Chapter 5

Finite Automata

This chapter introduces the notions of deterministic and nondeterministic finite automata and explores connections among several classes of languages: Regular, DFA-Acceptable, and NFA-acceptable. Along the way, we describe an important algorithm for converting an NFA into an equivalent DFA; that is a DFA that accepts the very same language as the NFA. We shall also derive a very powerful tool, commonly known as the *Pumping Lemma*, which will allow us to prove that certain languages are inherently too complex for any DFA or NFA to recognize.

5.1 Deterministic Finite Automata

An *automaton*[1] is a machine that behaves according to a precise set of instructions. A *finite automaton*[2] is a mathematical abstraction of a specific type of simple machine; one with extremely limited capabilities, but with many theoretical and practical applications. We shall first study the *deterministic finite automaton*, or DFA, in which every action of the machine is uniquely determined, and then the non-deterministic version, or NFA, in which some actions of the machine may be chosen arbitrarily.

Finite automata[3] are machines that can *recognize* languages possessing relatively simple patterns. The machine "reads" the symbols of a string, which we imagine to be printed on a "tape" that extends infinitely to the right. We refer to each location on the tape where a symbol can be written as a "cell." All the cells following the last symbol of the string are considered to be "blank." See Figure 5.1. Beginning in its *initial state*, or *start state*, the machine reads the tape from the left-most symbol, moving a *read head* along it in discrete steps, one symbol at a time. Every time the machine reads a symbol, two things happen simultaneously:

1. The *state* of the machine changes based on its current state and the symbol that was just read.

[1] Automaton, pronounced *aw-TOM-uh-TAWN*, is the singular form of automata.
[2] Another common name for a finite automaton is a *finite state machine*.
[3] The plural of automaton is pronounced *aw-TOM-uh-tuh*, not *auto-MAYTA*. There is no "auto" is automata.

2. The read head moves to the right, so that it is positioned over the next symbol on the tape, or the first blank cell following the string if the last symbol of the string has just been read.

This process continues until the last symbol of the string has been read, at which point the read head is positioned over the "blank" cell just past the end of the string. The state that the machine is left in at this point can be used to classify the string; in particular, the machine distinguishes between strings that leave the machine in an accept state and those that do not. The deterministic version of such a machine is formally specified by the 5-tuple $(\Sigma, Q, q_0, A, \delta)$, where

1. Σ is a finite non-empty set called the *alphabet*.
2. Q is a finite non-empty set of *states*.
3. q_0 is the *initial* state; the state of the machine before reading each string.
4. $A \subseteq Q$ is a special set of states called the *accept* states.
5. δ is the *transition function* that maps $Q \times \Sigma$ to Q.

Notice that there is no explicit mention of the tape or read head in this formalism. The idea of an infinite tape arises from the fact that the input string is not part of the machine definition; instead, we conceive of the machine as being capable of processing arbitrarily long (albeit finite) input strings that are written on its tape.

The operation of this machine can also be given a completely formal definition that captures the intuitive description given above. For this we will require the notion of a *configuration*, or instantaneous snapshot of the machine, and a *yields relation*, which will define the sequence of configurations that a machine enters as it operates. We first provide some basic notation and a convenient means of representing a DFA diagrammatically.

Let M denote the machine $(\Sigma, Q, q_0, A, \delta)$, and let $w \in \Sigma^*$ be some string. We shall use the notation $\mathcal{S}_{\text{M}}(w)$ to denote the *set of states* of machine M after reading string w; of course, for a DFA, this set will always be a singleton, as it is always left in precisely one state. If the single element of $\mathcal{S}_{\text{M}}(w)$ is in the set A of the accept states, we say that the machine *accepts* or *recognizes* the string w. We can also state this condition as $\mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset$. The set of all recognized strings constitutes a language associated with M. More precisely, we define

$$\mathcal{L}(M) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset\} \tag{5.1}$$

to be the language *accepted* (or *recognized*) by the machine M.[4] Such languages form an important class, so we shall give them a special name.

Definition 11 A language is called *DFA-acceptable* if and only if a DFA (Deterministic Finite Automaton) can be constructed that *recognizes* it.

[4]We could have also written the criterion in Equation (5.1) as $\mathcal{S}_{\text{M}}(w) \subset A$. While these definitions are equivalent in the case of DFAs, we will find that they are not equivalent for NFAs. In particular, $\mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset$ generalizes immediately to NFAs, while $\mathcal{S}_{\text{M}}(w) \subset A$ does not.

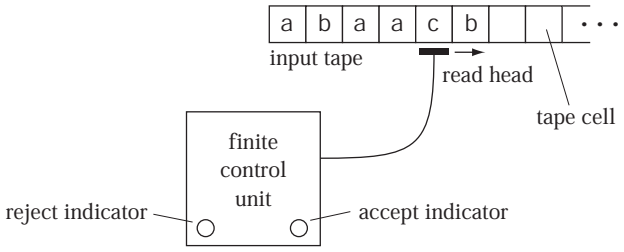

Figure 5.1: *Conceptually, a finite automaton consists of a finite control, a read head, and an input tape that is infinite in one direction (by convention it is always drawn so that it is infinite to the right). In each discrete step, the read head moves exactly one cell to the right until the end of the input string is reached. After each discrete transition, one of the two indicator lights is lit up. Once the string has been read, we say that the machine "accepts" or "rejects" the string, depending on which of the two lights is left on.*

The transition function $\delta : Q \times \Sigma \rightarrow Q$ defines the behavior of the DFA as the tape is being read. If the machine is in state q and the symbol s is read, the machine enters state $\delta(q, s)$. This transition can (and usually does) influence the states entered as subsequent symbols are read. As a simple example, consider the language a^*b^* over the alphabet $\Sigma = \{a, b\}$. We can define a simple DFA that accepts this language using only three states, $Q = \{q_0, q_1, q_2\}$, where q_0 is the start state and both q_0 and q_1 are accept states; thus, $A = \{q_0, q_1\}$. The transition function $\delta \subset Q \times \Sigma$ of this machine can be defined as follows:

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

The state q_2 is known as a *trap state*, since the machine never leaves that state once it enters it. Defining the transition function δ using a table as above can be extremely cumbersome for more complex machines. Fortunately, there is a much more intuitive yet complete representation available for defining such machines; namely, state diagrams.

5.1.1 State Diagrams

Figure 5.2 shows a *state diagram* for a DFA that accepts a simple language over the alphabet $\{a, b\}$ that includes strings such as *aba* and *baabb*. Each state q is depicted by a circle with the label "q", and each transition $\delta(q, a) = q'$ is depicted by an arrow from state "q" to state "q'" with the label "a". The initial state is indicated with an inward-pointing arrow with no label, and the accept states are indicated by double circles. An arrow with label "a" leaving state "q" indicates which state to move to upon reading an "a" from the tape when the machine is in state "q". Different strings will cause the machine to follow different paths through the directed graph of states, starting from the initial state each time.

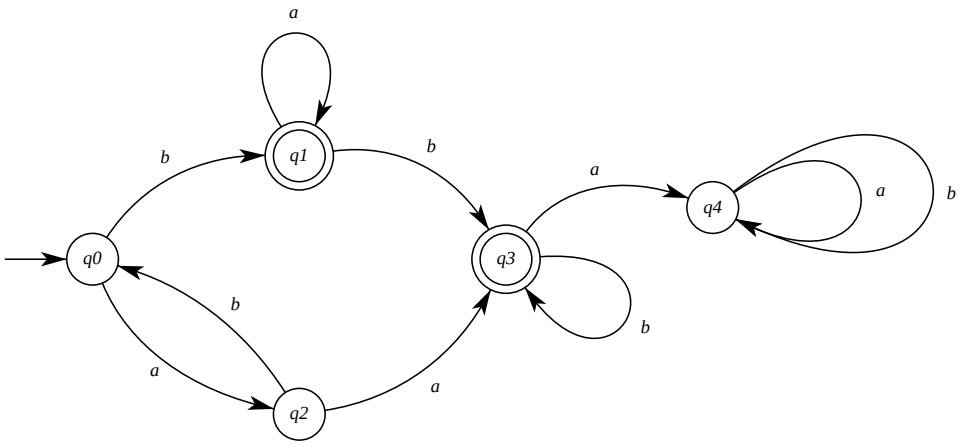

Figure 5.2: *A state diagram for a DFA that accepts a simple language over the alphabet* $\{a, b\}$*. Here state* q_0 *is the "initial state," states* q_2 *and* q_3 *are "accept" states, and state* q_4 *is a non-accepting "trap."*

More formally, the machine would be written as the 5-tuple $(\Sigma, Q, q_0, A, \delta)$, where

$$\begin{aligned}
\Sigma &= \{a, b\} \\
Q &= \{q_0, q_1, q_2, q_3, q_4\} \\
A &= \{q_2, q_3\} \\
\delta &= \{(q_0, a, q_2), (q_0, b, q_1), (q_1, a, q_1), (q_1, b, q_3), \\
&\quad (q_2, a, q_3), (q_2, b, q_1), (q_3, a, q_4), (q_3, b, q_3), \\
&\quad (q_4, a, q_4), (q_4, b, q_4)\}.
\end{aligned}$$

Here each ordered pair $((q, a), q')$ defined by the function $\delta : Q \times \Sigma \rightarrow Q$ has been abbreviated to (q, a, q'). Since δ is a function defined on $Q \times \Sigma$, as a set it must contain *exactly one* tuple for every $(q, a) \in Q \times \Sigma$. In terms of the state diagram, this means that every state is represented by a circle, and each circle must have an arrow leaving it for each letter in the alphabet. The state diagram is actually a complete specification of the DFA, but one that is far easier to think about. Given the state diagram of a DFA, we can always construct the corresponding 5-tuple, and vise versa.

