_0, x, Z_0) \overset{i-1}{\underset{M}{\vdash}} (q', \epsilon, \beta).$$

By the definition of δ, the fact that $([p', q'], a, \beta) \underset{M'}{\vdash} ([p, q], \epsilon, \gamma)$ tells us that $\delta_A(p', a) = p$ and $(q', a, \beta) \underset{M}{\vdash} (q, \epsilon, \gamma)$. Thus $\delta_A(p_0, w) = p$ and

$$(q_0, w, Z_0) \overset{i}{\underset{M}{\vdash}} (q, \epsilon, \gamma).$$

The converse, showing that $(q_0, w, Z_0) \overset{i}{\underset{M}{\vdash}} (q, \epsilon, \gamma)$ and $\delta_A(p_0, w) = p$ imply

$$([p_0, q_0], w, Z_0) \overset{i}{\underset{M'}{\vdash}} ([p, q], \epsilon, \gamma),$$

is similar and left as an exercise. □

Use of closure properties

We conclude this section with an example illustrating the use of closure properties of context-free languages to prove that certain languages are not context free.

Example 6.5 Let $L = \{ww \mid w \text{ is in } (\mathbf{a}+\mathbf{b})^*\}$. That is, L consists of all words whose first and last halves are the same. Suppose L were context free. Then by Theorem 6.5, $L_1 = L \cap \mathbf{a}^+\mathbf{b}^+\mathbf{a}^+\mathbf{b}^+$ would also be a CFL. But $L_1 = \{a^ib^ja^ib^j \mid i \geq 1, j \geq 1\}$. L_1 is almost the same as the language proved not to be context free in Example 6.2, using the pumping lemma. The same argument shows that L_1 is not a CFL. We thus contradict the assumption that L is a CFL.

If we did not want to use the pumping lemma on L_1, we could reduce it to $L_2 = \{a^ib^jc^id^j \mid i \geq 1 \text{ and } j \geq 1\}$, the exact language discussed in Example 6.2. Let h be the homomorphism $h(a) = h(c) = a$ and $h(b) = h(d) = b$. Then $h^{-1}(L_1)$ consists of all words of the form $x_1x_2x_3x_4$, where x_1 and x_3 are of the same length and in $(\mathbf{a}+\mathbf{c})^+$, and x_2 and x_4 are of equal length and in $(\mathbf{b}+\mathbf{d})^+$. Then $h^{-1}(L_1) \cap \mathbf{a}^*\mathbf{b}^*\mathbf{c}^*\mathbf{d}^* = L_2$. By Theorems 6.3 and 6.5, if L_1 were a CFL, so would be L_2. Since L_2 is known not to be a CFL, we conclude that L_1 is not a CFL.

6.3 DECISION ALGORITHMS FOR CFL's

There are a number of questions about CFL's we can answer. These include whether a given CFL is empty, finite, or infinite and whether a given word is in a given CFL. There are, however, certain questions about CFL's that no algorithm can answer. These include whether two CFG's are equivalent, whether a CFL is cofinite, whether the complement of a given CFL is also a CFL, and whether a given CFG is ambiguous. In the next two chapters we shall develop tools for showing that no algorithm to do a particular job exists. In Chapter 8 we shall actually prove that the above questions and others have no algorithms. In this chapter we shall content ourselves with giving algorithms for some of the questions that have algorithms.

As with regular sets, we have several representations for CFL's, namely context-free grammars and pushdown automata accepting by empty stack or by final state. As the constructions of Chapter 5 are all effective, an algorithm that uses one representation can be made to work for any of the others. We shall use the CFG representation in this section.

Theorem 6.6 There are algorithms to determine if a CFL is (a) empty, (b) finite, or (c) infinite.

Proof The theorem can be proved by the same technique (Theorem 3.7) as the analogous result for regular sets, by making use of the pumping lemma. However, the resulting algorithms are highly inefficient. Actually, we have already given a better algorithm to test whether a CFL is empty. For a CFG $G = (V, T, P, S)$, the test of Lemma 4.1 determines if a variable generates any string of terminals. Clearly, $L(G)$ is nonempty if and only if the start symbol S generates some string of terminals.

