Chapter 2

Finite Automata

2.1 The Basic Model

Finite automata model very simple computational devices or processes. These devices have a constant amount of memory and process their input in an online manner. By that we mean that the input is read a character at a time, and for each character, as it is read, a constant amount of processing is performed.

One might wonder whether such simple devices could be useful. In fact, they are widely used as controllers in mechanical devices such as elevators and automatic doors.

Example. A controller for the doors on the exit hatchway of a spaceship.

The rather simple spaceship in this example has an exit area with two doors: one leads out into space, the other leads into the spaceship interior, as shown in Figure 2.1.

The critical rule is that at most one door may be open at any given time (for otherwise all the air in the spaceship could vent out into the vacuum of space). The state of the doors is managed by a controller, a simple device which we describe next. The controller can receive signals or requests to open or close a door. For simplicity, we suppose that only one request can be received at one time. The diagram in Figure 2.2 specifies the operation of the controller. It is simply a directed graph with edge labels.

![Spaceship Exit Doors](image)

The vertex names in this graph specify the door state to which the vertex corresponds. The edges exiting a vertex indicate the actions taken by the controller on receiving the request labeling the edge. So for example, if both doors are closed and a request to open the exit door is received, then the controller opens this door and the state changes accordingly. On the other hand, if currently the interior door is open and a request to open the exit door
2.1. THE BASIC MODEL

is made, then nothing changes; i.e. the interior door remains open and the exit door remains closed. Note that there are many requests that result in no change. We can simplify the diagram so as to show only those requests that cause a change of state, as shown in Figure 2.3.

As requests are received the controller cycles through the appropriate states. To implement the device it suffices to have a 2-bit memory to record the current state (2 bits can distinguish 4 states) plus a little logic to decide what state change to make as requests are received.

Comment. The same design of controller would be needed for the doors on a torpedo hatchway in a submarine.

Many controllers need similarly modest amounts of memory and logic. They are more readily understood as simple devices, specified by means of a graph, rather than as general purpose computers.
Similar devices are also very useful as a front end in many programs, where they are used to partition the input into meaningful tokens. For example, a compiler, which is a program that processes input programs so they can then be executed, needs to partition the input into keywords, variables, numbers, operators, comments, etc.

For example, we could imagine recognizing the keywords if, then, else, end with the help of the following device or procedure, represented as a graph.

The processing begins at the vertex start and follows the path labeled by the input characters as they are read. If there is no edge to follow, then the computation fails; it recognizes a keyword if it finishes at a vertex represented by a double circle. For example, on reading the input string if, the finite automata goes from the start vertex to the vertex labeled if. Note that it is conventional for the start vertex, which need not be named start, to be indicated in the graph by means of a double arrow.

There are more details to handle in a compiler. In particular, the compiler has to be able to decide when it has reached the end of a keyword rather than being in the middle of a variable (e.g. ended) whose name begins with the keyword. But these essentially straightforward and tedious, albeit important, details are not the focus here.

Variables are another collection of strings that compilers need to recognize. Let us suppose variables consist of all strings of (lower-case) letters other than keywords, and to simplify the illustration below let us suppose there is just one keyword: if. Then the following procedure, shown in Figure 2.5, will recognize variables.

Notice that the graph has a self-loop; also, some of its edges have more than one label (alternatively, we could think of this as a collection of parallel edges, each with a distinct label). Again, to use the variable recognizer, the compiler follows the path spelled out by
the input. Any input other than the word *if* leads to a vertex represented as a double circle
and hence represents a variable.

### 2.1.1 A Precise Specification

We need to introduce a little notation and terminology. Recall that an *alphabet* is a set of
characters. For example:

- Binary = \{0,1\}
- Boolean = \{T,F\}
- English = \{a, b, \ldots, z\}

The usual way of writing an unspecified \(k\)-character alphabet is as \(\Sigma = \{a_1, a_2, \ldots, a_k\}\).

By convention, a finite automaton is named \(M\), or \(M_1, M_2, \ldots\) if there are several. As
we shall see subsequently, other devices will also be named by \(M\). A finite automaton is a
directed graph plus a finite alphabet \(\Sigma\) in which:

- One vertex is designated as the start vertex.
- A subset of the vertices, conventionally called \(F\), form the collection of Recognizing or
  *Final* vertices.
- Each edge is labeled by a character of \(\Sigma\).
- Each vertex has \(|\Sigma|\) outgoing edges, one for each character in \(\Sigma\).

In the corresponding drawing of the automaton, the start vertex is indicated by a double
arrow, and the vertices in \(F\) are indicated by drawing them with double circles.

\(M\) processes an input string \(s\) as follows: it determines the end of the path, which begins
at the start vertex and follows the labels spelled out by reading \(s\). For example, in the
Keyword Recognizer (see Figure 2.4), the input *if* specifies the path from the vertex *start*
to the vertex named *if*. If the end of the path is a vertex in \(F\) then \(M\) recognizes or *accepts*
\(s\); otherwise \(M\) *rejects* \(s\). Continuing the example, input *if* is recognized by the automaton
as the vertex *if* is in \(F\).

Thus we can view \(M\) as partitioning the set of all possible strings into those it accepts
and those it rejects. The set of strings accepted by \(M\) is often called the *language* accepted
by \(M\), and is sometimes written as \(L(M)\) or \(L\) for short. The topic we will study is what
sorts of collections of strings finite automata can accept.

One way to think about \(M\)’s processing is procedurally. At any point in time, having
read a portion \(u\) of its input, \(M\) will have one *currently reached* vertex, or *current* vertex for
short, namely the vertex \(p\) it reaches on reading string \(u\). It then reads the next character of
its input, \(a\) say, and follows the edge labeled \(a\) leaving vertex \(p\), which brings it to vertex \(q\).
\(q\) is now its current vertex. Given an input string \(s\), \(M\) will read it character by character,
advancing from vertex to vertex as described above, until all of $s$ it read, which brings it to some last vertex $r$, say. Then $s$ is accepted according to whether $r$ is in the accepting or final set of vertices, $F$.

Conventionally, the vertices of a finite automata are called its states and are written as $Q = \{q_1, q_2, \ldots, q_r\}$ rather than $V = \{v_1, v_2, \ldots, v_r\}$. The index $n$ tends to be reserved for the length of the input string. However, we will tend to use more descriptive names for the vertices in our examples.

For simplicity in drawing we replace multiple parallel edges by a single edge with multiple labels as shown in Figure 2.6.

Also, it is often convenient to omit a “sink” state (a non-final state which cannot be left); see Figure 2.7.

Note the descriptive names being used for the vertices. A description for a vertex $v$ specifies for which strings the automata reaches vertex $v$ starting from its start vertex. For example, in Figure 2.7, the vertex labeled ‘ye’ is reached on reading the input string ‘ye’ and no other input string causes this vertex to be reached.

Finally, it’s helpful to have a notation for representing labeled edges. We write $\delta(p, a) = q$ to mean that there is an edge labeled $a$ from vertex $p$ to vertex $q$; a more active interpretation is that starting at vertex (state) $p$, on reading $a$ the automata goes to vertex (state) $q$. Conventionally $\delta$ is called the transition function.

It is also useful to extend the destination (or transition) function to the reading of strings; this is denoted by $\delta^*$. $\delta^*(p, s) = r$ means that vertex $r$ is the destination on reading string $s$ starting at vertex $p$. For example, in the Variable Recognizer (see Figure 2.5), $\delta^*(\text{start}, \text{it}) = \text{“all other variables”}$.$^1$ Observe that $\delta^*(p, \lambda) = p$: reading the empty string, i.e. reading no characters, leaves the currently reached vertex unchanged.

---

$^1$When a name or string includes spaces we will often put quote marks around it to avoid ambiguity.
Some Particular Languages

- \( L(M) = \phi \) — This is the empty language, the language with no strings in it. So the finite automata \( M \) accepts nothing.

- \( L(M) = \{ \lambda \} \) — This is the language comprising the single string \( \lambda \). Note that \( \phi \neq \{ \lambda \} \); these two languages are distinct.

2.1.2 Examples of Automata

\( M_1 \), shown in Figure 2.8, accepts the following strings: those strings with at least one \( a \) and an even number of \( b \)'s following the last \( a \).

\[ \text{Figure 2.8: Example Automata } M_1 \]

\( M_2 \), shown in Figure 2.9, recognizes the set \( L(M_2) = \{ x \mid x \in \{ a, b \}^* \text{ and } x \text{ ends with an } a \} \).

\[ \text{Figure 2.9: Example Automata } M_2 \]

\( M_3 \), shown in Figure 2.10, recognizes the complement of the language recognized by \( M_2 \).

\( M_4 \), shown in Figure 2.11, keeps a running count mod 3 of the sum of the characters in its input that it has read, where the input alphabet \( \Sigma = \{ 0, 1, 2 \} \); it accepts a string if the corresponding total is 0 mod 3.