5.1.2 Configurations and the Yields Relation

Once a symbol is read, it is never considered by the machine again, because the read head moves only to the right. Consequently, when a symbol is read, its influence is recorded solely in the machine state that is entered next. All subsequent computation is completely determined by the current state of the machine, q, and the unread portion of the tape, u. We combine this information into an ordered pair (q, u) called the *configuration* of the machine; the sequence of configurations completely describes the behavior of the machine on a given string. If $\delta(q, a) = q'$ and $w = au$, where w and u are strings in Σ^*, then one can think of δ as defining a "transformation" from one configuration to another, which we write as

$$(q, w) \vdash (q', u), \tag{5.2}$$

which is read "configuration (q, w) *yields* configuration (q', u) in one step." The symbol $\vdash$ actually represents a binary relation on $Q \times \Sigma^*$, which is formally defined by

$$(q_1, w_1) \vdash (q_2, w_2) \iff [\, \delta(q_1, a) = q_2 \ \wedge \ w_1 = a w_2 \,],$$

where w_1 and w_2 are strings in Σ^*, and $a \in \Sigma$. Here w_2 can be any string at all, including the empty string, because the next transition made by the machine is completely determined by the current state q_1 and the current symbol a (although subsequent transitions will depend on w_2). It follows that

$$(q_1, au) \vdash (q_2, u) \iff (q_1, av) \vdash (q_2, v)$$

for all strings u and v in Σ^*. While fairly obvious, this simple fact is one of the tools that allows us to reason formally about the behavior of a DFA. In particular, the implication

$$(q_1, a) \vdash (q_2, \varepsilon) \implies (q_1, aw) \vdash (q_2, w)$$

for any $w \in \Sigma^*$ will be a useful transformation in constructing formal proofs about DFAs. However, we also need a way to reason about the *composition* of many steps in a computation. To this end, we define the *reflexive transitive closure* of the binary relation $\vdash$.

From the relation $\vdash$ we define a new relation $\overset{*}{\vdash}$ by forming the *reflexive transitive closure* of $\vdash$, which extends it to include all *finite sequences* of steps that the machine M can take. That is, for a given machine M, the configuration (q_1, u) *eventually* reaches the configuration (q_2, v) if and only if

$$(q_1, u) \overset{*}{\vdash} (q_2, v),$$

which we read as "configuration (q_1, u) yields configuration (q_2, v) after zero or more steps (of machine M)". More formally, we may give an inductive definition of this relation as

$$(q_1, u) \vdash (q_2, v) \implies (q_1, u) \overset{*}{\vdash} (q_2, v) \tag{5.3}$$

$$q \in Q \ \wedge \ a \in \Sigma \implies (q, a) \overset{*}{\vdash} (q, a) \tag{5.4}$$

$$\left[(q_1, u) \vdash (q_2, v) \ \wedge \ (q_2, v) \overset{*}{\vdash} (q_3, w) \right] \implies (q_1, u) \overset{*}{\vdash} (q_3, w) \tag{5.5}$$

where u, v, and w are strings in Σ^*. Implication (5.3) makes the new relation an *extension* (that is, a superset) of the old relation, implication (5.4) makes the new relation *reflexive*, and implication (5.5) makes the new relation *transitive* (that is, closed under composition). Taken together, these implications completely specify the binary relation $\overset{*}{\vdash}$ for a given machine M. From this definition it follows that $w \in \mathcal{L}(M)$ if and only if

$$(q_0, w) \overset{*}{\vdash} (q', \varepsilon) \tag{5.6}$$

for some $q' \in A$.

5.2 Nondeterminism

A *Nondeterministic Finite Automaton* (NFA) is a generalization of a deterministic finite automaton (DFA) in which any given transition may be incompletely determined by the current state of the

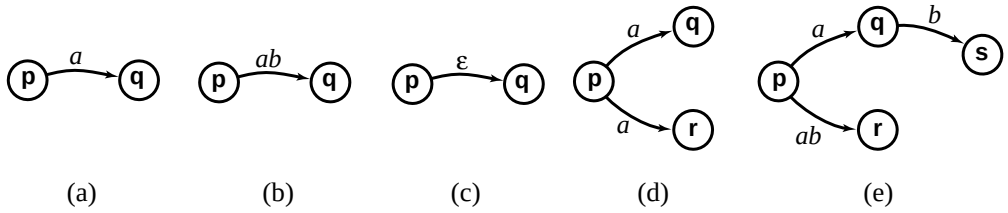

Figure 5.3: *Four complete NFAs, with $\Sigma = \{a, b\}$, that demonstrate the generality of nondeterministic machines: (a) missing transitions, such as an arrow labeled "a" leaving state q, (b) transitions with strings as labels, not just single characters, (c) null transitions, which have empty strings as labels, (d) simple ambiguous transitions, and (e) more complex ambiguous transitions.*

machine and the symbol currently under the read head; for example, there may be *several* equally valid alternatives to choose from, or perhaps none at all. The various possibilities are depicted in Figure 5.3. In contrast, a DFA always has exactly one valid transition at each step of the computation.

When an ambiguous transition is encountered in a nondeterministic machine, such as the situations depicted in (d) and (e) of Figure 5.3, we can imagine the machine behaving in one of several different ways:

- **Random Selection:** The machine arbitrarily chooses one of the applicable transitions.
- **Sequential Search:** The machine follows each of the applicable branches, one after the other, to determine which states may ultimately be reached.
- **Parallel Search:** The machine "clones" itself as many times as necessary and follows all applicable branches simultaneously to determine which states may ultimately be reached.
- **Consult an Oracle:** The machine consults an all-knowing "oracle," which magically tells it the "right" transition to follow (e.g. the transition that will ultimately lead it to an accept state).

As it happens, all of these interpretations will be equivalent when studying the possible behavior of such a machine without regard to "time;" that is, when we are concerned only with which outcomes are possible and when we disregard the number of steps the machine takes in reading a string on the tape. As time will be irrelevant in our current discussion, we may choose the interpretation that seems most helpful or intuitive in each situation.

There are exactly four ways in which the definition of an NFA can differ from that of a DFA, and all of them are *generalizations* of the DFA. These are listed below, and correspond to the simple machines shown in Figure 5.3.

1. Not all transitions need be specified, as in Figure 5.3(a). All unspecified transitions are understood to lead to a "reject" trap.

2. Transitions may be labeled with sequences of symbols (i.e. strings), as in Figure 5.3(b), as well as with single symbols. Such transitions read an entire prefix from the tape as opposed to a single symbol.

3. Transitions may be labeled with the null string, as in Figure 5.3(c). Such a transition leaves the read head fixed while changing the state.

4. There may be multiple transitions with the same label leaving a single state, as in Figure 5.3(d), or otherwise equivalent paths leaving a single state, as in Figure 5.3(e).

These generalizations require that we abandon the notion of a transition "function," since ambiguous transitions produce multiple values; for example, we would have both $\delta(p, a) = q$ and $\delta(p, a) = r$ in the example (d) of Figure 5.3. To remedy this, we replace the function δ with the *relation* Δ. Thus, an NFA is formally defined to be a 5-tuple $(\Sigma, Q, q_0, A, \Delta)$, where all of the symbols have the same meanings as in the DFA, with the exception that Δ is now the *transition relation* on the set $Q \times \Sigma^* \times Q$. Each 3-tuple in this relation represents the current state, the string labeling a transition, and the state the transition leads to. Notice that this one change accommodates all four of the generalizations above.

Let M denote an NFA. We shall say that M *accepts* (or *recognizes*) a string $w \in \Sigma^*$ if and only if there is *some* sequence of nondeterministic choices that M could make in processing w that would lead to an accept state. Equivalently, M accepts w if and only if at least one of the "clones" generated via nondeterminism ends in an accept state. It is easy to see how all of the interpretations of nondeterminism mentioned above lead to exactly the same conclusions regarding acceptance of strings.

As in the deterministic case we shall let $\mathcal{S}_{\text{M}}(w)$ denote the set of states that an NFA may be left in after reading the string w; while this set will always be a singleton for a DFA, in the case of an NFA it may have any number of elements, including zero. The elements of this set correspond to all possible states that can be reached by making different decisions while reading the string w. This property leads to a slightly different notion of what it means for an NFA to "accept" a string. We have the following definitions for NFAs, which parallel those for DFAs.

Definition 12 Let machine M be an NFA. We say that M *accepts* a string $w \in \Sigma^*$ if and only if $\mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset$. That is, if and only if at least one sequence of choices can be made while reading the string that leaves the machine in an accept state after reading the string.

