Unit-1

Theorem 2.11: If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, then $L(D) = L(N)$.

PROOF: What we actually prove first, by induction on $|w|$, is that

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Notice that each of the $\hat{\delta}$ functions returns a set of states from Q_N, but $\hat{\delta}_D$ interprets this set as one of the states of Q_D (which is the power set of Q_N), while $\hat{\delta}_N$ interprets this set as a subset of Q_N.

BASIS: Let $|w| = 0$; that is, $w = \epsilon$. By the basis definitions of $\hat{\delta}$ for DFA's and NFA's, both $\hat{\delta}_D(\{q_0\}, \epsilon)$ and $\hat{\delta}_N(q_0, \epsilon)$ are $\{q_0\}$.

INDUCTION: Let w be of length $n + 1$, and assume the statement for length n. Break w up as $w = xa$, where a is the final symbol of w. By the inductive hypothesis, $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Let both these sets of N's states be $\{p_1, p_2, \ldots, p_k\}$.

The inductive part of the definition of $\hat{\delta}$ for NFA's tells us

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

The subset construction, on the other hand, tells us that

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Now, let us use (2.3) and the fact that $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ in the inductive part of the definition of $\hat{\delta}$ for DFA's:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D\big(\hat{\delta}_D(\{q_0\}, x), a\big) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.4}$$

Thus, Equations (2.2) and (2.4) demonstrate that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. When we observe that D and N both accept w if and only if $\hat{\delta}_D(\{q_0\}, w)$ or $\hat{\delta}_N(q_0, w)$, respectively, contain a state in F_N, we have a complete proof that $L(D) = L(N)$. $\square$

Theorem 2.22: A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA.

PROOF: (If) This direction is easy. Suppose $L = L(D)$ for some DFA. Turn D into an ϵ-NFA E by adding transitions $\delta(q, \epsilon) = \emptyset$ for all states q of D. Technically, we must also convert the transitions of D on input symbols, e.g., $\delta_D(q, a) = p$ into an NFA-transition to the set containing only p, that is $\delta_E(q, a) = \{p\}$. Thus, the transitions of E and D are the same, but E explicitly states that there are no transitions out of any state on ϵ.

(Only-if) Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ-NFA. Apply the modified subset construction described above to produce the DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

We need to show that $L(D) = L(E)$, and we do so by showing that the extended transition functions of E and D are the same. Formally, we show $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ by induction on the length of w.

BASIS: If $|w| = 0$, then $w = \epsilon$. We know $\hat{\delta}_E(q_0, \epsilon) = \text{ECLOSE}(q_0)$. We also know that $q_D = \text{ECLOSE}(q_0)$, because that is how the start state of D is defined. Finally, for a DFA, we know that $\hat{\delta}(p, \epsilon) = p$ for any state p, so in particular, $\hat{\delta}_D(q_D, \epsilon) = \text{ECLOSE}(q_0)$. We have thus proved that $\hat{\delta}_E(q_0, \epsilon) = \hat{\delta}_D(q_D, \epsilon)$.

INDUCTION: Suppose $w = xa$, where a is the final symbol of w, and assume that the statement holds for x. That is, $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$. Let both these sets of states be $\{p_1, p_2, \ldots, p_k\}$.

By the definition of $\hat{\delta}$ for ϵ-NFA's, we compute $\hat{\delta}_E(q_0, w)$ by:

1. Let $\{r_1, r_2, \ldots, r_m\}$ be $\bigcup_{i=1}^{k} \delta_E(p_i, a)$.
2. Then $\hat{\delta}_E(q_0, w) = \text{ECLOSE}(\{r_1, r_2, \ldots, r_m\})$.

If we examine the construction of DFA D in the modified subset construction above, we see that $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ is constructed by the same two steps (1) and (2) above. Thus, $\hat{\delta}_D(q_D, w)$, which is $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ is the same set as $\hat{\delta}_E(q_0, w)$. We have now proved that $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ and completed the inductive part. $\square$

Unit-2

Theorem 4.11: If L is a regular language, so is L^R.

PROOF: Assume L is defined by regular expression E. The proof is a structural induction on the size of E. We show that there is another regular expression E^R such that $L(E^R) = \big(L(E)\big)^R$; that is, the language of E^R is the reversal of the language of E.

BASIS: If E is ϵ, $\emptyset$, or $\mathbf{a}$, for some symbol a, then E^R is the same as E. That is, we know $\{\epsilon\}^R = \{\epsilon\}$, $\emptyset^R = \emptyset$, and $\{a\}^R = \{a\}$.