Suppose that we want to maintain the above running count, but mod \( k \), where the input alphabet is \( \Sigma = \{ 0, 1, \cdots, k-1 \} \). This can be done with a \( k \)-vertex automata. The automata has vertex set \( Q = \{ q_0, q_1, \cdots, q_{k-1} \} \); \( q_i \) will be its currently reached vertex if the running sum equals \( i \mod k \). It immediately follows that the start vertex is vertex \( q_0 \) and the final vertex set is \( F = \{ q_0 \} \). The transition function is also immediate: if the current running
Figure 2.10: Example Automata $M_3$: $L(M_3) = \{x \mid x$ is empty or $x$ ends with a $b\}$

Figure 2.11: Example Automata $M_4$: $L(M_4) = \{x \mid \sum_{i=1}^{x} x_i = 0 \mod 3$, where $x = x_1x_2 \cdots x_{|x|}\}$. Note that $\lambda \in L(M_4)$. 
sum equals \( h \pmod{k} \) and an \( i \) is the next input character read, then the new running sum is \( j = h + i \pmod{k} \); so the current vertex changes from \( q_h \) to \( q_j \). We write \( \delta(q_h, i) = q_j \) where \( j = h + i \pmod{k} \). (See Figure 2.12.)

\[
\begin{array}{ccc}
\text{digit sum} & \rightarrow & \text{digit sum} \\
= h \pmod{k} & \xrightarrow{i} & = h + i \pmod{k}
\end{array}
\]

Figure 2.12: Example Automata \( M_5 \): The Transition Function

### 2.1.3 Designing Automata

The first issue is to decide what has to be remembered as the input is read. The easiest thing, it might seem, is to remember the whole string. But one cannot in general, as the automata has a finite memory.

For example, with \( \Sigma = \{a, b\} \), consider recognizing those strings with an odd number of \( b \)'s. It suffices to keep track of whether an even or an odd number of \( b \)'s have been read so far. This leads to the transition diagram shown in Figure 2.13(a). To obtain the full automata, it’s enough to note that at the start zero \( b \)'s have been read (an even number), and the Final vertex corresponds to an odd number of \( b \)'s having been read; see Figure 2.13(b).

![Automata recognizing strings with an odd number of b's](image)

Next, consider the language \( L(M) = \{w \mid w \text{ contains at least one } a\} \), with \( \Sigma = \{a, b\} \). The natural vertices correspond to “no \( a \) read” and “at least one \( a \) read”. The transitions are now evident and are shown in Figure 2.14.

Finally, consider the language \( L(M) = \{w \mid w \text{ contains at least two } b \text{’s}\} \), with \( \Sigma = \{a, b\} \). The natural three vertices correspond to: “no \( b \) read”, “one \( b \) read”, and “at least two \( b \)’s read”. Again, the transitions are evident and are shown in Figure 2.15.

The vertex descriptors help to check that the automata is doing what it is supposed to do. In particular, when designing an automata, you should check that the edge labels and the descriptors are consistent.
More specifically, suppose that \((p, q)\) is an edge with label \(a\). Let \(S_p\) be \(p\)’s descriptor and let \(S_q\) be \(q\)’s. e.g. in Figure 2.8, the start vertex descriptor is the set of all strings with no \(a\)’s. i.e. \(\{\lambda, b, bb, bbb, \cdots\}\).

Then you want to confirm that for each string \(s \in S_p\), string \(sa \in S_q\); i.e. appending an \(a\) to a string in \(S_p\) always yields a string in \(S_q\).

This may seem very difficult to do, but in fact it tends to be straightforward. For example, consider Figure 2.8 again, and the edge (start, even) labeled \(a\). We need to confirm that the descriptor ‘even number of \(b\)’s after last \(a\)’ includes all strings of the form ‘no \(a\)’ (strings in \(S_{\text{start}}\)) followed by one \(a\). Clearly, there are zero \(b\)’s after the last (just read) \(a\), and zero is an even number of \(b\)’s. Of course it would be rather tedious to write out many such checks; one just wants to do them in one’s head. However, it is all too easy to assume one has got the design right and not bother with the checks; this is how many simple mistakes are overlooked.

There are three other very simple checks.

1. When nothing has been read, the automata is at the start vertex. Consequently, the empty string must be in the descriptor for the start vertex.

2. Each string causes exactly one vertex to be reached. Consequently, the descriptors for distinct vertices are disjoint. In addition, the union of the descriptors is the set of all strings.

3. The union of the sets of strings specified by the descriptors for the set of final vertices specifies exactly those strings that you want the automata to recognize. More formally, let \(F = \{q_1, q_2, \cdots, q_l\}\); then \(S_{q_1} \cup S_{q_2} \cup \cdots \cup S_{q_l}\) is the set of strings recognized by the automata.
In Section 2.5 we will show that performing the full set of checks ensures the correctness of the automata.

2.1.4 Recognizing More Complex Sets

We consider sets (languages) built up by means of the following three operations: union, concatenation, and star (defined below). These are called the Regular Operations.

- Union: \[ A \cup B = \{ w \mid w \in A \text{ or } w \in B \} \]
- Concatenation: \[ A \circ B = \{ uv \mid u \in A \text{ and } v \in B \} \] This is the set of strings comprising a string from \( A \) followed by a string from \( B \).
- Star: \[ A^* = \{ x_1x_2\cdots x_k \mid k \geq 0 \text{ and } x_i \in A, \text{ for } i = 1, 2, \ldots, k \} \] Note that setting \( k = 0 \) shows \( \lambda \in A^* \). This is the set of strings consisting of the concatenation of zero or more strings from \( A \).

As we will see, if the sets \( A \) and \( B \) can be recognized by finite automata, then so can their union, concatenation, and the “star” of each set.

Recognizing \( A \cup B \) The approach is natural: we use a device \( M \) that comprises two finite automata: one, \( M_A \), that accepts \( A \), and one, \( M_B \), that accepts \( B \). The device \( M \) runs \( M_A \) and \( M_B \) in parallel (simultaneously) on its input and accepts the input exactly if at least one of \( M_A \) and \( M_B \) reaches a final vertex or state on this input. (In this proof, for clarity, we use the term states rather than vertices.)

The only challenge is to show how to have \( M \) be a finite automata. To describe this we need some more notation. Let \( M_A = (Q_A, \Sigma, \delta_A, \text{start}_A, F_A) \), \( M_B = (Q_B, \Sigma, \delta_B, \text{start}_B, F_B) \), and \( M = (Q, \Sigma, \delta, \text{start}, F) \). The states of \( M \) are pairs, where each pair consists of a state of \( M_A \) and a state of \( M_B \); we call these pair-states for clarity. The purpose of the \( M_A \) state in the pair-state is to simulate the computation of \( M_A \), and similarly that of the \( M_B \) state is to simulate \( M_B \)'s computation. Formally, we define \( Q = Q_A \times Q_B \): each pair-state of \( M \) is a 2-tuple, consisting of a state of \( M_A \) and a state of \( M_B \) (in that order). The idea is that:

Assertion 2.1.1. \( M \) will be in state \( q = (q_A, q_B) \) on reading input \( w \) exactly if \( M_A \) is in state \( q_A \) and \( M_B \) is in state \( q_B \) after reading input \( w \).

The definitions of start, \( \delta \), and \( F \) are now immediate.

- The start pair-state of \( M \) is the pair consisting of the start states of \( M_A \) and \( M_B \): \( \text{start} = (\text{start}_A, \text{start}_B) \).
- An edge of \( M \), labeled \( a \) say, connects pair state \( p = (p_A, p_B) \) to pair-state \( q = (q_A, q_B) \), exactly if \( p_A \) is connected to \( q_A \) by an \( a \)-labeled edge in \( M_A \) and \( p_B \) is connected to \( q_B \) by an \( a \)-labeled edge in \( M_B \); in other words, \( \delta(p, a) = \delta((p_A, p_B), a) = (\delta_A(p_A, a), \delta_B(p_B, a)) = (q_A, q_B) = q. \)
The final pair-states of $M$ are those pairs that include either a final state of $M_A$ or a final state of $M_B$ (or both): $F = (F_A \times Q_B) \cup (Q_A \times F_B)$.

**Lemma 2.1.2.** $M$ recognizes the language $A \cup B$, $L(M) = A \cup B$ for short.

**Proof.** We need to do two things:

1. Show that if $w$ is recognized by $M$, $w \in L(M)$, then $w \in A \cup B$, and

2. show that if $w \in A \cup B$, then $w \in L(M)$.

The first substep is to verify Assertion 2.1.1. But this is easy to see: consider the computation of $M$ on input $w$, but only pay attention to what is happening to the first component of its pair-state: this is identical to the computation of $M_A$ on input $w$. Likewise, if we pay attention only to the second component of the pair state, we see that the computation is identical to that of $M_B$ on input $w$. In particular, $M$ reaches pair-state $q = (q_A, q_B)$ on reading input $w$ exactly if $M_A$ is in state $q_A$ and $M_B$ is in state $q_B$ after reading input $w$, which is what Assertion 2.1.1 states.

Next, we show (1). So suppose that $M$ recognizes $w$, that is, on input $w$, $M$ reaches a final pair-state, $q = (q_A, q_B)$ say. By Assertion 2.1.1, $M$ reaches pair-state $q = (q_A, q_B)$ on input $w$ exactly if $M_A$ reaches state $q_A$ on input $w$ and $M_B$ reaches state $q_B$. Now, by definition, $q$ is a final pair-state of $M$ exactly if $q \in (F_A \times Q_B) \cup (Q_A \times F_B)$. Consider the case that $q = (q_A, q_B) \in F_A \times Q_B$; then $q_A \in F_A$, that is $M_A$ reaches a final state on input $w$, which happens exactly if $w \in A$. Similarly, in the case that $q = (q_A, q_B) \in Q_A \times F_B$, $q_B \in F_B$, that is $M_B$ reaches a final state on input $w$, which happens exactly if $w \in B$. This shows that if $M$ recognizes input $w$, then either $w \in A$ or $w \in B$ (or possibly both); this means that $w \in A \cup B$, which is the statement in (1) above.