Definition 13 The language *accepted* (or *recognized*) by an NFA $M = (\Sigma, Q, q_0, A, \Delta)$ is defined to be the set $\{w \in \Sigma^* \mid \mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset\}$, which is denoted by $\mathcal{L}(M)$, as in the deterministic case.

Definition 14 A language is called *NFA-acceptable* if and only if an NFA (Nondeterministic Finite Automaton) can be constructed that accepts it.

Clearly all DFA-acceptable languages are also NFA-acceptable. This follows immediately from the definition of NFAs, which are simply generalizations of DFAs. Thus, every DFA is also a valid NFA; one in which none of generalizations noted above are actually exploited.

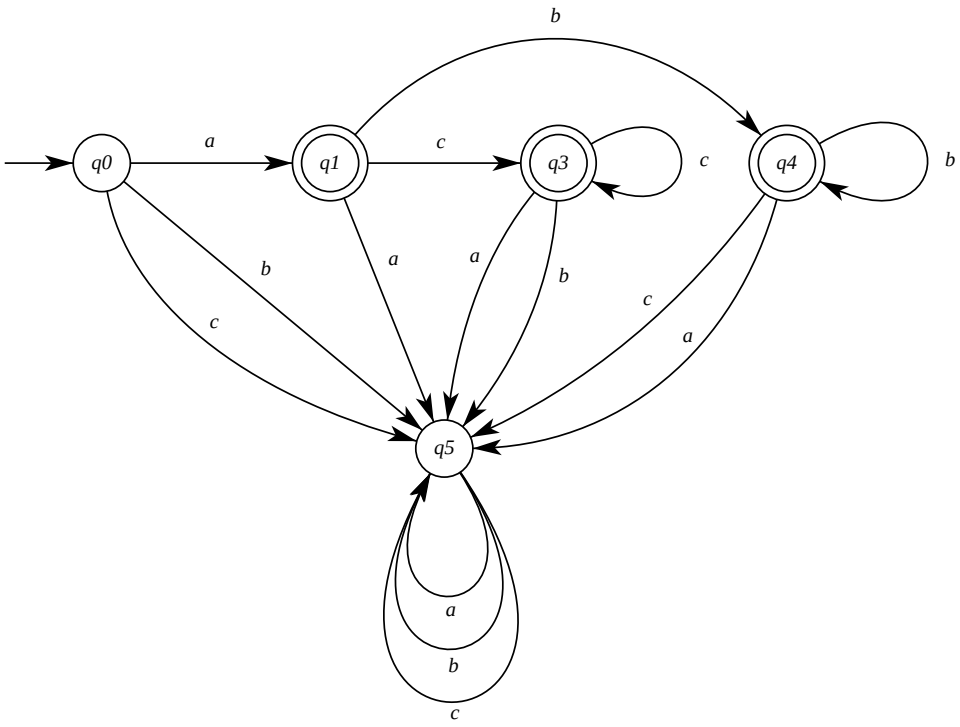

Figure 5.4: *A DFA that recognizes the language* $ab^* \cup ac^*$.

Nondeterminism is *not* intended to model randomness that may occur in actual machines, such as quantum effects or hardware failures. As we shall see, nondeterminism is primarily a powerful *theoretical* tool; it will allow us to prove things about finite automata that would otherwise be very cumbersome. Nondeterminism is also a convenient notational device.

As a first concrete example of the utility of nondeterminism, we note that it frequently allows for great economy in specifying machines. Consider the DFA shown in Figure 5.4, which recognizes the simple language $ab^* \cup ac^*$ over the alphabet $\{a, b, c\}$. Here we have labeled some of the transitions with *sets* as an abbreviation for multiple arrows that share the same beginning and ending states; that is, the set notation encodes "parallel" edges, whereas a string of characters (in an NFA) encodes multiple edges that are "in sequence."

Using the conventions of an NFA, the same language can be recognized with a much simpler machine. In the NFA shown in Figure 5.5, only three states suffice, thanks to the conveniences afforded by nondeterminism. The trap has been removed by simply not specifying transitions that cannot lead to an accept state. The resulting NFA has both fewer states and fewer transitions.

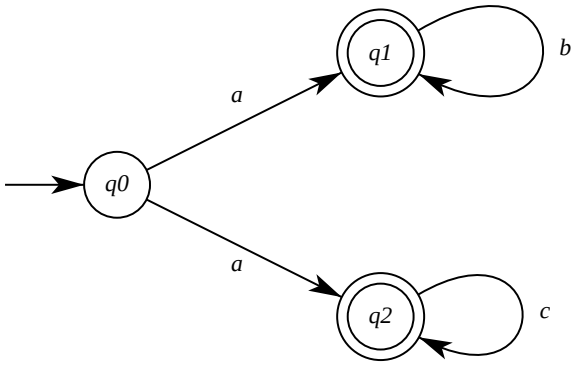

Figure 5.5: *An NFA that recognizes the language $ab^* \cup ac^*$. Nondeterminism allows us to create such a machine using fewer states and transitions than the equivalent DFA.*

5.3 Finite Automata and Regular Languages

We have now defined several classes of languages either explicitly by giving a method for generating strings in the language (e.g. regular expressions) or implicitly by constructing an automaton (DFA or NFA) that recognizes strings in the language. The classes we have defined are

1. Regular
2. DFA-acceptable
3. NFA-acceptable

We will now demonstrate the these classes of languages are identical. We will prove this by demonstrating three inclusions: DFA-acceptable $\subseteq$ Regular, Regular $\subseteq$ NFA-acceptable, and NFA-acceptable $\subseteq$ DFA-acceptable. Each of these inclusions requires a different style of proof, as illustrated in the following sections. As a consequence of this equivalence we are justified in referring to the languages accepted by both DFAs and NFAs as "regular."

5.3.1 Regular $\subseteq$ NFA-acceptable

In this section we shall prove that the set of regular languages is a subset of the NFA-acceptable languages. In other words, we will prove the following theorem.

Theorem 15 *Every regular language is accepted by some NFA.*

Proof: We proceed by first showing that the NFA-acceptable languages include the null language and all the singleton languages; then we'll show that NFA-acceptable languages possess all the necessary closure properties. In particular, we'll show that if L_1 and L_2 are both NFA-acceptable languages, then so are the languages

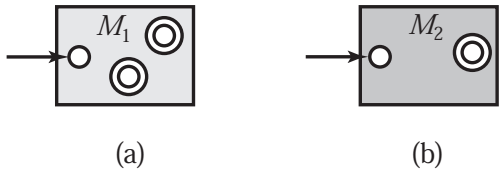

Figure 5.6: *Diagrammatic representations of two finite automata M_1 and M_2 (either deterministic or nondeterministic). The only features that are relevant are the start states, shown as single circles, and the accept states, shown as double circles.*

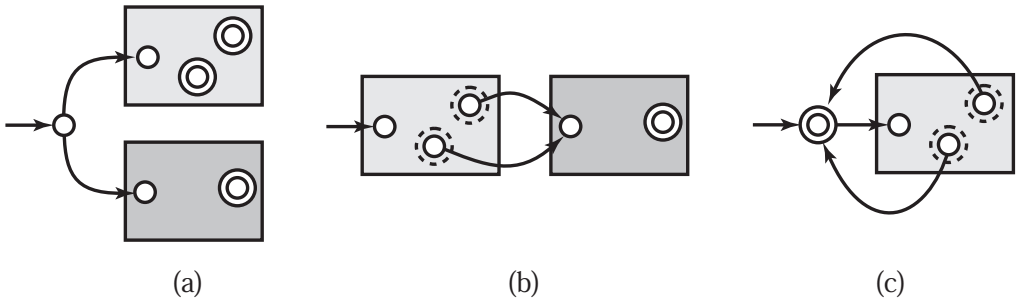

Figure 5.7: *New machines built from M_1 and M_2 that accept the (a) union of the two original languages, the (b) concatenation, and the (c) Kleene star of one of the original languages. The dashed circles indicate states that are no longer accepting states.*

1. $L_1 \circ L_2$ (showing closure with respect to concatenation)
2. $L_1 \cup L_2$ (showing closure with respect to union)
3. ${L_1}^*$ (showing closure with respect to Kleene star)

First, observe that an NFA with no accept states accepts the empty language $\emptyset$, and that for any $a \in \Sigma$ the machine shown in Figure 5.3(a) with p as initial state and q as an accept state accepts the singleton language $\{a\}$. Thus, all of the trivial regular languages are also NFA-acceptable.

We next demonstrate that all of the closure properties listed above also hold, from which it will follow that *all* regular languages are NFA-acceptable. To do this, let us depict machines M_1 and M_2, which accept languages L_1 and L_2, respectively, as in Figure 5.6. These simplified illustrations depict only the initial state and accepting states, of which there may be any number.

Without loss of generality, let us assume that the state labels appearing in the two diagrams are distinct. Then, from these two state diagrams we may construct state diagrams depicting new machines by adding states and/or edges. Figure 5.7(a) shows the state diagram for a machine that accepts the language $L_1 \cup L_2$, obtained by adding a new start state and two null transitions. Figure 5.7(b) shows a similar construction for a machine that accepts the language $L_1 \circ L_2$. Finally, Figure 5.7(c) shows a machine that accepts the language ${L_1}^*$.

