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) $\{a^ib^jc^k \mid i < j < k\}$.

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

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) $\{0^i1^j \mid j = i^2\}$.

! e) $\{a^nb^nc^i \mid n \leq i \leq 2n\}$.

! f) $\{ww^Rw \mid w \text{ is a string of 0's and 1's}\}$. 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) $\{0^n1^n \mid n \geq 1\}$.

* 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 $z = a^nb^nc^{n+n!}$.

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 $a^n b^n aa$ and $a^n b^{n+2}$, where $n \geq 1$.

Now, suppose $L = L(\mathbf{0}^*)$, that is, the set of all strings of 0's. Then $s(L) = \big(s(0)\big)^*$. 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.

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 $s(L)$.

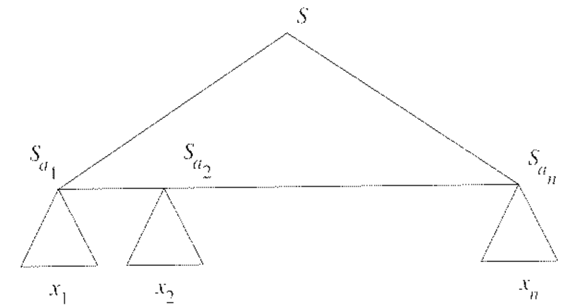

Figure 7.8: A parse tree in G' begins with a parse tree in G and finishes with many parse trees, each in one of the grammars G_a

(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 $L_1 L_2$ 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:

$$\begin{aligned} S &\to AB \\ A &\to 0A1 \mid 01 \\ B &\to 2B \mid 2 \end{aligned}$$

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:

$$\begin{aligned} S &\to AB \\ A &\to 0A \mid 0 \\ B &\to 1B2 \mid 12 \end{aligned}$$

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.

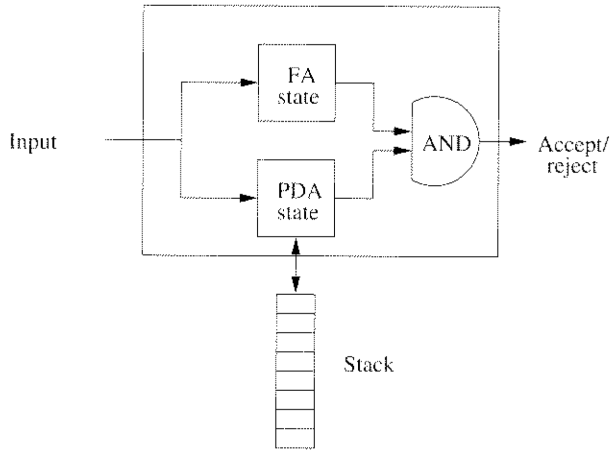

Figure 7.9: A PDA and a FA can run in parallel to create a new PDA

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.

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, $\hat{\delta}(p, a) = \delta_A(p, a)$, 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 $p = \hat{\delta}(q_A, w)$. 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 $L \cap R$. $\square$

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. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$.

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 $\mathbf{i}^*\mathbf{e}^*$, 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: $\delta((p,s),\epsilon,X_0) = \{((q,s),ZX_0)\}$.

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

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

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

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

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 an 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 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.

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

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

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.

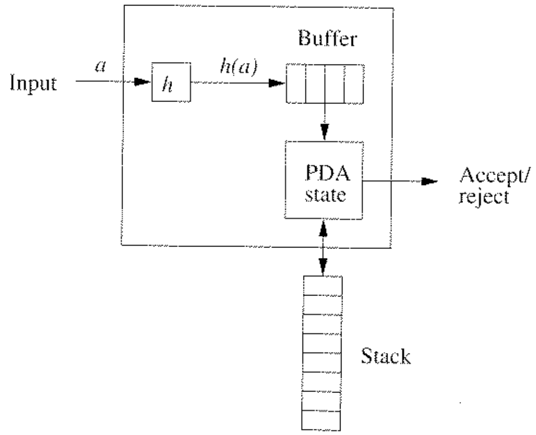

Figure 7.10: Constructing a PDA to accept the inverse homomorphism of what a given PDA accepts

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) $\delta'\big((q,\epsilon),a,X\big) = \Big\{\Big(\big(q,h(a)\big),X\Big)\Big\}$ 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.