NAVODAYA INSTITUTE OF TECHNOLOGY, RAICHUR

DEPARMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE 4

Chapter 7

Properties of Context-Free Languages

We shall complete our study of context-free languages by learning some of their properties. Our first task is to simplify context-free grammars; these simplifications make it easier to prove facts about CFL's, since we can claim that if a language is a CFL, then it has a grammar in some special form.

We then prove a "pumping lemma" for CFL's. This theorem is in the same spirit as Theorem 4.1 for regular languages, but can be used to prove a language not to be context-free. Next, we consider the sorts of properties that we studied in Chapter 4 for the regular languages: closure properties and decision properties. We shall see that some, but not all, of the closure properties that the regular languages have are also possessed by the CFL's. Likewise, some questions about CFL's can be decided by algorithms that generalize the tests we developed for regular languages, but there are also certain questions about CFL's that we cannot answer.

7.1 Normal Forms for Context-Free Grammars

The goal of this section is to show that every CFL (without ϵ) is generated by a CFG in which all productions are of the form $A \to BC$ or $A \to a$, where A, B, and C are variables, and a is a terminal. This form is called *Chomsky Normal Form*. To get there, we need to make a number of preliminary simplifications, which are themselves useful in various ways:

1. We must eliminate *useless symbols*, those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.

2. We must eliminate *ϵ-productions*, those of the form $A \to \epsilon$ for some variable A.

3. We must eliminate *unit productions*, those of the form $A \to B$ for variables A and B.

7.1.1 Eliminating Useless Symbols

We say a symbol X is *useful* for a grammar $G = (V, T, P, S)$ if there is some derivation of the form $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w$, where w is in T^*. Note that X may be in either V or T, and the sentential form $\alpha X \beta$ might be the first or last in the derivation. If X is not useful, we say it is *useless*. Evidently, omitting useless symbols from a grammar will not change the language generated, so we may as well detect and eliminate all useless symbols.

Our approach to eliminating useless symbols begins by identifying the two things a symbol has to be able to do to be useful:

1. We say X is *generating* if $X \stackrel{*}{\Rightarrow} w$ for some terminal string w. Note that every terminal is generating, since w can be that terminal itself, which is derived by zero steps.

2. We say X is *reachable* if there is a derivation $S \stackrel{*}{\Rightarrow} \alpha X \beta$ for some α and β.

Surely a symbol that is useful will be both generating and reachable. If we eliminate the symbols that are not generating first, and then eliminate from the remaining grammar those symbols that are not reachable, we shall, as will be proved, have only the useful symbols left.

Example 7.1: Consider the grammar:

$$S \to AB \mid a$$
$$A \to b$$

All symbols but B are generating; a and b generate themselves; S generates a, and A generates b. If we eliminate B, we must eliminate the production $S \to AB$, leaving the grammar:

$$S \to a$$
$$A \to b$$

Now, we find that only S and a are reachable from S. Eliminating A and b leaves only the production $S \to a$. That production by itself is a grammar whose language is $\{a\}$, just as is the language of the original grammar.

Note that if we start by checking for reachability first, we find that all symbols of the grammar

$$S \to AB \mid a$$
$$A \to b$$

are reachable. If we then eliminate the symbol B because it is not generating, we are left with a grammar that still has useless symbols, in particular, A and b. □

Theorem 7.2: Let $G = (V, T, P, S)$ be a CFG, and assume that $L(G) \neq \emptyset$; i.e., G generates at least one string. Let $G_1 = (V_1, T_1, P_1, S)$ be the grammar we obtain by the following steps:

1. First eliminate nongenerating symbols and all productions involving one or more of those symbols. Let $G_2 = (V_2, T_2, P_2, S)$ be this new grammar. Note that S must be generating, since we assume $L(G)$ has at least one string, so S has not been eliminated.

2. Second, eliminate all symbols that are not reachable in the grammar G_2.

Then G_1 has no useless symbols, and $L(G_1) = L(G)$.

PROOF: Suppose X is a symbol that remains; i.e., X is in $V_1 \cup T_1$. We know that $X \underset{G}{\overset{*}{\Rightarrow}} w$ for some w in T^*. Moreover, every symbol used in the derivation of w from X is also generating. Thus, $X \underset{G_2}{\overset{*}{\Rightarrow}} w$.

Since X was not eliminated in the second step, we also know that there are α and β such that $S \underset{G_2}{\overset{*}{\Rightarrow}} \alpha X \beta$. Further, every symbol used in this derivation is reachable, so $S \underset{G_1}{\overset{*}{\Rightarrow}} \alpha X \beta$.

We know that every symbol in $\alpha X \beta$ is reachable, and we also know that all these symbols are in $V_2 \cup T_2$, so each of them is generating in G_2. The derivation of some terminal string, say $\alpha X \beta \underset{G_2}{\overset{*}{\Rightarrow}} xwy$, involves only symbols that are reachable from S, because they are reached by symbols in $\alpha X \beta$. Thus, this derivation is also a derivation of G_1; that is,

$$S \underset{G_1}{\overset{*}{\Rightarrow}} \alpha X \beta \underset{G_1}{\overset{*}{\Rightarrow}} xwy$$

We conclude that X is useful in G_1. Since X is an arbitrary symbol of G_1, we conclude that G_1 has no useless symbols.

The last detail is that we must show $L(G_1) = L(G)$. As usual, to show two sets the same, we show each is contained in the other.

$L(G_1) \subseteq L(G)$: Since we have only eliminated symbols and productions from G to get G_1, it follows that $L(G_1) \subseteq L(G)$.

$L(G) \subseteq L(G_1)$: We must prove that if w is in $L(G)$, then w is in $L(G_1)$. If w is in $L(G)$, then $S \underset{G}{\overset{*}{\Rightarrow}} w$. Each symbol in this derivation is evidently both reachable and generating, so it is also a derivation of G_1. That is, $S \underset{G_1}{\overset{*}{\Rightarrow}} w$, and thus w is in $L(G_1)$. □

7.1.2 Computing the Generating and Reachable Symbols

Two points remain. How do we compute the set of generating symbols of a grammar, and how do we compute the set of reachable symbols of a grammar? For both problems, the algorithm we use tries its best to discover symbols of these types. We shall show that if the proper inductive constructions of these sets fails to discover a symbol to be generating or reachable, respectively, then the symbol is not of these types.

Let $G = (V, T, P, S)$ be a grammar. To compute the generating symbols of G, we perform the following induction.

BASIS: Every symbol of T is obviously generating; it generates itself.

INDUCTION: Suppose there is a production $A \to \alpha$, and every symbol of α is already known to be generating. Then A is generating. Note that this rule includes the case where $\alpha = \epsilon$; all variables that have ϵ as a production body are surely generating.

Example 7.3: Consider the grammar of Example 7.1. By the basis, a and b are generating. For the induction, we can use the production $A \to b$ to conclude that A is generating, and we can use the production $S \to a$ to conclude that S is generating. At that point, the induction is finished. We cannot use the production $S \to AB$, because B has not been established to be generating. Thus, the set of generating symbols is $\{a, b, A, S\}$. □

Theorem 7.4: The algorithm above finds all and only the generating symbols of G.

PROOF: For one direction, it is an easy induction on the order in which symbols are added to the set of generating symbols that each symbol added really is generating. We leave to the reader this part of the proof.

For the other direction, suppose X is a generating symbol, say $X \stackrel{*}{\underset{G}{\Rightarrow}} w$. We prove by induction on the length of this derivation that X is found to be generating.

BASIS: Zero steps. Then X is a terminal, and X is found in the basis.

INDUCTION: If the derivation takes n steps for $n > 0$, then X is a variable. Let the derivation be $X \Rightarrow \alpha \stackrel{*}{\Rightarrow} w$; that is, the first production used is $X \to \alpha$. Each symbol of α derives some terminal string that is a part of w, and that derivation must take fewer than n steps. By the inductive hypothesis, each symbol of α is found to be generating. The inductive part of the algorithm allows us to use production $X \to \alpha$ to infer that X is generating. □

Now, let us consider the inductive algorithm whereby we find the set of reachable symbols for the grammar $G = (V, T, P, S)$. Again, we can show that by trying our best to discover reachable symbols, any symbol we do not add to the reachable set is really not reachable.

BASIS: S is surely reachable.

INDUCTION: Suppose we have discovered that some variable A is reachable. Then for all productions with A in the head, all the symbols of the bodies of those productions are also reachable.

Example 7.5: Again start with the grammar of Example 7.1. By the basis, S is reachable. Since S has production bodies AB and a, we conclude that A, B, and a are reachable. B has no productions, but A has $A \to b$. We therefore conclude that b is reachable. Now, no more symbols can be added to the reachable set, which is $\{S, A, B, a, b\}$. □

Theorem 7.6: The algorithm above finds all and only the reachable symbols of G.

PROOF: This proof is another pair of simple inductions akin to Theorem 7.4. We leave these arguments as an exercise. □

7.1.3 Eliminating ϵ-Productions

Now, we shall show that ϵ-productions, while a convenience in many grammar-design problems, are not essential. Of course without a production that has an ϵ body, it is impossible to generate the empty string as a member of the language. Thus, what we actually prove is that if language L has a CFG, then $L - \{\epsilon\}$ has a CFG without ϵ-productions. If ϵ is not in L, then L itself is $L - \{\epsilon\}$, so L has a CFG with out ϵ-productions.

Our strategy is to begin by discovering which variables are "nullable." A variable A is *nullable* if $A \stackrel{*}{\Rightarrow} \epsilon$. If A is nullable, then whenever A appears in a production body, say $B \to CAD$, A might (or might not) derive ϵ. We make two versions of the production, one without A in the body ($B \to CD$), which corresponds to the case where A would have been used to derive ϵ, and the other with A still present ($B \to CAD$). However, if we use the version with A present, then we cannot allow A to derive ϵ. That proves not to be a problem, since we shall simply eliminate all productions with ϵ bodies, thus preventing any variable from deriving ϵ.

Let $G = (V, T, P, S)$ be a CFG. We can find all the nullable symbols of G by the following iterative algorithm. We shall then show that there are no nullable symbols except what the algorithm finds.

BASIS: If $A \to \epsilon$ is a production of G, then A is nullable.

INDUCTION: If there is a production $B \to C_1 C_2 \cdots C_k$, where each C_i is nullable, then B is nullable. Note that each C_i must be a variable to be nullable, so we only have to consider productions with all-variable bodies.

Theorem 7.7: In any grammar G, the only nullable symbols are the variables found by the algorithm above.

PROOF: For the "if" direction of the implied "A is nullable if and only if the algorithm identifies A as nullable," we simply observe that, by an easy induction on the order in which nullable symbols are discovered, that the each such symbol truly derives ϵ. For the "only-if" part, we can perform an induction on the length of the shortest derivation $A \stackrel{*}{\Rightarrow} \epsilon$.

