In particular, $0 < length(v) \leq k$. This observation can be used to place an upper bound on the length of uv^2w:

$$\begin{aligned} length(uv^2w) &= length(uvw) + length(v) \\ &= k^2 + length(v) \\ &\leq k^2 + k \\ &< k^2 + 2k + 1 \\ &= (k+1)^2. \end{aligned}$$

The length of uv^2w is greater than k^2 and less than $(k+1)^2$ and therefore is not a perfect square. Thus the string uv^2w obtained by pumping v once is not in L. We have shown that there is no decomposition of z that satisfies the conditions of the pumping lemma. The assumption that L is regular leads to a contradiction, establishing the nonregularity of L. □

Example 6.6.2

To show that the language $L = \{a^i \mid i \text{ is prime}\}$ is not regular, we assume that there is a DFA with some number k states that accepts it. Let n be a prime greater than k. The pumping lemma implies that a^n can be decomposed into substrings uvw, $v \neq \lambda$, such that uv^iw is in L for all $i \geq 0$. Assume that such a decomposition exists.

If $uv^{n+1}w \in L$, then its length must be prime. But

$$\begin{aligned} length(uv^{n+1}w) &= length(uvv^nw) \\ &= length(uvw) + length(v^n) \\ &= n + n(length(v)) \\ &= n(1 + length(v)). \end{aligned}$$

Since its length is not prime, $uv^{n+1}w$ is not in L. Thus there is no division of a^n into uvw that satisfies the pumping lemma and we conclude that L is not regular. □

In the preceding examples, the constraints on the length of the strings were sufficient to prove that the languages were not regular. Often the numeric relationships among the elements of a string are used to show that there is no substring that satisfies the conditions of the pumping lemma. We will now present another argument, this time using the pumping lemma, that demonstrates the nonregularity of $\{a^ib^i \mid i \geq 0\}$.

Example 6.6.3

To show that $L = \{a^ib^i \mid i \geq 0\}$ is not regular, we must find a string in L of appropriate length that has no pumpable substring. Assume that L is regular and let k be the number specified

by the pumping lemma. Let z be the string a^kb^k. Any decomposition of uvw of z satisfying the conditions of the pumping lemma must have the form

$$\begin{array}{ccc} u & v & w \\ a^i & a^j & a^{k-i-j}b^k, \end{array}$$

where $i + j \leq k$ and $j > 0$. Pumping any substring of this form produces $uv^2w = a^ia^ja^ja^{k-i-j}b^k = a^ka^jb^k$, which is not in L. Since $z \in L$ has no decomposition that satisfies the conditions of the pumping lemma, we conclude that L is not regular. □

Example 6.6.4

The language $L = \{a^ib^mc^n \mid 0 < i,\ 0 < m < n\}$ is not regular. Assume that L is accepted by a DFA with k states. Then, by the pumping lemma, every string $z \in L$ with length k or more can be written $z = uvw$, with $length(uv) \leq k$, $length(v) > 0$, and $uv^iw \in L$ for all $i \geq 0$.

Consider the string $z = ab^kc^{k+1}$, which is in L. We must show that there is no suitable decomposition of z. Any decomposition of z must have one of two forms, and the cases are examined separately.

Case 1: A decomposition in which $a \notin v$ has the form

$$\begin{array}{ccc} u & v & w \\ ab^i & b^j & b^{k-i-j}c^{k+1} \end{array}$$

where $i + j \leq k - 1$ and $j > 0$. Pumping v produces $uv^2w = ab^ib^jb^jb^{k-i-j}c^{k+1} = ab^kb^jc^{k+1}$, which is not in L.

Case 2: A decomposition of z in which $a \in v$ has the form

$$\begin{array}{ccc} u & v & w \\ \lambda & ab^i & b^{k-i}c^{k+1} \end{array}$$

where $i \leq k - 1$. Pumping v zero times produces $uv^0w = b^{k-i}c^{k+1}$, which is not in L since it does not contain an a.

Since ab^kc^{k+1} has no decomposition with a "pumpable" substring, L is not regular. □

The pumping lemma can be used to determine the size of the language accepted by a DFA. Pumping a string generates an infinite sequence of strings that are accepted by the DFA. To determine whether a regular language is finite or infinite it is only necessary to determine if it contains a pumpable string.

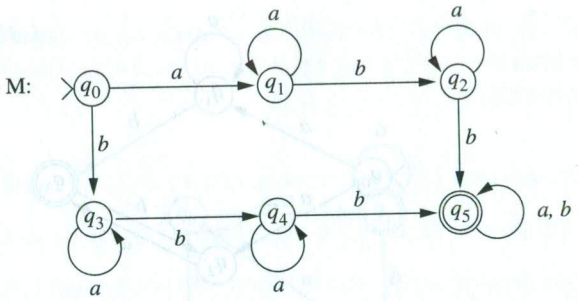
Theorem 6.6.4

Let M be a DFA with k states.

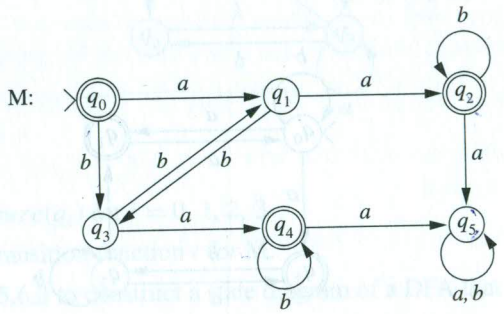
i) L(M) is not empty if, and only if, M accepts a string z with $length(z) < k$.

ii) L(M) has an infinite number of members if, and only if, M accepts a string z where $k \leq length(z) < 2k$.

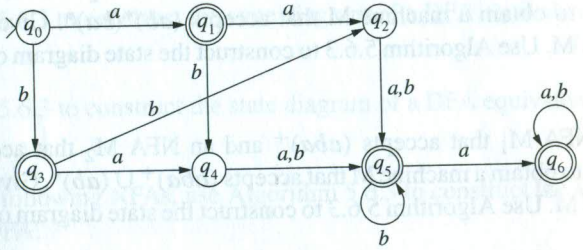
a)



b)



c)



Bibliographic Notes

Alternative interpretations of the result of finite-state computations were studied in Mealy [1955] and Moore [1956]. Transitions in Mealy machines are accompanied by the generation of output. A two-way automaton allows the tape head to move in both directions. A proof that two-way and one-way automata accept the same languages can be found in Rabin and Scott [1959] and Sheperdson [1959]. Nondeterministic finite automata were introduced by Rabin and Scott [1959]. The algorithm for minimizing the number of states in a DFA was presented in Nerode [1958]. The algorithm of Hopcroft [1971] increases the efficiency of the minimization technique.

The theory and applications of finite automata are developed in greater depth in the books by Minsky [1967]; Salomaa [1973]; Denning, Dennis, and Qualitz [1978]; and Bavel [1983].

CHAPTER 6

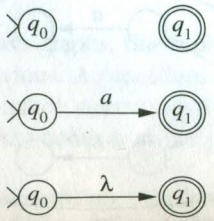
Properties of Regular Languages

Grammars were introduced as language generators, finite automata as language acceptors, and regular expressions as pattern descriptors. This chapter develops the relationship between these three approaches to language definition and explores the limitations of finite automata as language acceptors.

6.1 Finite-State Acceptance of Regular Languages

In this section we show that an NFA-λ can be constructed to accept any regular language. Regular sets are built recursively from ∅, {λ}, and singleton sets containing elements from the alphabet by applications of union, concatenation, and the Kleene star operation (Definition 2.3.2). The construction of an NFA-λ that accepts a regular set can be obtained following the steps of its recursive generation, but using state diagrams as the building blocks rather than sets.

State diagrams for machines that accept ∅, {λ}, and singleton sets {a} are

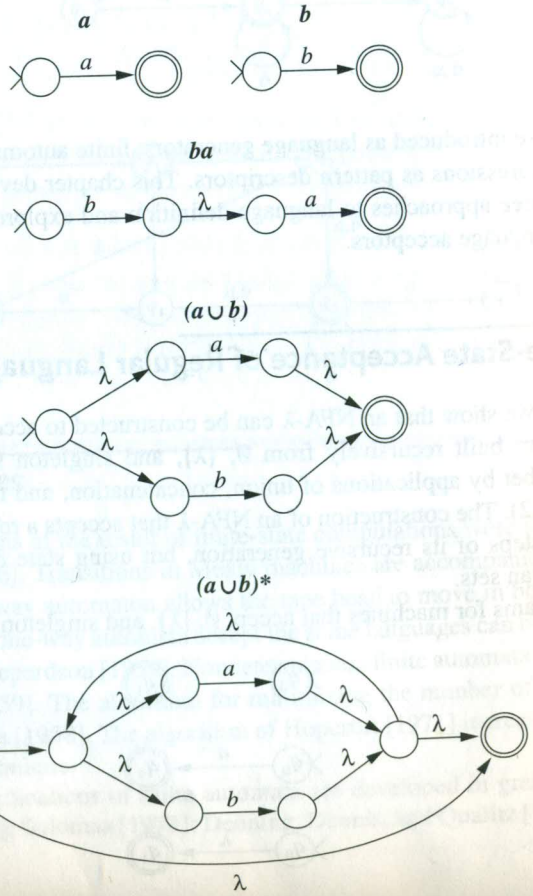


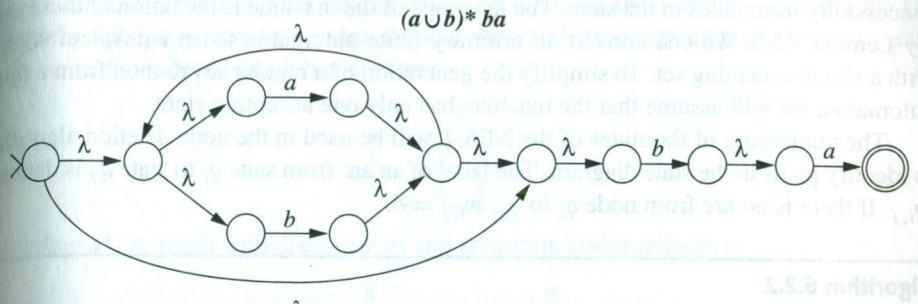
Note that each of these machines satisfies the restrictions described in Lemma 5.5.2. That is, the machines contain a single accepting state and there are no arcs entering the start state or leaving the accepting state.