To test whether $L(G)$ is finite, use the algorithm of Theorem 4.5 to find a CFG $G' = (V', T, P', S)$ in CNF and with no useless symbols, generating $L(G) - \{\epsilon\}$. $L(G')$ is finite if and only if $L(G)$ is finite. A simple test for finiteness of a CNF grammar with no useless symbols is to draw a directed graph with a vertex for each variable and an edge from A to B if there is a production of the form $A \to BC$ or $A \to CB$ for any C. Then the language generated is finite if and only if this graph has no cycles.

If there is a cycle, say $A_0, A_1, \ldots, A_n, A_0$, then

$$A_0 \Rightarrow \alpha_1 A_1 \beta_1 \Rightarrow \alpha_2 A_2 \beta_2 \cdots \Rightarrow \alpha_n A_n \beta_n \Rightarrow \alpha_{n+1} A_0 \beta_{n+1},$$

where the α's and β's are strings of variables, with $|\alpha_i \beta_i| = i$. Since there are no useless symbols, $\alpha_{n+1} \overset{*}{\Rightarrow} w$ and $\beta_{n+1} \overset{*}{\Rightarrow} x$ for some terminal strings w and x of total length at least $n+1$. Since $n \geq 0$, w and x cannot both be ϵ. Next, as there are no useless symbols, we can find terminal strings y and z such that $S \overset{*}{\Rightarrow} yA_0z$, and a terminal string v such that $A_0 \overset{*}{\Rightarrow} v$. Then for all i,

$$S \overset{*}{\Rightarrow} yA_0z \overset{*}{\Rightarrow} ywA_0xz \overset{*}{\Rightarrow} yw^2A_0x^2z \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} yw^iA_0x^iz \overset{*}{\Rightarrow} yw^ivx^iz.$$

As $|wx| > 0$, yw^ivx^iz cannot equal yw^jvx^jz if $i \neq j$. Thus the grammar generates an infinite number of strings.

Conversely, suppose the graph has no cycles. Define the *rank* of a variable A to be the length of the longest path in the graph beginning at A. The absence of cycles implies that the rank of A is finite. We also observe that if $A \to BC$ is a production, then the rank of B and C must be strictly less than the rank of A, because for every path from B or C, there is a path of length one greater from A. We show by induction on r that if A has rank r, then no terminal string derived from A has length greater than 2^r.

Basis $r = 0$. If A has rank 0, then its vertex has no edges out. Therefore all A-productions have terminals on the right, and A derives only strings of length 1.

Induction $r > 0$. If we use a production of the form $A \to a$, we may derive only a string of length 1. If we begin with $A \to BC$, then as B and C are of rank $r - 1$ or less, by the inductive hypothesis, they derive only strings of length 2^{r-1} or less. Thus BC cannot derive a string of length greater than 2^r.

Since S is of finite rank r_0, and in fact, is of rank no greater than the number of variables, S derives strings of length no greater than 2^{r_0}. Thus the language is finite. □

Example 6.6 Consider the grammar

$$S \to AB$$
$$A \to BC \mid a$$
$$B \to CC \mid b$$
$$C \to a$$

whose graph is shown in Fig. 6.7(a). This graph has no cycles. The ranks of S, A, B, and C are 3, 2, 1, and 0, respectively. For example, the longest path from S is S, A, B, C. Thus this grammar derives no string of length greater than $2^3 = 8$ and therefore generates a finite language. In fact, a longest string generated from S is

$$S \Rightarrow AB \Rightarrow BCB \Rightarrow CCCB \Rightarrow CCCCC \overset{*}{\Rightarrow} aaaaa.$$

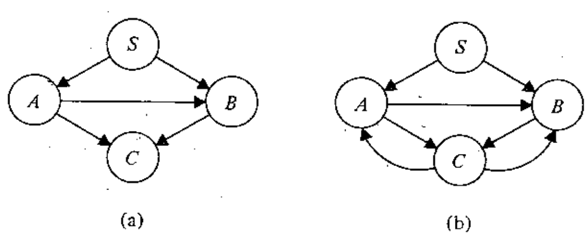


Fig. 6.7 Graphs corresponding to CNF grammars.