Finally, we show (2). If $w \in A \cup B$, then either $w \in A$ or $w \in B$ (or possibly both). Consider the case that $w \in A$. Then $w$ is recognized by $M_A$. That is, on input $w$, $M_A$ reaches a final vertex (state), $q_A$ say, with $q_A \in F_A$. At the same time, $M_B$, on input $w$, reaches some vertex $q_B \in Q_B$. By Assertion 2.1.1, $M$, on input $w$ reaches the pair-state $(q_A, q_B) \in F_A \times Q_B$. So $M$ recognizes $w$ since $(q_A, q_B) \in F = (F_A \times Q_B) \cup (Q_A \times F_B)$. An essentially identical argument shows that if $w \in B$ then $M$ again recognizes $w$. In sum, if $w \in A \cup B$, then $M$ recognizes $w$, that is $w \in L(M)$.

**Corollary 2.1.3.** Let $A$ and $B$ be languages recognized by finite automata. Then there is another finite automata recognizing the language $A \cup B$.

**Recognizing** $A \circ B$ Again, we suppose we have finite automata $M_A$ recognizing $A$ and $M_B$ recognizing $B$. We then build a device $M$ to recognize $A \circ B$, or $AB$ for short. $w \in AB$ exactly if we can divide $w$ into two parts, $u$ and $v$, with $u \in A$ and $v \in B$. This immediately suggests how $M$ will try to use $M_A$ and $M_B$. First it runs $M_A$ on substring $u$ and then it runs $M_B$ on substring $v$. The requirements are that $M_A$ recognize $u$ and $M_B$ recognize $v$; that is,
on input $u$, $M_A$ goes from its start state to a final state, and on input $v$, $M_B$ also goes from its start state to a final state. The difficulty is for $M$ to know when $u$ ends, for there may be multiple possibilities. For example, consider the languages $A = \{ \text{strings of one or more a's} \}$ and $B = \{ \text{strings of one or more b's} \}$. On the string $aabb$ it would not be correct to switch from simulating $M_A$ to simulating $M_B$ after reading a single $a$, while it would be correct on the string $abb$. The solution is to keep track of all possibilities. We do not explore this further at this point, as this is more readily understood using the technique of nondeterminism, which is the topic of the next subsection.

Likewise, we defer the description of how to recognize $A^*$ to the next subsection.

### 2.2 Nondeterministic Finite Automata

Non-deterministic Finite Automata, NFAs for short, are a generalization of the machines we have already defined, which are often called Deterministic Finite Automata by contrast, or DFAs for short. The reason for the name will become clear later.

As with a DFA, an NFA is simply a graph with edges labeled by single letters from the input alphabet $\Sigma$. There is one structural change.

For each vertex $v$ and each character $a \in \Sigma$, the number of edges exiting $v$ labeled with $a$ is unconstrained; it could be 0, 1, 2 or more edges.

This obliges us to redefine what an automaton $M$ is doing, given an input $x$. Quite simply, $M$ on input $x$ determines all the vertices that can be reached by paths starting at the start vertex, $start$, and labeled $x$. If any of these reachable vertices is in the set of Accepting or Final vertices then $M$ is defined to accept $x$. Another way of looking at this is that $M$ recognizes $x$ exactly if there is some path (and possibly more than one) labeled $x$, going from $start$ to a vertex (state) $q \in F$; such a path is called a recognizing or accepting path.

**Example 2.2.1.** Let $A = \{ w \mid \text{the third to last character in } w \text{ is a } \text{“b”} \}$. The machine $M$ with $L(M) = A$ is shown in Figure 2.16.

Note that on input $abbb$ all four vertices can be reached, whereas on input $abab$ the second from rightmost vertex cannot be reached.

![Figure 2.16: NFA recognizing $\{ w \mid \text{the third to last character in } w \text{ is a } \text{“b”} \}$](image)
CHAPTER 2. FINITE AUTOMATA

As we will see later, the collection of languages recognized by NFAs is exactly the collection of languages recognized by DFAs. That is, for each NFA $N$ there is a DFA $M$ with $L(M) = L(N)$. Thus NFAs do not provide additional computational power. However, they can be more compact and easier to understand, and as a result they are quite useful.

Indeed, the language from Example 2.2.1 is recognized by the DFA shown in Figure 2.17.

![Figure 2.17: DFA recognizing $\{w \mid$ the third to last character in $w$ is a $b\}$](image)

By naming the last three characters or all the characters in the string, each vertex specifies the strings for which the DFA reaches it.

It is helpful to consider how one might implement an NFA. One way is to maintain a bit vector over the vertices, so as to record which vertices are reachable currently. That is, suppose that reading input $w$ can lead to any of the vertices in set $R$, say. The set $S$ of vertices reachable on reading a further character $a$ is obtained by following all the edges labeled $a$ that exit any of the vertices in $R$. So in the automata of Figure 2.16, the set of vertices reachable on reading input $b$ is \{“any string”, “last char read is a $b$”\}, and the set of vertices reachable on reading $ba$ is \{“any string”, “next to last char read is a $b$”\}.

We define recognition for NFAs as follows: An NFA $M$ recognizes input string $w$ if there is a path from $M$’s start vertex to a final vertex (state) such that the labels along the edges of the path, when concatenated, form the string $w$. That is, were we to follow this path, reading the edge labels as we go, the string read would be exactly $w$; we say this path has label $w$. The implementation described above checks every path with label $w$; for acceptance, one needs to have at least one label $w$ path leading to a final vertex. We call such a path a recognizing or accepting path for $w$.

To define NFAs formally, we need to redefine the destination (transition) functions $\delta$. Now $\delta$ takes as its arguments a set of vertices (states) $R \subseteq V$ and a character $a \in \Sigma$ and produces as output another set of vertices (states) $S \subseteq V$: $\delta(R, a) = S$. The meaning is that the set of vertices (states) reachable from $R$ on reading $a$ is exactly the set of vertices (states) $S$. 
δ* is also redefined. δ*(R, w) = S means that the set of vertices (states) S are the possible destinations on reading w when starting from a vertex (state) in R. So, in particular, w is recognized by M, w ∈ L(M), exactly if δ*(\{start\}, w) ∩ F ≠ φ; i.e., when starting at vertex start, on reading w, at least one destination vertex (state) is an accepting or final vertex (state).

δ is often defined formally as follows: δ : 2^Q × Σ → 2^Q. In this context, 2^Q means the collection of all possible subsets of Q, and this is the set of states of the NFA, so δ does exactly what it should. Its inputs are (i) a set of vertices (states), that is one of the elements of the collection 2^Q, and (ii) a character in Σ; its output is also a set of vertices (states).

Next, we add one more option to NFAs: edges labeled with the empty string, as shown in Figure 2.18. We call these edges λ-edges for short.

The meaning is that if vertex (state) p is reached, then vertex (state) q can also be reached without reading any more input. Again, the same languages as before are recognized by NFAs with λ-edges and by DFAs; however, the λ-edges are very convenient in creating understandable machines. The meanings of the destination functions δ and δ* are unchanged.

**Example 2.2.2.** Let A and B be languages over the alphabet Σ that are recognized by NFAs M_A and M_B, respectively. Then the NFA shown in Figure 2.19 recognizes the language A ∪ B.

Its first step, prior to reading any input, is to go to the start vertices for machines M_A and M_B. Then the computation in these two machines is performed simultaneously. The set of final vertices for M is the union of the final vertices for M_A and M_B; thus M reaches a final vertex on input w exactly if at least one of M_A or M_B reaches a final vertex on input w. In other words L(M) = L(M_A) ∪ L(M_B).

**Deterministic vs. Nondeterministic** Another way of viewing the computation of an NFA M on input x is that the task is to find a path labeled x from the start vertex to a final vertex if there is one, and this is done correctly by (inspired) guessing. This process of correct guessing is called *Non-deterministic computation*. Correspondingly, if there is no choice or uncertainty in the computation, it is said to be *Deterministic*. This is not an imple-
mentable view; it is just a convenient way to think about what nondeterminism achieves.

### 2.2.1 Closure Properties

**Definition 2.2.3.** A language is said to be **regular** if it can be recognized by an NFA.

The regular languages obey three closure properties: if $A$ and $B$ are regular, then so are $A \cup B$, $A \circ B$ and $A^*$. We have already seen a demonstration of the first of these. We now give a simpler demonstration of the first property and then show the other two.

First, the following technical lemma is helpful.

**Lemma 2.2.4.** Let $M$ be an NFA. Then there is another NFA, $N$, which has just one final vertex, and which recognizes the same language as $M$: $L(N) = L(M)$.

**Proof.** The idea is very simple: $N$ is a copy of $M$ with one new vertex, new\_final, added; new\_final is $N$’s only final vertex. $\lambda$-edges are added from the vertices that were final vertices in $M$ to new\_final, as illustrated in Figure 2.20.

![Figure 2.20: NFA N with a single final vertex](image)

Recall that a recognizing path in an NFA goes from its start vertex to a final vertex. It is now easy to see that $M$ and $N$ recognize the same language. The argument has two parts.

First, any recognizing path in $M$, for string $w$ say, can be extended by a single $\lambda$-edge to reach $N$’s final vertex, and thereby becomes a recognizing path in $N$; in addition, the edge addition leaves the path label unchanged (as $w\lambda = w$). It follows that if $M$ recognizes string $w$ then so does $N$. In other words, $L(M) \subseteq L(N)$.

Second, removing the last edge from a recognizing path in $N$ yields a recognizing path in $M$ having the same path label (for the removed edge has label $\lambda$). It follows that if $N$ recognizes string $w$ then so does $M$. In other words, $L(N) \subseteq L(M)$.

Together, these two parts yield that $L(M) = L(N)$, as claimed.

We are now ready to demonstrate the closure properties.
If $A$ and $B$ are regular then so is $A \cup B$. We show this using the NFA $M_{A\cup B}$ in Figure 2.21. $M_{A\cup B}$ is built using $M_A$ and $M_B$, NFAs with single final vertices recognizing $A$ and $B$, respectively. It has an additional start vertex $\text{start}$, connected to $M_A$ and $M_B$’s start vertices by $\lambda$-edges, and an additional final vertex $\text{final}$, to which $M_A$ and $M_B$’s final vertices are connected by $\lambda$-edges.