As shown in Theorem 5.5.3, λ-transitions can be used to combine machines of this form to produce machines that accept more complex languages. Using repeated applications of these techniques, the construction of the regular expression from the basis elements can be mimicked by the corresponding machine operations. This process is illustrated in the following example.

Example 6.1.1

An NFA-λ that accepts $(a \cup b)^*ba$ is constructed following the steps in the recursive definition of the regular expression. The language accepted by each intermediate machine is indicated by the regular expression above the state diagram.





□

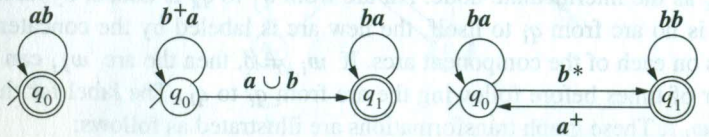
6.2 Expression Graphs

The construction in the previous section demonstrates that every regular language is recognized by a finite automaton. We will now show that every language accepted by a finite automaton is regular by constructing a regular expression for the language of the machine. To accomplish this, we extend the notion of a state diagram.

Definition 6.2.1

An **expression graph** is a labeled directed graph in which the arcs are labeled by regular expressions. An expression graph, like a state diagram, contains a distinguished start node and a set of accepting nodes.

The state diagram of a finite automaton with alphabet Σ is a special case of an expression graph; the labels consist of λ and expressions corresponding to the elements of Σ. Paths in expression graphs generate regular expressions. The language of an expression graph is the union of the regular expressions along paths from the start node to an accepting node. For example, the expression graphs



accept the languages $(ab)^*$, $(b^+a)^*(a \cup b)(ba)^*$, and $(ba)^*b^*(bb \cup (a^+(ba)^*b^*))^*$, respectively.

Because of the simplicity of the graphs, the expressions for the languages accepted by the previous examples were obvious. A procedure is developed to reduce an arbitrary expression graph to an expression graph containing at most two nodes. The reduction is accomplished by repeatedly removing nodes from the graph in a manner that preserves the language of the graph.

The state diagram of a finite automaton may have any number of accepting states. Each of these states exhibits the acceptance of a set of strings, the strings whose processing

successfully terminates in the state. The language of the machine is the union of these sets. By Lemma 5.5.2, we can convert an arbitrary finite automaton to an equivalent NFA-λ with a single accepting set. To simplify the generation of a regular expression from a finite automaton, we will assume that the machine has only one accepting state.

The numbering of the states of the NFA-λ will be used in the node deletion algorithm to identify paths in the state diagram. The label of an arc from state q_i to state q_j is denoted $w_{i,j}$. If there is no arc from node q_i to q_j, $w_{i,j} = \emptyset$.

Algorithm 6.2.2
Construction of a Regular Expression from a Finite Automaton

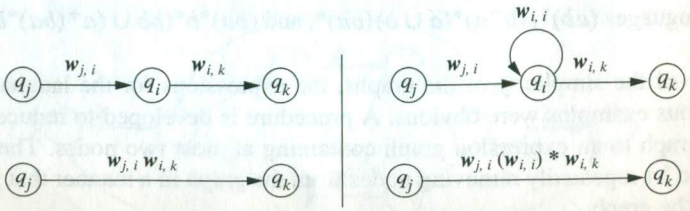
input: state diagram G of a finite automaton with one accepting state

Let q_0 be the start state and q_t the accepting state of G.

1. **repeat**
   - 1.1. choose a node q_i that is neither q_0 nor q_t
   - 1.2. delete the node q_i from G according to the following procedure:
     - 1.2.1 **for** every j, k not equal to i (this includes $j = k$) **do**
       - i) **if** $w_{j,i} \neq \emptyset$, $w_{i,k} \neq \emptyset$ and $w_{i,i} = \emptyset$, **then** add an arc from node j to node k labeled $w_{j,i}w_{i,k}$
       - ii) **if** $w_{j,i} \neq \emptyset$, $w_{i,k} \neq \emptyset$ and $w_{i,i} \neq \emptyset$, **then** add an arc from node q_j to node q_k labeled $w_{j,i}(w_{i,i})^*w_{i,k}$
       - iii) **if** nodes q_j and q_k have arcs labeled $w_1, w_2, \ldots, w_s$ connecting them, **then** replace the arcs by a single arc labeled $w_1 \cup w_2 \cup \cdots \cup w_s$
     - 1.2.2 remove the node q_i and all arcs incident to it in G

   until the only nodes in G are q_0 and q_t
2. determine the expression accepted by G

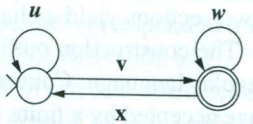
The deletion of node q_i is accomplished by finding all paths q_j, q_i, q_k of length two that have q_i as the intermediate node. An arc from q_j to q_k is added, bypassing the node q_i. If there is no arc from q_i to itself, the new arc is labeled by the concatenation of the expressions on each of the component arcs. If $w_{i,i} \neq \emptyset$, then the arc $w_{i,i}$ can be traversed any number of times before following the arc from q_i to q_k. The label for the new arc is $w_{j,i}(w_{i,i})^*w_{i,k}$. These graph transformations are illustrated as follows:



Step 2 in the algorithm may appear to be begging the question; the objective of the entire algorithm is to determine the expression accepted by G. After the node deletion process is

completed, the regular expression can easily be obtained from the resulting graph. The reduced graph has at most two nodes, the start node and the accepting node. If these are the same node, the reduced graph has the form

accepting u^*. A graph with distinct start and accepting nodes reduces to

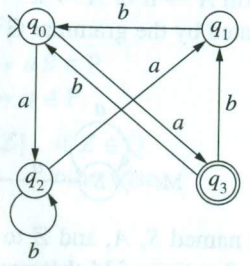


and accepts the expression $u^*v(w \cup xu^*v)^*$. This expression may be simplified if any of the arcs in the graph are labeled $\emptyset$.

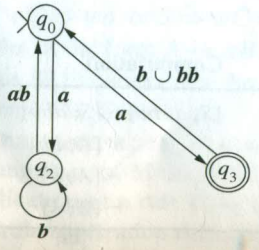
Algorithm 6.2.2 can also be used to construct the language of a finite state machine with multiple accepting states. For each accepting state, we can produce an expression for the strings accepted by that state. The language of the machine is simply the union of the regular expressions obtained for each accepting state.

Example 6.2.1

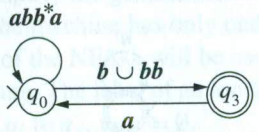
The reduction technique of Algorithm 6.2.2 is used to generate a regular expression for the language of the NFA with state diagram



Deleting node q_1 yields



The deletion of q_1 produced a second path from q_3 to q_0, which is indicated by the union in the expression on the arc from q_3 to q_0. Removing q_2 produces



with associated language $(abb^*a)^*(b \cup bb)(a(abb^*a)^*(b \cup bb))^*$. □

The results of the previous two sections yield a characterization of regular languages originally established by Kleene. The construction outlined in Section 6.1 can be used to build an NFA-λ to accept any regular language. Conversely, Algorithm 6.2.2 produces a regular expression for the language accepted by a finite automaton. Using the equivalence of deterministic and nondeterministic machines, Kleene's Theorem can be expressed in terms of languages accepted by deterministic finite automata.

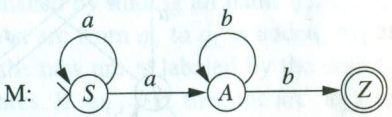
Theorem 6.2.3 (Kleene)
A language L is accepted by a DFA with alphabet Σ if, and only if, L is a regular language over Σ.

6.3 Regular Grammars and Finite Automata

A context-free grammar is called regular (Section 3.3) if each rule is of the form $A \to aB$, $A \to a$, or $A \to \lambda$. A string derivable in a regular grammar contains at most one variable which, if present, occurs as the rightmost symbol. A derivation is terminated by the application of a rule of the form $A \to a$ or $A \to \lambda$.

The language a^+b^+ is generated by the grammar G and accepted by the NFA M

G: $S \to aS \mid aA$
$A \to bA \mid b$



where the states of M have been named S, A, and Z to simplify the comparison of computation and generation. The computation of M that accepts $aabb$ is given along with the derivation that generates the string in G.

Derivation	Computation	String Processed
$S \Rightarrow aS$	$[S, aabb] \vdash [S, abb]$	a
$\Rightarrow aaA$	$\vdash [A, bb]$	aa
$\Rightarrow aabA$	$\vdash [A, b]$	aab
$\Rightarrow aabb$	$\vdash [Z, \lambda]$	$aabb$

A computation in an automaton begins with the input string, sequentially processes the leftmost symbol, and halts when the entire string has been analyzed. Generation, on the other hand, begins with the start symbol of the grammar and adds terminal symbols to the prefix of the derived sentential form. The derivation terminates with the application of a λ-rule or a rule whose right-hand side is a single terminal.

The example illustrates the correspondence between generating a terminal string with a regular grammar and processing the string by a computation of an automaton. The state of the automaton is identical to the variable in the derived string. A computation terminates when the entire string has been processed, and the result is designated by the final state. The accepting state Z, which does not correspond to a variable in the grammar, is added to M to represent the completion of the derivation of G.

The state diagram of an NFA M can be constructed directly from the rules of a grammar G. The states of the automaton consist of the variables of the grammar and, possibly, an additional accepting state. In the previous example, transitions $\delta(S, a) = S$, $\delta(S, a) = A$, and $\delta(A, b) = A$ of M correspond to the rules $S \to aS$, $S \to aA$, and $A \to bA$ of G. The left-hand side of the rule represents the current state of the machine. The terminal on the right-hand side is the input symbol. The state corresponding to the variable on the right-hand side of the rule is entered as a result of the transition.

Since the rule terminating a derivation does not add a variable to the string, the consequences of an application of a λ-rule or a rule of the form $A \to a$ must be incorporated into the construction of the corresponding automaton.

Theorem 6.3.1
Let G = (V, Σ, P, S) be a regular grammar. Define the NFA M = (Q, Σ, δ, S, F) as follows:

i) $Q = \begin{cases} V \cup \{Z\} & \text{where } Z \notin V, \text{ if P contains a rule } A \to a \\ V & \text{otherwise.} \end{cases}$

ii) $\delta(A, a) = B$ whenever $A \to aB \in P$
$\delta(A, a) = Z$ whenever $A \to a \in P$.

iii) $F = \begin{cases} \{A \mid A \to \lambda \in P\} \cup \{Z\} & \text{if } Z \in Q \\ \{A \mid A \to \lambda \in P\} & \text{otherwise.} \end{cases}$

Then L(M) = L(G).