BASIS: One step. Then $A \to \epsilon$ must be a production, and A is discovered in the basis part of the algorithm.

INDUCTION: Suppose $A \stackrel{*}{\Rightarrow} \epsilon$ by n steps, where $n > 1$. The first step must look like $A \Rightarrow C_1C_2 \cdots C_k \stackrel{*}{\Rightarrow} \epsilon$, where each C_i derives ϵ by a sequence of fewer than n steps. By the inductive hypothesis, each C_i is discovered by the algorithm to be nullable. Thus, by the inductive step, A, thanks to the production $A \to C_1C_2 \cdots C_k$, is found to be nullable. □

Now we give the construction of a grammar without ϵ-productions. Let $G = (V, T, P, S)$ be a CFG. Determine all the nullable symbols of G. We construct a new grammar $G_1 = (V, T, P_1, S)$, whose set of productions P_1 is determined as follows.

For each production $A \to X_1X_2 \cdots X_k$ of P, where $k \geq 1$, suppose that m of the k X_i's are nullable symbols. The new grammar G_1 will have 2^m versions of this production, where the nullable X_i's, in all possible combinations are present or absent. There is one exception: if $m = k$, i.e., all symbols are nullable, then we do not include the case where all X_i's are absent. Also, note that if a production of the form $A \to \epsilon$ is in P, we do not place this production in P_1.

Example 7.8: Consider the grammar

$$\begin{aligned} S &\to AB \\ A &\to aAA \mid \epsilon \\ B &\to bBB \mid \epsilon \end{aligned}$$

First, let us find the nullable symbols. A and B are directly nullable because they have productions with ϵ as the body. Then, we find that S is nullable, because the production $S \to AB$ has a body consisting of nullable symbols only. Thus, all three variables are nullable.

Now, let us construct the productions of grammar G_1. First consider $S \to AB$. All symbols of the body are nullable, so there are four ways we could choose present or absent for A and B, independently. However, we are not allowed to choose to make all symbols absent, so there are only three productions:

$$S \to AB \mid A \mid B$$

Next, consider production $A \to aAA$. The second and third positions hold nullable symbols, so again there are four choices of present/absent. In this case,

all four choices are allowable, since the nonnullable symbol a will be present in any case. Our four choices yield productions:

$$A \to aAA \mid aA \mid aA \mid a$$

Note that the two middle choices happen to yield the same production, since it doesn't matter which of the A's we eliminate if we decide to eliminate one of them. Thus, the final grammar G_1 will only have three productions for A.

Similarly, the production B yields for G_1:

$$B \to bBB \mid bB \mid b$$

The two ϵ-productions of G yield nothing for G_1. Thus, the following productions:

$$\begin{aligned} S &\to AB \mid A \mid B \\ A &\to aAA \mid aA \mid a \\ B &\to bBB \mid bB \mid b \end{aligned}$$

constitute G_1. □

We conclude our study of the elimination of ϵ-productions by proving that the construction given above does not change the language, except that ϵ is no longer present if it was in the language of G. Since the construction obviously eliminates ϵ-productions, we shall have a complete proof of the claim that for every CFG G, there is a grammar G_1 with no ϵ-productions, such that

$$L(G_1) = L(G) - \{\epsilon\}$$

Theorem 7.9: If the grammar G_1 is constructed from G by the above construction for eliminating ϵ-productions, then $L(G_1) = L(G) - \{\epsilon\}$.

PROOF: We must show that if $w \neq \epsilon$, then w is in $L(G_1)$ if and only if w is in $L(G)$. As is often the case, we find it easier to prove a more general statement. In this case, we need to talk about the terminal strings that each variable generates, even though we only care what the start symbol S generates. Thus, we shall prove:

- $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ if and only if $A \overset{*}{\underset{G}{\Rightarrow}} w$ and $w \neq \epsilon$.

In each case, the proof is an induction on the length of the derivation.

(Only-if) Suppose that $A \overset{*}{\underset{G_1}{\Rightarrow}} w$. Then surely $w \neq \epsilon$, because G_1 has no ϵ-productions. We must show by induction on the length of the derivation that $A \overset{*}{\underset{G}{\Rightarrow}} w$.

BASIS: One step. Then there is a production $A \to w$ in G_1. The construction of G_1 tells us that there is some production $A \to \alpha$ of G, such that α is w, with zero or more nullable variables interspersed. Then in G, $A \underset{G}{\Rightarrow} \alpha \overset{*}{\underset{G}{\Rightarrow}} w$, where the steps after the first, if any, derive ϵ from whatever variables there are in α.

INDUCTION: Suppose the derivation takes $n > 1$ steps. Then the derivation looks like $A \underset{G_1}{\Rightarrow} X_1X_2\cdots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$. The first production used must come from a production $A \to Y_1Y_2\cdots Y_m$, where the Y's are the X's, in order, with zero or more additional, nullable variables interspersed. Also, we can break w into $w_1w_2\cdots w_k$, where $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ for $i = 1, 2, \ldots, k$. If X_i is a terminal, then $w_i = X_i$, and if X_i is a variable, then the derivation $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ takes fewer than n steps. By the inductive hypothesis, we can conclude $X_i \overset{*}{\underset{G}{\Rightarrow}} w_i$.

Now, we construct a corresponding derivation in G as follows:

$$A \underset{G}{\Rightarrow} Y_1Y_2\cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} X_1X_2\cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w_1w_2\cdots w_k = w$$

The first step is application of the production $A \to Y_1Y_2\cdots Y_k$ that we know exists in G. The next group of steps represents the derivation of ϵ from each of the Y_j's that is not one of the X_i's. The final group of steps represents the derivations of the w_i's from the X_i's, which we know exist by the inductive hypothesis.

(If) Suppose $A \overset{*}{\underset{G}{\Rightarrow}} w$ and $w \neq \epsilon$. We show by induction on the length n of the derivation, that $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

BASIS: One step. Then $A \to w$ is a production of G. Since $w \neq \epsilon$, this production is also a production of G_1, and $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

INDUCTION: Suppose the derivation takes $n > 1$ steps. Then the derivation looks like $A \underset{G}{\Rightarrow} Y_1Y_2\cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} w$. We can break $w = w_1w_2\cdots w_m$, such that $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ for $i = 1, 2, \ldots, m$. Let $X_1, X_2, \ldots, X_k$ be those of the Y_j's, in order, such that $w_j \neq \epsilon$. We must have $k \geq 1$, since $w \neq \epsilon$. Thus, $A \to X_1X_2\cdots X_k$ is a production of G_1.

We claim that $X_1X_2\cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w$, since the only Y_j's that are not present among the X's were used to derive ϵ, and thus do not contribute to the derivation of w. Since each of the derivations $Y_j \overset{*}{\underset{G}{\Rightarrow}} w_j$ takes fewer than n steps, we may apply the inductive hypothesis and conclude that, if $w_j \neq \epsilon$, then $Y_j \overset{*}{\underset{G_1}{\Rightarrow}} w_j$. Thus, $A \underset{G_1}{\Rightarrow} X_1X_2\cdots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$.

Now, we complete the proof as follows. We know w is in $L(G_1)$ if and only if $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Letting $A = S$ in the above, we know that w is in $L(G_1)$ if and only if $S \overset{*}{\underset{G}{\Rightarrow}} w$ and $w \neq \epsilon$. That is, w is in $L(G_1)$ if and only if w is in $L(G)$ and $w \neq \epsilon$. □

7.1.4 Eliminating Unit Productions

A *unit production* is a production of the form $A \to B$, where both A and B are variables. These productions can be useful. For instance, in Example 5.27, we

saw how using unit productions $E \to T$ and $T \to F$ allowed us to create an unambiguous grammar for simple arithmetic expressions:

$$\begin{array}{lcl} I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \to & I \mid (E) \\ T & \to & F \mid T * F \\ E & \to & T \mid E + T \end{array}$$

However, unit productions can complicate certain proofs, and they also introduce extra steps into derivations that technically need not be there. For instance, we could expand the T in production $E \to T$ in both possible ways, replacing it by the two productions $E \to F \mid T * F$. That change still doesn't eliminate unit productions, because we have introduced unit production $E \to F$ that was not previously part of the grammar. Further expanding $E \to F$ by the two productions for F gives us $E \to I \mid (E) \mid T * F$. We still have a unit production; it is $E \to I$. But if we further expand this I in all six possible ways, we get

$$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F$$

Now the unit production for E is gone. Note that $E \to a$ is *not* a unit production, since the lone symbol in the body is a terminal, rather than a variable as is required for unit productions.

The technique suggested above — expand unit productions until they disappear — often works. However, it can fail if there is a cycle of unit productions, such as $A \to B$, $B \to C$, and $C \to A$. The technique that is guaranteed to work involves first finding all those pairs of variables A and B such that $A \stackrel{*}{\Rightarrow} B$ using a sequence of unit productions only. Note that it is possible for $A \stackrel{*}{\Rightarrow} B$ to be true even though no unit productions are involved. For instance, we might have productions $A \to BC$ and $C \to \epsilon$.

Once we have determined all such pairs, we can replace any sequence of derivation steps in which $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_n \Rightarrow \alpha$ by a production that uses the nonunit production $B_n \to \alpha$ directly from A; that is, $A \to \alpha$. To begin, here is the inductive construction of the pairs (A, B) such that $A \stackrel{*}{\Rightarrow} B$ using only unit productions. Call such a pair a *unit pair*.

BASIS: (A, A) is a unit pair for any variable A. That is, $A \stackrel{*}{\Rightarrow} A$ by zero steps.

INDUCTION: Suppose we have determined that (A, B) is a unit pair, and $B \to C$ is a production, where C is a variable. Then (A, C) is a unit pair.

Example 7.10: Consider the expression grammar of Example 5.27, which we reproduced above. The basis gives us the unit pairs (E, E), (T, T), (F, F), and (I, I). For the inductive step, we can make the following inferences:

1. (E, E) and the production $E \to T$ gives us unit pair (E, T).
2. (E, T) and the production $T \to F$ gives us unit pair (E, F).
3. (E, F) and the production $F \to I$ gives us unit pair (E, I).

4. (T, T) and the production $T \to F$ gives us unit pair (T, F).
5. (T, F) and the production $F \to I$ gives us unit pair (T, I).
6. (F, F) and the production $F \to I$ gives us unit pair (F, I).

There are no more pairs that can be inferred, and in fact these ten pairs represent all the derivations that use nothing but unit productions. □

The pattern of development should by now be familiar. There is an easy proof that our proposed algorithm does get all the pairs we want. We then use the knowledge of those pairs to remove unit productions from a grammar and show that the language of the two grammars is the same.

Theorem 7.11: The algorithm above finds exactly the unit pairs for a CFG G.

PROOF: In one direction, it is an easy induction on the order in which the pairs are discovered, that if (A, B) is found to be a unit pair, then $A \overset{*}{\underset{G}{\Rightarrow}} B$ using only unit productions. We leave this part of the proof to you.