As in the earlier construction, each recognizing path in $M_{A\cup B}$ corresponds to a recognizing path in one of $M_A$ and $M_B$, and vice-versa, and thus $w \in L(M_{A\cup B})$ if and only if $w \in A \cup B$. We now show this more formally.

**Lemma 2.2.5.** $M_{A\cup B}$ recognizes $A \cup B$.

**Proof.** We show that $A \cup B = L(M_{A\cup B})$ by showing that $L(M_{A\cup B}) \subseteq A \cup B$ and that $A \cup B \subseteq L(M_{A\cup B})$.

To show that $L(M_{A\cup B}) \subseteq A \cup B$, it is enough to show that if $w \in L(M_{A\cup B})$ then $w \in A \cup B$ also. So let $w \in L(M_{A\cup B})$ and let $P$ be a recognizing path for $w$ in $M_{A\cup B}$. Removing the first and last edges of $P$, which both have label $\lambda$, yields a recognizing path in one of $M_A$ or $M_B$, also having label $w$. This shows that if $w \in L(M_{A\cup B})$, then either $w \in L(M_A) = A$ or $w \in L(M_B) = B$ (of course, it might be in both, but this argument does not reveal this); in other words, if $w \in L(M_{A\cup B})$ then $w \in A \cup B$.

To show that $A \cup B \subseteq L(M_{A\cup B})$, it is enough to show that if $w \in A \cup B$ then $w \in L(M_{A\cup B})$ also. Now, if $w \in A \cup B$, then either $w \in A$ or $w \in B$ (or possibly both). Consider the case that $w \in A$. Then there is a recognizing path $P_A$ for $w$ in $M_A$. Preceding $P_A$ with the appropriate $\lambda$-edge from $\text{start}$ and following it with the $\lambda$-edge to $\text{final}$, yields a recognizing path in $M_{A\cup B}$, also having label $w$. This shows that if $w \in L(M_A) = A$ then $w \in L(M_{A\cup B})$ also. Similarly, if $w \in B$ then $w \in L(M_{A\cup B})$ too. Together, this gives that if $w \in A \cup B$ then $w \in L(M_{A\cup B})$. 

If $A$ and $B$ are regular then so is $A \circ B$. We show this using the NFA $M_{AB}$ displayed in Figure 2.22. $M_{AB}$ is built using $M_A$ and $M_B$, NFAs with single final vertices, recognizing $A$ and $B$, respectively. $M_{AB}$ comprises a copy of $M_A$ plus a copy $M_B$, plus one additional edge. $M_A$’s start vertex is also $M_{AB}$’s start vertex, and $M_B$’s final vertex is $M_{AB}$’s only final vertex. Finally, the new edge, $e$, joins $M_A$’s final vertex to $M_B$’s start vertex.
CHAPTER 2. FINITE AUTOMATA

The idea of the construction is that a recognizing path in $M_{AB}$ corresponds to recognizing paths in $M_A$ and $M_B$ joined by edge $e$. It then follows that $w \in L(M_{AB})$ if and only if $w$ is the concatenation of strings $u$ and $v$, $w = uv$, with $u \in A$ and $v \in B$. We now show this more formally.

Lemma 2.2.6. $M_{AB}$ recognizes $A \circ B$.

Proof. We show that $L(M_{AB}) = A \circ B$ by showing that $L(M_{AB}) \subseteq A \circ B$ and that $A \circ B \subseteq L(M_{AB})$.

To show that $L(M_{AB}) \subseteq A \circ B$, it is enough to show that if $w \in L(M_{AB})$ then $w \in A \circ B$ also. So let $w \in L(M_{AB})$ and let $P$ be a recognizing path for $w$ in $M_{AB}$. Removing edge $e$ from $P$ creates two paths $P_A$ and $P_B$, with $P_A$ being a recognizing path in $M_A$ and $P_B$ a recognizing path in $M_B$. Let $u$ and $v$ be the path labels for $P_A$ and $P_B$, respectively. So $u \in A$ and $v \in B$. As $e$ is a $\lambda$-edge, $w = u\lambda v = uv$. This shows that $w \in A \circ B$.

To show that $A \circ B \subseteq L(M_{AB})$, it is enough to show that if $u \in A$ and $v \in B$ then $uv \in L(M_{AB})$. So let $u \in A$, $v \in B$, let $P_A$ be a recognizing path for $u$ in $M_A$, and let $P_B$ be a recognizing path for $v$ in $M_B$. Then form the path $P = P_A, e, P_B$ in $M_{AB}$; clearly, this is a recognizing path. Further, it has label $u\lambda v = uv$. So $uv \in L(M_{AB})$.

If $A$ is regular then so is $A^*$

$M_{A^*}$ is built using $M_A$, an NFA with a single final vertex recognizing $A$. $M_{A^*}$ comprises a copy of $M_A$ plus a new start vertex, plus two additional $\lambda$-edges, $e$ and $f$. $e$ joins $M_{A^*}$’s start vertex to $M_A$’s start vertex, and $f$ joins $M_A$’s final vertex to $M_{A^*}$’s start vertex. $M_{A^*}$’s start vertex is also its final vertex.

The construction is based on the observation that re-

We show this using the NFA $M_{A^*}$ in Figure 2.23.
Lemma 2.2.7. \( M_{A^*} \) recognizes \( A^* \).

Proof. We show that \( L(M_{A^*}) = A^* \) by showing that \( L(M_{A^*}) \subseteq A^* \) and that \( A^* \subseteq L(M_{A^*}) \).

To show that \( L(M_{A^*}) \subseteq A^* \), it is enough to show that if \( w \in L(M_{A^*}) \) then \( w \in A^* \) also. So let \( w \in L(M_{A^*}) \) and let \( P \) be a recognizing path for \( w \) in \( M_{A^*} \). Removing all instances of edges \( e \) and \( f \) from \( P \) creates \( k \) subpaths, for some \( k \geq 0 \), where each subpath is a recognizing path in \( M_A \). Let the path labels on these \( k \) subpaths be \( u_1, \ldots, u_k \), respectively. So \( u_i \in A \), for \( 1 \leq i \leq k \). As \( e \) and \( f \) are \( \lambda \)-edges, \( w = \lambda u_1 \lambda u_2 \lambda \cdots \lambda u_k \lambda = u_1 u_2 \cdots u_k \). Thus \( w \in A^* \).

To show that \( A^* \subseteq L(M_{A^*}) \), it is enough to show that if \( w \in A^* \) then \( w \in L(M_{A^*}) \). If \( w \in A^* \) then \( w = u_1 u_2 \cdots u_k \), with \( u_1, u_2, \ldots, u_k \in A \), for some \( k \geq 0 \). As \( u_i \in A \), there is an accepting path \( P_i \) in \( M_A \) for \( u_i \). Let \( P \) be the following path in \( M_{A^*} \): \( e, P_1, f, e, P_2, f, \cdots, e, P_k, f \) (for \( k = 0 \) we intend the path of zero edges). \( P \) is a recognizing path in \( M_{A^*} \), and it has label \( u_1 u_2 \cdots u_k = w \), as \( e \) and \( f \) are \( \lambda \)-edges. Thus \( w \in L(M_{A^*}) \).

1.2.2 Every regular language is recognized by a DFA

Let \( N \) be an NFA. We show how to construct a DFA \( M \) with \( L(M) = L(N) \).

Recall that to implement an NFA we keep track of the set of currently reachable vertices. This is what \( M \) will do with its vertices or states, which we call superstates henceforth. Each of \( M \)'s superstates is a subset of the set \( Q \) of vertices or states in \( N \). So \( M \)'s collection (set) of superstates is the power set of \( Q \), \( 2^Q \).

The relation between superstates of \( M \) and sets of vertices in \( N \) is specified by the following assertion.

Assertion 2.2.8. On input \( w \), \( M \) reaches superstate \( R \) if and only if \( N \) can reach the set of states \( R \).

To achieve Assertion 2.2.8, we define the transition function \( \delta \) for \( M \) to be identical to the transition function \( \delta \) for \( N \). Note that this specifies where \( M \)'s edges go.

To make sure the assertion is correct initially, that is for \( w = \lambda \), we set \( M \)'s start superstate to be the set of \( N \)'s vertices that \( N \) can reach on input \( \lambda \). Finally, we set the final superstates of \( M \) to be those superstates that include one or more of the final vertices of \( N \); so if \( F \) is the set of \( N \)'s final vertices, then \( R \) is a final superstate for \( M \) exactly if \( R \cap F \neq \phi \).

Lemma 2.2.9. \( L(M) = L(N) \).
Proof. We begin by arguing by induction on the length of the input string read so far that Assertion 2.2.8 is true. The base case is for the empty string: the claim is true by construction. For the inductive step, if the claim is true for strings of length \( k \), then we argue that it is also true for strings of length \( k + 1 \). By the inductive hypothesis, on an input \( w \) of length \( k \), \( R \) is the superstate reached by \( M \) if and only if \( R \) is the set of vertices reachable by \( N \). But then, on input \( wa \), \( M \) reaches superstate \( \delta(R, a) \), and \( N \) can reach the set of vertices \( \delta(R, a) \), so the claim is true for strings of length \( k + 1 \) also.