Proof. The construction of the machine transitions from the rules of the grammar allows every derivation of G to be traced by a computation in M. The derivation of a terminal string has the form $S \Rightarrow \lambda$, $S \overset{*}{\Rightarrow} wC \Rightarrow wa$, or $S \overset{*}{\Rightarrow} wC \Rightarrow w$ where the derivation $S \overset{*}{\Rightarrow} wC$ consists of the application of rules of the form $A \to aB$. Induction can be used to establish the existence of a computation in M that processes the string w and terminates in state C whenever wC is a sentential form of G (Exercise 6).

First we show that every string generated by G is accepted by M. If L(G) contains the null string, then S is an accepting state of M and $\lambda \in L(M)$. The derivation of a nonnull string is terminated by the application of a rule $C \to a$ or $C \to \lambda$. In a derivation of the form $S \overset{*}{\Rightarrow} wC \Rightarrow wa$, the final rule application corresponds to the transition $\delta(C, a) = Z$,

causing the machine to halt in the accepting state Z. A derivation of the form $S \overset{*}{\Rightarrow} wC \Rightarrow w$ is terminated by the application of a λ-rule. Since $C \to \lambda$ is a rule of G, the state C is accepting in M. The acceptance of w in M is exhibited by the computation that corresponds to the derivation $S \overset{*}{\Rightarrow} wC$.

Conversely, we must show that $L(M) \subseteq L(G)$. Let $w = ua$ be a string accepted by M. A computation accepting w has the form

$$[S, w] \overset{*}{\vdash} [B, \lambda], \quad \text{where } B \neq Z,$$

or

$$[S, w] \overset{*}{\vdash} [A, a] \vdash [Z, \lambda].$$

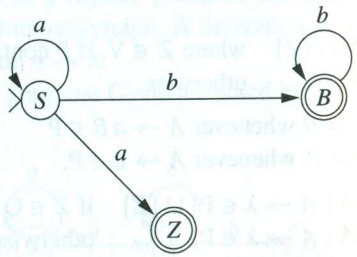
In the former case, B is the left-hand side of a λ-rule of G. The string wB can be derived by applying the rules that correspond to transitions in the computation. The generation of w is completed by the application of the λ-rule. Similarly, a derivation of uA can be constructed from the rules corresponding to the transitions in the computation $[S, w] \overset{*}{\vdash} [A, a]$. The string w is obtained by terminating this derivation with the rule $A \to a$. Thus every string accepted by M is in the language of G. ■

Example 6.3.1

The grammar G generates and the NFA M accepts the language $a^*(a \cup b^+)$.

G: $S \to aS \mid bB \mid a$
$B \to bB \mid \lambda$

M:



□

The preceding transformation can be reversed to construct a regular grammar from an NFA. The transition $\delta(A, a) = B$ produces the rule $A \to aB$. Since every transition results in a new machine state, no rules of the form $A \to a$ are produced. The rules obtained from the transitions generate derivations of the form $S \overset{*}{\Rightarrow} wC$ that mimic computations in the automaton. Rules must be added to terminate the derivations. When C is an accepting state, a computation that terminates in state C exhibits the acceptance of w. Completing the derivation $S \overset{*}{\Rightarrow} wC$ with the application of a rule $C \to \lambda$ generates w in G. The grammar is completed by adding λ-rules for all accepting states of the automaton. This informal argument justifies Theorem 6.3.2. The formal proof is left as an exercise.

Theorem 6.3.2

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Define a regular grammar $G = (V, \Sigma, P, q_0)$ as follows:

i) $V = Q$,

ii) $q_i \to aq_j \in P$ whenever $\delta(q_i, a) = q_j$,

iii) $q_i \to \lambda \in P$ if $q_i \in F$.

Then $L(G) = L(M)$.

The constructions outlined in Theorems 6.3.1 and 6.3.2 can be applied sequentially to shift from automaton to grammar and back again. Beginning with an NFA M, the sequence of transformations would have the form

$$M \longrightarrow G \longrightarrow M'.$$

Since G contains only rules of the form $A \to aB$ or $A \to \lambda$, the NFA M' is identical to M.

A regular grammar G can be converted to an NFA that, in turn, can be reconverted into a grammar G':

$$G \longrightarrow M \longrightarrow G'.$$

The grammar G' that results from these conversions can be obtained directly from G by adding a single new variable, call it Z, to the grammar and the rule $Z \to \lambda$. All rules $A \to a$ are then replaced by $A \to aZ$.

Example 6.3.2

The regular grammar G' that accepts L(M) is constructed from the automaton M from Example 6.3.1.

G': $S \to aS \mid bB \mid aZ$
$B \to bB \mid \lambda$
$Z \to \lambda$

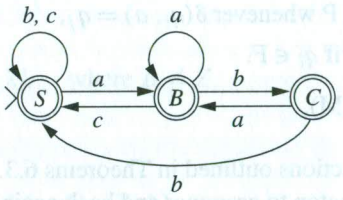
The transitions provide the S rules and the first B rule. The λ-rules are added since B and Z are accepting states. □

The two conversions allow us to conclude that the languages generated by regular grammars are precisely those accepted by finite automata. It follows from Theorems 6.2.3 and 6.3.1 that the language generated by a regular grammar is a regular set. The conversion from automaton to regular grammar guarantees that every regular set is generated by some regular grammar. This yields the characterization of regular languages promised in Section 3.3: the languages generated by regular grammars.

Example 6.3.3

The language of the regular grammar from Example 3.2.12 is the set of strings over $\{a, b, c\}$ that do not contain the substring abc. Theorem 6.3.1 is used to construct an NFA that accepts this language.

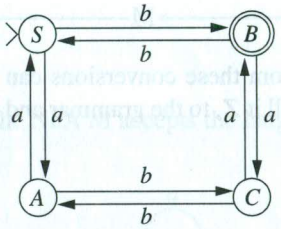
$$S \to bS \mid cS \mid aB \mid \lambda$$
$$B \to aB \mid cS \mid bC \mid \lambda$$
$$C \to aB \mid bS \mid \lambda$$



□

Example 6.3.4

A regular grammar with alphabet $\{a, b\}$ that generates strings with an even number of a's and an odd number of b's can be constructed from the DFA in Example 5.3.5. This machine is reproduced below with the states $[e_a, e_b]$, $[o_a, e_b]$, $[e_a, o_b]$, and $[o_a, o_b]$ renamed S, A, B, and C, respectively.



The associated grammar is

$$S \to aA \mid bB$$
$$A \to aS \mid bC$$
$$B \to bS \mid aC \mid \lambda$$
$$C \to aB \mid bA.$$

□

6.4 Closure Properties of Regular Languages

Regular languages have been defined, generated, and accepted. A language over an alphabet Σ is regular if it is

i) a regular set (expression) over Σ,

ii) accepted by a DFA, NFA, or NFA-λ, or

iii) generated by a regular grammar.

A family of languages is *closed* under an operation if the application of the operation to members of the family produces a member of the family. Each of the equivalent formulations of regularity will be used to demonstrate closure properties of the family of regular languages.

The recursive definition of regular sets establishes closure for the unary operation Kleene star and the binary operations union and concatenation. This was also proved in Theorem 5.5.3 using acceptance by finite-state machines.

Theorem 6.4.1

Let L_1 and L_2 be two regular languages. The languages $L_1 \cup L_2$, L_1L_2, and L_1^* are regular languages.

The regular languages are also closed under complementation. If L is regular over the alphabet Σ, then so is $\overline{L} = \Sigma^* - L$, the set containing all strings in Σ^* that are not in L. Theorem 5.3.3 used the properties of DFAs to construct a machine that accepts $\overline{L}$ from one that accepts L. Complementation and union combine to establish the closure of regular languages under intersection.

Theorem 6.4.2

Let L be a regular language over Σ. The language $\overline{L}$ is regular.

Theorem 6.4.3

Let L_1 and L_2 be regular languages over Σ. The language $L_1 \cap L_2$ is regular.

Proof. By DeMorgan's Law

$$L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}.$$

The right-hand side of the equality is regular since it is built from L_1 and L_2 using union and complementation. ■

Closure properties provide additional tools for establishing the regularity of languages. The operations of complementation and intersection, as well as union, concatenation, and Kleene star, preserve regularity when combining regular languages.

Example 6.4.1

Let L be the language over $\{a, b\}$ consisting of all strings that contain the substring aa but do not contain bb. The regular languages $L_1 = (a \cup b)^*aa(a \cup b)^*$ and $L_2 = (a \cup b)^*bb(a \cup b)^*$ consist of strings containing substrings aa and bb, respectively. Hence, $L = L_1 \cap \overline{L_2}$ is regular. □

Example 6.4.2

Let L be any regular language over $\{a, b\}$. The language

$$L_1 = \{u \mid u \in L \text{ and } u \text{ has exactly one } a\}$$

is regular. The regular expression b^*ab^* describes the set of strings with exactly one a. The language $L_1 = L \cap b^*ab^*$ is regular since it is the intersection of regular languages. □

The next example exhibits the robustness of the family of regular languages. Adding or removing a small number, in fact any finite number, of strings cannot turn a regular language into a nonregular language.

Example 6.4.3

Let L_1 be a regular language over an alphabet Σ and let $L_2 \subseteq \Sigma^*$ be any finite set of strings. Then $L_1 \cup L_2$ and $L_1 - L_2$ are both regular. The critical observation is that any finite language is regular. Why? The regularity of $L_1 \cup L_2$ and $L_1 - L_2$ then follows from the closure of the regular languages under union and set difference (Exercise 8). □

Example 6.4.4

The set SUF(L) $= \{v \mid uv \in L\}$ consists of all suffixes of strings of the langauge L. For example, if $aabb \in L$, then λ, b, bb, abb, and $aabb$ are in SUF(L). We will show that if L is regular, then so is SUF(L). Since L is regular, we know that it is defined by a regular expression, accepted by a finite automaton, and generated by a regular grammar. We may use any of these categorizations of regularity to show that SUF(L) is regular.

Using the grammatical characterization, we know that L is generated by a regular grammar $G = (V, \Sigma, P, S)$. We may assume that G has no useless symbols. If it did, we would use the algorithm from Section 4.4 to remove them while preserving the language.