INDUCTION: There are three cases, depending on the form of E.

1. $E = E_1 + E_2$. Then $E^R = E_1^R + E_2^R$. The justification is that the reversal of the union of two languages is obtained by computing the reversals of the two languages and taking the union of those languages.

2. $E = E_1E_2$. Then $E^R = E_2^R E_1^R$. Note that we reverse the order of the two languages, as well as reversing the languages themselves. For instance, if $L(E_1) = \{01, 111\}$ and $L(E_2) = \{00, 10\}$, then $L(E_1E_2) = \{0100, 0110, 11100, 11110\}$. The reversal of the latter language is

   $$\{0010, 0110, 00111, 01111\}$$

   If we concatenate the reversals of $L(E_2)$ and $L(E_1)$ in that order, we get

   $$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

   which is the same language as $\big(L(E_1E_2)\big)^R$. In general, if a word w in $L(E)$ is the concatenation of w_1 from $L(E_1)$ and w_2 from $L(E_2)$, then $w^R = w_2^R w_1^R$.

3. $E = E_1^*$. Then $E^R = (E_1^R)^*$. The justification is that any string w in $L(E)$ can be written as $w_1w_2 \cdots w_n$, where each w_i is in $L(E)$. But

   $$w^R = w_n^R w_{n-1}^R \cdots w_1^R$$

   Each w_i^R is in $L(E^R)$, so w^R is in $L\big((E_1^R)^*\big)$. Conversely, any string in $L\big((E_1^R)^*\big)$ is of the form $w_1w_2 \cdots w_n$, where each w_i is the reversal of a string in $L(E_1)$. The reversal of this string, $w_n^R w_{n-1}^R \cdots w_1^R$, is therefore a string in $L(E_1^*)$, which is $L(E)$. We have thus shown that a string is in $L(E)$ if and only if its reversal is in $L\big((E_1^R)^*\big)$.

Theorem 4.1: (The *pumping lemma for regular languages*) Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \epsilon$.
2. $|xy| \leq n$.
3. For all $k \geq 0$, the string xy^kz is also in L.

That is, we can always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language L.

PROOF: Suppose L is regular. Then $L = L(A)$ for some DFA A. Suppose A has n states. Now, consider any string w of length n or more, say $w = a_1a_2 \cdots a_m$, where $m \geq n$ and each a_i is an input symbol. For $i = 0, 1, \ldots, n$ define state p_i to be $\hat{\delta}(q_0, a_1a_2 \cdots a_i)$, where δ is the transition function of A, and q_0 is the start state of A. That is, p_i is the state A is in after reading the first i symbols of w. Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n+1$ different p_i's for $i = 0, 1, \ldots, n$ to be distinct, since there are only n different states. Thus, we can find two different integers i and j, with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1a_2 \cdots a_i$.
2. $y = a_{i+1}a_{i+2} \cdots a_j$.
3. $z = a_{j+1}a_{j+2} \cdots a_m$.

That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of w. The relationships among the strings and states are suggested by Fig. 4.1. Note that x may be empty, in the case that $i = 0$. Also, z may be empty if $j = n = m$. However, y can not be empty, since i is strictly less than j.

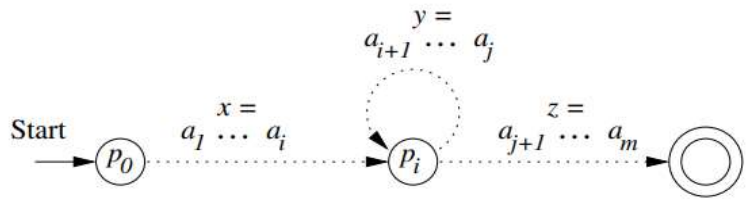


Figure 4.1: Every string longer than the number of states must cause a state to repeat

Now, consider what happens if the automaton A receives the input xy^kz for any $k \geq 0$. If $k = 0$, then the automaton goes from the start state q_0 (which is also p_0) to p_i on input x. Since p_i is also p_j, it must be that A goes from p_i to the accepting state shown in Fig. 4.1 on input z. Thus, A accepts xz.