It remains to consider which strings the two machines recognize. \( M \) recognizes input \( w \) if and only if on input \( w \) it reaches superstate \( R \) where \( R \cap F \neq \phi \); and \( N \) recognizes input \( w \) if and only if it can reach a set of vertices \( S \) where \( S \cap F \neq \phi \). But we have shown that \( S = R \). So the two machines recognize the same collection of strings, that is \( L(M) = L(N) \). \( \square \)

Implementation Remark The advantage of an NFA is that it may have far fewer vertices (states) than a DFA recognizing the same language, and thus use much less memory to store them. On the other hand, when running the machines, the NFA may be less efficient, as the set of reachable vertices had to be computed as the input is read, whereas in the DFA one just needs to follow a single edge. Which choice is better depends on the particulars of the language and the implementation environment.

2.3 Non Regular Languages

We turn now to a method for demonstrating that some languages are not regular. For example, as we shall see, \( L = \{a^n b^n | n \geq 1 \} \) is not a regular language. Intuition suggests that to recognize \( L \) we would need to count the number of \( a \)’s in the input string; in turn, this suggests that any automata recognizing \( L \) would need an unbounded number of vertices. But how do we turn this into a convincing argument?

We use a proof by contradiction. So suppose for a contradiction that \( L \) were regular. Then there must be a DFA \( M \) that accepts \( L \). \( M \) has some number \( k \) of vertices. Let’s consider feeding \( M \) the input \( a^k b^k \). Look at the sequence of \( k + 1 \) vertices \( M \) goes through on reading \( a^k \): \( r_0 = start, r_1, r_2, \cdots, r_k \), where some of the \( r_i \)’s may be a repeated instance of the same vertex. This is illustrated in Figure 2.24.

As there are only \( k \) distinct vertices, there is at least one vertex that is visited twice in this sequence, \( r_i = r_j \), say, for some pair \( i, j, 0 \leq i < j \leq k \). This is shown in Figure 2.25.

In fact we see that \( r_{j+1} = r_{h+1} \) if \( j + 1 \leq k \), \( r_{j+2} = r_{h+2} \) if \( j + 2 \leq k \), and so forth. But all we will need for our result is the presence of one loop in the path, so we will stick with the representation in the figure above.

It is helpful to partition the input into four pieces: \( a^i, a^{j-i}, a^{k-j}, b^k \). The first \( a^i \) takes
M from vertex $r_0$ to $r_i$ (to the start of the loop), the next $a^{j-i}$ takes $M$ from $r_i$ to $r_j$ (once around the loop), the final $a^{k-j}$ takes $M$ from $r_j$ to $r_k$, and $b^k$ takes $M$ from $r_k$ to a final vertex, as shown in figure 2.26.

![Figure 2.25: The path traversed on input $a^k$.](image)

![Figure 2.26: The path traversed on input $a^k$.](image)

What happens on input $a^i a^{j-i} a^{k-j} b^k = a^{k+j-i} b^k$? The initial $a^i$ takes $M$ from $r_0$ to $r_i$, the first $a^{j-i}$ takes $M$ from $r_i$ to $r_j = r_i$ (once around the loop), the second $a^{j-i}$ takes $M$ from $r_i$ to $r_j$ (around the loop again), the $a^{k-j}$ takes $M$ from $r_j$ to $r_k$, and the $b^k$ takes $M$ from $r_k$ to a final vertex. So $M$ accepts $a^{k+j-i} b^k$, which is not in $L$ as $j - i > 0$. This is a contradiction, and thus the initial assumption, that $L$ was regular, must be incorrect.

We now formalize the above approach in the following lemma.

**Lemma 2.3.1. (Pumping Lemma)** Let $L$ be a regular language. Then there is a number $p = p_L$, the pumping length for $L$, with the property that for each string $s$ in $L$ of length at least $p$, $s$ can be written as the concatenation of 3 substrings $x$, $y$, and $z$, that is $s = xyz$, and these substrings satisfy the following conditions.

1. $|y| > 0$,
2. $|xy| \leq p$,
3. For each integer $i \geq 0$, $xy^iz \in L$.

**Definition 2.3.2.** We say $s$ can be pumped if $s \in L$ can be written as the concatenation of substrings $x$, $y$ and $z$, and conditions 1–3 of the Pumping Lemma hold.

**Proof.** Again, we use a proof by contradiction. So suppose that $L$ were regular and let $M$ be a DFA accepting $L$. Now suppose that $M$ has $p$ states. $p$, the number of states in $M$, will be the pumping length for $L$. 

Let $s$ be a string in $L$ of length $n \geq p$. Write $s$ as $s = s_1s_2 \cdots s_n$, where each $s_i$ is a character in $\Sigma$, the alphabet for $L$. Consider the substring $s' = s_1s_2 \cdots s_p$. We look at the path $M$ follows on input $s'$. It must go through $p + 1$ vertices, and as $M$ has only $p$ vertices, at least one vertex must be repeated. Let \( \text{start} = r_0, r_1, \ldots, r_p \) be this sequence of vertices and suppose that $r_i = r_j$, where $0 \leq i < j \leq p$, is a repeated vertex, as shown in Figure 2.27. Let $x$ denote $s_1 \cdots s_i$, $y$ denote $s_{i+1} \cdots s_j$, and $z$ denote $s_{j+1} \cdots s_p s_{p+1} \cdots s_n$. The path traversed on input $s' = s_1s_2 \cdots s_p$ is illustrated in Figure 2.28.

As $j > i$, $|y| = j - i > 0$. Also $|xy| = |s_1 \cdots s_j| = j \leq p$. So (1) and (2) are true.

Clearly, $xz$ is recognized by $M$, for $x$ takes $M$ from $r_0$ to $r_i = r_j$ and $z$ takes $M$ from $r_j$ to a final vertex.

Similarly $xy^iz$ is recognized by $M$ for any $i \geq 1$, for $x$ takes $M$ from $r_0$ to $r_i$, each repetition of $y$ takes $M$ from $r_i$ (back) to $r_j = r_i$, and then the $z$ takes $M$ from $r_j$ to a final vertex. Thus (3) is also true, proving the result.

Now, to show that a language $L$ is non-regular we use the pumping Lemma in the following way. We begin by assuming that $L$ is regular so as to obtain a contradiction. Next, we assert that there is a pumping length $p$ such that for each string $s$ in $L$ of length at least $p$ the three conditions of the Pumping Lemma hold. The next (and more substantial) task is to choose a particular string $s$ to which we will apply the conditions of the Pumping Lemma, and condition (3) in particular, so as to obtain a contradiction.

**Example 2.3.3.** Let us look at the language $L = \{a^nb^n|n \geq 1\}$ again. The argument showing that $L$ is not regular proceeds as follows.
Step 1. Suppose, so as to obtain a contradiction, that $L$ were regular. Then $L$ must have a pumping length $p \geq 1$ such that for any string $s \in L$ with $|s| \geq p$, $s$ can be pumped.

Step 2. Choose $s$. Recall that we chose the string $s = a^p b^p$.

Step 3. By pumping $s$, obtain a contradiction. As $s$ can be pumped (for $s \in L$ and $|s| = 2p \geq p$), we can write $s = xyz$, with $|y| > 0$, $|xy| \leq p$, and $xy^iz \in L$ for all integer $i \geq 0$.

As the first $p$ characters of $s$ are all $a$’s, the substring $xy$ must also be a string of all $a$’s. By condition (3), with $i = 0$, we have that $xz \in L$; but the string $xz$ has removed $|y|$ a’s from $s$, that is $xz = a^{p-|y|} b^p$, and this string is not in $L$ since $p - |y| \neq p$. This is a contradiction for we have shown that $xz \in L$ and $xz \notin L$.

Step 4. Consequently the initial assumption is incorrect; that is, $L$ cannot be recognized by a DFA, so $L$ is not regular.

Let me stress the sequence in which the argument goes. First the existence of pumping length $p$ for $L$ is asserted (different regular languages $L$ may have different pumping lengths, but each regular language containing arbitrarily long strings will have a pumping length). Then a suitable string $s$ is chosen. The length of $s$ will be a function of $p$. $s$ is chosen so that when it is pumped a string outside of $L$ is obtained. An important point about the pumped substring $y$ is that while we know $y$ occurs among the first $p$ characters of $s$, we do not know exactly which ones form the substring $y$. Consequently, a contradiction must arise for every possible substring $y$ of the first $p$ characters in order to show that $L$ is not regular.

There is a distinction here: for a given value of $p$, $s$ is fully determined; for instance, in Example 2.3.3 for $p = 3$, $s = aaabba$. By contrast, all that we know about $x$ and $y$ is that together they contain between 1 and 3 characters, and that $y$ has at least one character. There are 6 possibilities in all for the pair $(x, y)$, namely: $(\lambda, a)$, $(\lambda, aa)$, $(\lambda, aaa)$, $(a, a)$, $(a, aa)$, $(aa, a)$. In general, for $|xy| \leq p$, there are $\frac{1}{2}p(p+1)$ choices of $x$ and $y$. The argument leading to a contradiction must work for every possible choice of $p$ and every possible partition of $s$ into $x$, $y$, and $z$. Of course, you are not going to give a separate argument for each case given that there are infinitely many cases. Rather, the argument must work regardless of the value of $p$ and regardless of which partition of $s$ is being considered.

Next, we show an alternative Step 3 for Example 2.3.3.

Alternative Step 3. In applying Condition 3, use $i = 2$ (instead of $i = 0$), giving $xyyz \in L$. But $xyyz$ adds $|y|$ a’s to $xyz$, namely $xyyz = a^{p+|y|} b^p$ and this string is not in $L$ since $p + |y| \neq p$. This is a contradiction for we have shown that $xyyz \in L$ and $xyyz \notin L$.

In Example 2.3.3 both pumping down ($i = 0$) and pumping up ($i = 2$) will yield a contradiction. This is not the case in every example. Sometimes only one direction works, and then only for the right choice of $s$.