In the other direction, suppose that $A \overset{*}{\underset{G}{\Rightarrow}} B$ using unit productions only. We can show by induction on the length of the derivation that the pair (A, B) will be found.

BASIS: Zero steps. Then $A = B$, and the pair (A, B) is added in the basis.

INDUCTION: Suppose $A \overset{*}{\Rightarrow} B$ using n steps, for some $n > 0$, each step being the application of a unit production. Then the derivation looks like

$$A \overset{*}{\Rightarrow} C \Rightarrow B$$

The derivation $A \overset{*}{\Rightarrow} C$ takes $n - 1$ steps, so by the inductive hypothesis, we discover the pair (A, C). Then the inductive part of the algorithm combines the pair (A, C) with the production $C \to B$ to infer the pair (A, B). □

To eliminate unit productions, we proceed as follows. Given a CFG $G = (V, T, P, S)$, construct CFG $G_1 = (V, T, P_1, S)$:

1. Find all the unit pairs of G.
2. For each unit pair (A, B), add to P_1 all the productions $A \to \alpha$, where $B \to \alpha$ is a nonunit production in P. Note that $A = B$ is possible; in that way, P_1 contains all the nonunit productions in P.

Example 7.12: Let us continue with Example 7.10, which performed step (1) of the construction above for the expression grammar of Example 5.27. Figure 7.1 summarizes step (2) of the algorithm, where we create the new set of productions by using the first member of a pair as the head and all the nonunit bodies for the second member of the pair as the production bodies.

The final step is to eliminate the unit productions from the grammar of Fig. 7.1. The resulting grammar:

Pair	Productions
(E,E)	$E \to E+T$
(E,T)	$E \to T*F$
(E,F)	$E \to (E)$
(E,I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T,T)	$T \to T*F$
(T,F)	$T \to (E)$
(T,I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F,F)	$F \to (E)$
(F,I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I,I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Figure 7.1: Grammar constructed by step (2) of the unit-production-elimination algorithm

$$
\begin{aligned}
&E \to E+T \mid T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1\\
&T \to T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1\\
&F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1\\
&I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1
\end{aligned}
$$

has no unit productions, yet generates the same set of expressions as the grammar of Fig. 5.19. □

Theorem 7.13: If grammar G_1 is constructed from grammar G by the algorithm described above for eliminating unit productions, then $L(G_1) = L(G)$.

PROOF: We show that w is in $L(G)$ if and only if w is in $L(G_1)$.

(If) Suppose $S \underset{G_1}{\overset{*}{\Rightarrow}} w$. Since every production of G_1 is equivalent to a sequence of zero or more unit productions of G followed by a nonunit production of G, we know that $\alpha \underset{G_1}{\Rightarrow} \beta$ implies $\alpha \underset{G}{\overset{*}{\Rightarrow}} \beta$. That is, every step of a derivation in G_1 can be replaced by one or more derivation steps in G. If we put these sequences of steps together, we conclude that $S \underset{G}{\overset{*}{\Rightarrow}} w$.

(Only-if) Suppose now that w is in $L(G)$. Then by the equivalences in Section 5.2, we know that w has a leftmost derivation, i.e., $S \underset{lm}{\Rightarrow} w$. Whenever a unit production is used in a leftmost derivation, the variable of the body becomes the leftmost variable, and so is immediately replaced. Thus, the leftmost derivation in grammar G can be broken into a sequence of steps in which zero or more unit productions are followed by a nonunit production. Note that any nonunit production that is not preceded by a unit production is a "step" by itself. Each of these steps can be performed by one production of G_1, because the construction of G_1 created exactly the productions that reflect zero or more unit productions followed by a nonunit production. Thus, $S \underset{G_1}{\overset{*}{\Rightarrow}} w$. □

We can now summarize the various simplifications described so far. We want to convert any CFG G into an equivalent CFG that has no useless symbols, ϵ-productions, or unit productions. Some care must be taken in the order of application of the constructions. A safe order is:

1. Eliminate ϵ-productions.
2. Eliminate unit productions.
3. Eliminate useless symbols.

You should notice that, just as in Section 7.1.1, where we had to order the two steps properly or the result might have useless symbols, we must order the three steps above as shown, or the result might still have some of the features we thought we were eliminating.

Theorem 7.14: If G is a CFG generating a language that contains at least one string other than ϵ, then there is another CFG G_1 such that $L(G_1) = L(G) - \{\epsilon\}$, and G_1 has no ϵ-productions, unit productions, or useless symbols.

PROOF: Start by eliminating the ϵ-productions by the method of Section 7.1.3. If we then eliminate unit productions by the method of Section 7.1.4, we do not introduce any ϵ-productions, since the bodies of the new productions are each identical to some body of an old production. Finally, we eliminate useless symbols by the method of Section 7.1.1. As this transformation only eliminates productions and symbols, never introducing a new production, the resulting grammar will still be devoid of ϵ-productions and unit productions. □

7.1.5 Chomsky Normal Form

We complete our study of grammatical simplifications by showing that every nonempty CFL without ϵ has a grammar G in which all productions are in one of two simple forms, either:

1. $A \to BC$, where A, B, and C, are each variables, or
2. $A \to a$, where A is a variable and a is a terminal.

Further, G has no useless symbols. Such a grammar is said to be in *Chomsky Normal Form*, or CNF.[1]

To put a grammar in CNF, start with one that satisfies the restrictions of Theorem 7.14; that is, the grammar has no ϵ-productions, unit productions, or useless symbols. Every production of such a grammar is either of the form $A \to a$, which is already in a form allowed by CNF, or it has a body of length 2 or more. Our tasks are to:

[1] N. Chomsky is the linguist who first proposed context-free grammars as a way to describe natural languages, and who proved that every CFG could be converted to this form. Interestingly, CNF does not appear to have important uses in natural linguistics, although we shall see it has several other uses, such as an efficient test for membership of a string in a context-free language (Section 7.4.4).

a) Arrange that all bodies of length 2 or more consist only of variables.

b) Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

The construction for (a) is as follows. For every terminal a that appears in a body of length 2 or more, create a new variable, say A. This variable has only one production, $A \to a$. Now, we use A in place of a everywhere a appears in a body of length 2 or more. At this point, every production has a body that is either a single terminal or at least two variables and no terminals.

For step (b), we must break those productions $A \to B_1 B_2 \cdots B_k$, for $k \geq 3$, into a group of productions with two variables in each body. We introduce $k-2$ new variables, $C_1, C_2, \ldots, C_{k-2}$. The original production is replaced by the $k-1$ productions

$$A \to B_1 C_1, \quad C_1 \to B_2 C_2, \ldots, C_{k-3} \to B_{k-2} C_{k-2}, \quad C_{k-2} \to B_{k-1} B_k$$

Example 7.15: Let us convert the grammar of Example 7.12 to CNF. For part (a), notice that there are eight terminals, a, b, 0, 1, +, *, (, and), each of which appears in a body that is not a single terminal. Thus, we must introduce eight new variables, corresponding to these terminals, and eight productions in which the new variable is replaced by its terminal. Using the obvious initials as the new variables, we introduce:

$$\begin{array}{llll} A \to a & B \to b & Z \to 0 & O \to 1 \\ P \to + & M \to * & L \to (& R \to) \end{array}$$

If we introduce these productions, and replace every terminal in a body that is other than a single terminal by the corresponding variable, we get the grammar shown in Fig. 7.2.

$$\begin{array}{lll} E & \to & EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ T & \to & TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ F & \to & LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ I & \to & a \mid b \mid IA \mid IB \mid IZ \mid IO \\ A & \to & a \\ B & \to & b \\ Z & \to & 0 \\ O & \to & 1 \\ P & \to & + \\ M & \to & * \\ L & \to & (\\ R & \to &) \end{array}$$

Figure 7.2: Making all bodies either a single terminal or several variables

Now, all productions are in Chomsky Normal Form except for those with the bodies of length 3: EPT, TMF, and LER. Some of these bodies appear in more than one production, but we can deal with each body once, introducing one extra variable for each. For EPT, we introduce new variable C_1, and replace the one production, $E \to EPT$, where it appears, by $E \to EC_1$ and $C_1 \to PT$.

For TMF we introduce new variable C_2. The two productions that use this body, $E \to TMF$ and $T \to TMF$, are replaced by $E \to TC_2$, $T \to TC_2$, and $C_2 \to MF$. Then, for LER we introduce new variable C_3 and replace the three productions that use it, $E \to LER$, $T \to LER$, and $F \to LER$, by $E \to LC_3$, $T \to LC_3$, $F \to LC_3$, and $C_3 \to ER$. The final grammar, which is in CNF, is shown in Fig. 7.3. □

$$
\begin{array}{lcl}
E & \to & EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
T & \to & TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
F & \to & LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
I & \to & a \mid b \mid IA \mid IB \mid IZ \mid IO \\
A & \to & a \\
B & \to & b \\
Z & \to & 0 \\
O & \to & 1 \\
P & \to & + \\
M & \to & * \\
L & \to & (\\
R & \to &) \\
C_1 & \to & PT \\
C_2 & \to & MF \\
C_3 & \to & ER
\end{array}
$$

Figure 7.3: Making all bodies either a single terminal or two variables

Theorem 7.16: If G is a CFG whose language contains at least one string other than ϵ, then there is a grammar G_1 in Chomsky Normal Form, such that $L(G_1) = L(G) - \{\epsilon\}$.

PROOF: By Theorem 7.14, we can find CFG G_2 such that $L(G_2) = L(G) - \{\epsilon\}$, and such that G_2 has no useless symbols, ϵ-productions, or unit productions. The construction that converts G_2 to CNF grammar G_1 changes the productions in such a way that each production of G_1 can be simulated by one or more productions of G_2. Conversely, the introduced variables of G_2 each have only one production, so they can only be used in the manner intended. More formally, we prove that w is in $L(G_2)$ if and only if w is in $L(G_1)$.

(Only-if) If w has a derivation in G_2, it is easy to replace each production used, say $A \to X_1X_2\cdots X_k$, by a sequence of productions of G_1. That is, one step in the derivation in G_2 becomes one or more steps in the derivation of w using the productions of G_1. First, if any X_i is a terminal, we know G_1 has a corresponding variable B_i and a production $B_i \to X_i$. Then, if $k > 2$, G_1 has productions $A \to B_1C_1$, $C_1 \to B_2C_2$, and so on, where B_i is either the introduced variable for terminal X_i or X_i itself, if X_i is a variable. These productions simulate in G_1 one step of a derivation of G_2 that uses $A \to X_1X_2\cdots X_k$. We conclude that there is a derivation of w in G_1, so w is in $L(G_1)$.

(If) Suppose w is in $L(G_1)$. Then there is a parse tree in G_1, with S at the root and yield w. We convert this tree to a parse tree of G_2 that also has root S and yield w.