If $k > 0$, then A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k, and then goes to the accepting state on input z. Thus, for any $k \geq 0$, xy^kz is also accepted by A; that is, xy^kz is in L. □

Closure Under Intersection

Now, let us consider the intersection of two regular languages. We actually have little to do, since the three boolean operations are not independent. Once we have ways of performing complementation and union, we can obtain the intersection of languages L and M by the identity

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

In general, the intersection of two sets is the set of elements that are not in the complement of either set. That observation, which is what Equation (4.1) says, is one of *DeMorgan's laws*. The other law is the same with union and intersection interchanged; that is, $L \cup M = \overline{\overline{L} \cap \overline{M}}$.

However, we can also perform a direct construction of a DFA for the intersection of two regular languages. This construction, which essentially runs two DFA's in parallel, is useful in its own right. For instance, we used it to construct the automaton in Fig. 2.3 that represented the "product" of what two participants — the bank and the store — were doing. We shall make the *product construction* formal in the next theorem.

Theorem 4.8 : If L and M are regular languages, then so is $L \cap M$.

PROOF: Let L and M be the languages of automata $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ and $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Notice that we are assuming that the alphabets of both automata are the same; that is, Σ is the union of the alphabets of L and M, if those alphabets are different. The product construction actually works for NFA's as well as DFA's, but to make the argument as simple as possible, we assume that A_L and A_M are DFA's.

For $L \cap M$ we shall construct an automaton A that simulates both A_L and A_M. The states of A are pairs of states, the first from A_L and the second from A_M. To design the transitions of A, suppose A is in state (p, q), where p is the state of A_L and q is the state of A_M. If a is the input symbol, we see what A_L does on input a; say it goes to state s. We also see what A_M does on input a; say it makes a transition to state t. Then the next state of A will be (s, t). In that manner, A has simulated the effect of both A_L and A_M. The idea is sketched in Fig. 4.3.

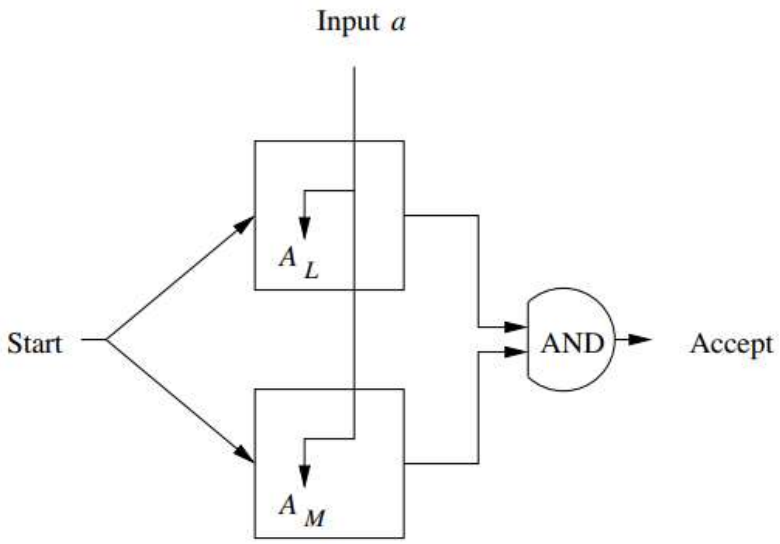


Figure 4.3: An automaton simulating two other automata and accepting if and only if both accept

The remaining details are simple. The start state of A is the pair of start states of A_L and A_M. Since we want to accept if and only if both automata accept, we select as the accepting states of A all those pairs (p, q) such that p is an accepting state of A_L and q is an accepting state of A_M. Formally, we define:

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M)$$

where $\delta\big((p, q), a\big) = \big(\delta_L(p, a), \delta_M(q, a)\big)$.

To see why $L(A) = L(A_L) \cap L(A_M)$, first observe that an easy induction on $|w|$ proves that $\hat{\delta}\big((q_L, q_M), w\big) = \big(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\big)$. But A accepts w if and only if $\hat{\delta}\big((q_L, q_M), w\big)$ is a pair of accepting states. That is, $\hat{\delta}_L(q_L, w)$ must be in F_L, and $\hat{\delta}_M(q_M, w)$ must be in F_M. Put another way, w is accepted by A if and only if both A_L and A_M accept w. Thus, A accepts the intersection of

Theorem 4.16: If h is a homomorphism from alphabet Σ to alphabet T, and L is a regular language over T, then $h^{-1}(L)$ is also a regular language.

PROOF: The proof starts with a DFA A for L. We construct from A and h a DFA for $h^{-1}(L)$ using the plan suggested by Fig. 4.6. This DFA uses the states of A but translates the input symbol according to h before deciding on the next state.