A common mistake. Not infrequently, an attempted solution may try to specify how $s$ is partitioned into $x$, $y$, and $z$. In Example 2.3.3, this might take the form of stating that $x = \lambda$, $y = a^p$, and $z = b^p$, and then obtaining a contradiction for this partition. This is an incomplete argument, however. All that the Pumping Lemma states is that there is a
partition; it does not tell you what the partition is. The argument showing a contradiction must work for every possible partition.

Example 2.3.4. Let $K = \{ww^R \mid w \in \{a, b\}^*\}$. For example, $\lambda, abba, abaaba \in K$, $ab, a, aba \notin K$. We show that $K$ is not regular.

Step 1. Suppose, so as to obtain a contradiction, that $K$ were regular. Then $K$ must have a pumping length $p \geq 1$ such that for any string $s \in K$ with $|s| \geq p$, $s$ can be pumped.

Step 2. Choose $s$. We choose the string $s = a^p bba^p$.

Step 3. By pumping $s$, obtain a contradiction. As $s$ can be pumped (for $|s| = 2p + 2 \geq p$), we can write $s = xyz$, with $|y| > 0$, $|xy| \leq p$, and $xy^iz \in K$ for all integer $i \geq 0$.

As the first $p$ characters of $s$ are all $a$'s, the substring $xy$ must also be a string of all $a$'s. By condition (3), with $i = 0$, we have that $xz \in K$; but the string $xz$ has removed $|y|$ $a$'s from $s$, that is $xz = a^{p-|y|} bba^p$, and this string is not in $K$ since it is not in the form $ww^R$ for any $w$. (To see this, note that if it were in the form $ww^R$, as $xz$ contains two $b$'s, one of them would be in the $w$ and the other in the $w^R$; this forces $w = a^{p-|y|} b$ and $w^R = ba^p$, and this is not possible as $p - |y| \neq p$. This is a very detailed explanation, which we will not spell out to this extent in future. Noting that $p - |y| \neq p$ will suffice.)

This is a contradiction for we have shown that $xz \in K$ and $xz \notin K$.

Step 4. Consequently the initial assumption is incorrect; that is, $K$ cannot be recognized by a DFA, so $K$ is not regular.

Another common mistake. I have seen attempted solutions for the above example that set $s = w^p(w^R)^p$. This is not a legitimate definition of $s$. For $w$ is an arbitrary string, so such a definition does not fully specify the string $s$, that is it does not spell out the characters forming $s$. To effectively apply the Pumping Lemma you are going to need to choose an $s$ in which only $p$ is left unspecified. So if you are told, for example, that $p = 3$, then you must be able to write down $s$ as a specific string of characters (in Example 2.3.4, with $p = 3$, $s = aaabbbaaa$).

For the assumption in the application of the Pumping Lemma is that the language $L$ under consideration is recognized by a DFA. What is not known is the assumed size of the DFA. What the argument leading to a contradiction shows is that regardless of its size, the supposed DFA cannot recognize $L$, but this requires the argument to work for any value of $p$.

Example 2.3.5. $J = \{w \mid w$ has equal numbers of $a$’s and $b$’s\}. We show that $J$ is not regular. To do this we introduce a new technique. Note that if $A$ and $B$ are regular then so is $A \cap B = \overline{(A \cup \overline{B})}$ (for the union and complement of regular languages are themselves regular; alternatively, see Problem 3).

Now note that $R = \{a^i b^j \mid i, j \geq 0\}$ is regular. Thus if $J$ were regular, then $J \cap R = \{a^i b^j \mid i \geq 0\}$ would also be regular. But we have already shown that $J \cap R$ was not regular in Example 2.3.3. Thus $J$ cannot be regular either. (Strictly, this is a proof by contradiction.)
We could also proceed as in Example 2.3.3, applying the Pumping Lemma to string \( s = a^p b^p \). The exact same argument will work.

**Question**  Does the proof of the Pumping Lemma work if we consider a \( p \)-state NFA that accepts \( L \), rather than a \( p \)-state DFA? Justify your answer.

### 2.4  Regular expressions

Regular expressions provide another, elegant and simple way of describing regular languages. We denote regular expressions by small letters such as \( r, s, r_1, \) etc. As with regular languages, they are defined with respect to a specified alphabet \( \Sigma \). They are defined by the following six rules, comprising three base cases and three combining rules.

We begin with the base cases.

1. \( \phi \) is a regular expression; it represents \( \phi \), the empty set.
2. \( \lambda \) is a regular expression; it represents the language \( \{ \lambda \} \), the language containing the empty string alone.
3. \( a \) is a regular expression; it represents the language \( \{ a \} \).

For the combining rules, let regular expressions \( r \) and \( s \) represent languages \( R \) and \( S \). The rules follow.

4. \( r \cup s \) is a regular expression; it represents language \( R \cup S \).
5. \( r \circ s \) is a regular expression; it represents language \( R \circ S \). We write \( rs \) for short.
6. \( r^* \) is a regular expression; it represents language \( R^* \).

Parentheses are used to indicate the scope of an operator. It is also convenient to introduce the notation \( r^+ \); while not a standard regular expression, it is shorthand for \( rr^* \), which represents the language \( RR^* \). Also \( \Sigma \) is used as a shorthand for \( a_1 \cup a_2 \cup \cdots \cup a_k \), where \( \Sigma = \{ a_1, a_2, \cdots, a_k \} \). Finally, \( L(r) \) denotes the language represented by regular expression \( r \).

**Examples.** Let \( \Sigma = \{ a, b \} \).

1. \( a^* ba^* = \{ w \mid w \text{ contains exactly one } b \} \).
2. \( \Sigma^* bab \Sigma^* = \{ w \mid w \text{ contains } bab \text{ as a substring} \} \).
3. \( (\Sigma \Sigma)^* = \{ w \mid w \text{ has even length} \} \).
4. \( a \Sigma^* a \cup b \Sigma^* b \cup a \cup b = \{ w \mid w \text{ has the same first and last character} \} \).
5. \((a \cup \lambda)b^* = ab^* \cup b^*\).

6. \(a^*\phi = \phi\). (Why?\(^2\))

Next we show that regular expressions represent exactly the regular languages.

**Lemma 2.4.1.** Let \(r\) be a regular expression. There is an NFA \(N_r\) that recognizes the language represented by \(r\): \(L(N_r) = L(r)\).

**Proof.** The proof is by induction on the number of operators (union, concatenation, and star) in regular expression \(r\).

The base case is for zero operators, which are also the base cases for specifying regular expressions. It is easy to give DFAs that recognize the languages specified in each of these base cases and this is left as an exercise for the reader.

For the inductive step, suppose that \(r\) is given by one of the recursive definitions, \(r_1 \cup r_2\), \(r_1 \circ r_2\), or \(r^*\). Since \(r_1\) and \(r_2\) if it occurs, contain fewer operators than \(r\), we can assume by the inductive hypothesis that there are NFAs recognizing the languages represented by regular expressions \(r_1\) and \(r_2\). Then Lemmas 2.2.5–2.2.7 provide the NFAs recognizing the languages \(r_1\) and \(r_2\).

We can conclude that there is an NFA recognizing \(L(r)\).

To prove the converse, that every regular language can be represented by a regular expression takes more effort. To this end, we introduce yet another variant of NFAs, called GNFAs (for Generalized NFAs).

In a GNFA each edge is labeled by a regular expression \(r\) rather than by one of \(\lambda\) or a character \(a \in \Sigma\). We can think of an edge labeled by regular expression \(r\) being traversable on reading string \(x\) exactly if \(x\) is in the language represented by \(r\). String \(w\) is recognized by a GNFA \(M\) if there is a path \(P\) in \(M\) from its start vertex to a final vertex such that \(P\)'s label, the concatenation of the labels on \(P\)'s edges, forms a regular expression that includes \(w\) among the set of strings it represents.

In more detail, suppose \(P\) consists of edges \(e_1, e_2, \cdots, e_k\), with labels \(r_1, r_2, \cdots, r_k\), respectively; then \(P\) is a \(w\)-recognizing path if \(w\) can be written as \(w = w_1w_2\cdots w_k\) and each \(w_i\) is in the language represented by \(r_i\), for \(1 \leq i \leq k\).

Clearly, every NFA is a GNFA, so it will be enough to show that any language recognized by a GNFA can also be represented by a regular expression.

We begin with two simple technical lemmas.

**Lemma 2.4.2.** Suppose that GNFA \(M\) has two vertices \(p\) and \(q\) with \(h\) edges from \(p\) to \(q\), edges \(e_1, e_2, \cdots, e_h\). Suppose further that these edges are labeled by regular expressions \(r_1, r_2, \cdots, r_h\), respectively. Then replacing these \(h\) edges by a new edge \(e\) labeled \(r_1 \cup r_2 \cup \cdots \cup r_h\), or \(r\) for short, yields a GNFA \(N\) recognizing the same language as \(M\).

\(^2\)As there is no string in \(\phi\), to follow a string in \(a^*\) with a string from \(\phi\) is not possible.
2.4. REGULAR EXPRESSIONS

Proof. The edge replacement is illustrated in Figure 2.29.

First we show that \( L(M) \subseteq L(N) \). Suppose that \( M \) recognizes string \( w \). Let \( P \) be a \( w \)-recognizing path in \( M \). Replacing each instance of edge \( e_i \), for \( 1 \leq i \leq h \), by edge \( e \) yields a new path \( P' \) in \( N \), with \( P' \) being \( w \)-recognizing (for a substring \( w_i \) read on traversing edge \( e_i \), and hence represented by \( r_i \), is also represented by \( r \) and so can be read on traversing edge \( e \) also). This shows \( w \in L(N) \).