It is now easy to see that the NFA-acceptable languages includes all of the regular languages, since we can construct an NFA that accepts any language that can be represented by a regular expression.

We need only echo each of the operations represented in the regular expression as an operation applied to machines; the result will be an NFA that accepts exactly the same language as that represented by the regular expression.

5.3.2 DFA-acceptable $\subseteq$ Regular

In this section we shall use induction to prove that the set of DFA-acceptable languages is a subset of the regular languages. That is, we shall prove the following theorem.

Theorem 16 *The language accepted by every DFA is regular.*

Proof: Let $M = (Q, \Sigma, q_0, A, \delta)$ be a DFA with states $Q = \{1, 2, 3, \ldots, n\}$ and initial state $q_0 = 1$. Let $R_k(i, j)$ denote the set of strings that would cause M to pass from state i to state j without ever passing *through* a state labeled k or higher. That is, $w \in R_k(i, j)$ if and only if machine M, when started in state i, would end up in state j after reading w, but without ever leaving or entering states k through n (with the possible exceptions of the first or last transitions, respectively). Then

$$\mathcal{L}(M) = \bigcup_{m \in A} R_{n+1}(1, m), \tag{5.7}$$

since $R_{n+1}(1, m)$ excludes no intermediate states, and all accepting states are accounted for. We shall now show that $R_k(i, j)$ is always regular using induction on k. As the base case, observe that when $k = 1$ the machine cannot pass through *any* intermediate states, so we have

$$R_1(i, j) = \{a \in \Sigma \mid \delta(i, a) = j\} \ \cup \ E(i, j), \tag{5.8}$$

where $E(i, j) = \{\varepsilon\}$ when $i = j$, and $\emptyset$ otherwise. That is, $R_1(i, j)$ consists of all symbols for which there is a direct transition from i to j, plus the empty string ε in the case where $i = j$. As a finite language, $R_1(i, j)$ is clearly regular. Now suppose that $R_1(i, j)$, $R_2(i, j), \ldots,$ $R_k(i, j)$ are all regular as the inductive hypothesis, and observe that

$$R_{k+1}(i, j) \ = \ R_k(i, j) \ \cup \ R_k(i, k) \circ R_k(k, k)^* \circ R_k(k, j). \tag{5.9}$$

It then follows from Equation (5.9) that $R_{k+1}(i, j)$ is also regular, as it consists of concatenations and a Kleene star of regular languages. Finally, we conclude from Equation (5.7) that $\mathcal{L}(M)$ is also regular, as it is a union of regular languages. $\square$

5.3.3 NFA-acceptable $\subseteq$ DFA-acceptable

In this section we shall prove that the set of NFA-acceptable languages is a subset of the DFA-acceptable languages. Since inclusion in the opposite direction is obvious (as noted above), we have the following theorem.

Theorem 17 *A language is accepted by an NFA if and only if it is accepted by some DFA.*

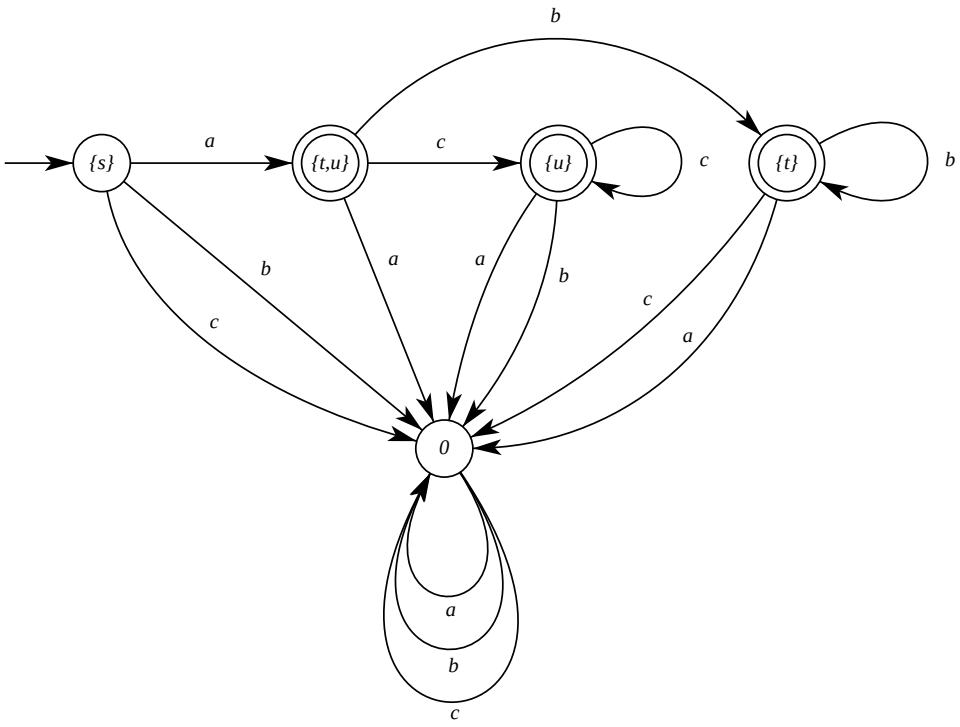

Figure 5.8: *This is the DFA that is obtained by performing the NFA-to-DFA conversion procedure on the NFA shown in Figure 5.5. Aside from state labels, it matches the DFA shown in Figure 5.4.*

Consequently, the utility of nondeterminism in the context of finite automaton lies in ease of expression and greater simplicity, not the ability to describe a wider variety of languages. That is, it is typically much easier to construct an NFA than the corresponding DFA, and the resulting machine is frequently much simpler to understand. For most theoretical purposes, such as proving that languages are DFA-acceptable, we needn't ever perform an explicit conversion to a DFA; it is sufficient to know that an equivalent DFA exists.

Proof: We need only show that every NFA can be replaced with a DFA that accepts exactly the same language (since the converse is immediate). The following algorithm describes the steps for creating the state diagram of an equivalent DFA from the state diagram of an NFA. The sates of the constructed DFA are labeled with the elements of 2^Q, where Q is the set of states of the NFA. That is, the new labels are *sets* of labels that appear in the original NFA. This labeling reflects the fact that with each transition the equivalent DFA simulates progress along multiple paths in the NFA simultaneously. This strategy eliminates nondeterminism at the cost of possibly introducing *exponentially* more states.

To construct a DFA that is equivalent to a given NFA:

1. Create a start state for the DFA and label it with the set of all states in the original NFA that can be reached without moving the read head (i.e. without reading a single symbol). This set consists of the NFA's start state and all the states that can be reached from it via null

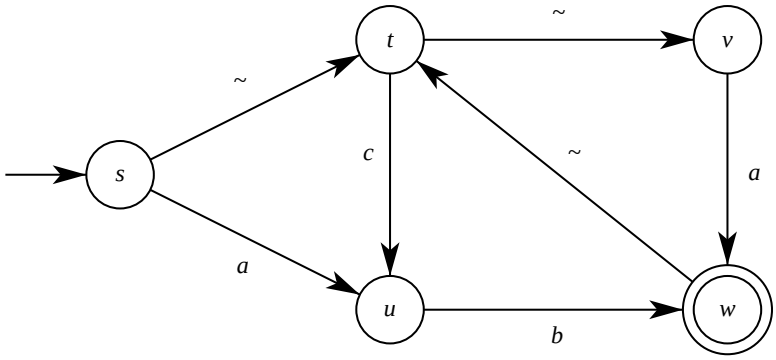

Figure 5.9: *An example of an NFA with multiple null transitions. Such transitions complicate the conversion of the machine to a DFA; at each step we must find all states that are reachable by these transitions as well.*

transitions.

2. Expand all edges in the NFA that are labeled with strings; that is, replace them with a sequence of edges, each labeled with single characters. Give each of the new intermediate states a unique label. We shall call this new and equivalent machine the *expanded NFA*.

3. Loop forever:

   (a) Let S be a state in the DFA under construction that has no outgoing edge labeled "a", where "a" is some symbol in Σ. If there is no such state, then exit the loop.

   (b) Let T be the set of all states reachable in the expanded NFA via exactly one a-transition in combination with any number of null transitions, starting from any of the states $s \in S$.

   (c) If a state with the label T does not already exist in the DFA, create one. Note that T may be the empty set, $\emptyset$.

   (d) Add an edge from S to T and label it with the symbol a.

4. If the DFA has a state that is labeled $\emptyset$, add a loop back to itself for each $a \in \Sigma$. That is, make it a trap.

5. Let A be the set of all states in the DFA whose labels contain one or more accept states of the original NFA. Let A be the set of accept states of the DFA.

By applying this algorithm to the previous NFA, which accepted the language $ab^* \cup ac^*$, we arrive at the DFA shown in Figure 5.8. Note that step number 1 above did not apply here, since every edge in the original NFA was labeled by a single symbol. Coincidentally, the resulting DFA is identical to the one in Figure 5.4, except for the state labels. This will not always be the case. Frequently the above algorithm will introduce far more states than are necessary and be more complex than a machine built directly from the language description. The purpose of this conversion algorithm is primarily theoretical; it simply demonstrates the equivalence of NFAs and DFAs.