First, we "undo" part (b) of the CNF construction. That is, suppose there is a node labeled A, with two children labeled B_1 and C_1, where C_1 is one of the variables introduced in part (b). Then this portion of the parse tree must look like Fig. 7.4(a). That is, because these introduced variables each have only one production, there is only one way that they can appear, and all the variables introduced to handle the production $A \to B_1B_2\cdots B_k$ must appear together, as shown.

Any such cluster of nodes in the parse tree may be replaced by the production that they represent. The parse-tree transformation is suggested by Fig. 7.4(b).

The resulting parse tree is still not necessarily a parse tree of G_2. The reason is that step (a) in the CNF construction introduced other variables that derive single terminals. However, we can identify these in the current parse tree and replace a node labeled by such a variable A and its one child labeled a, by a single node labeled a. Now, every interior node of the parse tree forms a production of G_2. Since w is the yield of a parse tree in G_2, we conclude that w is in $L(G_2)$. □

7.1.6 Exercises for Section 7.1

Exercise 7.1.1: Find a grammar equivalent to

$$\begin{array}{lcl} S & \to & AB \mid CA \\ A & \to & a \\ B & \to & BC \mid AB \\ C & \to & aB \mid b \end{array}$$

with no useless symbols.

Greibach Normal Form

There is another interesting normal form for grammars that we shall not prove. Every nonempty language without ϵ is $L(G)$ for some grammar G each of whose productions are of the form $A \to a\alpha$, where a is a terminal and α is a string of zero or more variables. Converting a grammar to this form is complex, even if we simplify the task by, say, starting with a Chomsky-Normal-Form grammar. Roughly, we expand the first variable of each production, until we get a terminal. However, because there can be cycles, where we never reach a terminal, it is necessary to "short-circuit" the process, creating a production that introduces a terminal as the first symbol of the body and has variables following it to generate all the sequences of variables that might have been generated on the way to generation of that terminal.

This form, called *Greibach Normal Form*, after Sheila Greibach, who first gave a way to construct such grammars, has several interesting consequences. Since each use of a production introduces exactly one terminal into a sentential form, a string of length n has a derivation of exactly n steps. Also, if we apply the PDA construction of Theorem 6.13 to a Greibach-Normal-Form grammar, then we get a PDA with no ϵ-rules, thus showing that it is always possible to eliminate such transitions of a PDA.

* **Exercise 7.1.2:** Begin with the grammar:

$$\begin{array}{lcl} S & \to & ASB \mid \epsilon \\ A & \to & aAS \mid a \\ B & \to & SbS \mid A \mid bb \end{array}$$

a) Eliminate ϵ-productions.

b) Eliminate any unit productions in the resulting grammar.

c) Eliminate any useless symbols in the resulting grammar.

d) Put the resulting grammar into Chomsky normal form.

Exercise 7.1.3: Repeat Exercise 7.1.2 for the following grammar:

$$\begin{array}{lcl} S & \to & 0A0 \mid 1B1 \mid BB \\ A & \to & C \\ B & \to & S \mid A \\ C & \to & S \mid \epsilon \end{array}$$

Exercise 7.1.4: Repeat Exercise 7.1.2 for the following grammar:

$$\begin{aligned} S &\rightarrow AAA \mid B \\ A &\rightarrow aA \mid B \\ B &\rightarrow \epsilon \end{aligned}$$

Exercise 7.1.5: Repeat Exercise 7.1.2 for the following grammar:

$$\begin{aligned} S &\rightarrow aAa \mid bBb \mid \epsilon \\ A &\rightarrow C \mid a \\ B &\rightarrow C \mid b \\ C &\rightarrow CDE \mid \epsilon \\ D &\rightarrow A \mid B \mid ab \end{aligned}$$

Exercise 7.1.6: Design a CNF grammar for the set of strings of balanced parentheses. You need not start from any particular non-CNF grammar.

!! **Exercise 7.1.7:** Suppose G is a CFG with p productions, and no production body longer than n. Show that if $A \overset{*}{\underset{G}{\Rightarrow}} \epsilon$, then there is a derivation of ϵ from A of no more than $(n^p - 1)/(n - 1)$ steps. How close can you actually come to this bound?

! **Exercise 7.1.8:** Suppose we have a grammar G with n productions, none of them ϵ-productions, and we we convert this grammar to CNF.

a) Show that the CNF grammar has at most $O(n^2)$ productions.

b) Show that it is possible for the CNF grammar to have a number of productions proportional to n^2. *Hint*: Consider the construction that eliminates unit productions.

Exercise 7.1.9: Provide the inductive proofs needed to complete the following theorems:

a) The part of Theorem 7.4 where we show that discovered symbols really are generating.

b) Both directions of Theorem 7.6, where we show the correctness of the algorithm in Section 7.1.2 for detecting the reachable symbols.

c) The part of Theorem 7.11 where we show that all pairs discovered really are unit pairs.

*! **Exercise 7.1.10:** Is it possible to find, for every context-free language without ϵ, a grammar such that all its productions are either of the form $A \rightarrow BCD$ (i.e., a body consisting of three variables), or $A \rightarrow a$ (i.e., a body consisting of a single terminal)? Give either a proof or a counterexample.

Exercise 7.1.11: In this exercise, we shall show that for every context-free language L containing at least one string other than ϵ, there is a CFG in Greibach normal form that generates $L - \{\epsilon\}$. Recall that a Greibach normal form (GNF) grammar is one where every production body starts with a terminal. The construction will be done using a series of lemmas and constructions.

a) Suppose that a CFG G has a production $A \to \alpha B \beta$, and all the productions for B are $B \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$. Then if we replace $A \to \alpha B \beta$ by all the productions we get by substituting some body of a B-production for B, that is, $A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \cdots \mid \alpha\gamma_n\beta$, the resulting grammar generates the same language as G.

In what follows, assume that the grammar G for L is in Chomsky Normal Form, and that the variables are called $A_1, A_2, \ldots, A_k$.

*! b) Show that, by repeatedly using the transformation of part (a), we can convert G to an equivalent grammar in which every production body for A_i either starts with a terminal or starts with A_j, for some $j \geq i$. In either case, all symbols after the first in any production body are variables.

! c) Suppose G_1 is the grammar that we get by performing step (b) on G. Suppose that A_i is any variable, and let $A \to A_i\alpha_1 \mid \cdots \mid A_i\alpha_m$ be all the A_i-productions that have a body beginning with A_i. Let

$$A_i \to \beta_1 \mid \cdots \mid \beta_p$$

be all the other A_i-productions. Note that each β_j must start with either a terminal or a variable with index higher than j. Introduce a new variable B_i, and replace the first group of m productions by

$$\begin{aligned} A_i &\to \beta_1 B_i \mid \cdots \mid \beta_p B_i \\ B_i &\to \alpha_1 B_i \mid \alpha_1 \mid \cdots \mid \alpha_m B_i \mid \alpha_m \end{aligned}$$

Prove that the resulting grammar generates the same language as G and G_1.

*! d) Let G_2 be the grammar that results from step (c). Note that all the A_i productions have bodies that begin with either a terminal or an A_j for $j > i$. Also, all the B_i productions have bodies that begin with either a terminal or some A_j. Prove that G_2 has an equivalent grammar in GNF. *Hint*: First fix the productions for A_k, then A_{k-1}, and so on, down to A_1, using part (a). Then fix the B_i productions in any order, again using part (a).

Exercise 7.1.12: Use the construction of Exercise 7.1.11 to convert the grammar

$$\begin{aligned} S &\to AA \mid 0 \\ A &\to SS \mid 1 \end{aligned}$$

to GNF.

7.2 The Pumping Lemma for Context-Free Languages

Now, we shall develop a tool for showing that certain languages are not context-free. The theorem, called the "pumping lemma for context-free languages," says that in any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings, that we can "pump" in tandem. That is, we may repeat both of the strings i times, for any integer i, and the resulting string will still be in the language.

We may contrast this theorem with the analogous pumping lemma for regular languages, Theorem 4.1, which says we can always find one small string to pump. The difference is seen when we consider a language like $L = \{0^n1^n \mid n \geq 1\}$. We can show it is not regular, by fixing n and pumping a substring of 0's, thus getting a string with more 0's than 1's. However, the CFL pumping lemma states only that we can find two small strings, so we might be forced to use a string of 0's and a string of 1's, thus generating only strings in L when we "pump." That outcome is fortunate, because L is a CFL, and thus we should not be able to use the CFL pumping lemma to construct strings not in L.

7.2.1 The Size of Parse Trees

Our first step in deriving a pumping lemma for CFL's is to examine the shape and size of parse trees. One of the uses of CNF is to turn parse trees into binary trees. These trees have some convenient properties, one of which we exploit here.

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w. If the length of the longest path is n, then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n.

BASIS: $n = 1$. Recall that the length of a path in a tree is the number of edges, i.e., one less than the number of nodes. Thus, a tree with a maximum path length of 1 consists of only a root and one leaf labeled by a terminal. String w is this terminal, so $|w| = 1$. Since $2^{n-1} = 2^0 = 1$ in this case, we have proved the basis.

INDUCTION: Suppose the longest path has length n, and $n > 1$. The root of the tree uses a production, which must be of the form $A \to BC$, since $n > 1$; i.e., we could not start the tree using a production with a terminal. No path in the subtrees rooted at B and C can have length greater than $n - 1$, since these paths exclude the edge from the root to its child labeled B or C. Thus, by the inductive hypothesis, these two subtrees each have yields of length at most 2^{n-2}. The yield of the entire tree is the concatenation of these two yields,

and therefore has length at most $2^{n-2} + 2^{n-2} = 2^{n-1}$. Thus, the inductive step is proved. □

7.2.2 Statement of the Pumping Lemma

The pumping lemma for CFL's is quite similar to the pumping lemma for regular languages, but we break each string z in the CFL L into five parts, and we pump the second and fourth, in tandem.

Theorem 7.18: (The pumping lemma for context-free languages) Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n, then we can write $z = uvwxy$, subject to the following conditions:

1. $|vwx| \leq n$. That is, the middle portion is not too long.
2. $vx \neq \epsilon$. Since v and x are the pieces to be "pumped," this condition says that at least one of the strings we pump must not be empty.
3. For all $i \geq 0$, uv^iwx^iy is in L. That is, the two strings v and x may be "pumped" any number of times, including 0, and the resulting string will still be a member of L.

PROOF: Our first step is to find a Chomsky-Normal-Form grammar G for L. Technically, we cannot find such a grammar if L is the CFL $\emptyset$ or $\{\epsilon\}$. However, if $L = \emptyset$ then the statement of the theorem, which talks about a string z in L surely cannot be violated, since there is no such z in $\emptyset$. Also, the CNF grammar G will actually generate $L - \{\epsilon\}$, but that is again not of importance, since we shall surely pick $n > 0$, in which case z cannot be ϵ anyway.