Next we show that \( L(N) \subseteq L(M) \). Suppose that \( N \) recognizes string \( w \). Let \( P \) be a \( w \)-recognizing path in \( N \). Suppose that edge \( e \) is traversed \( j \) times in \( P \), and let \( w_1, w_2, \ldots, w_j \) be the substrings of \( w \) read on these \( j \) traversals. Each substring \( w_g \), \( 1 \leq g \leq j \), is represented by regular expression \( r \), and as \( r = r_1 \cup \cdots \cup r_h \), each \( w_g \) is represented by one of \( r_1, r_2, \ldots, r_h \); so let \( w_g \) be represented by \( r_{i_g} \), where \( 1 \leq i_g \leq h \). Replacing the corresponding instance of \( e \) in \( P \) by \( e_{i_g} \) yields a path \( P' \) in \( M \) which is also \( w \)-recognizing. This shows that \( w \in L(M) \). \( \square \)

Lemma 2.4.3. Suppose that GNFA \( M \) has three vertices \( u, v, q \), connected by edges labeled as shown in Figure 2.30 and further suppose that they are the only edges incident on \( q \). Then removing vertex \( q \) and creating edge \( (u, v) \) with label \( r_1r_2^*r_3 \) yields a GNFA \( N \) recognizing the same language as \( M \).

Proof. First we show that \( L(M) \subseteq L(N) \). Suppose that \( w \in L(M) \), and let \( P \) be a \( w \)-recognizing path in \( M \). We build a \( w \)-recognizing path \( P' \) in \( N \). Consider a segment of the path from \( u \) to \( v \) going through \( q \) in \( P \). It consists of edge \( (u, q) \), followed by some \( k \geq 0 \) repetitions of edge \( (q, q) \), followed by edge \( (q, v) \). This subpath has label \( r_1r_2^*r_3 \). But all strings represented by \( r_1r_2^*r_3 \) are represented by \( r_1r_2^*r_3 \), and so we can replace this subpath by the new edge \( (u, v) \) in \( N \). Thus \( w \) is recognized by \( N \).

Next, we show that \( L(N) \subseteq L(M) \). Suppose that \( w \in L(N) \), and let \( P \) be a \( w \)-recognizing path in \( N \). We build a \( w \)-recognizing path \( P' \) in \( M \). Consider an instance of edge \( e \) with \( e = (u, v) \) on \( P \), if any. Suppose string \( w_i \) is read on traversing \( e \). Then \( w_i \) is in the
CHAPTER 2. FINITE AUTOMATA

language represented by \( r_1 r_2^k r_3 \) for some \( k \geq 0 \) (for this is what \( r_1 r_2^3 r_3 \) means: \( r_1 r_2^3 r_3 = r_1 r_3 \cup r_1 r_2 r_3 \cup r_1 r_2^2 r_3 \cup r_1 r_2^3 r_3 \cup \cdots \)). So we can replace this instance of edge \( e \) with the edge sequence \((u, q)\) followed by \( k \) instances of edge \((q, q)\), followed by edge \((q, v)\). When all instances of \( e \) in \( P \) are replaced in this manner, the resulting path \( P' \) in \( M \) is still \( w \)-recognizing, so \( w \in M \).

\[ \text{Lemma 2.4.4. Let } M \text{ be a GNFA. Then there is a regular expression } r \text{ representing the language recognized by } M: L(r) = L(M). \]

**Proof.** We begin by modifying \( M \) so that its start vertex has no in-edge (if need be by introducing a new start vertex) and so that it has a single final vertex with no out-edges (again, by adding a new vertex, if needed). Also, for each pair of vertices, if there are several edges between them, they are replaced by a single edge by applying Lemma 2.4.2. Let \( N_0 \) be the resulting machine and suppose that it has \( n + 2 \) vertices \{start, \( q_1 \), \( q_2 \), \ldots, \( q_n \), final\}.

We construct a sequence \( N_1, N_2, \ldots, N_n \) of GNFAs, where \( N_i \) is \( N_{i-1} \) with \( q_i \) removed and otherwise modified so that \( N_{i-1} \) and \( N_i \) recognize the same language, for \( 1 \leq i \leq n \).

Thus \( L(N_n) = L(M) \). But \( L_n \) has two vertices, start and final, joined by a single edge labeled by a regular expression \( r \), say. Clearly, the language recognized by \( N_n \) is \( L(r) \). So it remains to show how to construct \( N_i \) given \( N_{i-1} \).

**Step 1.** Let \( q = q_i \) be the vertex being removed. For each pair of vertices \( u, v \neq q \) such that there is a path \( u, q, v \) we make a new copy \( q_{u,v} \) of \( q \) as shown in Figure 2.31, and then remove the vertex \( q \) and the edges incident on it.

![Figure 2.31: Vertex duplication in Step 1.](image)

Clearly, a path segment \((u, q \text{ repeated } k \geq 1 \text{ times}, v)\) in \( N_{i-1} \) has the same label as the path segment \((u, q_{u,v} \text{ repeated } k \text{ times}, v)\), and consequently the labels on the recognizing paths in \( N_{i-1} \) and the machine following the Step 1 modification are the same.

**Step 2.** In turn, for each vertex \( q_{u,v} \), apply Lemmas 2.4.3 and 2.4.2 to the subgraph formed by \( u, q_{u,v} \), and \( v \) for each pair of vertices \( u, v \). The resulting GNFA is \( N_i \). Clearly, \( L(N_i) = L(N_{i-1}) \). The effect of the application of these lemmas is illustrated in Figure 2.32.

Note that in fact we could have performed the change achieved by Step 2 without introducing the vertices \( q_{u,v} \). They are here just to simplify the explanation.

\( \square \)
2.5 Correctness of Vertex Specifications

Given a Finite Automata, with a vertex specification $S_v$ for each vertex $v$, how can one determine whether the vertex specifications are correct, in the sense of exactly specifying those strings for which the automata can reach $v$, when starting at the start vertex?

**Definition 2.5.1.** A specification $S_v$ for a vertex $v$ in a DFA or NFA is a set of strings $s$.

\[ S_v = \{ \text{even length strings} \} \]

For the specifications to be correct they will need to satisfy three properties, including forward and backward consistency, which we define next. We begin by considering automata with no $\lambda$-edges.

Let $e = (u, v)$ be an edge of the finite automata labeled by $a$. Suppose that $u$ and $v$ have specifications $S_u$ and $S_v$, respectively.

**Definition 2.5.2.** $S_u$ and $S_v$ are forward-consistent with respect to edge $e$ if for each $s \in S_u$, $sa \in S_v$.

**Definition 2.5.3.** $S_v$ is backward-consistent if for each $s \in S_v$ such that $|s| \geq 1$, there is a vertex $u$ and an edge $(u, v)$ labeled $a$ for some $a \in \Sigma$, with the property that $s = s'a$ and $s' \in S_u$.

**Definition 2.5.4.** The specifications for a finite automata are consistent if

1. The specifications are forward-consistent with respect to every edge.
2. The specifications are backward-consistent.
3. $\lambda \in S_{\text{start}}$ and $\lambda \notin S_v$ for $v \neq \text{start}$.
Lemma 2.5.5. The specifications for a finite automata with no $\lambda$-edges are consistent exactly if they specify the strings that can reach each vertex.

Proof. Suppose for a contradiction that some specification is incorrect, i.e. it fails to specify exactly those strings that can reach its vertex.

Case 1. There is a shortest string $s$ that is incorrectly included in some specification $S_v$.
By (3), $s \neq \lambda$. So $s$ can be written as $s = s'a$, where $a \in \Sigma$. By backward-consistency, $s' \in S_u$ for some vertex $u$ where $(u, v)$ is an edge labeled $a$. Now $s'$ is correctly included in $S_u$ as $s$ was a shortest incorrectly placed string; but as $s'$ is correctly in $S_u$, then in fact $s$ is correctly in $S_v$. So this case does not arise.

Case 2. There is a shortest string $s$ that is incorrectly missing from some specification $S_v$.
Again, $s \neq \lambda$. So $s$ can be written as $s = s'a$, where $a \in \Sigma$. Let $U$ be the set of vertices with an edge labeled $a$ into $v$ (i.e. for each $u \in U$, there is an edge $(u, v)$ labeled by $a$). Then $s'$ reaches some vertex in $U$, vertex $u$ say. As $|s'| < |s|$, $s' \in S_u$. But then $s \in S_v$ by forward consistency. So this case does not occur either.

We conclude that all the specifications are correct. \hfill \square

We account for the $\lambda$-edges by making appropriate small changes to the definition of consistency. What we want to do is to modify the automata by removing all $\lambda$-edges and introducing appropriate replacements. Let $M$ be an NFA; we build an equivalent NFA $M'$ with no $\lambda$ edges, as follows.

1. Make all vertices reachable on input $\lambda$ into start vertices.

2. For each pair of vertices $(u, v)$ and for each label $a \in \Sigma$, if there is path from $u$ to $v$ labeled by $a$, introduce an edge $(u, v)$ labeled by $a$, and then remove all $\lambda$-edges.

We need to explain what it means to have more than one start vertex: $M'$ is simply understood as starting at all its start vertices simultaneously.

It is not hard to see that $M'$ recognizes exactly the same strings as the original machine. The result of Lemma 2.5.5 applies to $M'$. Note that the vertex specifications used for $M$ are the ones being used for $M'$ too. Consequently, one can check the correctness of the specifications for $M$ by applying Lemma 2.5.5 to the corresponding machine $M'$.

Exercises

1. Consider the Finite Automata shown in Figure 2.33.
   Answer the following questions regarding each of the automata.
   i. Name its start vertex.
   ii. List its final vertices.
   iii. List the series of vertices the finite automata goes through as the following input is read: $ababb$. 
iv. Is $ababb$ recognized by the automata?