A suffix of v of G is produced by a derivation of the form

$$S \overset{*}{\Rightarrow} uA \overset{*}{\Rightarrow} uv.$$

Intuitively, we would like to add a rule $S \to A$ to G to directly generate the suffix

$$S \Rightarrow A \overset{*}{\Rightarrow} v.$$

Unfortunately, the resulting grammar would not be regular. To fix that problem, we will use grammar transformations from Chapter 4.

We begin by defining a new grammar $G' = (V', \Sigma, P', S')$ by

$$V' = V \cup \{S'\}$$

$$P' = P \cup \{S' \to A \mid A \in V\}.$$

A derivation in G′ uses only one rule not in G. Any string in L is produced by a derivation of the form

$$S' \Rightarrow S \overset{*}{\Rightarrow} w,$$

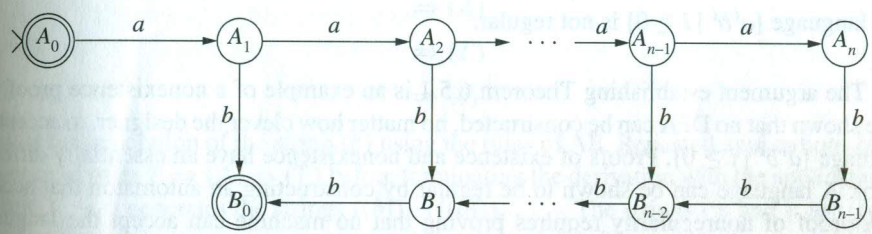
while the remaining suffixes are generated by

$$S' \Rightarrow A \overset{*}{\Rightarrow} w.$$

Consequently, L(G) = SUF(L). We can obtain an equivalent regular grammar by removing λ-rules and chain rules from G′. □

6.5 A Nonregular Language

The incompletely specified DFA



accepts the language $\{a^ib^i \mid i \leq n\}$. The states A_i count the number of leading a's in the input string. Upon processing the first b, the machine enters the sequence of states labeled B_i. The accepting state B_0 is entered when an equal number of b's are processed. This strategy cannot be extended to accept the language $L = \{a^ib^i \mid i \geq 0\}$ since it would require infinitely many states. However, there may be other strategies and machines that accept L that only require finitely many states. We will show that this is not the case, that L is not accepted by any DFA and therefore is not a regular language.

The proof of the nonregularity of the language $L = \{a^ib^i \mid i \geq 0\}$ is by contradiction. We assume that there is a DFA that accepts L and show that it must have states that record the number of a's in the same manner as the states $A_1, A_2, \ldots$ in the preceding diagram. It follows that the machine must have infinitely many states, which contradicts the requirement that a DFA has only finitely many states. The contradiction allows us to conclude that no DFA can accept L.

We begin with the assumption that L is accepted by some DFA, call it M. The extended transition function $\hat{\delta}$ is used to show that the automaton M must have an infinite number of states. Let A_i be the state of the machine entered upon processing the string a^i; that is, $\hat{\delta}(q_0, a^i) = A_i$. For all $i, j \geq 0$ with $i \neq j$, $a^ib^i \in L$ and $a^jb^i \notin L$. Hence, $\hat{\delta}(q_0, a^ib^i) \neq \hat{\delta}(q_0, a^jb^i)$ since the former is an accepting state and the latter rejecting. Now

$$\hat{\delta}(q_0, a^ib^i) = \hat{\delta}(\hat{\delta}(q_0, a^i), b^i) = \hat{\delta}(A_i, b^i) \in L$$

and

$$\hat{\delta}(q_0, a^j b^i) = \hat{\delta}(\hat{\delta}(q_0, a^j), b^i) = \hat{\delta}(A_j, b^i) \notin \mathrm{L}.$$

Consequently, $\hat{\delta}(A_i, b^i) \neq \hat{\delta}(A_j, b^i)$. In a deterministic machine, two computations that begin in the same state and process the same string must end in the same state. Since the computations $\hat{\delta}(A_i, b^i)$ and $\hat{\delta}(A_j, b^i)$ process the same string but terminate in different states, we conclude that $A_i \neq A_j$.

We have shown that states A_i and A_j are distinct for all values of $i \neq j$. Any deterministic finite-state machine that accepts L must contain an infinite sequence of states corresponding to A_0, A_1, A_2, ... This violates the restriction that limits a DFA to a finite number of states. Consequently, there is no DFA that accepts L, or equivalently, L is not regular. The preceding argument justifies Theorem 6.5.1.

Theorem 6.5.1

The language $\{a^i b^i \mid i \geq 0\}$ is not regular.

The argument establishing Theorem 6.5.1 is an example of a nonexistence proof. We have shown that no DFA can be constructed, no matter how clever the designer, to accept the language $\{a^i b^i \mid i \geq 0\}$. Proofs of existence and nonexistence have an essentially different flavor. A language can be shown to be regular by constructing an automaton that accepts it. A proof of nonregularity requires proving that no machine can accept the language. Theorem 6.5.1 can be generalized to establish the nonregularity of a number of languages.

Corollary 6.5.2 (to the proof of Theorem 6.5.1)

Let L be a language over Σ. If there are sequences of distinct strings $u_i \in \Sigma^*$ and $v_i \in \Sigma^*$, $i \geq 0$, with $u_i v_i \in \mathrm{L}$ and $u_i v_j \notin \mathrm{L}$ for $i \neq j$, then L is not a regular language.

The proof is identical to that of Theorem 6.5.1, with u_i replacing a^i and v_i replacing b^i.

Example 6.5.1

The set L of palindromes over $\{a, b\}$ is not regular. By Corollary 6.5.2, it is sufficient to discover two sequences of strings u_i and v_i that satisfy $u_i v_i \in \mathrm{L}$ and $u_i v_j \notin \mathrm{L}$ for all $i \neq j$. The strings

$$u_i = a^i b$$
$$v_i = a^i$$

fulfill these requirements. □

Example 6.5.2

Grammars were introduced as a formal structure for defining the syntax of languages. Corollary 6.5.2 can be used to show that regular grammars are not a sufficiently powerful tool to define programming languages containing arithmetic or Boolean expressions in infix form. The grammar AE

$$\begin{aligned} \text{AE: } & S \rightarrow A \\ & A \rightarrow T \mid A + T \\ & T \rightarrow b \mid (A) \end{aligned}$$

generates additive expressions using +, parentheses, and the operand b. For example, (b), $b + (b)$, and $((b))$ are in L(AE).

Infix notation permits—in fact, requires—the nesting of parentheses. The derivation

$$\begin{aligned} S &\Rightarrow T \\ &\Rightarrow (A) \\ &\Rightarrow (T) \\ &\Rightarrow (b) \end{aligned}$$

exhibits the generation of the string (b) using the rules of AE. Repeated applications of the sequence of rules $T \Rightarrow (A) \Rightarrow (T)$ before terminating the derivation with the application of the rule $T \rightarrow b$ generates the strings $((b))$, $(((b)))$, ... The strings $({}^i b$ and $)^i$ satisfy the requirements of the sequences u_i and v_i of Corollary 6.5.2. Thus the language defined by the grammar AE is not regular. A similar argument can be used to show that programming languages such as C, C++, and Java, among others, are not regular. □

Just as the closure properties of regular languages can be used to establish regularity, they can also be used to demonstrate the nonregularity of languages.

Example 6.5.3

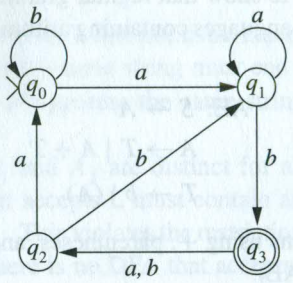
The language $\mathrm{L} = \{a^i b^j \mid i, j \geq 0 \text{ and } i \neq j\}$ is not regular. If L is regular then, by Theorems 6.4.2 and 6.4.3, so is $\overline{\mathrm{L}} \cap a^* b^*$. But $\overline{\mathrm{L}} \cap a^* b^* = \{a^i b^i \mid i \geq 0\}$, which we know is not regular. □

6.6 The Pumping Lemma for Regular Languages

The existence of nonregular languages was established in the previous section by demonstrating the impossibility of constructing a DFA to accept the language. In this section a more general criterion for establishing nonregularity is developed. The main result, the pumping lemma for regular languages, requires strings in a regular language to admit decompositions satisfying certain repetition properties.

Pumping a string refers to constructing new strings by repeating (pumping) substrings in the original string. Acceptance in the state diagram of the DFA

M:



illustrates pumping strings. Consider the string $z = ababbaaab$ in L(M). This string can be decomposed into substrings u, v, and w where $u = a$, $v = bab$, $w = baaab$, and $z = uvw$. The strings $a(bab)^i baaab$ are obtained by pumping the substring bab in $ababbaaab$.

As usual, processing z in the DFA M corresponds to generating a path in the state diagram of M. The decomposition of z into u, v, and w breaks the path in the state diagram into three subpaths. The subpaths generated by the computation of substrings $u = a$ and $w = baaab$ are q_0, q_1 and q_1, q_3, q_2, q_0, q_1, q_3. Processing the second component of the decomposition generates the cycle q_1, q_3, q_2, q_1. The pumped strings $uv^i w$ are also accepted by the DFA since the repetition of the substring v simply adds additional trips around the cycle q_1, q_3, q_2, q_1 before the processing of w terminates the computation in state q_3.

The pumping lemma requires the existence of such a decomposition for all sufficiently long strings in the language of a DFA. Two lemmas are presented establishing conditions guaranteeing the existence of cycles in paths in the state diagram of a DFA. The proofs utilize a simple counting argument known as the *pigeonhole principle*. This principle is based on the observation that given a number of boxes and a greater number of items to be distributed among them, at least one of the boxes must receive more than one item.

Lemma 6.6.1

Let G be the state diagram of a DFA with k states. Any path of length k in G contains a cycle.

Proof. A path of length k contains $k + 1$ nodes. Since there are only k nodes in G, there must be a node, call it q_i, that occurs in at least two positions in the path. The subpath from the first occurrence of q_i to the second produces the desired cycle. ■

Paths with length greater than k can be divided into an initial subpath of length k and the remainder of the path. Lemma 6.6.1 guarantees the existence of a cycle in the initial subpath. The preceding remarks are formalized in Corollary 6.6.2.

Corollary 6.6.2

Let G be the state diagram of a DFA with k states and let **p** be a path of length k or more. The path **p** can be decomposed into subpaths **q**, **r**, and **s** where **p** = **qrs**, the length of **qr** is less than or equal to k, and **r** is a cycle.