Now, starting with a CNF grammar $G = (V, T, P, S)$ such that $L(G) = L - \{\epsilon\}$, let G have m variables. Choose $n = 2^m$. Next, suppose that z in L is of length at least n. By Theorem 7.17, any parse tree whose longest path is of length m or less must have a yield of length $2^{m-1} = n/2$ or less. Such a parse tree cannot have yield z, because z is too long. Thus, any parse tree with yield z has a path of length at least $m + 1$.

Figure 7.5 suggests the longest path in the tree for z, where k is at least m and the path is of length $k+1$. Since $k \geq m$, there are at least $m+1$ occurrences of variables $A_0, A_1, \ldots, A_k$ on the path. As there are only m different variables in V, at least two of the last $m + 1$ variables on the path (that is, A_{k-m} through A_k, inclusive) must be the same variable. Suppose $A_i = A_j$, where $k - m \leq i < j \leq k$.

Then it is possible to divide the tree as shown in Fig. 7.6. String w is the yield of the subtree rooted at A_j. Strings v and x are the strings to the left and right, respectively, of w in the yield of the larger subtree rooted at A_i. Note that, since there are no unit productions, v and x could not both be ϵ, although one could be. Finally, u and y are those portions of z that are to the left and right, respectively, of the subtree rooted at A_i.

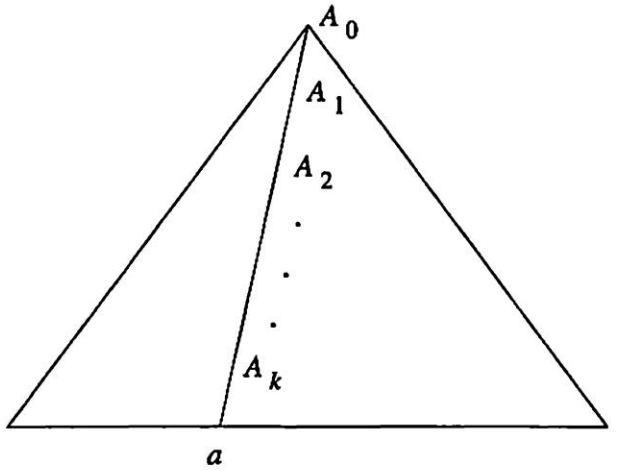

Figure 7.5: Every sufficiently long string in L must have a long path in its parse tree

If $A_i = A_j = A$, then we can construct new parse trees from the original tree, as suggested in Fig. 7.7(a). First, we may replace the subtree rooted at A_i, which has yield vwx, by the subtree rooted at A_j, which has yield w. The reason we can do so is that both of these trees have root labeled A. The resulting tree is suggested in Fig. 7.7(b); it has yield uwy and corresponds to the case $i = 0$ in the pattern of strings uv^iwx^iy.

Another option is suggested by Fig. 7.7(c). There, we have replaced the subtree rooted at A_j by the entire subtree rooted at A_i. Again, the justification is that we are substituting one tree with root labeled A for another tree with the same root label. The yield of this tree is uv^2wx^2y. Were we to then replace the subtree of Fig. 7.7(c) with yield w by the larger subtree with yield vwx, we would have a tree with yield uv^3wx^3y, and so on, for any exponent i. Thus, there are parse trees in G for all strings of the form uv^iwx^iy, and we have almost proved the pumping lemma.

The remaining detail is condition (1), which says that $|vwx| \leq n$. However, we picked A_i to be close to the bottom of the tree; that is, $k - i \leq m$. Thus, the longest path in the subtree rooted at A_i is no greater than $m + 1$. By Theorem 7.17, the subtree rooted at A_i has a yield whose length is no greater than $2^m = n$. □

7.2.3 Applications of the Pumping Lemma for CFL's

Notice that, like the earlier pumping lemma for regular languages, we use the CFl pumping lemma as an "adversary game, as follows."

1. We pick a language L that we want to show is not a CFL.

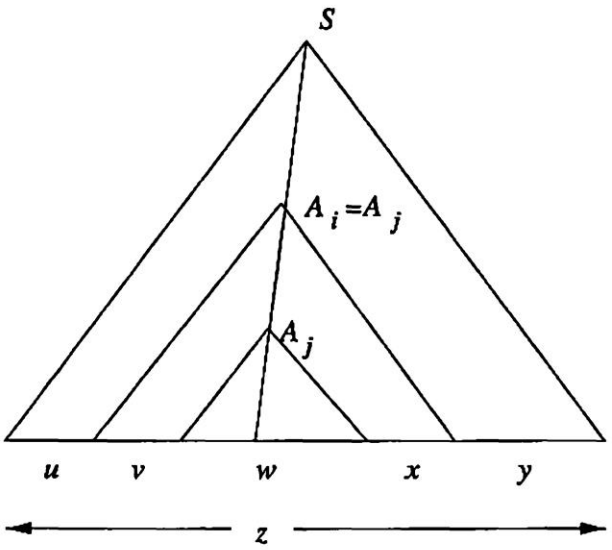

Figure 7.6: Dividing the string w so it can be pumped

2. Our "adversary" gets to pick n, which we do not know, and we therefore must plan for any possible n.

3. We get to pick z, and may use n as a parameter when we do so.

4. Our adversary gets to break z into $uvwxy$, subject only to the constraints that $|vwx| \leq n$ and $vx \neq \epsilon$.

5. We "win" the game, if we can, by picking i and showing that uv^iwx^iy is not in L.

We shall now see some examples of languages that we can prove, using the pumping lemma, not to be context-free. Our first example shows that, while context-free languages can match two groups of symbols for equality or inequality, they cannot match three such groups.

Example 7.19: Let L be the language $\{0^n1^n2^n \mid n \geq 1\}$. That is, L consists of all strings in $0^+1^+2^+$ with an equal number of each symbol, e.g., 012, 001122, and so on. Suppose L were context-free. Then there is an integer n given to us by the pumping lemma.[2] Let us pick $z = 0^n1^n2^n$.

Suppose the "adversary" breaks z as $z = uvwxy$, where $|vwx| \leq n$ and v and x are not both ϵ. Then we know that vwx cannot involve both 0's and 2's, since the last 0 and the first 2 are separated by $n + 1$ positions. We shall prove that L contains some string known not to be in L, thus contradicting the assumption that L is a CFL. The cases are as follows:

[2]Remember that this n is the constant provided by the pumping lemma, and it has nothing to do with the local variable n used in the definition of L itself.

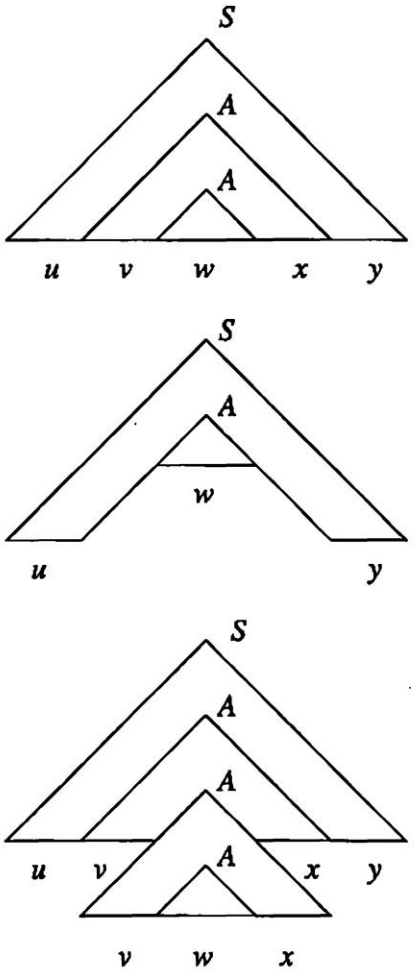

Figure 7.7: Pumping strings v and x zero times and pumping them twice

1. vwx has no 2's. Then vx consists of only 0's and 1's, and has at least one of these symbols. Then uwy, which would have to be in L by the pumping lemma, has n 2's, but has fewer than n 0's or fewer than n 1's, or both. It therefore does not belong in L, and we conclude L is not a CFL in this case.

2. vwx has no 0's. Similarly, uwy has n 0's, but fewer 1's or fewer 2's. It therefore is not in L.

Whichever case holds, we conclude that L has a string we know not to be in L. This contradiction allows us to conclude that our assumption was wrong; L is *not* a CFL. □

Another thing that CFL's cannot do is match two pairs of equal numbers of symbols, provided that the pairs interleave. The idea is made precise in the following example of a proof of non-context-freeness using the pumping lemma.

Example 7.20: Let L be the language $\{0^i1^j2^i3^j \mid i \geq 1 \text{ and } j \geq 1\}$. If L is context-free, let n be the constant for L, and pick $z = 0^n1^n2^n3^n$. We may write $z = uvwxy$ subject to the usual constraints $|vwx| \leq n$ and $vx \neq \epsilon$. Then vwx is either contained in the substring of one symbol, or it straddles two adjacent symbols.

If vwx consists of only one symbol, then uwy has n of three different symbols and fewer than n of the fourth symbol. Thus, it cannot be in L. If vwx straddles two symbols, say the 1's and 2's, then uwy is missing either some 1's or some 2's, or both. Suppose it is missing 1's. As there are n 3's, this string cannot be in L. Similarly, if it is missing 2's, then as it has n 0's, uwy cannot be in L. We have contradicted the assumption that L is a CFL and conclude that it is not. □

As a final example, we shall show that CFL's cannot match two strings of arbitrary length, if the strings are chosen from an alphabet of more than one symbol. An implication of this observation, incidentally, is that grammars are not a suitable mechanism for enforcing certain "semantic" constraints in programming languages, such as the common requirement that an identifier be declared before use. In practice, another mechanism, such as a "symbol table" is used to record declared identifiers, and we do not try to design a parser that, by itself, checks for "definition prior to use."

Example 7.21: Let $L = \{ww \mid w \text{ is in } \{0,1\}^*\}$. That is, L consists of repeating strings, such as ϵ, 0101, 00100010, or 110110. If L is context-free, then let n be its pumping-lemma constant. Consider the string $z = 0^n1^n0^n1^n$. This string is 0^n1^n repeated, so z is in L.

Following the pattern of the previous examples, we can break $z = uvwxy$, such that $|vwx| \leq n$ and $vx \neq \epsilon$. We shall show that uwy is not in L, and thus show L not to be a context-free language, by contradiction.

First, observe that, since $|vwx| \leq n$, $|uwy| \geq 3n$. Thus, if uwy is some repeating string, say tt, then t is of length at least $3n/2$. There are several cases to consider, depending where vwx is within z.

1. Suppose vwx is within the first n 0's. In particular, let vx consist of k 0's, where $k > 0$. Then uwy begins with $0^{n-k}1^n$. Since $|uwy| = 4n - k$, we know that if $uwy = tt$, then $|t| = 2n - k/2$. Thus, t does not end until after the first block of 1's; i.e., t ends in 0. But uwy ends in 1, and so it cannot equal tt.