As a final example, consider the NFA shown in Figure 5.9, which has three null transitions. Note, in particular, that two additional states are now reachable from the start state via null transitions,

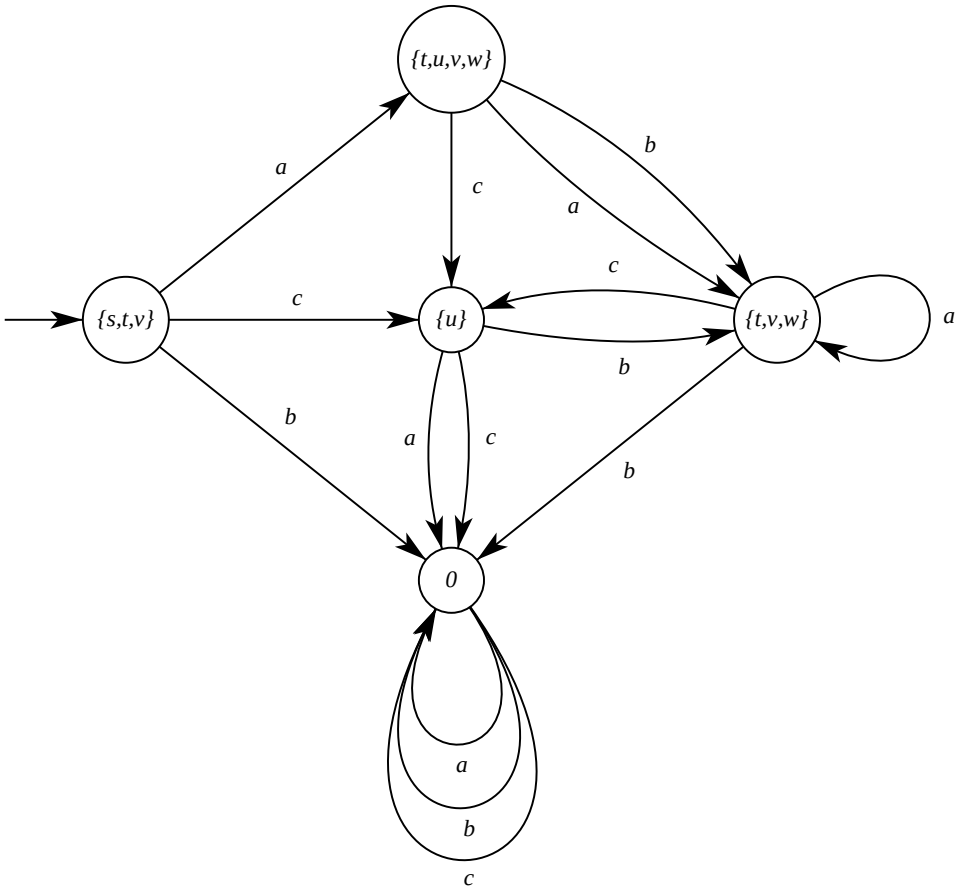

Figure 5.10: *This is the machine obtained by converting the NFA shown in Figure 5.9 to a DFA. Note that all null transitions are gone, and all missing transitions have been filled in.*

so they are included in the start of the constructed DFA. Converting this NFA to a DFA, we arrive at the machine shown in Figure 5.10. In finding the three state labels along the top it was necessary to follow multiple null transitions in the original NFA.

5.3.4 Correctness of the NFA Conversion Algorithm

In the previous section we showed that the class of NFA-acceptable languages was a subset of the DFA-acceptable languages by giving a conversion algorithm from NFAs to equivalent DFAs. We'll now show that the DFA constructed by the algorithm in fact recognize the same language as the original NFA.

Theorem 18 *Let $M = (Q, \Sigma, q_0, A, \Delta)$ be an arbitrary nondeterministic finite automaton (NFA)*

and let $M' = (Q', \Sigma', q_0', A', \delta')$ *be a deterministic finite automaton (DFA) where*

$$\begin{aligned} Q' &= \{\, \mathcal{S}_{\text{M}}(w) \mid w \in \Sigma^* \,\} && (5.10)\\ \Sigma' &= \Sigma && (5.11)\\ q_0' &= \mathcal{S}_{\text{M}}(\varepsilon) && (5.12)\\ A' &= \{\, \mathcal{S}_{\text{M}}(w) \mid w \in \Sigma^* \ \wedge\ \mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset \,\} && (5.13)\\ \delta' &= \{\, (\, \mathcal{S}_{\text{M}}(u),\, a,\, \mathcal{S}_{\text{M}}(ua)\,) \mid u \in \Sigma^* \ \wedge\ a \in \Sigma \,\} && (5.14) \end{aligned}$$

Then M *and* M' *are equivalent in that they accept the same language.*

Proof: We first verify that the DFA defined in Theorem 18 is properly formed. Recall that $\mathcal{S}_{\text{M}}(w)$ denotes the *set* of states that machine M can reach by reading string w (which is a singleton set for a deterministic machine). So, by Equation (5.10) each element of Q' is a *subset* of Q. Thus, Q' is finite, since $Q' \subseteq 2^Q$ and Q is finite. It is also easy to check that $A' \subseteq Q'$, $q_0' \in Q'$, and that δ' is a total function; that is, $\delta' : Q' \times \Sigma' \ \rightarrow\ Q'$ is single-valued and defined over all of $Q' \times \Sigma'$.

We now show that $\mathcal{L}(M') = \mathcal{L}(M)$. To do this, we first prove that

$$\mathcal{S}_{\text{M}'}(w) = \{\, \mathcal{S}_{\text{M}}(w) \,\} \quad \forall w \in \Sigma^*. \tag{5.15}$$

That is, after reading the string w, the deterministic machine M' is left in the state with label $\mathcal{S}_{\text{M}}(w)$, which is the set of states in M that are reachable by reading the string w. This is easily established by induction on the length of w. As the base case, when $|w| = 0$, or $w = \varepsilon$, note that

$$\begin{aligned} \mathcal{S}_{\text{M}'}(\varepsilon) &= \{\, q_0' \,\} && \textit{by the definition of a DFA} && (5.16)\\ &= \{\, \mathcal{S}_{\text{M}}(\varepsilon) \,\}. && \textit{by Equation (5.12)} && (5.17) \end{aligned}$$

As the inductive hypothesis, assume that $\mathcal{S}_{\text{M}'}(w) = \{\, \mathcal{S}_{\text{M}}(w) \,\}$ for all $w \in \Sigma^*$ with $|w| \leq n$. Now, let $w = ua$, where $u \in \Sigma^*$, $|u| = n$, and $a \in \Sigma$. Let q be the state that M' is left in after reading the string u; that is, $\mathcal{S}_{\text{M}'}(u) = \{q\}$. Then

$$\begin{aligned} \mathcal{S}_{\text{M}'}(ua) &= \{\, \delta'(q, a) \,\} && \textit{by the definition of a DFA}\\ &= \{\, \delta'(\mathcal{S}_{\text{M}}(u), a) \,\} && \textit{by the inductive hypothesis}\\ &= \{\, \mathcal{S}_{\text{M}}(ua) \,\}. && \textit{by Equation (5.14)} \end{aligned}$$

Thus, Equation (5.15) holds by induction. Finally, we show that $w \in \mathcal{L}(M) \iff w \in \mathcal{L}(M')$ by a sequence of double implications:

$$\begin{aligned} w \in \mathcal{L}(M) &\iff \mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset && \textit{by the definition of } \mathcal{L}(M)\\ &\iff \mathcal{S}_{\text{M}}(w) \in A' && \textit{by Equation (5.13)}\\ &\iff \mathcal{S}_{\text{M}'}(w) \cap A' \neq \emptyset && \textit{by Equation (5.15)}\\ &\iff w \in \mathcal{L}(M'). && \textit{by the definition of } \mathcal{L}(M') \end{aligned}$$

Consequently, $\mathcal{L}(M) = \mathcal{L}(M')$, which demonstrates that M and M' are equivalent.

Theorem 19 *The algorithm described earlier for converting an NFA to an equivalent DFA constructs the machine defined in Theorem 18.*

Proof: (To be supplied.) □

5.4 Closure Properties of Regular Languages

By their very definition the set of regular languages over a common alphabet are closed with respect to concatenation, union, and Kleene star. In this section we extend this list to include several more common operations on languages.

Theorem 20 *Regular languages are closed under complementation, intersection, subtraction, and symmetric difference.*

Proof: First, we shall use the fact that every regular language is accepted by some DFA to prove that regular languages are closed under complementation. Let L be a regular language and let $M = (Q, \Sigma, q_0, A, \delta)$ be a DFA that accepts it. Now define another DFA $M' = (\Sigma, Q, q_0, Q - A, \delta)$. We claim that M' accepts the language $\Sigma^* - L$, which is the complement $\overline{L}$ of the language L. This is easy to see, since $\mathcal{S}_{M'}(w) = \mathcal{S}_M(w)$ for every $w \in \Sigma^*$, so $\mathcal{S}_{M'}(w) \notin Q - A$ if and only if $\mathcal{S}_M(w) \in A$. Therefore, M' rejects a string w if and only if M accepts it.