Theorem 6.6.3 (Pumping Lemma for Regular Languages)

Let L be a regular language that is accepted by a DFA M with k states. Let z be any string in L with $length(z) \geq k$. Then z can be written uvw with $length(uv) \leq k$, $length(v) > 0$, and $uv^i w \in$ L for all $i \geq 0$.

Proof. Let $z \in$ L be a string with length $n \geq k$. Processing z in M generates a path of length n in the state diagram of M. By Corollary 6.6.2, this path can be broken into subpaths **q**, **r**, and **s**, where **r** is a cycle in the state diagram. The decomposition of z into u, v, and w consists of the strings spelled by the paths **q**, **r**, and **s**. ■

The paths corresponding to the strings $uv^i w$ begin and end at the same nodes as the computation for uvw. The sole difference is the number of trips around the cycle **r**. Consequently, if uvw is accepted by M, then so is $uv^i w$.

Properties of the particular DFA that accepts the language L are not specifically mentioned in the proof of the pumping lemma. The argument holds for all such DFAs, including the DFA with the minimal number of states. The statement of the theorem could be strengthened to specify k as the number of states in the minimal DFA accepting L.

The pumping lemma is a powerful tool for proving that languages are not regular. Every string of length k or more in a regular language, where k is the value specified by the pumping lemma, must have an appropriate decomposition. To show that a language is not regular, it suffices to find one string that does not satisfy the conditions of the pumping lemma. The use of the pumping lemma to establish nonregularity is illustrated in the following examples. The technique consists of choosing a string z in L and showing that there is no decomposition uvw of z for which $uv^i w$ is in L for all $i \geq 0$.

The first two examples show that computations of a finite state machine are not sufficiently powerful to determine whether a number is a perfect square or a prime.

Example 6.6.1

Let L = $\{z \in \{a\}^* \mid length(z)$ is a perfect square$\}$. Assume that L is regular. This implies that L is accepted by some DFA. Let k be the number of states of the DFA. By the pumping lemma, every string $z \in$ L of length k or more can be decomposed into substrings u, v, and w such that $length(uv) \leq k$, $v \neq \lambda$, and $uv^i w \in$ L for all $i \geq 0$.

Consider the string $z = a^{k^2}$ of length k^2. Since z is in L and its length is greater than k, z can be written $z = uvw$ where the u, v, and w satisfy the conditions of the pumping lemma.

In particular, $0 < length(v) \leq k$. This observation can be used to place an upper bound on the length of uv^2w:

$$\begin{aligned} length(uv^2w) &= length(uvw) + length(v) \\ &= k^2 + length(v) \\ &\leq k^2 + k \\ &< k^2 + 2k + 1 \\ &= (k+1)^2. \end{aligned}$$

The length of uv^2w is greater than k^2 and less than $(k+1)^2$ and therefore is not a perfect square. Thus the string uv^2w obtained by pumping v once is not in L. We have shown that there is no decomposition of z that satisfies the conditions of the pumping lemma. The assumption that L is regular leads to a contradiction, establishing the nonregularity of L. □

Example 6.6.2

To show that the language $L = \{a^i \mid i \text{ is prime}\}$ is not regular, we assume that there is a DFA with some number k states that accepts it. Let n be a prime greater than k. The pumping lemma implies that a^n can be decomposed into substrings uvw, $v \neq \lambda$, such that uv^iw is in L for all $i \geq 0$. Assume that such a decomposition exists.

If $uv^{n+1}w \in L$, then its length must be prime. But

$$\begin{aligned} length(uv^{n+1}w) &= length(uvv^nw) \\ &= length(uvw) + length(v^n) \\ &= n + n(length(v)) \\ &= n(1 + length(v)). \end{aligned}$$

Since its length is not prime, $uv^{n+1}w$ is not in L. Thus there is no division of a^n into uvw that satisfies the pumping lemma and we conclude that L is not regular. □

In the preceding examples, the constraints on the length of the strings were sufficient to prove that the languages were not regular. Often the numeric relationships among the elements of a string are used to show that there is no substring that satisfies the conditions of the pumping lemma. We will now present another argument, this time using the pumping lemma, that demonstrates the nonregularity of $\{a^ib^i \mid i \geq 0\}$.

Example 6.6.3

To show that $L = \{a^ib^i \mid i \geq 0\}$ is not regular, we must find a string in L of appropriate length that has no pumpable substring. Assume that L is regular and let k be the number specified

by the pumping lemma. Let z be the string a^kb^k. Any decomposition of uvw of z satisfying the conditions of the pumping lemma must have the form

$$\begin{array}{lll} u & v & w \\ a^i & a^j & a^{k-i-j}b^k, \end{array}$$

where $i + j \leq k$ and $j > 0$. Pumping any substring of this form produces $uv^2w = a^ia^ja^ja^{k-i-j}b^k = a^ka^jb^k$, which is not in L. Since $z \in L$ has no decomposition that satisfies the conditions of the pumping lemma, we conclude that L is not regular. □

Example 6.6.4

The language $L = \{a^ib^mc^n \mid 0 < i,\ 0 < m < n\}$ is not regular. Assume that L is accepted by a DFA with k states. Then, by the pumping lemma, every string $z \in L$ with length k or more can be written $z = uvw$, with $length(uv) \leq k$, $length(v) > 0$, and $uv^iw \in L$ for all $i \geq 0$.

Consider the string $z = ab^kc^{k+1}$, which is in L. We must show that there is no suitable decomposition of z. Any decomposition of z must have one of two forms, and the cases are examined separately.

Case 1: A decomposition in which $a \notin v$ has the form

$$\begin{array}{lll} u & v & w \\ ab^i & b^j & b^{k-i-j}c^{k+1} \end{array}$$

where $i + j \leq k - 1$ and $j > 0$. Pumping v produces $uv^2w = ab^ib^jb^jb^{k-i-j}c^{k+1} = ab^kb^jc^{k+1}$, which is not in L.

Case 2: A decomposition of z in which $a \in v$ has the form

$$\begin{array}{lll} u & v & w \\ \lambda & ab^i & b^{k-i}c^{k+1} \end{array}$$

where $i \leq k - 1$. Pumping v zero times produces $uv^0w = b^{k-i}c^{k+1}$, which is not in L since it does not contain an a.

Since ab^kc^{k+1} has no decomposition with a "pumpable" substring, L is not regular. □

The pumping lemma can be used to determine the size of the language accepted by a DFA. Pumping a string generates an infinite sequence of strings that are accepted by the DFA. To determine whether a regular language is finite or infinite it is only necessary to determine if it contains a pumpable string.

Theorem 6.6.4

Let M be a DFA with k states.

i) L(M) is not empty if, and only if, M accepts a string z with $length(z) < k$.

ii) L(M) has an infinite number of members if, and only if, M accepts a string z where $k \leq length(z) < 2k$.

Proof.

i) L(M) is clearly not empty if a string of length less than k is accepted by M.

Now let M be a machine whose language is not empty and let z be the smallest string in L(M). Assume that the length of z is greater than $k-1$. By the pumping lemma, z can be written uvw where $uv^iw \in L$. In particular, $uv^0w = uw$ is a string smaller than z in L. This contradicts the assumption of the minimality of the length of z. Therefore, $length(z) < k$.

ii) If M accepts a string z with $k \leq length(z) < 2k$, then z can be written uvw where u, v, and w satisfy the conditions of the pumping lemma. This implies that the strings uv^iw are in L for all $i \geq 0$.

Assume that L(M) is infinite. We must show that there is a string whose length is between k and $2k-1$ in L(M). Since there are only finitely many strings over a finite alphabet with length less than k, L(M) must contain strings of length greater than $k-1$. Choose a string $z \in L(M)$ whose length is as small as possible but greater than $k-1$. If $k \leq length(z) < 2k$, there is nothing left to show. Assume that $length(z) \geq 2k$. By the pumping lemma, $z = uvw$, $length(v) \leq k$, and $uv^0w = uw \in L(M)$. But this is a contradiction since uw is a string whose length is greater than $k-1$ but strictly smaller than the length of z. ∎

The preceding result establishes a decision procedure for determining the cardinality of the language of a DFA. If k is the number of states and j the size of the alphabet of the automaton, there are $(j^k - 1)/(j-1)$ strings having length less than k. By Theorem 6.6.4, testing each of these determines whether the language is empty. Testing all strings with length between k and $2k-1$ resolves the question of finite or infinite. This, of course, is an extremely inefficient procedure. Nevertheless, it is effective, yielding the following corollary.

Corollary 6.6.5

Let M be a DFA. There is an algorithm that determines whether L(M) is empty, finite, or infinite.

The closure properties of regular language can be combined with Corollary 6.6.5 to develop a decision procedure that determines whether two DFAs accept the same language.

Corollary 6.6.6

Let M_1 and M_2 be two DFAs. There is a decision procedure to determine whether M_1 and M_2 are equivalent.

Proof. Let L_1 and L_2 be the languages accepted by M_1 and M_2. By Theorems 6.4.1, 6.4.2, and 6.4.3, the language

$$L = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$$

is regular. L is empty if, and only if, L_1 and L_2 are identical. By Corollary 6.6.5, there is a decision procedure to determine whether L is empty, or equivalently, whether M_1 and M_2 accept the same language. ∎

6.7 The Myhill-Nerode Theorem

Kleene's Theorem established the relationship between regular languages and finite automata. In this section regularity is characterized by the existence of an equivalence relation on the strings of the language. This characterization provides a method for obtaining the minimal state DFA that accepts a regular language and provides the justification for the DFA minimization presented in Algorithm 5.7.2.

Definition 6.7.1

Let L be a language over Σ. Strings $u, v \in \Sigma^*$ are indistinguishable in L if, for every $w \in \Sigma^*$, either uw and vw are both in L or neither uw nor vw is in L.

Using membership in L as the criterion for differentiating strings, u and v are distinguishable if there is some string w whose concatenation with u and v produces strings with different membership values in L. That is, w distinguishes u and v if one of uw and vw is in L and the other is not.

Indistinguishability in a language L defines a binary relation $\equiv_L$ on Σ^*; $u \equiv_L v$ if u and v are indistinguishable. It is easy to see that $\equiv_L$ is reflexive, symmetric, and transitive. These observations provide the basis for Lemma 6.7.2.

Lemma 6.7.2

For any language L, the relation $\equiv_L$ is an equivalence relation.