If we add production $C \to AB$, we get the graph of Fig. 6.7(b). This new graph has several cycles, such as A, B, C, A. Thus we can find a derivation $A \overset{*}{\Rightarrow} \alpha_3 A \beta_3$, in particular $A \Rightarrow BC \Rightarrow CCC \Rightarrow CABC$, where $\alpha_3 = C$ and $\beta_3 = BC$. Since $C \overset{*}{\Rightarrow} a$ and $BC \overset{*}{\Rightarrow} ba$, we have $A \overset{*}{\Rightarrow} aAba$. Then as $S \overset{*}{\Rightarrow} Ab$ and $A \overset{*}{\Rightarrow} a$, we now have $S \overset{*}{\Rightarrow} a^ia(ba)^ib$ for every i. Thus the language is infinite.

Membership

Another question we may answer is: Given a CFG $G = (V, T, P, S)$ and string x in T^*, is x in $L(G)$? A simple but inefficient algorithm to do so is to convert G to $G' = (V', T, P', S)$, a grammar in Greibach normal form generating $L(G) - \{\epsilon\}$. Since the algorithm of Theorem 4.3 tests whether $S \overset{*}{\Rightarrow} \epsilon$, we need not concern ourselves with the case $x = \epsilon$. Thus assume $x \neq \epsilon$, so x is in $L(G')$ if and only if x is in $L(G)$. Now, as every production of a GNF grammar adds exactly one terminal to the string being generated, we know that if x has a derivation in G', it has one with exactly $|x|$ steps. If no variable of G' has more than k productions, then there are at most $k^{|x|}$ leftmost derivations of strings of length $|x|$. We may try them all systematically.

However, the above algorithm can take time which is exponential in $|x|$. There are several algorithms known that take time proportional to the cube of $|x|$ or even a little less. The bibliographic notes discuss some of these. We shall here present a simple cubic time algorithm known as the Cocke-Younger-Kasami or *CYK* algorithm. It is based on the dynamic programming technique discussed in the solution to Exercise 3.23. Given x of length $n \geq 1$, and a grammar G, which we may assume is in Chomsky normal form, determine for each i and j and for each variable A, whether $A \overset{*}{\Rightarrow} x_{ij}$, where x_{ij} is the substring of x of length j beginning at position i.

We proceed by induction on j. For $j = 1$, $A \overset{*}{\Rightarrow} x_{ij}$ if and only if $A \to x_{ij}$ is a production, since x_{ij} is a string of length 1. Proceeding to higher values of j, if $j > 1$, then $A \overset{*}{\Rightarrow} x_{ij}$ if and only if there is some production $A \to BC$ and some k, $1 \leq k < j$, such that B derives the first k symbols of x_{ij} and C derives the last $j - k$ symbols of x_{ij}. That is, $B \overset{*}{\Rightarrow} x_{ik}$ and $C \overset{*}{\Rightarrow} x_{i+k,j-k}$. Since k and $j - k$ are both less than j, we already know whether each of the last two derivations exists. We may thus determine whether $A \overset{*}{\Rightarrow} x_{ij}$. Finally, when we reach $j = n$, we may determine whether $S \overset{*}{\Rightarrow} x_{1n}$. But $x_{1n} = x$, so x is in $L(G)$ if and only if $S \overset{*}{\Rightarrow} x_{1n}$.

To state the CYK algorithm precisely, let V_{ij} be the set of variables A such that $A \overset{*}{\Rightarrow} x_{ij}$. Note that we may assume $1 \leq i \leq n - j + 1$, for there is no string of length greater than $n - i + 1$ beginning at position i. Then Fig. 6.8 gives the CYK algorithm formally.

Steps (1) and (2) handle the case $j = 1$. As the grammar G is fixed, step (2) takes a constant amount of time. Thus steps (1) and (2) take $O(n)$ time. The nested for-loops of lines (3) and (4) cause steps (5) through (7) to be executed at most n^2 times, since i and j range in their respective for-loops between limits that are at

```
    begin
1)    for i:= 1 to n do
2)      V_i1:= {A | A → a is a production and the ith symbol of x is a};
3)    for j:= 2 to n do
4)      for i:= 1 to n − j + 1 do
          begin
5)          V_ij:= ∅;
6)          for k:= 1 to j − 1 do
7)            V_ij:= V_ij ∪ {A | A → BC is a production, B is in V_ik and C
                is in V_{i+k,j−k}}
          end
    end
```

Fig. 6.8. The CYK algorithm.

most n apart. Step (5) takes constant time at each execution, so the aggregate time spent at step (5) is $0(n^2)$. The for-loop of line (6) causes step (7) to be executed n or fewer times. Since step (7) takes constant time, steps (6) and (7) together take $0(n)$ time. As they are executed $0(n^2)$ times, the total time spent in step (7) is $0(n^3)$. Thus the entire algorithm is $0(n^3)$.