Given the closure properties we have established thus far, the remaining operations are easily seen to be closed. We need only observe that each can be expressed in terms of other operations under which regular languages have been previously shown to be closed. Thus,

$$L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})} \tag{5.18}$$

$$L_1 - L_2 = \{w \in L_1 \mid w \notin L_2\} = L_1 \cap \overline{L_2} \tag{5.19}$$

$$L_1 \ominus L_2 = \{w \in L_1 \cup L_2 \mid w \notin L_1 \cap L_2\} = (L_1 - L_2) \cup (L_2 - L_1). \tag{5.20}$$

The operation denoted by "$\ominus$" is known as *symmetric difference*. It is important to observe that the method we used above to construct a DFA that accepts $\overline{L}$ from one that accepts L does not carry over directly to NFAs. That is, if we simply change accepting states into non-accepting states, and vise versa, in an arbitrary NFA, the resulting machine will in general *not* accept the complement of the original language.

5.5 The Pumping Lemma

We start by establishing several elementary rules governing the operation of finite automata that will prove useful later. Let Σ be an alphabet, and let a and b denote arbitrary symbols in Σ. From the definition of the "yields" relation for the finite automaton M, denoted by $\vdash$, we may obtain the elementary composition rule

$$(q_1, a) \vdash (q_2, \varepsilon) \implies (q_1, ab) \vdash (q_2, b), \tag{5.21}$$

which simply affirms that the next step taken by machine M is unaffected by the symbol immediately following the current position of the read head. From the definition of the relation $\overset{*}{\vdash}$, rule (5.21) may be inductively extended to strings, yielding the rule

$$(q_1, u) \overset{*}{\vdash} (q_2, \varepsilon) \implies (q_1, uv) \overset{*}{\vdash} (q_2, v), \tag{5.22}$$

where u and v now denote strings in Σ^*. By the transitivity of $\overset{*}{\vdash}$, we may then immediately formulate the fundamental composition rule for finite automata:

$$\left[(q_1, u) \overset{*}{\vdash} (q_2, \varepsilon) \ \wedge \ (q_2, v) \overset{*}{\vdash} (q_3, \varepsilon) \right] \implies (q_1, uv) \overset{*}{\vdash} (q_3, \varepsilon). \tag{5.23}$$

From rule (5.23) it follows by induction on n that

$$(q, v) \overset{*}{\vdash} (q, \varepsilon) \implies (q, v^n) \overset{*}{\vdash} (q, \varepsilon) \tag{5.24}$$

for all $n \in \mathbb{N}$. Finally, from rules (5.23) and (5.24) together, we obtain

$$\left[(q_1, u) \overset{*}{\vdash} (q_2, \varepsilon) \ \wedge \ (q_2, v) \overset{*}{\vdash} (q_2, \varepsilon) \right] \implies (q_1, uv^n) \overset{*}{\vdash} (q_2, \varepsilon) \tag{5.25}$$

for all $n \in \mathbb{N}$. When q_1 is the initial state of machine M, rule (5.25) may be be more succinctly stated as

$$\mathcal{S}_{\text{M}}(u) = \mathcal{S}_{\text{M}}(uv) \implies \mathcal{S}_{\text{M}}(u) = \mathcal{S}_{\text{M}}(uv^n) \tag{5.26}$$

for all $n \in \mathbb{N}$. Rule (5.26) will be an essential element in establishing the *first pumping lemma*, which is a powerful tool for identifying languages that cannot be recognized by finite automata. This lemma is proven in the following section.

While it is easy to see from a counting argument that there are uncountably many non-regular (i.e. non-DFA-acceptable) languages, such an argument provides no means of identifying a specific non-regular language. To prove that a language is *not* regular, we must show that *no* DFA recognizes it. As is frequently the case, a negative result (e.g. that *no* DFA accepts language L) is more challenging to prove than the corresponding positive result (e.g. that machine M accepts language L), since the latter can be demonstrated by simply exhibiting a machine that accepts L. To attack this problem, we first introduce the notion of a *pumping property* that may or may not be possessed by any given string in a language.

Definition 15 Let k be a natural number and let L be a language. Then we shall say that $w \in L$ has the *first pumping property of length k* (with respect to the language L) if and only if there exist strings x, y, and z (not necessarily in the language L) such that $w = xyz$, $|xy| \leq k$, $|y| \geq 1$, and $xy^n z \in L$ for all $n \geq 0$.

The word "pumping" in this definition is motivated by the fact that any number of copies of the string y may be inserted (or "pumped") into the machine M, following the prefix x, without changing the state of M.

The following powerful lemma provides a method for demonstrating non-regularity of a large class of (but not all of) the non-regular languages.

Lemma 1 (The Pumping Lemma for Regular Languages) *Let L be a regular language. Then for some $k \in \mathbb{N}$, every $w \in L$ with $|w| \geq k$ has the first pumping property of length k.*

Proof: Let L be a regular language, and let $M = (\Sigma, Q, q_0, A, \delta)$ be a DFA that recognizes L; that is, $L = \mathcal{L}(M)$. Let $\ell = |Q|$ and let w be any string in L with $|w| = k$ where $k \geq \ell$. If no such string exists, then the theorem holds trivially, so let us assume that such a string does exist. Define the function $f_w : \{0, 1, 2, \ldots, k\} \rightarrow Q$ by

$$f_w(i) = q \quad \text{such that} \quad \mathcal{S}_{\text{M}}(w[1, i]) = \{q\}\,.$$

Here $w[i, j]$ denotes the substring $w[i]w[i+1]\cdots w[j]$ when $i \leq j$, and ε when $i > j$. More formally, we may define $w[i, j]$ inductively by

$$w[i, j] \stackrel{\text{def}}{=} \begin{cases} w[i] \circ w[i+1, j] & \text{if } i \leq j \\ \varepsilon & \text{if } i > j \end{cases}$$

Thus, $f_w(i)$ is the state of machine M after reading the first i symbols of the string w. Since M is deterministic, there is exactly one such state. As special cases, note that $f_w(0) = q_0$, where q_0 is the initial state, and $f_w(k) \in A$, since $w \in L$ by definition. Now observe that

$$|\,\{0, 1, 2, \ldots, k\}\,| \;=\; k + 1 \;>\; |Q|\,,$$

because w was chosen so that $k \geq |Q|$. Therefore, by the pigeonhole principle it follows that f_w cannot be injective. From this we may conclude that there exist distinct integers i and j such that $0 \leq i < j \leq k$ and $f_w(i) = f_w(j)$. We now claim that the substrings x, y, and z given by

$$\begin{aligned} x &= w[1, i] \\ y &= w[i+1, j] \\ z &= w[j+1, k] \end{aligned}$$

satisfy all the criteria necessary to show that w has the pumping property of length k. Obviously, $w = xyz$, $|xy| = j \leq k$, and $|y| = j - i > 0$, by the very definition of the substrings. Moreover, since $\mathcal{S}_{\text{M}}(x) = \{f_w(i)\} = \{f_w(j)\} = \mathcal{S}_{\text{M}}(xy)$, it follows from Equation (5.26) that $\mathcal{S}_{\text{M}}(xy) = \mathcal{S}_{\text{M}}(xy^n)$ for all $n \in \mathbb{N}$, and consequently, $\mathcal{S}_{\text{M}}(xyz) = \mathcal{S}_{\text{M}}(xy^n z)$ for all $n \in \mathbb{N}$. Since $w \in L$, it follows that $\mathcal{S}_{\text{M}}(xy^n z) = \mathcal{S}_{\text{M}}(w)$. Moreover, since $\mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset$, where A is the set of accept states in M, it follows that $xy^n z \in L$ for all $n \geq 0$. We have thus shown that for any $w \in L$ with $|w| \geq \ell$, where ℓ is the number of states in *some* machine that accepts L, the string w must have the pumping property of length k, which proves the lemma. □

Observe that lemma 1 is useful in showing that some languages are *not* regular, but it is of no use in showing that a given language *is* regular. That is, the "pumping property" is a *necessary* but not a *sufficient* condition for a language to be regular.

5.5.1 Predicate and Contrapositive Form

To actually apply the lemma 1, we will need to change into its contrapositive form. To simplify this manipulation, it will be convenient to introduce some new predicate notation. The first such predicate is

$$\Pi_k^L(w), \tag{5.27}$$

which is a predicate on strings, w, which is defined to be true if and only if w has the first pumping property of length k with respect to the language L. Note that the predicate depends on both the length k and the language L, which we make explicit by affixing k and L as subscript and superscript, respectively. Next, let **Reg** be a predicate on languages such that **Reg**(L) is true if and only if L is regular. We may now state the first pumping lemma more succinctly in terms of these predicates.