Example 6.7.1

Let L be the regular language $a(a \cup b)(bb)^*$. Strings aa and ab are indistinguishable since, for any w, aaw and abw are either both in L or both not in L. The former arises when w consists of an even number of b's and the latter for any other string. The pair of strings b and ba are also indistinguishable in L since bw and baw are not in L for any string w. Strings a and ab are distinguishable in L since concatenating bb to a produces $abb \notin L$ and to ab produces $abbb \in L$.

The equivalence classes of $\equiv_L$ are

Representative Element	Equivalence Class
$[\lambda]_{\equiv_L}$	λ
$[b]_{\equiv_L}$	$b(a \cup b)^* \cup a(a \cup b)(bb)^*a(a \cup b)^* \cup a(a \cup b)(bb)^*ba(a \cup b)^*$
$[a]_{\equiv_L}$	a
$[aa]_{\equiv_L}$	$a(a \cup b)(bb)^*$
$[aab]_{\equiv_L}$	$a(a \cup b)b(bb)^*$

□

Example 6.7.2

Let L be the language $\{a^i b^i \mid i \geq 0\}$. The strings a^i and a^j, where $i \neq j$, are distinguishable in L. Concatenating b^i produces $a^i b^i \in$ L and $a^j b^i \notin$ L. Thus each string a^i, $i = 0, 1, \ldots$, is in a different equivalence class. This example shows that the indistinguishability relation $\equiv_L$ may generate infinitely many equivalence classes. □

The equivalence relation $\equiv_L$ defines indistinguishability on the basis of membership in the language L. We now define the indistinguishability of strings on the basis of computations of a DFA.

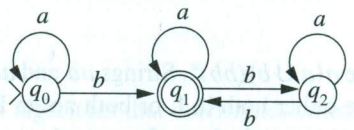
Definition 6.7.3

Let M = (Q, Σ, δ, q_0, F) be a DFA that accepts L. Strings $u, v \in \Sigma^*$ are indistinguishable by M if $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$.

Strings u and v are indistinguishable by M if the computation of M with input u halts in the same state as the computation with v. It is easy to see that indistinguishability defined in this manner is also an equivalence relation over Σ^*. Each state q_i of M that is reachable by computations of M has an associated equivalence class: the set of all strings whose computations halt in q_i. Thus the number of equivalence classes of a DFA M is at most the number of states of M. Indistinguishability by a machine M will be denoted $\equiv_M$.

Example 6.7.3

Let M be the DFA



that accepts the language $a^*ba^*(ba^*ba^*)^*$, the set of strings with an odd number of b's. The equivalence classes of Σ^* defined by the relation $\equiv_M$ are

State	Associated Equivalence Class
q_0	a^*
q_1	$a^*ba^*(ba^*ba^*)^*$
q_2	$a^*ba^*ba^*(ba^*ba^*)^*$

□

Indistinguishability relations can be used to provide additional characterizations of regularity. These characterizations use the *right-invariance* of the indistinguishability equivalence relations. An equivalence relation ≡ over Σ^* is said to be right-invariant if $u \equiv v$ implies $uw \equiv vw$ for every $w \in \Sigma^*$. Both $\equiv_L$ and $\equiv_M$ are right-invariant.

Theorem 6.7.4 (Myhill-Nerode)

The following are equivalent:

i) L is regular over Σ.

ii) There is a right-invariant equivalence relation ≡ on Σ^* with finitely many equivalence classes such that L is the union of a subset of the equivalences classes of ≡.

iii) $\equiv_L$ has finitely many equivalence classes.

Proof.

Condition (i) implies condition (ii): Since L is regular, it is accepted by some DFA M = (Q, Σ, δ, q_0, F). We will show that $\equiv_M$ satisfies the conditions of statement (ii). As previously noted, $\equiv_M$ has at most as many equivalence classes as M has states. Consequently, the number of equivalence classes of $\equiv_M$ is finite. Right-invariance follows from the determinism of the computations of M, which ensures that $\hat{\delta}(q_0, uw) = \hat{\delta}(q_0, vw)$ whenever $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$.

It remains to show that L is the union of some of the equivalence classes of $\equiv_M$. For each state q_i of M, there is an equivalence class consisting of the strings whose computations halt in q_i. The language L is the union of the equivalence classes associated with the accepting states of M.

Condition (ii) implies condition (iii): Let ≡ be an equivalence relation that satisfies (ii). We begin by showing that every ≡ equivalence class $[u]_{\equiv}$ is a subset of the $\equiv_L$ equivalence class $[u]_{\equiv_L}$.

Let u and v be any two strings from $[u]_{\equiv}$; that is, $u \equiv v$. By right-invariance, $uw \equiv vw$ for any $w \in \Sigma^*$. Thus uw and vw are in the same ≡ equivalence class. Since L is the union of some set of equivalence classes of ≡, every string in a particular ≡ equivalence class has the same membership value in L. Consequently, uw and vw are either both in L or both not in L. It follows that u and v are in the same equivalence class of $\equiv_L$.

Since $[u]_{\equiv} \subseteq [u]_{\equiv_L}$ for every string $u \in \Sigma^*$, there is at least one ≡ equivalence class in each of the $\equiv_L$ equivalence classes. It follows that the number of equivalence classes of $\equiv_L$ is no greater than the number of equivalence classes of ≡, which is finite.

Condition (iii) implies condition (i): To prove that L is regular when $\equiv_L$ has only a finite number of equivalence classes, we will build a DFA M_L that accepts L. The alphabet of M_L consists of the symbols in L and the states are the equivalence classes of $\equiv_L$. The start state is the equivalence class containing λ. An equivalence class is an accepting state if it contains an element $u \in$ L. All that remains is to define the transition function and show that the language of M_L is L.

For a symbol $a \in \Sigma$, we define $\delta([u]_{\equiv_L}, a) = [ua]_{\equiv_L}$. By this definition, the result of a transition from state $[u]_{\equiv_L}$ with symbol a is the equivalence class $[ua]_{\equiv_L}$. We must show that the definition of the transition is independent of the choice of a particular element from the equivalence class $[u]_{\equiv_L}$.

Let u and v be two strings in that are $\equiv_L$ equivalent. For the transition function δ to be well defined, $[ua]_{\equiv_L}$ must be the same equivalence class as $[va]_{\equiv_L}$, or equivalently,

$ua \equiv_L va$. To establish this, we need to show that for any string $x \in \Sigma^*$, uax and vax are either both in L or both not in L. By the definition of $\equiv_L$, uw and vw are both in L or both not in L for any $w \in \Sigma^*$. Letting $w = ax$ gives the desired result.

All that remains is to show that $L(M_L) = L$. For any string u, $\hat{\delta}([\lambda]_{\equiv_L}, u) = [u]_{\equiv_L}$. If u is in L, the computation $\hat{\delta}([\lambda]_{\equiv_L}, u)$ halts in the accepting state $[u]_{\equiv_L}$. Exercise 25 shows that either all of the elements in an equivalence $[u]_{\equiv_L}$ are in L or none of the elements are in L. Thus if $u \notin L$, then $[u]_{\equiv_L}$ is not an accepting state. It follows that a string u is accepted by M_L if, and only if, $u \in L$.

Note that the equivalence classes of $\equiv_L$ are precisely those of $\equiv_{M_L}$, the indistinguishability relation over Σ^* generated by the machine M_L. ■

Example 6.7.4

The DFA M from Example 5.7.1 accepts the language $(a \cup b)(a \cup b^*)$. The eight equivalence classes of the relation $\equiv_M$ with the associated states of M are

State	Equivalence Class	State	Equivalence Class
q_0	λ	q_4	b
q_1	a	q_5	ba
q_2	aa	q_6	bb^+
q_3	ab^+	q_7	$(aa(a \cup b) \cup ab^+a \cup ba(a \cup b) \cup bb^+a)(a \cup b)^*$

The equivalence relation $\equiv_L$ identifies strings u and v as indistinguishable if for any w, either both uw and vw are in L or both are not in L. The $\equiv_L$ equivalence classes of the language $(a \cup b)(a \cup b^*)$ are

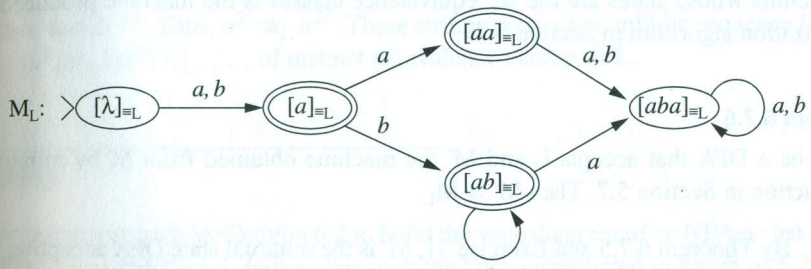
$\equiv_L$ Equivalence Classes	
$[\lambda]_{\equiv_L}$	λ
$[a]_{\equiv_L}$	$a \cup b$
$[aa]_{\equiv_L}$	$aa \cup ba$
$[ab]_{\equiv_L}$	$ab^+ \cup bb^+$
$[aba]_{\equiv_L}$	$(aa(a \cup b) \cup ab^+a \cup ba(a \cup b) \cup bb^+a)(a \cup b)^*$

where the string inside the brackets is a representative element of the class. It is easy to see that the strings within an equivalence class are indistinguishable and that strings from different classes are distinguishable.

If we denote the $\equiv_M$ equivalence class of strings whose computations halt in state q_i by $cl_M(q_i)$, the relationship between the equivalence classes of $\equiv_L$ and $\equiv_M$ is

$$
\begin{aligned}
[\lambda]_{\equiv_L} &= cl_M(q_0) \\
[a]_{\equiv_L} &= cl_M(q_1) \cup cl_M(q_4) \\
[aa]_{\equiv_L} &= cl_M(q_2) \cup cl_M(q_5) \\
[ab]_{\equiv_L} &= cl_M(q_3) \cup cl_M(q_6) \\
[aba]_{\equiv_L} &= cl_M(q_7).
\end{aligned}
$$

Using the technique outlined in the Myhill-Nerode Theorem, we can construct a DFA M_L accepting L from the equivalence classes of $\equiv_L$. The DFA obtained by this construction is



which is identical to the DFA M′ in Example 5.7.1 obtained using the minimization technique presented in Section 5.7. □

Theorem 6.7.5 shows that the DFA M_L obtained from the $\equiv_L$ equivalence classes is the minimal state DFA that accepts L.