Example 6.7 Consider the CFG

$$S \to AB \mid BC$$
$$A \to BA \mid a$$
$$B \to CC \mid b$$
$$C \to AB \mid a$$

and the input string *baaba*. The table of V_{ij}'s is shown in Fig. 6.9. The top row is filled in by steps (1) and (2) of the algorithm in Fig. 6.8. That is, for positions 1 and 4, which are b, we set $V_{11} = V_{41} = \{B\}$, since B is the only variable which derives b.

	b	*a*	*a*	*b*	*a*
$j \downarrow$ / $i \to$	1	2	3	4	5
1	B	A, C	A, C	B	A, C
2	S, A	B	S, C	S, A	
3	$\varnothing$	B	B		
4	$\varnothing$	S, A, C			
5	S, A, C				

Fig. 6.9 Table of V_{ij}'s.

Similarly, $V_{21} = V_{31} = V_{51} = \{A, C\}$, since only A and C have productions with a on the right.

To compute V_{ij} for $j > 1$, we must execute the for-loop of steps (6) and (7). We must match V_{ik} against $V_{i+k,j-k}$ for $k = 1, 2, \ldots, j-1$, seeking variable D in V_{ik} and E in $V_{i+k,j-k}$ such that DE is the right side of one or more productions. The left sides of these productions are adjoined to V_{ij}. The pattern in the table which corresponds to visiting V_{ik} and $V_{i+k,j-k}$ for $k = 1, 2, \ldots, j-1$ in turn is to simultaneously move down column i and up the diagonal extending from V_{ij} to the right, as shown in Fig. 6.10.

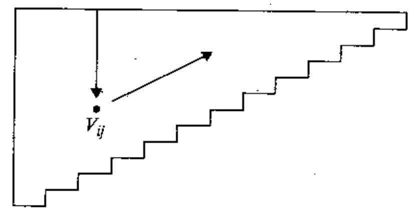


Fig. 6.10 Traversal pattern for computation of V_{ij}.

For example, let us compute V_{24}, assuming that the top three rows of Fig. 6.9 are filled in. We begin by looking at $V_{21} = \{A, C\}$ and $V_{33} = \{B\}$. The possible right-hand sides in $V_{21}V_{33}$ are AB and CB. Only the first of these is actually a right side, and it is a right side of two productions $S \to AB$ and $C \to AB$. Hence we add S and C to V_{24}. Next we consider $V_{22}V_{42} = \{B\}\{S, A\} = \{BS, BA\}$. Only BA is a right side, so we add the corresponding left side A to V_{24}. Finally, we consider $V_{23}V_{51} = \{B\}\{A, C\} = \{BA, BC\}$. BA and BC are each right sides, with left sides A and S, respectively. These are already in V_{24}, so we have $V_{24} = \{S, A, C\}$. Since S is a member of V_{15}, the string *baaba* is in the language generated by the grammar.

EXERCISES

6.1 Show that the following are not context-free languages.

a) $\{a^ib^jc^k \mid i < j < k\}$
b) $\{a^ib^j \mid j = i^2\}$
c) $\{a^i \mid i \text{ is a prime}\}$
d) the set of strings of a's, b's, and c's with an equal number of each
e) $\{a^nb^nc^m \mid n \le m \le 2n\}$

* **6.2** Which of the following are CFL's?

a) $\{a^ib^j \mid i \ne j \text{ and } i \ne 2j\}$
b) $(\mathbf{a} + \mathbf{b})^* - \{(a^nb^n)^n \mid n \ge 1\}$
c) $\{ww^Rw \mid w \text{ is in } (\mathbf{a} + \mathbf{b})^*\}$
d) $\{b_i \# b_{i+1} \mid b_i \text{ is } i \text{ in binary}, i \ge 1\}$