Lemma 2 (Predicate form of Lemma 1) *Let L be any language. Then*

$$\mathbf{Reg}(L) \implies \left[\exists\, k \in \mathbb{N},\ \forall w \in L,\ |w| \geq k\ \supset\ \Pi_k^L(w)\right]. \tag{5.28}$$

Since every element of a regular language must possess the "pumping property" for sufficiently large k, a language that fails to have this property for all k cannot be regular. Stated in this way, it is clearly the contrapositive of lemma 2 that allows us to draw conclusions about a language. Thus, we state the contrapositive as a lemma in its own right.

Lemma 3 (Contrapositive of Lemma 2) *Let L be any language. Then*

$$\left[\ \forall\, k \in \mathbb{N},\ \exists w \in L,\ |w| \geq k\ \wedge\ \neg\Pi_k^L(w)\ \right] \implies \neg\mathbf{Reg}(L). \tag{5.29}$$

The bracketed expression in Equation (5.29) is the negation of the bracketed expression in Equation (5.28)[5]. To make use of lemma 3, we must show that the *negation* of the predicate $\Pi_k^L(w)$ holds for *some* sufficiently long string $w \in L$. We now look at what is entailed by this negation. To make the notation a bit more concise, we'll encapsulate various properties as predicates on strings or languages. First, we define the "partition" and the "pump" predicates by

$$\mathbf{P}_k(w,x,y,z) \stackrel{\text{def}}{=} [\ w = xyz\ \wedge\ |xy| \leq k\ \wedge\ |y| \geq 1\ ] \tag{5.30}$$

$$\mathbf{Q}_L(x,y,z) \stackrel{\text{def}}{=} \forall n \in \mathbb{N}\,(xy^n z \in L) \tag{5.31}$$

respectively, where $k \in \mathbb{N}$. Note that equations Equation (5.30) and (5.31) actually define families of predicates. The predicate $\mathbf{P}_k(w,x,y,z)$ is true if and only if the strings x, y, and z form a "partition" of the string w that satisfies all of the language-independent[6] prerequisites of the first pumping property of length k, and $\mathbf{Q}_L(x,y,z)$ is true if and only if the remaining language-dependent[7] property holds. We may then define the complete "pumping property" predicate by

$$\Pi_k^L(w) \stackrel{\text{def}}{=} \exists\, x,y,z\,(\ \mathbf{P}_k(w,x,y,z)\ \wedge\ \mathbf{Q}_L(x,y,z)\ ), \tag{5.32}$$

which is true if and only if the string $w \in L$ possesses the pumping property of length k (with respect to the language L), as defined earlier. Recall that x, y, and z may be any strings in Σ^*, and needn't

[5] Recall that $A \supset B$ is simply shorthand for $\neg A \vee B$. Hence, the negation of $\exists x\,\forall y\,A(x,y) \supset B(x,y)$ is $\forall x\,\exists y\,A(x,y) \wedge \neg B(x,y)$.

[6] We refer to these properties as "language-independent" because they merely require xyz to be a proper partition of w with restrictions on the lengths of xy and y. These properties do not mention the language L at all.

[7] This predicate is always relative to a specific language L, as it requires all of the "pumped" strings to be in L. Hence, we refer to it as "language-dependent."

be elements of L themselves. We now have

$$\begin{aligned}
\neg\Pi_k^L(w) &\iff \neg\exists\, x,y,z\,[\,\mathbf{P}_k(w,x,y,z)\ \wedge\ \mathbf{Q}_L(x,y,z)\,]\\
&\iff \forall\, x,y,z\,[\,\neg\mathbf{P}_k(w,x,y,z)\ \vee\ \neg\mathbf{Q}_L(x,y,z)\,]\\
&\iff \forall\, x,y,z\,[\,\mathbf{P}_k(w,x,y,z)\ \supset\ \neg\mathbf{Q}_L(x,y,z)\,]\\
&\iff \forall\, x,y,z\,[\,\mathbf{P}_k(w,x,y,z)\ \supset\ \exists n\,\neg\,(xy^nz\in L)\,]\\
&\iff \forall\, x,y,z\,[\,\mathbf{P}_k(w,x,y,z)\ \supset\ \exists n\ (xy^nz\notin L)\,]\,.
\end{aligned}$$

We now assemble the various pieces, and state the final predicate form of the pumping lemma for regular languages as another lemma.

Lemma 4 (Contrapositive of Lemma 2, Version 2) *Let L be any language. If*

$$\forall\, k\in\mathbb{N},\ \exists w\in L,\ |w|\geq k\ \wedge\ \forall\, x,y,z\,[\,\mathbf{P}_k(w,x,y,z)\ \supset\ \exists n\ (xy^nz\notin L)\,]\,, \tag{5.33}$$

then L is not regular.

Thus, to show that a language L is not *not* regular, using the contrapositive of the first pumping lemma, we must show that for every $k\in\mathbb{N}$, we can find *some* string $w\in L$ with $|w|\geq k$ that fails to have the pumping property of length k. That is, *for all valid partitionings* of w into xyz (meaning $w=xyz$, $|xy|\leq k$, and $|y|\geq 1$), there is *some* $n\in\mathbb{N}$ such that $xy^nz\notin L$. We now show how to apply this lemma.

5.5.2 Applying the Pumping Lemma

Theorem 21 *The language $L_1=\{a^nb^n\mid n\geq 1\}$ is not regular.*

Proof: Let $k\in\mathbb{N}$ be given. We shall now select a string $w\in L_1$ that is of length at least k; since we are at liberty to chose any such w, we'll choose one that will make the rest of the proof as easy as possible. Let $w=a^kb^k$, which clearly has length greater than k. Now consider *any* partition $w=xyz$ such that $|xy|\leq k$ and $|y|\geq 1$. Then y must consist entirely of a's. (This is precisely why the string a^kb^k was convenient for this particular proof). This implies that xy^2z will have more a's than b's. More precisely

$$\left|xy^2z\right|_a\ =\ |w|_a+|y|_a\ >\ |w|_b+|y|_b\ =\ \left|xy^2z\right|_b\,, \tag{5.34}$$

since $|y|_a\geq 1$ while $|y|_b=0$. It follows immediately that xy^2z cannot be of the form a^nb^n, which always has the same number of a's ad b's. Consequently, $xy^2z\notin L_1$. We have thus shown that for any $k\in\mathbb{N}$, there is a string of length at least k (namely a^kb^k) that fails to have the pumping property of length k. Therefore, L_1 is not regular. □

Theorem 22 *The language $L_2=\{a^n\mid n\in\mathbb{N}\wedge n$ is prime$\}$ is not regular.*

Proof: Let $k \in \mathbb{N}$ be given. Let $w = a^p$ where p is any prime greater than $\max(2, k)$. Then $w \in L_2$ with $|w| \geq k$, as required. Now suppose that $w = xyz$ is any partitioning of w such that $|y| \geq 1$. Then

$$|xy^n z| \;=\; |xyz| + (n-1)\,|y| \;=\; p + (n-1)m,$$

where $m \geq 1$. We must now show that for at least one choice of $n \in \mathbb{N}$, $p + (n-1)m$ is not prime, from which it will follow that $xy^n z$ has non-prime length, and therefore is not in L_2. But this is easy to do, for if we let $n = p + 1$, then we have

$$\left|xy^{p+1}z\right| \;=\; p + pm \;=\; p(m+1), \tag{5.35}$$

which is a composite number, since both factors are greater than one. Therefore, we have thus shown that for any $k \in \mathbb{N}$, there is a $w \in L_2$ of length at least k (namely a^p, where $p \geq \max(2, k)$ is any prime) that fails to have the pumping property of length k. Therefore, L_2 is not regular. □

Notice that in the above proof, we did not require the partition to have the property that $|xy| \leq k$, as this constraint was of no help whatsoever. By omitting the constraint, we were actually considering a larger class of partitions of w, yet even among this larger class, none of them possessed the pumping property. In general, then, for the sake of clarity we may omit the constraint if we do not make use of it in any way.

5.5.3 Limitations of the First Pumping Lemma

It is important to remember that there are some non-regular languages that the first pumping lemma will fail to identify as such. For example, consider the language

$$L_3 \quad = \quad \left\{c^{n+k}a^n b^n \mid n > 0 \text{ and } k \geq 1\right\}.$$

Intuitively, this language is not regular, since no single finite automaton can ensure that the string of a's and the string of b's are of equal length for all $n > 0$; for large enough n the machine's capacity to "count" is exceeded. However, the First Pumping Lemma cannot demonstrate (directly) that L_3 is not regular. To see this, note that for any $w \in L_3$, we can always write $w = xyz$ where $x = \varepsilon$, $y = c$, and z is the remainder of the string. Then $xy^n z \in L_3$ for all $n \in \mathbb{N}$. Consequently, L_3 satisfies the pumping lemma, even though it is not regular.

5.6 Summary

Here is a summary of the definitions and behaviors of both DFAs and NFAs.

5.7 Exercises

1. Define a DFA that accepts each of the following languages, where the alphabet is assumed to be $\Sigma = \{a, b\}$. You need only draw the state diagrams.