Theorem 6.7.5

Let L be a regular language and $\equiv_L$ the indistinguishability relation defined by L. The minimal state DFA accepting L is the machine M_L defined from the equivalence classes of $\equiv_L$ as specified in Theorem 6.7.4.

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be any DFA that accepts L and let $\equiv_M$ be the equivalence relation generated by M. By the Myhill-Nerode Theorem, each equivalence class of $\equiv_M$ is a subset of an equivalence class of $\equiv_L$. Since the equivalence classes of both $\equiv_M$ and $\equiv_L$ partition Σ^*, $\equiv_M$ must have at least as many equivalence classes as $\equiv_L$. Combining the preceding observation with the construction of M_L from the equivalence classes of $\equiv_L$, we see that

$$
\begin{aligned}
&\text{the number of states of M} \\
\geq\ &\text{the number of equivalence classes of } \equiv_M \\
\geq\ &\text{the number of equivalence classes of } \equiv_L \\
=\ &\text{the number of states of } M_L.
\end{aligned}
$$

Thus a DFA M that accepts L may not have fewer states than M_L, and we conclude that M_L is the minimal state DFA that accepts L. ■

The statement of Theorem 6.7.5 asserts that the M_L is *the* minimal state DFA that accepts L. Exercise 31 establishes that all minimal state DFAs accepting L are identical to M_L, except possibly for the names assigned to the states.

Theorems 6.7.4 and 6.7.5 establish the existence of a unique minimal state DFA M_L that accepts a language L. The minimal state machine can be constructed from the equivalence classes of the relation $\equiv_L$. Unfortunately, to this point we have not provided a straightforward method to obtain these equivalence classes. Theorem 6.7.6 shows that the machine whose states are the $\equiv_L$ equivalence classes is the machine produced by the minimization algorithm in Section 5.7.

Theorem 6.7.6

Let M be a DFA that accepts L and M′ the machine obtained from M by minimization construction in Section 5.7. Then $M' = M_L$.

Proof. By Theorem 6.7.5 and Exercise 31, M′ is the minimal state DFA accepting L if the number of states of M′ is the same as the number of equivalence classes of $\equiv_L$. Following Definition 6.7.3, there is an equivalence relation $\equiv_{M'}$ that associates a set of strings with each state of M′. The equivalence class of $\equiv_{M'}$ associated with state $[q_i]$ is

$$cl_{M'}([q_i]) = \{u \mid \hat{\delta}'([q_0], u) = [q_i]\} = \bigcup_{q_j \in [q_i]} \{u \mid \hat{\delta}(q_0, u) = q_j\},$$

where $\hat{\delta}'$ and $\hat{\delta}$ are the extended transition functions of M′ and M, respectively. By the Myhill-Nerode Theorem, $cl_{M'}([q_i])$ is a subset of an equivalence class of $\equiv_{M_L}$.

Assume that the number of states of M′ is greater than the number of equivalence classes of $\equiv_L$. Then there are two states $[q_i]$ and $[q_j]$ of M′ such that $cl_{M'}([q_i])$ and $cl_{M'}([q_j])$ are both subsets of the same equivalence class of $\equiv_L$. This implies that there are strings u and v such that $\hat{\delta}(q_0, u) = q_i$, $\hat{\delta}(q_0, v) = q_j$, and $u \equiv_L v$.

Since $[q_i]$ and $[q_j]$ are distinct states in M′, there is a string w that distinguishes these states. That is, either $\hat{\delta}(q_i, w)$ is accepting and $\hat{\delta}(q_j, w)$ is nonaccepting or vice versa. It follows that uw and vw have different membership values in L. This is a contradiction since $u \equiv_L v$ implies that uw and vw have the same membership value in L for all strings w. Consequently, the assumption that the number of states of M′ is greater than the number of equivalence classes of $\equiv_L$ must be false. ■

The characterization of regularity in the Myhill-Nerode Theorem gives another method for establishing the nonregularity of a language. A language L is not regular if the equiva- [illegible] has infinitely many equivalence classes.

Example 6.7.5

In Example 6.7.2, it was shown that the language $\{a^i b^i \mid i \geq 0\}$ has infinitely many $\equiv_L$ equivalence classes and therefore is not regular. □

Example 6.7.6

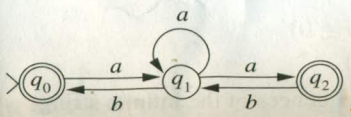
The Myhill-Nerode Theorem will be used to show that the language $L = \{a^{2^i} \mid i \geq 0\}$ is not regular. To accomplish this, we show that a^{2^i} and a^{2^j} are distinguishable by the $\equiv_L$ equivalence relation whenever $i < j$. Concatenating a^{2^i} with each of these strings produces $a^{2^i}a^{2^i} = a^{2^{i+1}} \in L$ and $a^{2^j}a^{2^i} \notin L$. The latter string is not in L since it has length greater than 2^j but less than 2^{j+1}. Thus, $a^{2^i} \not\equiv_L a^{2^j}$. These strings produce an infinite sequence $[a^0] \equiv_L$, $[a^1] \equiv_L$, $[a^2] \equiv_L$, $[a^4] \equiv_L$, . . . of distinct equivalence classes of L. □

Exercises

1. Use the technique from Section 6.2 to build the state diagram of an NFA-λ that accepts the language $(ab)^*ba$. Compare this with the DFA constructed in Exercise 5.22(a).
2. For each of the state diagrams in Exercise 5.40, use Algorithm 6.2.2 to construct a regular expression for the language accepted by the automaton.
3. The language of the DFA M in Example 5.37 consists of all strings over $\{a, b\}$ with an even number of a's and an odd number of b's. Use Algorithm 6.2.2 to construct a regular expression for L(M). Exercise 2.38 requested a nonalgorithmic construction of a regular expression for this language, which, as you now see, is a formidable task.
4. Let G be the grammar

$$G: \quad S \rightarrow aS \mid bA \mid a$$
$$A \rightarrow aS \mid bA \mid b.$$

   a) Use Theorem 6.3.1 to build an NFA M that accepts L(G).
   b) Using the result of part (a), build a DFA M′ that accepts L(G).
   c) Construct a regular grammar from M that generates L(M).
   d) Construct a regular grammar from M′ that generates L(M′).
   e) Give a regular expression for L(G).
5. Let M be the NFA



Thus a DFA M that accepts L may not have fewer states than M_L, and we conclude that M_L is the minimal state DFA that accepts L. ■

The statement of Theorem 6.7.5 asserts that the M_L is *the* minimal state DFA that accepts L. Exercise 31 establishes that all minimal state DFAs accepting L are identical to M_L, except possibly for the names assigned to the states.

Theorems 6.7.4 and 6.7.5 establish the existence of a unique minimal state DFA M_L that accepts a language L. The minimal state machine can be constructed from the equivalence classes of the relation $\equiv_L$. Unfortunately, to this point we have not provided a straightforward method to obtain these equivalence classes. Theorem 6.7.6 shows that the machine whose states are the $\equiv_L$ equivalence classes is the machine produced by the minimization algorithm in Section 5.7.

Theorem 6.7.6

Let M be a DFA that accepts L and M′ the machine obtained from M by minimization construction in Section 5.7. Then $M' = M_L$.

Proof. By Theorem 6.7.5 and Exercise 31, M′ is the minimal state DFA accepting L if the number of states of M′ is the same as the number of equivalence classes of $\equiv_L$. Following Definition 6.7.3, there is an equivalence relation $\equiv_{M'}$ that associates a set of strings with each state of M′. The equivalence class of $\equiv_{M'}$ associated with state $[q_i]$ is

$$cl_{M'}([q_i]) = \{u \mid \hat{\delta}'([q_0], u) = [q_i]\} = \bigcup_{q_j \in [q_i]} \{u \mid \hat{\delta}(q_0, u) = q_j\},$$

where $\hat{\delta}'$ and $\hat{\delta}$ are the extended transition functions of M′ and M, respectively. By the Myhill-Nerode Theorem, $cl_{M'}([q_i])$ is a subset of an equivalence class of $\equiv_{M_L}$.

Assume that the number of states of M′ is greater than the number of equivalence classes of $\equiv_L$. Then there are two states $[q_i]$ and $[q_j]$ of M′ such that $cl_{M'}([q_i])$ and $cl_{M'}([q_j])$ are both subsets of the same equivalence class of $\equiv_L$. This implies that there are strings u and v such that $\hat{\delta}(q_0, u) = q_i$, $\hat{\delta}(q_0, v) = q_j$, and $u \equiv_L v$.

Since $[q_i]$ and $[q_j]$ are distinct states in M′, there is a string w that distinguishes these states. That is, either $\hat{\delta}(q_i, w)$ is accepting and $\hat{\delta}(q_j, w)$ is nonaccepting or vice versa. It follows that uw and vw have different membership values in L. This is a contradiction since $u \equiv_L v$ implies that uw and vw have the same membership value in L for all strings w. Consequently, the assumption that the number of states of M′ is greater than the number of equivalence classes of $\equiv_L$ must be false. ■

The characterization of regularity in the Myhill-Nerode Theorem gives another method for establishing the nonregularity of a language. A language L is not regular if the equivalence relation $\equiv_L$ has infinitely many equivalence classes.