2. Suppose vwx straddles the first block of 0's and the first block of 1's. It may be that vx consists only of 0's, if $x = \epsilon$. Then, the argument that uwy is not of the form tt is the same as case (1). If vx has at least one

1, then we note that t, which is of length at least $3n/2$, must end in 1^n, because uwy ends in 1^n. However, there is no block of n 1's except the final block, so t cannot repeat in uwy.

3. If vwx is contained in the first block of 1's, then the argument that uwy is not in L is like the second part of case (2).

4. Suppose vwx straddles the first block of 1's and the second block of 0's. If vx actually has no 0's, then the argument is the same as if vwx were contained in the first block of 1's. If vx has at least one 0, then uwy starts with a block of n 0's, and so does t if $uwy = tt$. However, there is no other block of n 0's in uwy for the second copy of t. We conclude in this case too, that uwy is not in L.

5. In the other cases, where vwx is in the second half of z, the argument is symmetric to the cases where vwx is contained in the first half of z.

Thus, in no case is uwy in L, and we conclude that L is not context-free. □

7.2.4 Exercises for Section 7.2

Exercise 7.2.1: Use the CFL pumping lemma to show each of these languages not to be context-free:

* a) $\{a^i b^j c^k \mid i < j < k\}$.

b) $\{a^n b^n c^i \mid i \leq n\}$.

c) $\{0^p \mid p \text{ is a prime}\}$. *Hint*: Adapt the same ideas used in Example 4.3, which showed this language not to be regular.

*! d) $\{0^i 1^j \mid j = i^2\}$.

! e) $\{a^n b^n c^i \mid n \leq i \leq 2n\}$.

! f) $\{ww^R w \mid w \text{ is a string of 0's and 1's}\}$. That is, the set of strings consisting of some string w followed by the same string in reverse, and then the string w again, such as 001100001.

! **Exercise 7.2.2:** When we try to apply the pumping lemma to a CFL, the "adversary wins," and we cannot complete the proof. Show what goes wrong when we choose L to be one of the following languages:

a) $\{00, 11\}$.

* b) $\{0^n 1^n \mid n \geq 1\}$.

* c) The set of palindromes over alphabet $\{0, 1\}$.

! **Exercise 7.2.3:** There is a stronger version of the CFL pumping lemma known as *Ogden's lemma*. It differs from the pumping lemma we proved by allowing us to focus on any n "distinguished" positions of a string z and guaranteeing that the strings to be pumped have between 1 and n distinguished positions. The advantage of this ability is that a language may have strings consisting of two parts, one of which can be pumped without producing strings not in the language, while the other *does* produce strings outside the language when pumped. Without being able to insist that the pumping take place in the latter part, we cannot complete a proof of non-context-freeness. The formal statement of Ogden's lemma is: If L is a CFL, then there is a constant n, such that if z is any string of length at least n in L, in which we select at least n positions to be *distinguished*, then we can write $z = uvwxy$, such that:

1. vwx has at most n distinguished positions.
2. *vx has at least one distinguished position.*
3. For all i, uv^iwx^iy is in L.

Prove Ogden's lemma. *Hint*: The proof is really the same as that of the pumping lemma of Theorem 7.18 if we pretend that the nondistinguished positions of z are not present as we select a long path in the parse tree for z.

* **Exercise 7.2.4:** Use Ogden's lemma (Exercise 7.2.3) to simplify the proof in Example 7.21 that $L = \{ww \mid w \text{ is in } \{0,1\}^*\}$ is not a CFL. *Hint*: With $z = 0^n1^n0^n1^n$, make the two middle blocks distinguished.

Exercise 7.2.5: Use Ogden's lemma (Exercise 7.2.3) to show the following languages are not CFL's:

! a) $\{0^i1^j0^k \mid j = \max(i,k)\}$.

!! b) $\{a^nb^nc^i \mid i \neq n\}$. *Hint*: If n is the constant for Ogden's lemma, consider the string $z = a^nb^nc^{n!}$.

7.3 Closure Properties of Context-Free Languages

We shall now consider some of the operations on context-free languages that are guaranteed to produce a CFL. Many of these closure properties will parallel the theorems we had for regular languages in Section 4.2. However, there are some differences.

First, we introduce an operation called substitution, in which we replace each symbol in the strings of one language by an entire language. This operation, a generalization of the homomorphism that we studied in Section 4.2.3, is useful in *proving some other closure properties of CFL's*, such as the regular-expression operations: union, concatenation, and closure. We show that CFL's are closed

under homomorphisms and inverse homomorphisms. Unlike the regular languages, the CFL's are not closed under intersection or difference. However, the intersection or difference of a CFL and a regular language is always a CFL.

7.3.1 Substitutions

Let Σ be an alphabet, and suppose that for every symbol a in Σ, we choose a language L_a. These chosen languages can be over any alphabets, not necessarily Σ and not necessarily the same. This choice of languages defines a function s (a *substitution*) on Σ, and we shall refer to L_a as $s(a)$ for each symbol a.

If $w = a_1a_2\cdots a_n$ is a string in Σ^*, then $s(w)$ is the language of all strings $x_1x_2\cdots x_n$ such that string x_i is in the language $s(a_i)$, for $i = 1, 2, \ldots, n$. Put another way, $s(w)$ is the concatenation of the languages $s(a_1)s(a_2)\cdots s(a_n)$. We can further extend the definition of s to apply to languages: $s(L)$ is the union of $s(w)$ for all strings w in L.

Example 7.22: Suppose $s(0) = \{a^nb^n \mid n \geq 1\}$ and $s(1) = \{aa, bb\}$. That is, s is a substitution on alphabet $\Sigma = \{0, 1\}$. Language $s(0)$ is the set of strings with one or more a's followed by an equal number of b's, while $s(1)$ is the finite language consisting of the two strings aa and bb.

Let $w = 01$. Then $s(w)$ is the concatenation of the languages $s(0)s(1)$. To be exact, $s(w)$ consists of all strings of the forms a^nb^naa and a^nb^{n+2}, where $n \geq 1$.

Now, suppose $L = L(0^*)$, that is, the set of all strings of 0's. Then $s(L) = (s(0))^*$. This language is the set of all strings of the form

$$a^{n_1}b^{n_1}a^{n_2}b^{n_2}\cdots a^{n_k}b^{n_k}$$

for some $k \geq 0$ and any sequence of choices of positive integers $n_1, n_2, \ldots, n_k$. It includes strings such as ϵ, *aabbaaabbb*, and *abaabbabab*. □

Theorem 7.23: If L is a context-free language over alphabet Σ, and s is a substitution on Σ such that $s(a)$ is a CFL for each a in Σ, then $s(L)$ is a CFL.

PROOF: The essential idea is that we may take a CFG for L and replace each terminal a by the start symbol of a CFG for language $s(a)$. The result is a single CFG that generates $s(L)$. However, there are a few details that must be gotten right to make this idea work.

More formally, start with grammars for each of the relevant languages, say $G = (V, \Sigma, P, S)$ for L and $G_a = (V_a, T_a, P_a, S_a)$ for each a in Σ. Since we can choose any names we wish for variables, let us make sure that the sets of variables are disjoint; that is, there is no symbol A that is in two or more of V and any of the V_a's. The purpose of this choice of names is to make sure that when we combine the productions of the various grammars into one set of productions, we cannot get accidental mixing of the productions from two grammars and thus have derivations that do not resemble the derivations in any of the given grammars.

We construct a new grammar $G' = (V', T', P', S)$ for $s(L)$, as follows:

- V' is the union of V and all the V_a's for a in Σ.
- T' is the union of all the T_a's for a in Σ.
- P' consists of:
  1. All productions in any P_a, for a in Σ.
  2. The productions of P, but with each terminal a in their bodies replaced by S_a everywhere a occurs.

Thus, all parse trees in grammar G' start out like parse trees in G, but instead of generating a yield in Σ^*, there is a frontier in the tree where all nodes have labels that are S_a for some a in Σ. Then, dangling from each such node is a parse tree of G_a, whose yield is a terminal string that is in the language $s(a)$. The typical parse tree is suggested in Fig. 7.8.

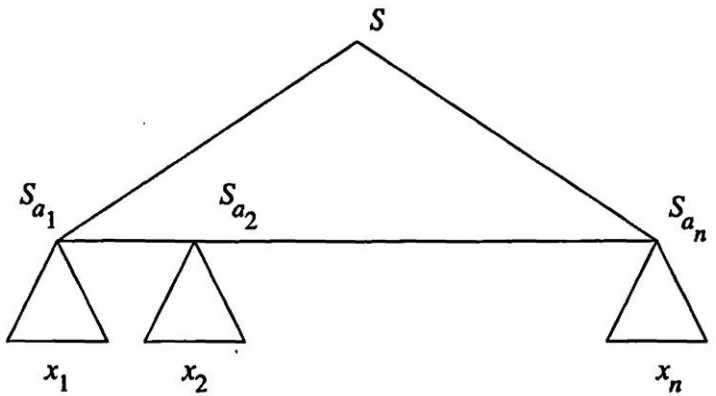

Figure 7.8: A parse tree in G' begins with a parse tree in G and finishes with many parse trees, each in one of the grammars G_a

Now, we must prove that this construction works, in the sense that G' generates the language $s(L)$. Formally:

- A string w is in $L(G')$ if and only if w is in $s(L)$.

(If) Suppose w is in $s(L)$. Then there is some string $x = a_1a_2\cdots a_n$ in L, and strings x_i in $s(a_i)$ for $i = 1, 2, \ldots, n$, such that $w = x_1x_2\cdots x_n$. Then the portion of G' that comes from the productions of G with S_a substituted for each a will generate a string that looks like x, but with S_a in place of each a. This string is $S_{a_1}S_{a_2}\cdots S_{a_n}$. This part of the derivation of w is suggested by the upper triangle in Fig. 7.8.

Since the productions of each G_a are also productions of G', the derivation of x_i from S_{a_i} is also a derivation in G'. The parse trees for these derivations are suggested by the lower triangles in Fig. 7.8. Since the yield of this parse tree of G' is $x_1x_2\cdots x_n = w$, we conclude that w is in $L(G')$.

(Only-if) Now suppose w is in $L(G')$. We claim that the parse tree for w must look like the tree of Fig. 7.8. The reason is that the variables of each of the grammars G and G_a for a in Σ are disjoint. Thus, the top of the tree, starting from variable S, must use only productions of G until some symbol S_a is derived, and below that S_a only productions of grammar G_a may be used. As a result, whenever w has a parse tree T, we can identify a string $a_1a_2\cdots a_n$ in $L(G)$, and strings x_i in language $s(a_i)$, such that

1. $w = x_1x_2\cdots x_n$, and
2. The string $S_{a_1}S_{a_2}\cdots S_{a_n}$ is the yield of a tree that is formed from T by deleting some subtrees (as suggested by Fig. 7.8).