Formally, let L be $L(A)$, where DFA $A = (Q, T, \delta, q_0, F)$. Define a DFA

$$B = (Q, \Sigma, \gamma, q_0, F)$$

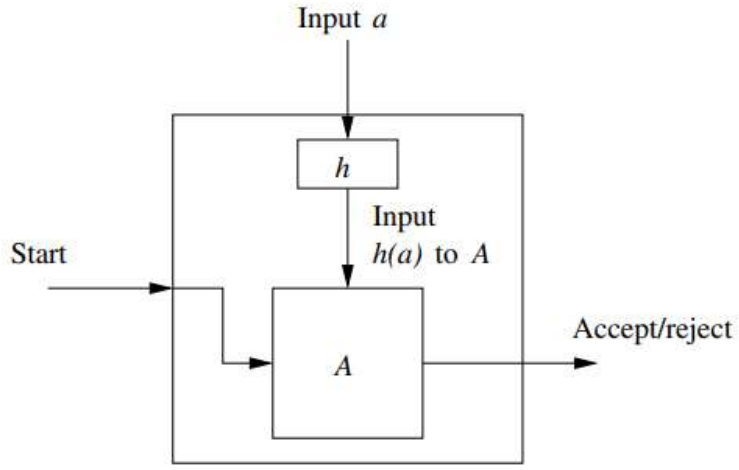


Figure 4.6: The DFA for $h^{-1}(L)$ applies h to its input, and then simulates the DFA for L

where transition function γ is constructed by the rule $\gamma(q, a) = \hat{\delta}\big(q, h(a)\big)$. That is, the transition B makes on input a is the result of the sequence of transitions that A makes on the string of symbols $h(a)$. Remember that $h(a)$ could be ϵ, it could be one symbol, or it could be many symbols, but $\hat{\delta}$ is properly defined to take care of all these cases.

It is an easy induction on $|w|$ to show that $\hat{\gamma}(q_0, w) = \hat{\delta}\big(q_0, h(w)\big)$. Since the accepting states of A and B are the same, B accepts w if and only if A accepts $h(w)$. Put another way, B accepts exactly those strings w that are in $h^{-1}(L)$.

Unit-3

Theorem 5.12: Let $G = (V, T, P, S)$ be a CFG. If the recursive inference procedure tells us that terminal string w is in the language of variable A, then there is a parse tree with root A and yield w.

PROOF: The proof is an induction on the number of steps used to infer that w is in the language of A.

BASIS: One step. Then only the basis of the inference procedure must have been used. Thus, there must be a production $A \to w$. The tree of Fig. 5.8, where there is one leaf for each position of w, meets the conditions to be a parse tree for grammar G, and it evidently has yield w and root A. In the special case that $w = \epsilon$, the tree has a single leaf labeled ϵ and is a legal parse tree with root A and yield w.

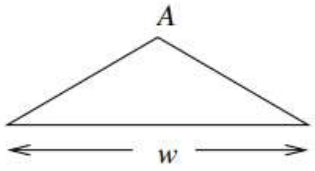


Figure 5.8: Tree constructed in the basis case of Theorem 5.12

INDUCTION: Suppose that the fact w is in the language of A is inferred after $n+1$ inference steps, and that the statement of the theorem holds for all strings x and variables B such that the membership of x in the language of B was inferred using n or fewer inference steps. Consider the last step of the inference that w is in the language of A. This inference uses some production for A, say $A \to X_1 X_2 \cdots X_k$, where each X_i is either a variable or a terminal.

We can break w up as $w_1 w_2 \cdots w_k$, where:

1. If X_i is a terminal, then $w_i = X_i$; i.e., w_i consists of only this one terminal from the production.

2. If X_i is a variable, then w_i is a string that was previously inferred to be in the language of X_i. That is, this inference about w_i took at most n of the $n+1$ steps of the inference that w is in the language of A. It cannot take all $n+1$ steps, because the final step, using production $A \to X_1 X_2 \cdots X_k$, is surely not part of the inference about w_i. Consequently, we may apply the inductive hypothesis to w_i and X_i, and conclude that there is a parse tree with yield w_i and root X_i.