Example 6.7.5

In Example 6.7.2, it was shown that the language $\{a^i b^i \mid i \geq 0\}$ has infinitely many $\equiv_L$ equivalence classes and therefore is not regular. □

Example 6.7.6

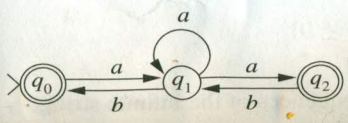
The Myhill-Nerode Theorem will be used to show that the language $L = \{a^{2^i} \mid i \geq 0\}$ is not regular. To accomplish this, we show that a^{2^i} and a^{2^j} are distinguishable by the $\equiv_L$ equivalence relation whenever $i < j$. Concatenating a^{2^i} with each of these strings produces $a^{2^i}a^{2^i} = a^{2^{i+1}} \in L$ and $a^{2^j}a^{2^i} \notin L$. The latter string is not in L since it has length greater than 2^j but less than 2^{j+1}. Thus, $a^{2^i} \not\equiv_L a^{2^j}$. These strings produce an infinite sequence $[a^0] \equiv_L$, $[a^1] \equiv_L$, $[a^2] \equiv_L$, $[a^4] \equiv_L$, . . . of distinct equivalence classes of L. □

Exercises

1. Use the technique from Section 6.2 to build the state diagram of an NFA-λ that accepts the language $(ab)^*ba$. Compare this with the DFA constructed in Exercise 5.22(a).
2. For each of the state diagrams in Exercise 5.40, use Algorithm 6.2.2 to construct a regular expression for the language accepted by the automaton.
3. The language of the DFA M in Example 5.37 consists of all strings over $\{a, b\}$ with an even number of a's and an odd number of b's. Use Algorithm 6.2.2 to construct a regular expression for L(M). Exercise 2.38 requested a nonalgorithmic construction of a regular expression for this language, which, as you now see, is a formidable task.
4. Let G be the grammar

$$G: \; S \to aS \mid bA \mid a$$
$$A \to aS \mid bA \mid b.$$

   a) Use Theorem 6.3.1 to build an NFA M that accepts L(G).
   b) Using the result of part (a), build a DFA M′ that accepts L(G).
   c) Construct a regular grammar from M that generates L(M).
   d) Construct a regular grammar from M′ that generates L(M′).
   e) Give a regular expression for L(G).
5. Let M be the NFA



a) Construct a regular grammar from M that generates L(M).

b) Give a regular expression for L(M).

*6. Let G be a regular grammar and M the NFA obtained from G according to Theorem 6.3.1. Prove that if $S \stackrel{*}{\Rightarrow} wC$, then there is a computation $[S, w] \vdash^* [C, \lambda]$ in M.

7. Let L be a regular language over $\{a, b, c\}$. Show that each of the following sets is regular.

a) $\{w \mid w \in L \text{ and } w \text{ ends with } aa\}$

b) $\{w \mid w \in L \text{ or } w \text{ contains an } a\}$

c) $\{w \mid w \notin L \text{ and } w \text{ does not contain an } a\}$

d) $\{uv \mid u \in L \text{ and } v \notin L\}$

8. Prove that the family of regular languages is closed under the operation of set difference.

9. Prove that the family of regular languages is not closed under intersection with context-free languages. That is, if L is regular and L_1 context-free, $L \cap L_1$ need not be regular.

10. Is the family of regular languages closed under infinite unions? That is, if $L_0, L_1, L_2, \ldots$ are regular, is $\bigcup_{i=0}^{\infty} L_i$ necessarily regular? If so, prove it. If not, give a counterexample.

11. Let L be a regular language. Show that the following languages are regular.

a) The set $P = \{u \mid uv \in L\}$ of prefixes of strings in L.

b) The set $L^R = \{w^R \mid w \in L\}$ of reversals of strings in L.

c) The set $E = \{uv \mid v \in L\}$ of strings that have a suffix in L.

d) The set $SUB = \{v \mid uvw \in L\}$ of strings that are substrings of a string in L.

12. Let L be a regular language containing only strings of even length. Let L′ be the language $\{u \mid uv \in L \text{ and } length(u) = length(v)\}$. L′ is the set of all strings that contain the first half of strings from L. Prove that L′ is regular.

13. Use Corollary 6.5.2 to show that each of the following sets is not regular.

a) The set of strings over $\{a, b\}$ with the same number of a's and b's.

b) The set of palindromes of even length over $\{a, b\}$.

c) The set of strings over $\{(,)\}$ in which the parentheses are paired, for example, λ, ( ), ( )( ), (( ))( ).

d) The language $\{a^i(ab)^j(ca)^{2i} \mid i, j > 0\}$.

14. Use the pumping lemma to show that each of the following sets is not regular.

a) The set of palindromes over $\{a, b\}$

b) $\{a^n b^m \mid n < m\}$

c) $\{a^i b^j c^{2j} \mid i \geq 0,\ j \geq 0\}$

d) $\{ww^R \mid w \in \{a, b\}^*\}$

*e) The set of initial sequences of the infinite string

$$abaabaaabaaaab \ldots ba^n ba^{n+1} b \ldots$$

f) The set of strings over $\{a, b\}$ in which the number of a's is a perfect cube

15. Prove that the set of nonpalindromes over $\{a, b\}$ is not a regular language.

16. Let L be a regular language and let $L_1 = \{uu \mid u \in L\}$ be the language L "doubled." Is L_1 necessarily regular? Prove your answer.

17. Let L_1 be a nonregular language and L_2 an arbitrary finite language.

a) Prove that $L_1 \cup L_2$ is nonregular.

b) Prove that $L_1 - L_2$ is nonregular.

c) Show that the conclusions of parts (a) and (b) are not true if L_2 is not assumed to be finite.

18. Give examples of languages L_1 and L_2 over $\{a, b\}$ that satisfy the following descriptions.

a) L_1 is regular, L_2 is nonregular, and $L_1 \cup L_2$ is regular.

b) L_1 is regular, L_2 is nonregular, and $L_1 \cup L_2$ is nonregular.

c) L_1 is regular, L_2 is nonregular, and $L_1 \cap L_2$ is regular.

d) L_1 is nonregular, L_2 is nonregular, and $L_1 \cup L_2$ is regular.

e) L_1 is nonregular and L_1^* is regular.

*19. Let Σ_1 and Σ_2 be two alphabets. A **string homomorphism** is a total function h from Σ_1^* to Σ_2^* that preserves concatenation. That is, h satisfies

i) $h(\lambda) = \lambda$

ii) $h(uv) = h(u)h(v)$.

a) Let $L_1 \subseteq \Sigma_1^*$ be a regular language. Show that the set $\{h(w) \mid w \in L_1\}$ is regular over Σ_2. This set is called the *homomorphic image* of L_1 under h.

b) Let $L_2 \subseteq \Sigma_2^*$ be a regular language. Show that the set $\{w \in \Sigma_1^* \mid h(w) \in L_2\}$ is regular. This set is called the *inverse image* of L_2 under h.

20. A context-free grammar $G = (V, \Sigma, P, S)$ is called **right-linear** if each rule is of the form

i) $A \to u$, or

ii) $A \to uB$,

where $A, B \in V$, and $u \in \Sigma^*$. Use the techniques from Section 6.3 to show that the right-linear grammars generate precisely the regular sets.

*21. A context-free grammar $G = (V, \Sigma, P, S)$ is called **left-regular** if each rule is of the form

i) $A \to \lambda$,

ii) $A \to a$, or

iii) $A \to Ba$,

where $A, B \in V$, and $a \in \Sigma$.

a) Design an algorithm to construct an NFA that accepts the language of a left-regular grammar.

b) Show that the left-regular grammars generate precisely the regular sets.

22. A context-free grammar $G = (V, \Sigma, P, S)$ is called **left-linear** if each rule is of the form

    i) $A \rightarrow u$, or

    ii) $A \rightarrow Bu$,

    where $A, B \in V$, and $u \in \Sigma^*$. Show that the left-linear grammars generate precisely the regular sets.

23. Give a regular language L such that $\equiv_L$ has exactly three equivalence classes.

24. Give the $\equiv_L$ equivalence classes of the language a^+b^+.

25. Let $[u]_{\equiv_L}$ be a $\equiv_L$ equivalence class of a language L. Show that if $[u]_{\equiv_L}$ contains one string $v \in L$, then every string in $[u]_{\equiv_L}$ is in L.

26. Prove that $\equiv_L$ is right-invariant for any regular language L. That is, if $u \equiv_L v$, then $ux \equiv_L vx$ for any $x \in \Sigma^*$, where Σ is the alphabet of the language L.

27. Use the Myhill-Nerode Theorem to prove that the language $\{a^i \mid i \text{ is a perfect square}\}$ is not regular.

28. Let $u \in [ab]_{\equiv_M}$ and $v \in [aba]_{\equiv_M}$ be strings from the equivalence classes of $(a \cup b)(a \cup b^*)$ defined in Example 6.7.4. Show that u and v are distinguishable.

29. Give the equivalence classes defined by the relation $\equiv_M$ for the DFA in Example 5.3.1.

30. Give the equivalence classes defined by the relation $\equiv_M$ for the DFA in Example 5.3.3.

*31. Let M_L be the minimal state DFA that accepts a language L defined in Theorems 6.7.4 and 6.7.5. Let M be another DFA that accepts L with the same number of states as M_L. Prove that M_L and M are identical except (possibly) for the names assigned to the states. Two such DFAs are said to be *isomorphic*.

Bibliographic Notes

The equivalence of regular sets and languages accepted by finite automata was established by Kleene [1956]. The proof given in Section 6.2 is modeled after that of McNaughton and Yamada [1960]. Chomsky and Miller [1958] established the equivalence of the languages generated by regular grammars and accepted by finite automata. Closure under homomorphisms (Exercise 19) is from Ginsburg and Rose [1963b]. The closure of regular sets under reversal was noted by Rabin and Scott [1959]. Additional closure results for regular sets can be found in Bar-Hillel, Perles, and Shamir [1961], Ginsburg and Rose [1963b], and Ginsburg [1966]. The pumping lemma for regular languages is from Bar-Hillel, Perles, and Shamir [1961]. The relationship between the number of equivalence classes of a language and regularity was established in Myhill [1957] and Nerode [1958].

CHAPTER 7

Pushdown Automata and Context-Free Languages

Regular languages have been characterized as the languages generated by regular grammars and accepted by finite automata. This chapter presents a class of machines, the pushdown automata, that accepts the context-free languages. A pushdown automaton is a finite-state machine augmented with an external stack memory. The addition of a stack provides the pushdown automaton with a last-in, first-out memory management capability. The combination of stack and states overcomes the memory limitations that prevented the acceptance of the language $\{a^i b^i \mid i \geq 0\}$ by a deterministic finite automaton.

As with regular languages, a pumping lemma for context-free languages ensures the existence of repeatable substrings in strings of a context-free language. The pumping lemma provides a technique for showing that many easily definable languages are not context-free.

7.1 Pushdown Automata

Theorem 6.5.1 established that the language $\{a^i b^i \mid i \geq 0\}$ is not accepted by any finite automaton. To accept this language, a machine needs the ability to record the processing of any finite number of a's. The restriction of having finitely many states does not allow the automaton to "remember" the number of leading a's in an arbitrary input string. A new type of automaton is constructed that augments the state-input transitions of a finite automaton with the ability to utilize unlimited memory.

A pushdown stack, or simply a stack, is added to a finite automaton to construct a new machine known as a pushdown automaton (PDA). Stack operations affect only the top item of the stack; a pop removes the top element from the stack and a push places an element