But the string $x_1x_2\cdots x_n$ is in $s(L)$, since it is formed by substituting strings x_i for each of the a_i's. Thus, we conclude w is in $s(L)$. □

7.3.2 Applications of the Substitution Theorem

There are several familiar closure properties, which we studied for regular languages, that we can show for CFL's using Theorem 7.23. We shall list them all in one theorem.

Theorem 7.24: The context-free languages are closed under the following operations:

1. Union.
2. Concatenation.
3. Closure (*), and positive closure ($^+$).
4. Homomorphism.

PROOF: Each requires only that we set up the proper substitution. The proofs below each involve substitution of context-free languages into other context-free languages, and therefore produce CFL's by Theorem 7.23.

1. *Union*: Let L_1 and L_2 be CFL's. Then $L_1 \cup L_2$ is the language $s(L)$, where L is the language $\{1,2\}$, and s is the substitution defined by $s(1) = L_1$ and $s(2) = L_2$.
2. *Concatenation*: Again let L_1 and L_2 be CFL's. Then L_1L_2 is the language $s(L)$, where L is the language $\{12\}$, and s is the same substitution as in case (1).
3. *Closure and positive closure*: If L_1 is a CFL, L is the language $\{1\}^*$, and s is the substitution $s(1) = L_1$, then $L_1^* = s(L)$. Similarly, if L is instead the language $\{1\}^+$, then $L_1^+ = s(L)$.

4. Suppose L is a CFL over alphabet Σ, and h is a homomorphism on Σ. Let s be the substitution that replaces each symbol a in Σ by the language consisting of the one string that is $h(a)$. That is, $s(a) = \{h(a)\}$, for all a in Σ. Then $h(L) = s(L)$.

□

7.3.3 Reversal

The CFL's are also closed under reversal. We cannot use the substitution theorem, but there is a simple construction using grammars.

Theorem 7.25: If L is a CFL, then so is L^R.

PROOF: Let $L = L(G)$ for some CFL $G = (V, T, P, S)$. Construct $G^R = (V, T, P^R, S)$, where P^R is the "reverse" of each production in P. That is, if $A \to \alpha$ is a production of G, then $A \to \alpha^R$ is a production of G^R. It is an easy induction on the lengths of derivations in G and G^R to show that $L(G^R) = L^R$. Essentially, all the sentential forms of G^R are reverses of sentential forms of G, and vice-versa. We leave the formal proof as an exercise. □

7.3.4 Intersection With a Regular Language

The CFL's are not closed under intersection. Here is a simple example that proves they are not.

Example 7.26: We learned in Example 7.19 that the language

$$L = \{0^n 1^n 2^n \mid n \geq 1\}$$

is not a context-free language. However, the following two languages *are* context-free:

$$L_1 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$$
$$L_2 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$$

A grammar for L_1 is:

$$S \to AB$$
$$A \to 0A1 \mid 01$$
$$B \to 2B \mid 2$$

In this grammar, A generates all strings of the form $0^n 1^n$, and B generates all strings of 2's. A grammar for L_2 is:

$$S \to AB$$
$$A \to 0A \mid 0$$
$$B \to 1B2 \mid 12$$

It works similarly, but with A generating any string of 0's, and B generating matching strings of 1's and 2's.

However, $L = L_1 \cap L_2$. To see why, observe that L_1 requires that there be the same number of 0's and 1's, while L_2 requires the numbers of 1's and 2's to be equal. A string in both languages must have equal numbers of all three symbols and thus be in L.

If the CFL's were closed under intersection, then we could prove the false statement that L is context-free. We conclude by contradiction that the CFL's are not closed under intersection. □

On the other hand, there is a weaker claim we can make about intersection. The context-free languages are closed under the operation of "intersection with a regular language." The formal statement and proof is in the next theorem.

Theorem 7.27: If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

PROOF: This proof requires the pushdown-automaton representation of CFL's, as well as the finite-automaton representation of regular languages, and generalizes the proof of Theorem 4.8, where we ran two finite automata "in parallel" to get the intersection of their languages. Here, we run a finite automaton "in parallel" with a PDA, and the result is another PDA, as suggested in Fig. 7.9.

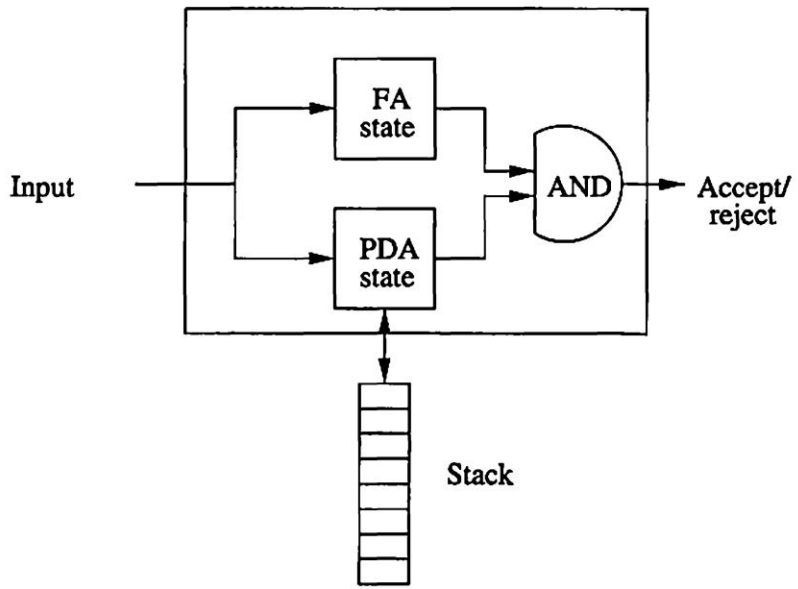

Figure 7.9: A PDA and a FA can run in parallel to create a new PDA

Formally, let

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

be a PDA that accepts L by final state, and let

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

be a DFA for R. Construct PDA

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$$

where $\delta\big((q,p),a,X\big)$ is defined to be the set of all pairs $\big((r,s),\gamma\big)$ such that:

1. $s = \hat{\delta}_A(p,a)$, and
2. Pair (r,γ) is in $\delta_P(q,a,X)$.

That is, for each move of PDA P, we can make the same move in PDA P', and in addition, we carry along the state of the DFA A in a second component of the state of P'. Note that a may be a symbol of Σ, or $a = \epsilon$. In the former case, $\hat{\delta}(p,a) = \delta_A(p)$, while if $a = \epsilon$, then $\hat{\delta}(p,a) = p$; i.e., A does not change state while P makes moves on ϵ input.

It is an easy induction on the numbers of moves made by the PDA's that $(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \gamma)$ if and only if $\big((q_P, q_A), w, Z_0\big) \overset{*}{\underset{P'}{\vdash}} \big((q,p), \epsilon, \gamma\big)$, where $p = \hat{\delta}(p_A, w)$. We leave these inductions as exercises. Since (q,p) is an accepting state of P' if and only if q is an accepting state of P, and p is an accepting state of A, we conclude that P' accepts w if and only if both P and A do; i.e., w is in $L \cap R$. □

Example 7.28: In Fig. 6.6 we designed a PDA called F to accept by final state the set of strings of i's and e's that represent minimal violations of the rule regarding how if's and else's may appear in C programs. Call this language L. The PDA F was defined by

$$P_F = (\{p,q,r\}, \{i,e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

where δ_F consists of the rules:

1. $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$.
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$.
3. $\delta_F(q, e, Z) = \{(q, \epsilon)\}$.
4. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon, \epsilon)\}$.

Now, let us introduce a finite automaton

$$A = (\{s,t\}, \{i,e\}, \delta_A, s, \{s,t\})$$

that accepts the strings in the language of $\mathbf{i^*e^*}$, that is, all strings of i's followed by e's. Call this language R. Transition function δ_A is given by the rules:

a) $\delta_A(s,i) = s$.

b) $\delta_A(s,e) = t$.

c) $\delta_A(t,e) = t$.

Strictly speaking, A is not a DFA, as assumed in Theorem 7.27, because it is missing a dead state for the case that we see input i when in state t. However, the same construction works even for an NFA, since the PDA that we construct is allowed to be nondeterministic. In this case, the constructed PDA is actually deterministic, although it will "die" on certain sequences of input.

We shall construct a PDA

$$P = (\{p,q,r\} \times \{s,t\}, \{i,e\}, \{Z,X_0\}, \delta, (p,s), X_0, \{r\} \times \{s,t\})$$

The transitions of δ are listed below and indexed by the rule of PDA F (a number from 1 to 4) and the rule of DFA A (a letter a, b, or c) that gives rise to the rule. In the case that the PDA F makes an ϵ-transition, there is no rule of A used. Note that we construct these rules in a "lazy" way, starting with the state of P that is the start states of F and A, and constructing rules for other states only if we discover that P can enter that pair of states.

1: $\delta\big((p,s),\epsilon,X_0\big) = \{\big((q,s),ZX_0\big)\}$.

2a: $\delta\big((q,s),i,Z\big) = \{\big((q,s),ZZ\big)\}$.

3b: $\delta\big((q,s),e,Z\big) = \{\big((q,t),\epsilon\big)\}$.

4: $\delta\big((q,s),\epsilon,X_0\big) = \{\big((r,s),\epsilon\big)\}$. Note: one can prove that this rule is never exercised. The reason is that it is impossible to pop the stack without seeing an e, and as soon as P sees an e the second component of its state becomes t.

3c: $\delta\big((q,t),e,Z\big) = \{\big((q,t),\epsilon\big)\}$.

4: $\delta\big((q,t),\epsilon,X_0\big) = \{\big((r,t),\epsilon\big)\}$.

The language $L \cap R$ is the set of strings with some number of i's followed by one more e, that is, $\{i^n e^{n+1} \mid n \geq 0\}$. This set is exactly those if-else violations that consist of a block of if's followed by a block of else's. The language is evidently a CFL, generated by the grammar with productions $S \to iSe \mid e$.

Note that the PDA P accepts this language $L \cap R$. After pushing Z onto the stack, it pushes more Z's onto the stack in response to inputs i, staying in state (q,s). As soon as it sees and e, it goes to state (q,t) and starts popping the stack. It dies if it sees an i until X_0 is exposed on the stack. At that point, it spontaneously transitions to state (r,t) and accepts. □

Since we know that the CFL's are not closed under intersection, but are closed under intersection with a regular language, we also know about the set-difference and complementation operations on CFL's. We summarize these properties in one theorem.

Theorem 7.29: The following are true about a CFL's L, L_1, and L_2, and a regular language R.

1. $L - R$ is a context-free language.
2. $\overline{L}$ is not necessarily a context-free language.
3. $L_1 - L_2$ is not necessarily context-free.