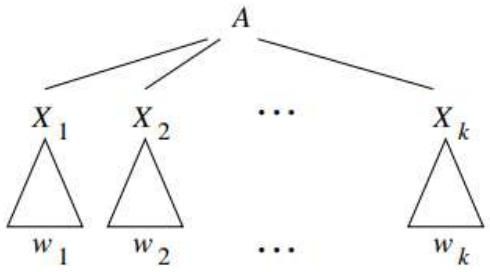


Figure 5.9: Tree used in the inductive part of the proof of Theorem 5.12

We then construct a tree with root A and yield w, as suggested in Fig. 5.9. There is a root labeled A, whose children are $X_1, X_2, \ldots, X_k$. This choice is valid, since $A \to X_1 X_2 \cdots X_k$ is a production of G.

The node for each X_i is made the root of a subtree with yield w_i. In case (1), where X_i is a terminal, this subtree is a trivial tree with a single node labeled X_i. That is, the subtree consists of only this child of the root. Since $w_i = X_i$ in case (1), we meet the condition that the yield of the subtree is w_i.

In case (2), X_i is a variable. Then, we invoke the inductive hypothesis to claim that there is some tree with root X_i and yield w_i. This tree is attached to the node for X_i in Fig. 5.9.

The tree so constructed has root A. Its yield is the yields of the subtrees, concatenated from left to right. That string is $w_1 w_2 \cdots w_k$, which is w. $\square$

Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. 6.4. We use a new symbol X_0, which must not be a symbol of Γ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack. That is, if P_F sees X_0 on top of its stack, then it knows that P_N would empty its stack on the same input.

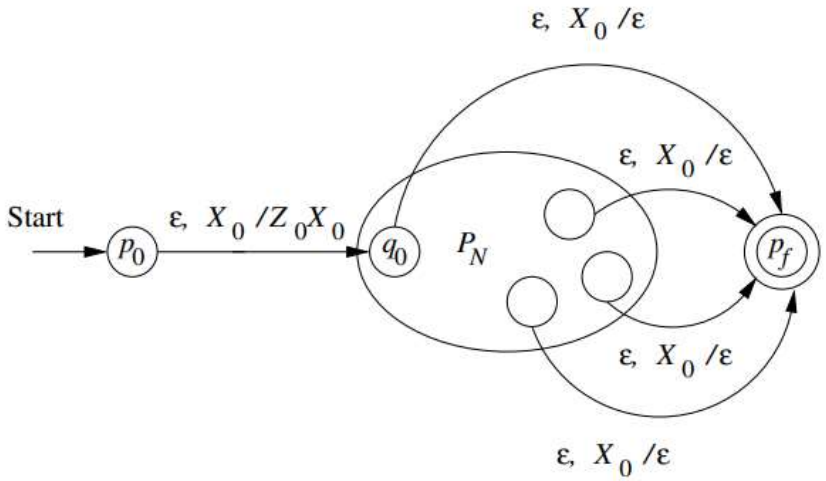


Figure 6.4: P_F simulates P_N and accepts if P_N empties its stack

We also need a new start state, p_0, whose sole function is to push Z_0, the start symbol of P_N, onto the top of the stack and enter state q_0, the start state of P_N. Then, P_F simulates P_N, until the stack of P_N is empty, which P_F detects because it sees X_0 on the top of the stack. Finally, we need another new state, p_f, which is the accepting state of P_F; this PDA transfers to state p_f whenever it discovers that P_N would have emptied its stack.

The specification of P_F is as follows:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

where δ_F is defined by:

1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N, pushing its start symbol Z_0 onto the stack.

2. For all states q in Q, inputs a in Σ or $a = \epsilon$, and stack symbols Y in Γ, $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$.

3. In addition to rule (2), $\delta_F(q, \epsilon, X_0)$ contains (p_f, ϵ) for every state q in Q.

We must show that w is in $L(P_F)$ if and only if w is in $N(P_N)$.

(If) We are given that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$ for some state q. Theorem 6.5 lets us insert X_0 at the bottom of the stack and conclude $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, X_0)$. Since by rule (2) above, P_F has all the moves of P_N, we may also conclude that $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0)$. If we put this sequence of moves together with the initial and final moves from rules (1) and (3) above, we get:

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0) \underset{P_F}{\vdash} (p_f, \epsilon, \epsilon) \tag{6.1}$$

Thus, P_F accepts w by final state.

(Only-if) The converse requires only that we observe the additional transitions of rules (1) and (3) give us very limited ways to accept w by final state. We must use rule (3) at the last step, and we can only use that rule if the stack of P_F contains only X_0. No X_0's ever appear on the stack except at the bottommost position. Further, rule (1) is only used at the first step, and it *must* be used at the first step.