Deterministic Finite Automata
Initialization • The state of the machine is set to the initial state, q_0. • The input string is written on the semi-infinite tape with its first symbol writen in the first cell of the tape. All other cells are considered blank. • The read head is positioned over the first cell of the tape, which is the first symbol of the input string.
Transitions depend on • The current state. • The symbol under the read head.
What happens at each step • The control unit advances to one of a finite number of states. • The read head moves exactly one cell to the right.
Halts when • The read head moves onto a blank cell of the tape.

Formal Definition of a Deterministic Finite Automaton as a 5-tuple: $(\Sigma, Q, q_0, A, \delta)$

Symbol	Name	Description
Σ	Input alphabet	Finite non-empty set for encoding input strings.
Q	State set	Finite non-empty set of state labels.
q_0	Initial state	The machine starts in this special state of Q.
A	Accept states	The machine accepts the string if it halts in one of these states.
δ	Transition function	A function from $Q \times \Sigma$ to Q.

(a) All strings in Σ^* that begin with aa or end with bb (or both).

(b) All strings in Σ^* that do not contain the substring aba.

(c) All strings in Σ^* with an even number of a's.

(d) All strings in Σ^* that contain no runs longer than two.

2. Construct a DFA or NFA that accepts each of the following languages over the alphabet $\Sigma = \{a, b, c\}$. You need only exhibit the state diagrams, but indicate whether it is a DFA or an

Nondeterministic Finite Automata
Initialization • *Exactly the same as a DFA.*
Transitions depend on • The current state. • Zero or more contiguous symbols, starting with the one under the read head and moving to the right. • A nondeterministic choice among available alternatives.
What happens at each step • The control unit advances to one of a finite number of states. • The read head moves zero or more cells to the right.
Halts when • The read head moves onto a blank cell of the tape. • An undefined transition is encountered.

Formal Definition of a Nondeterministic Finite Automaton as a 5-tuple: $(\Sigma, Q, q_0, A, \Delta)$

Symbol	Name	Description
Σ	Input alphabet	Finite non-empty set for encoding input strings.
Q	State set	Finite non-empty set of state labels.
q_0	Initial state	The machine starts in this special state of Q.
A	Accept states	The machine accepts the string if *some* sequence of nondeterministic choices allows the machine to halt (by reaching the end of the string) in one of these states.
Δ	Transition relation	A finite subset of $Q \times \Sigma^* \times Q$

NFA, and explain why it is one or the other. (We shall always regard a machine as a DFA if it fits the more restrictive definition, even though every DFA can also be regarded as an NFA.)

(a) $L_1 = a^*(bb)^*$

(b) $L_2 = (aa)^* \mid b^*$

(c) $L_3 = a^*(bb)^*(ccc)^*$

(d) $L_4 = \{\varepsilon\} \cup \left\{a^{2n}b^{2m+1} \mid n \geq 0 \text{ and } m \geq 0\right\}$

(e) $L_5 = \{w \in \Sigma^* \mid w \text{ does not contain two or more contiguous } a\text{s}\}$

(f) $L_6 = \{w \in \Sigma^* \mid w \text{ contains no runs of } a\text{'s or } b\text{'s longer than two}\}$

(g) $L_7 = \{w \in \Sigma^* \mid w \text{ does not contain all three letters}\}$

3. Let M_1 and M_2 be DFAs that accept the regular languages $a^*(bb)^*$ and $(aa)^*b^*$, respectively, where the alphabet is $\Sigma = \{a, b\}$. Use the construction described in section 5.3.1 to build an NFA that accepts the *union* of these two languages, then convert the NFA to a DFA. Draw all the relevant state diagrams to document the entire process.

4. Let M be an NFA. In each case below, briefly explain what must be true about the *state diagram* of M in order for the statement to be true.

(a) $\mathcal{L}(M)$ is the empty set.

(b) $\mathcal{L}(M)$ contains the empty string, ε.

(c) $\mathcal{L}(M)$ is infinite.

(d) The yields relation $\vdash$ is a partial function.

5. For each of the following languages over the alphabet $\Sigma = \{a, b, c\}$, either show that it is regular, or prove that it is not. To show that a language is regular, you may either give a regular expression for it or draw a state diagram of a DFA or NFA that accepts it.

(a) $L_1 = \{w \in \Sigma^* \mid |w| \text{ is a power of } 2\}$

(b) $L_2 = \{w \in \Sigma^* \mid |w| \text{ is a multiple of } 3\}$

(c) $L_3 = \{w \in \Sigma^* \mid |w|_a \text{ is even and } |w|_b \text{ is odd}\}$

(d) $L_4 = \{uvc \mid u \text{ and } v \text{ are strings in } \{a, b\}^* \text{ with } |u|_a = |v|_b\}$

(e) $L_5 = \{ucv \mid u \text{ and } v \text{ are strings in } \{a, b\}^* \text{ with } |u|_a = |v|_b\}$

Hint: of the two languages L_4 and L_5, one is regular and the other is not. Moreover, the one that is regular can be expressed in a *much* simpler way.

6. Given the 5-tuple descriptions of two DFAs, M_1 and M_2, over the same alphabet Σ, construct a 5-tuple for an NFA that

(a) Accepts $\mathcal{L}(M_1) \cup \mathcal{L}(M_2)$.

(b) Accepts $\mathcal{L}(M_1) \circ \mathcal{L}(M_2)$.

(c) Accepts $\overline{\mathcal{L}(M_1) \cap \mathcal{L}(M_2)}$.

In each case, construct the 5-tuples of the NFA by performing various set operations on the elements of the 5-tuples for M_1 and M_2. You may assume, without loss of generality, that the state sets of M_1 and M_2 are disjoint.

7. Consider the following modifications to the definition of a DFA. Would these changes affect the class of languages accepted by this type of machine? That is, would the modified class of machines be capable of accepting a larger, smaller, or the same set of languages? Give brief justifications for your answers.

(a) Only one accept state is allowed.

(b) No self-loops are allowed; that is, each transition must lead to a state that is distinct from the current state.

(c) No loops of any kind are allowed, making the state diagram a *directed acyclic graph*, or DAG.

Which of your answers change if these same modifications are applied to the definition of an NFA?

8. Consider the following modifications to the definition of an NFA. Would these changes affect the class of languages accepted by this type of machine? That is, would the modified class of machines be capable of accepting a larger, smaller, or the same set of languages? Give brief justifications for your answers.

   (a) A string is accepted if and only if *all* possible paths lead to an accept state.

   (b) Multiple start states are allowed, which are selected non-deterministically.

   (c) At most two transitions are allowed *out* of each state.

   (d) All accept states must be terminal; that is, no transitions are allowed *out* of an accept state.

9. Let M_1 and M_2 be DFAs that accept the regular languages aa^* and bb^*, respectively, where the alphabet is $\{a, b\}$. Build an NFA that accepts the *intersection* of these two languages by combining M_1 and M_2 appropriately. Show that this machine accepts the null language (that is, the language containing no strings, not even the empty string). The machine you will build could be denoted by

$$\overline{\left(\overline{M_1} \cup \overline{M_2}\right)},$$

   where the bar indicates the machine accepting the complement of the language accepted by the original machine.

10. Construct a DFA that recognizes strings in $\{0, 1\}^*$ with the following property. Each string, when interpreted as a binary number, is congruent to 3 modulo 5. That is, the remainder is 3 when divided by 5. The binary number should be written in the natural left-to-right manner; that is, with the most significant digits appearing first on the tape. Hint: if $s \in \{0, 1\}^*$ is congruent to k mod 5 when interpreted as a binary number, what is the string $s1$ congruent to? How about the string $s0$?

11. Give an example of a regular language L_1 and a non-regular language L_2 over the same alphabet such that the concatenation $L_1 \circ L_2$ is regular.

12. Give an example of a language over $\Sigma = \{a, b\}$ with the following property: if all the b's are *removed* from every string, the resulting language over $\Sigma' = \{a\}$ is not regular, but if all the b's are *replaced* with a's, then the resulting language is regular over Σ'.

13. Let L be any language over the alphabet Σ, and define L^P to be the set of *prefixes* of strings in L. That is,

$$L^P \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid wv = u \text{ for some } u \in L \text{ and } v \in \Sigma^*\}.$$

   Show that if L is regular, then L^P is also regular. Hint: To create a DFA that accepts L^P we can start with a DFA that accepts L and change nothing more than the set of accept states.

14. Given any two languages, L_1 and L_2, over the same alphabet Σ, we define their *quotient* to be

$$L_1/L_2 \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid wz \in L_1 \text{ for some } z \in L_2\}.$$

For example, $(a^*bc^*)/(bc)^* = a^* \mid a^*bc^*$. Show that if L_1 and L_2 are regular, so is L_1/L_2. Hint: As in problem 13, we can start with a DFA that accepts L_1 and simply modify the set of accept states. To assist in this, define $M^{(q)}$ to be $(\Sigma, Q, q, A, \delta)$; that is, the machine $M^{(q)}$ is exactly the same as M, but with the start state set to q. Now use various properties of regular languages to show that it is possible to construct a DFA that accepts L_1/L_2.

15. Let L be a regular language over the alphabet Σ and consider the language

$$L' \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid w \text{ is a substring of some } v \in L\}.$$

Is L' necessarily regular? Justify your answer.