PROOF: For (1), note that $L - R = L \cap \overline{R}$. If R is regular, so is $\overline{R}$ regular by Theorem 4.5. Then $L - R$ is a CFL by Theorem 7.27.

For (2), suppose that $\overline{L}$ is always context-free when L is. Then since

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

and the CFL's are closed under union, it would follow that the CFL's are closed under intersection. However, we know they are not from Example 7.26.

Lastly, let us prove (3). We know Σ^* is a CFL for every alphabet Σ; designing a grammar or PDA for this regular language is easy. Thus, if $L_1 - L_2$ were always a CFL when L_1 and L_2 are, it would follow that $\Sigma^* - L$ was always a CFL when L is. However, $\Sigma^* - L$ is $\overline{L}$ when we pick the proper alphabet Σ. Thus, we would contradict (2) and we have proved by contradiction that $L_1 - L_2$ is not necessarily a CFL. □

7.3.5 Inverse Homomorphism

Let us review from Section 4.2.4 the operation called "inverse homomorphism." If h is a homomorphism, and L is any language, then $h^{-1}(L)$ is the set of strings w such that $h(w)$ is in L. The proof that regular languages are closed under inverse homomorphism was suggested in Fig. 4.6. There, we showed how to design a finite automaton that processes its input symbols a by applying a homomorphism h to it, and simulating another finite automaton on the sequence of inputs $h(a)$.

We can prove this closure property of CFL's in much the same way, by using PDA's instead of finite automata. However, there is one problem that we face with PDA's that did not arise when we were dealing with finite automata. The action of a finite automaton on a sequence of inputs is a state transition, and thus looks, as far as the constructed automaton is concerned, just like a move that a finite automaton might make on a single input symbol.

When the automaton is a PDA, in contrast, a sequence of moves might not look like a move on one input symbol. In particular, in n moves, the PDA can pop n symbols off its stack, while one move can only pop one symbol. Thus,

the construction for PDA's that is analogous to Fig. 4.6 is somewhat more complex; it is sketched in Fig. 7.10. The key additional idea is that after input a is read, $h(a)$ is placed in a "buffer." The symbols of $h(a)$ are used one at a time, and fed to the PDA being simulated. Only when the buffer is empty does the constructed PDA read another of its input symbols and apply the homomorphism to it. We shall formalize this construction in the next theorem.

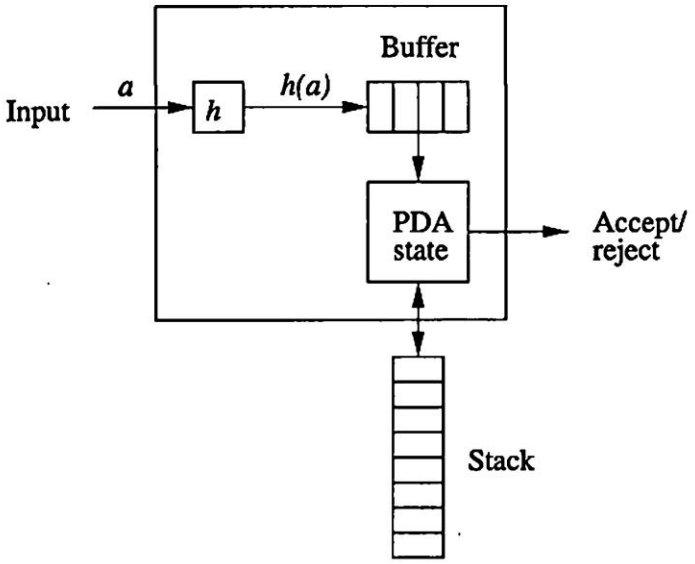

Figure 7.10: Constructing a PDA to accept the inverse homomorphism of what a given PDA accepts

Theorem 7.30: Let L be a CFL and h a homomorphism. Then $h^{-1}(L)$ is a CFL.

PROOF: Suppose h applies to symbols of alphabet Σ and produces strings in T^*. We also assume that L is a language over alphabet T. As suggested above, we start with a PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ that accepts L by final state. We construct a new PDA

$$P' = (Q', \Sigma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\}) \tag{7.1}$$

where:

1. Q' is the set of pairs (q, x) such that:
   (a) q is a state in Q, and
   (b) x is a suffix (not necessarily proper) of some string $h(a)$ for some input symbol a in Σ.

That is, the first component of the state of P' is the state of P, and the second component is the buffer. We assume that the buffer will periodically be loaded with a string $h(a)$, and then allowed to shrink from the front, as we use its symbols to feed the simulated PDA P. Note that since Σ is finite, and $h(a)$ is finite for all a, there are only a finite number of states for P'.

2. δ' is defined by the following rules:

   (a) $\delta'\big((q,\epsilon),a,X\big) = \{\big((q,h(a)\big),X\big)\}$ for all symbols a in Σ, all states q in Q, and stack symbols X in Γ. Note that a cannot be ϵ here. When the buffer is empty, P' can consume its next input symbol a and place $h(a)$ in the buffer.

   (b) If $\delta(q,b,X)$ contains (p,γ), where b is in T or $b=\epsilon$, then

   $$\delta'\big((q,bx),\epsilon,X\big)$$

   contains $\big((p,x),\gamma\big)$. That is, P' always has the option of simulating a move of P, using the front of its buffer. If b is a symbol in T, then the buffer must not be empty, but if $b=\epsilon$, then the buffer can be empty.

3. Note that, as defined in (7.1), the start state of P' is (q_0,ϵ); i.e., P' starts in the start state of P with an empty buffer.

4. Likewise, the accepting states of P', as per (7.1), are those states (q,ϵ) such that q is an accepting state of P.

The following statement characterizes the relationship between P' and P:

- $\big(q_0,h(w),Z_0\big) \overset{*}{\underset{P}{\vdash}} (p,\epsilon,\gamma)$ if and only if $\big((q_0,\epsilon),w,Z_0\big) \overset{*}{\underset{P'}{\vdash}} \big((p,\epsilon),\epsilon,\gamma\big)$.

The proofs in both directions are inductions on the number of moves made by the two automata. In the "if" portion, one needs to observe that once the buffer of P' is nonempty, it cannot read another input symbol and must simulate P, until the buffer has become empty (although when the buffer is empty, it may still simulate P). We leave further details as an exercise.

Once we accept this relationship between P' and P, we note that P accepts $h(w)$ if and only if P' accepts w, because of the way the accepting states of P' are defined. Thus, $L(P') = h^{-1}\big(L(P)\big)$. □

7.3.6 Exercises for Section 7.3

Exercise 7.3.1: Show that the CFL's are closed under the following operations:

* a) *init*, defined in Exercise 4.2.6(c). *Hint*: Start with a CNF grammar for the language L.

*!! b) The operation L/a, defined in Exercise 4.2.2. *Hint*: Again, start with a CNF grammar for L.

!! c) *cycle*, defined in Exercise 4.2.11. *Hint*: Try a PDA-based construction.

Exercise 7.3.2: Consider the following two languages:

$$L_1 = \{a^n b^{2n} c^m \mid n, m \geq 0\}$$
$$L_2 = \{a^n b^m c^{2m} \mid n, m \geq 0\}$$

a) Show that each of these languages is context-free by giving grammars for each.

! b) Is $L_1 \cap L_2$ a CFL? Justify your answer.

!! **Exercise 7.3.3:** Show that the CFL's are *not* closed under the following operations:

* a) *min*, as defined in Exercise 4.2.6(a).

b) *max*, as defined in Exercise 4.2.6(b).

c) *half*, as defined in Exercise 4.2.8.

d) *alt*, as defined in Exercise 4.2.7.

Exercise 7.3.4: The *shuffle* of two strings w and x is the set of all strings that one can get by interleaving the positions of w and x in any way. More precisely, *shuffle*(w, x) is the set of strings z such that

1. Each position of z can be assigned to w or x, but not both.
2. The positions of z assigned to w form w when read from left to right.
3. The positions of z assigned to x form x when read from left to right.

For example, if $w = 01$ and $x = 110$, then *shuffle*$(01, 110)$ is the set of strings $\{01110, 01101, 10110, 10101, 11010, 11001\}$. To illustrate the necessary reasoning, the third string, 10110, is justified by assigning the second and fifth positions to 01 and positions one, three, and four to 110. The first string, 01110 has three justifications. Assign the first position and either the second, third, or fourth to 01, and the other three to 110. We can also define the shuffle of languages, *shuffle*(L_1, L_2) to the the union over all pairs of strings, w from L_1 and x from L_2, of *shuffle*(w, x).

a) What is *shuffle*$(00, 111)$?

* b) What is *shuffle*(L_1, L_2) if $L_1 = L(\mathbf{0}^*)$ and $L_2 = \{0^n 1^n \mid n \geq 0\}$.

*!! c) Show that if L_1 and L_2 are both regular languages, then so is

$$shuffle(L_1, L_2)$$

Hint: Start with DFA's for L_1 and L_2.

! d) Show that if L is a CFL and R is a regular language, then $shuffle(L, R)$ is a CFL. *Hint*: start with a PDA for L and a DFA for R.

!! e) Give a counterexample to show that if L_1 and L_2 are both CFL's, then $shuffle(L_1, L_2)$ need not be a CFL.

*!! **Exercise 7.3.5:** A string y is said to be a *permutation* of the string x if the symbols of y can be reordered to make x. For instance, the permutations of string $x = 011$ are 110, 101, and 011. If L is a language, then $perm(L)$ is the set of strings that are permutations of strings in L. For example, if $L = \{0^n1^n \mid n \geq 0\}$, then $perm(L)$ is the set of strings with equal numbers of 0's and 1's.

a) Give an example of a regular language L over alphabet $\{0, 1\}$ such that $perm(L)$ is not regular. Justify your answer. *Hint*: Try to find a regular language whose permutations are all strings with an equal number of 0's and 1's.

b) Give an example of a regular language L over alphabet $\{0, 1, 2\}$ such that $perm(L)$ is not context-free.

c) Prove that for every regular language L over a two-symbol alphabet, $perm(L)$ is context-free.

Exercise 7.3.6: Give the formal proof of Theorem 7.25: that the CFL's are closed under reversal.

Exercise 7.3.7: Complete the proof of Theorem 7.27 by showing that

$$(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \gamma)$$

if and only if $((q_P, q_A), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((q, p), \epsilon, \gamma)$ and $p = \hat{\delta}(p_A, w)$.

7.4 Decision Properties of CFL's

Now, let us consider what kinds of questions we can answer about context-free languages. In analogy with Section 4.3 about decision properties of the regular languages, our starting point for a question is always some representation of a CFL — either a grammar or a PDA. Since we know from Section 6.3 that we can convert between grammars and PDA's, we may assume we are given either representation of a CFL, whichever is more convenient.

We shall discover that very little can be decided about a CFL; the major tests we are able to make are whether the language is empty and whether a given