Thus, any computation of P_F that accepts w must look like sequence (6.1). Moreover, the middle of the computation — all but the first and last steps — must also be a computation of P_N with X_0 below the stack. The reason is that, except for the first and last steps, P_F cannot use any transition that is not also a transition of P_N, and X_0 cannot be exposed or the computation would end at the next step. We conclude that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$. That is, w is in $N(P_N)$. □

Unit-4

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 $h(L) = s(L)$.

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 $m + 1$.

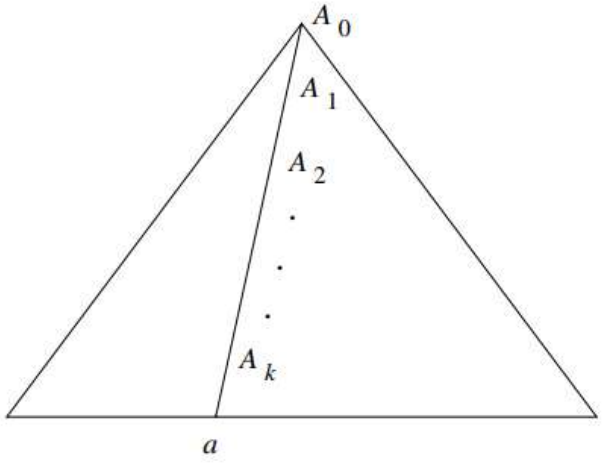


Figure 7.5: Every sufficiently long string in L must have a long path in its parse tree

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 $k - m \leq i < j \leq k$.

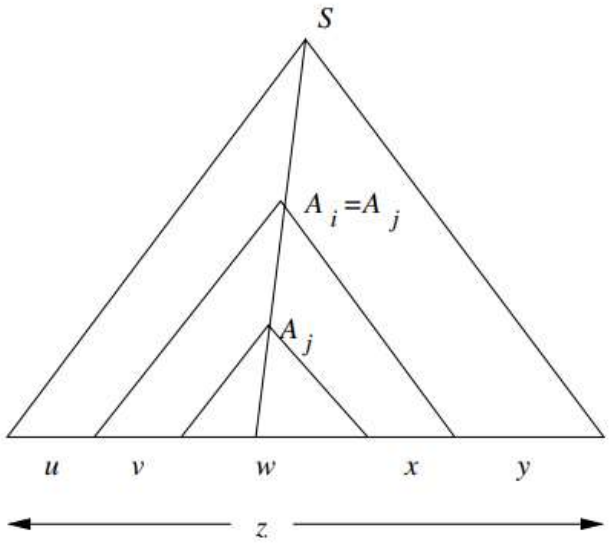


Figure 7.6: Dividing the string w so it can be pumped

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.

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 $2^m = n$. □

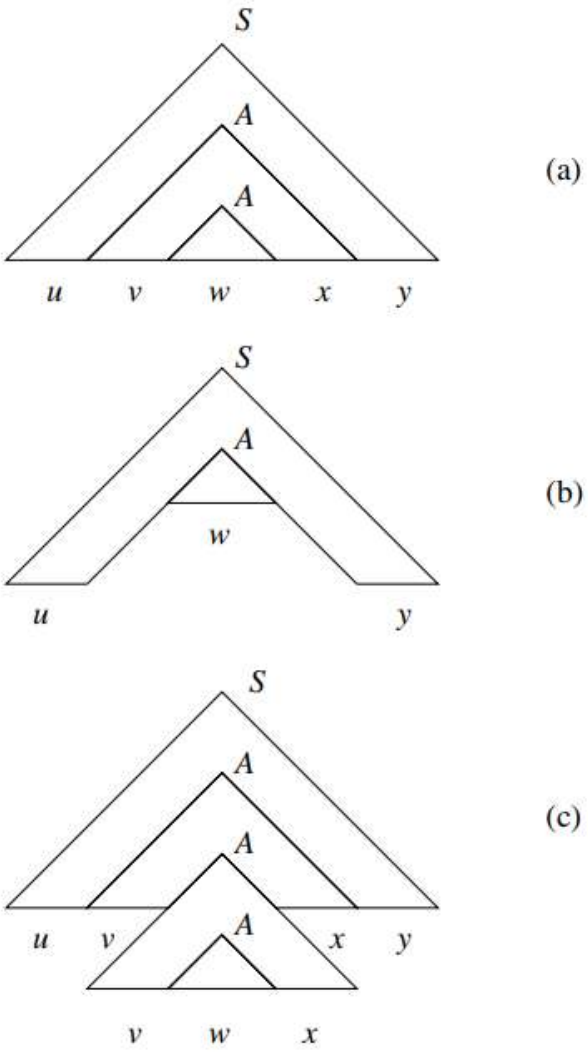


Figure 7.7: Pumping strings v and x zero times and pumping them twice

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

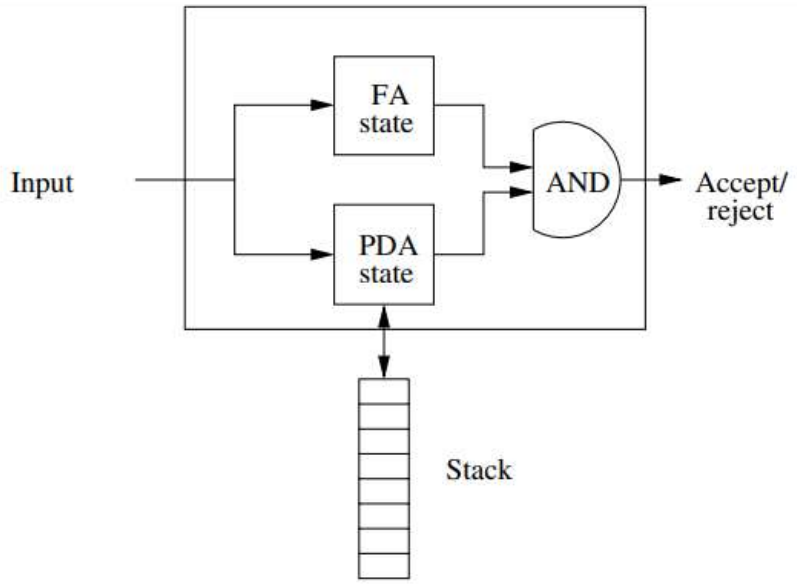


Figure 7.9: A PDA and a FA can run in parallel to create a new PDA

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 $L \cap R$. $\square$

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.

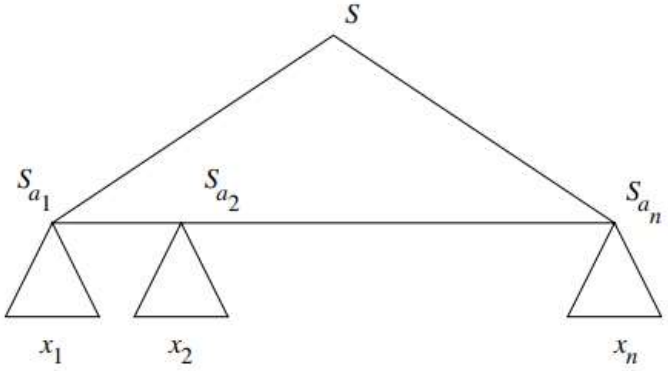
We construct a new grammar $G' = (V', T', P', S)$ for $s(L)$, as follows:

- 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 $s(L)$.



(If) Suppose w is in $s(L)$. Then there is some string $x = a_1 a_2 \cdots a_n$ in L, and strings x_i in $s(a_i)$ for $i = 1, 2, \ldots, n$, such that $w = x_1 x_2 \cdots x_n$. 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 $x_1 x_2 \cdots x_n = w$, 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 $a_1 a_2 \cdots a_n$ in $L(G)$, and strings x_i in language $s(a_i)$, such that

1. $w = x_1 x_2 \cdots x_n$, 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 $x_1 x_2 \cdots x_n$ 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 $s(L)$. $\square$

Unit-5

Theorem 9.3 : If L is a recursive language, so is $\overline{L}$.

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 9.3. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.
2. $\overline{M}$ has a new accepting state r; there are no transitions from r.
3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.

Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □

There is another important fact about complements of languages that further restricts where in the diagram of Fig. 9.2 a language and its complement can fall. We state this restriction in the next theorem.

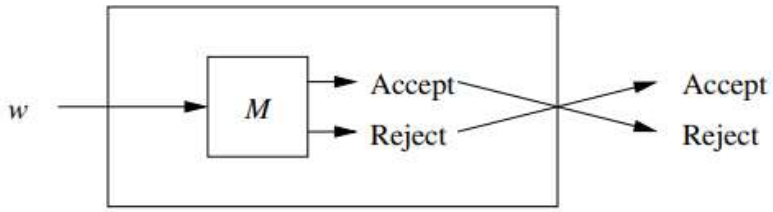


Figure 9.3: Construction of a TM accepting the complement of a recursive language

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\overline{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

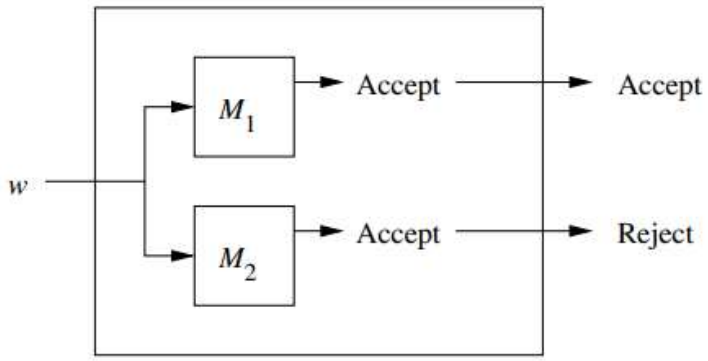


Figure 9.4: Simulation of two TM's accepting a language and its complement

If input w to M is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, then it is in $\overline{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and $L(M)$ is exactly L. Since M always halts, and $L(M) = L$, we conclude that L is recursive. □

We may summarize Theorems 9.3 and 9.4 as follows. Of the nine possible ways to place a language L and its complement $\overline{L}$ in the diagram of Fig. 9.2, only the following four are possible:

1. Both L and $\overline{L}$ are recursive; i.e., both are in the inner ring.
2. Neither L nor $\overline{L}$ is RE; i.e., both are in the outer ring.
3. L is RE but not recursive, and $\overline{L}$ is not RE; i.e., one is in the middle ring and the other is in the outer ring.
4. $\overline{L}$ is RE but not recursive, and L is not RE; i.e., the same as (3), but with L and $\overline{L}$ swapped.

In proof of the above, Theorem 9.3 eliminates the possibility that one language (L or $\overline{L}$) is recursive and the other is in either of the other two classes. Theorem 9.4 eliminates the possibility that both are RE but not recursive.

Theorem 9.8 : L_{ne} is recursively enumerable.

PROOF: We have only to exhibit a TM that accepts L_{ne}. It is easiest to describe a nondeterministic TM M, whose plan is shown in Fig. 9.8. By Theorem 8.11, M can be converted to a deterministic TM.

The operation of M is as follows.

1. M takes as input a TM code M_i.
2. Using its nondeterministic capability, M guesses an input w that M_i might accept.

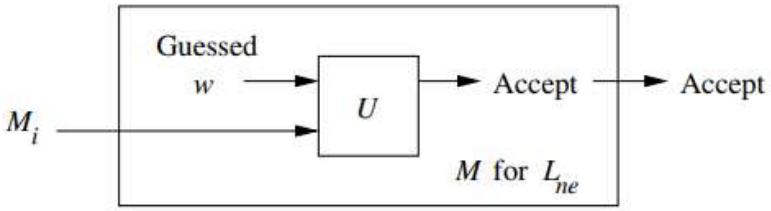


Figure 9.8: Construction of a NTM to accept L_{ne}

3. M tests whether M_i accepts w. For this part, M can simulate the universal TM U that accepts L_u.
4. If M_i accepts w, then M accepts its own input, which is M_i.

In this manner, if M_i accepts even one string, M will guess that string (among all others, of course), and accept M_i. However, if $L(M_i) = \emptyset$, then no guess w leads to acceptance by M_i, so M does not accept M_i. Thus, $L(M) = L_{ne}$. □

Our next step is to prove that L_{ne} is not recursive. To do so, we reduce L_u to L_{ne}. That is, we shall describe an algorithm that transforms an input (M, w) into an output M', the code for another Turing machine, such that w is in $L(M)$ if and only if $L(M')$ is not empty. That is, M accepts w if and only if M' accepts at least one string. The trick is to have M' ignore its input, and instead simulate M on input w. If M accepts, then M' accepts its own input; thus acceptance of w by M is tantamount to $L(M')$ being nonempty. If L_{ne} were recursive, then we would have an algorithm to tell whether or not M accepts w: construct M' and see whether $L(M') = \emptyset$.