2. Draw the graphs of Deterministic Finite Automata recognizing the following languages. In each subproblem, the alphabet being used is $\Sigma = \{a, b\}$.

i. The empty set $\phi$.

ii. The set containing just the empty string: $\{\lambda\}$.

iii. The set of all strings: $\Sigma^*$.

iv. The set of all strings having at least one character: $\{w \mid |w| \geq 1\}$.

**Sample solution.**

![Automata diagram]

v. The set of all strings of length two or more: $\{w \mid |w| \geq 2\}$.

vi. The set of all strings that begin with an $a$.

vii. The set of all strings that end with a $b$.

viii. The set of all strings containing $aa$ as a substring.

ix. The set of all strings containing at least four $a$’s.

x. The set of all strings either starting with an $a$ and ending with a $b$, or starting with a $b$ and ending with an $a$.

xi. The set of all strings such that no two $b$’s are adjacent.

xii. The set of all strings excepting $aba$: $\{w \mid w \neq aba\}$.

xiii. The set of all strings with alternating $a$’s and $b$’s.

xiv. The set of all strings with only $a$’s in the even positions.
xv. The set of all strings of even length whose second symbol is a $b$.
xvi. The set of all strings containing at least two $a$’s.
xvii. The set of all strings that contain $aba$ as a substring.
xviii. The set of strings in which all the $a$’s come before all the $b$’s.

3. Consider the construction from Section 2.1.4. Show how to modify this construction so as to give a (Deterministic) Finite Automata $M$ recognizing $A \cap B$, assuming that $A$ and $B$ can both be recognized by Finite Automata. Explain briefly why your construction is correct; that is, explain why $L(M) = A \cap B$.

4. Each of the following languages can be obtained by applying set operations (one of union, intersection, or complement) to simpler languages. By building Deterministic Finite Automata recognizing the simpler languages and then combining or modifying them, build Deterministic Finite Automata to recognize the following languages. For intersections, Problem 3 may be helpful. In each subproblem, the alphabet being used is $\Sigma = \{a, b\}$.

i. The set of all strings that contain at least one $a$ or at least two $b$’s.
ii. The set of all strings that do not contain the substring $aa$.
iii. The set of all strings other than the empty string: $\{w \mid w \neq \lambda\}$.
iv. The set of all strings other than $a$ or $bb$: $\{w \mid w \neq a, bb\}$.
v. The set of all strings containing at least one $a$ and at least one $b$.
vi. The set of all strings containing at least one $a$ and at most two $b$’s.

vii. The set of all strings with an even number of $a$’s and an odd number of $b$’s.
viii. The set of all strings with an even number of $a$’s and no adjacent $a$’s.
ix. The set of all strings that start with an $a$ and end with a $b$.
x. The set of all strings that have either 2 or 3 $b$’s and that have exactly 2 $a$’s.

xi. The set of all strings that include both an $a$ and a $b$.

5. Draw the graphs of NFAs recognizing the following languages.

i. $L = a^*$, using a 1-vertex NFA. Why is your solution not a DFA if the input alphabet is $\{a, b\}$?

ii. $L_1 = \{w \mid w$ has $aa$ as a substring$\}$, using a 3-vertex NFA.

iii. $L_2 = \{w \mid w$ ends with $bb\}$, using a 3-vertex NFA.

iv. $L_3 = \{w \mid w$ has both $aa$ and $bb$ as substrings$\}$, using a 9-vertex NFA (8 vertices is doable).

v. $L_4 = \{w \mid w$ is of even length or the second symbol in $w$ is a $b$ (or both)$\}$, using a 5-vertex NFA.
6. Using the methods of Section 2.2.1, give the graphs of NFAs that recognize the following languages.

i. $A \cup B$, where $A = \{w \mid w \text{ begins with an } a\}$, $B = \{x \mid x \text{ ends with a } b\}$, and $A, B \subseteq \{a, b\}^*$.

ii. $C \circ D$ where $C = \{w \mid |w| \geq 2\}$, $D = \{x \mid x \text{ contains } aa \text{ as a substring}\}$, and $C, D \subseteq \{a, b\}^*$.

iii. $E^*$, where $E = \{w \mid \text{ all characters in even positions in } w \text{ are } a's\}$, and $E \subseteq \{a, b\}^*$.

iv. $F = \emptyset^*$ (recall that $\emptyset$ is the empty language). Trust the construction. What strings, if any, are in $\emptyset^*$?

7. i. Construct an NFA recognizing the language $L = \{ba, bab\}^*$.

ii. Convert this NFA to a DFA recognizing the same language using the method of Section 2.2.2. You need show only the portion of the DFA reachable from the start vertex.

8. Let $L$ be a regular language. Define the reverse of $L$, $L^R = \{w \mid w^R \in L\}$, i.e. $L^R$ contains the reverse of strings in $L$ (for $(x^R)^R = x$ for any string $x$). Show that $L^R$ is also regular. Hint. Suppose that $M$ is a DFA (or an NFA if you prefer) recognizing $L$. Construct an NFA $M^R$ that recognizes $L^R$; $M^R$ will be based on $M$. Remember to argue that $L(M^R) = L^R$.

9. For each of the following regular expressions answer the following questions:

i. $(a \cup b)^* a (a \cup b)^*$.

ii. $(a \cup b)^* a (a \cup b)^* b (a \cup b)^*$.

iii. $(\lambda \cup a) b^*$.

iv. $(\lambda \cup aa) b^*$.

a. Give two strings in the language represented by the regular expression.

b. Give two strings not in the language represented by the regular expression.

c. Describe in English the language represented by the regular expression.

10. a. Using the method of Lemma 2.4.1, give an NFA to recognize the language represented by regular expression $(a \cup bb)^*(aba)$.

b. Repeat for the regular expression $(aba)(a \cup bb)^*$.

11. Using the method of Lemma 2.4.4, give a regular expression representing the language recognized by the NFA in figure 2.34.

12. Use the Pumping Lemma to show that the following languages are not regular.
i. \( L = \{a^ib^i \mid i \geq 0\} \).

**Sample solution.** Suppose, so as to obtain a contradiction, that \( L \) were regular. Then \( L \) must have a pumping length \( p \geq 1 \) such that for any string \( s \in L \) with \(|s| \geq p\), \( s \) can be pumped.

Choose \( s \) to be the string \( s = a^pb^p \).

As \( s \in L \) and \(|s| = 2p \geq p \), \( s \) can be pumped. In other words, we can write \( s = xyz \), with \(|y| > 0\), \(|xy| \leq p\), and \( xy^iz \in L \) for all integer \( i \geq 0 \).

By pumping \( s \), we will obtain a contradiction, as follows.

As the first \( p \) characters of \( s \) are all \( a \)'s, the substring \( xy \) must also be a string of all \( a \)'s. As \( xy^iz \in L \) for all integer \( i \geq 0 \), on setting \( i = 0 \), we have that \( xz \in L \); but the string \( xz \) has removed \(|y| a \)'s from \( s \), that is \( xz = a^{p-|y|}b^p \), and this string is not in \( L \) since \( p - |y| \neq p \). This is a contradiction for we have shown that \( xz \in L \) and \( xz \notin L \).

Consequently the initial assumption is incorrect; that is, \( L \) is not regular.

ii. \( A = \{a^iba^i \mid i \geq 0\} \).

iii. \( B = \{a^{2i}b^i \mid i \geq 1\} \).

iv. \( C = \{a^ib^ic^i \mid i \geq 1\} \).

v. \( D = \{ww \mid w \in \{a,b\}^*\} \).

vi. \( E = \{a^i \mid i \geq 0\} \), strings with \( a \) repeated \( 2^i \) times for some \( i \).

vii. \( F = \{a^q \mid q \text{ prime}\} \), strings containing a prime number of \( a \)'s.

viii. Let \( \Sigma = \{(, \}) \). Show \( G \) is non-regular, where \( G \) is the language of legal balanced parentheses: i.e., for each left parenthesis there is a matching right parenthesis to its right, and pairs of matched parentheses do not interleave. For example, the following are in \( G \): \( (), ()(), (((())))) \), and the following are not in \( G \): \( (), ((())))),((.

ix. Show that the following language is not regular: \( H = \{a^ib^i \mid i \neq j\} \).

13. If for language \( L \) the pumping length is 3 and \( a^3b^3 \in L \), then if you choose \( s = a^3b^3 \) and apply the Pumping Lemma to \( s \), how many pairs \( x, y \) are there, and what are these pairs?

14. Does the proof of the Pumping Lemma work using a \( p \)-state NFA accepting regular language \( L \) instead of a \( p \)-state DFA? Justify your answer briefly.

15. Let \( A = \{a^iw \mid w \in \{a,b\}^* \text{ and } w \text{ contains at most } i \text{ } a \text{’s, } i \geq 1\} \). Show that \( A \) is not regular. Remember that if \( R \) is regular and if \( A \) is regular then so is \( A \cap R \). It may be helpful to choose a suitable \( R \), and then show that \( A \cap R \) is not regular.
16. Show that the language $E = \{wxw \mid w, x \in \{a, b\}^*\}$ is not regular.

17. Let $w' = \text{Substitute}(a, b, w)$, or $\text{Subst}(a, b, w)$ for short, if $w'$ is obtained by replacing every $a$ in $w$ by a $b$. For example, $\text{Subst}(b, c, caca) = cbcbb$.
   Let $\text{Subst}(a, b, L) = \{w' \mid w' = \text{Subst}(a, b, w)\}$ for some $w \in L$. Show that if $L$ is regular then so is $\text{Subst}(a, b, L)$.

18. Use the Pumping Lemma to show that the following language is not regular.
   $$A = \{x = y \cdot z \mid x, y, z \in \{0, 1\}^* \text{ and } x = y \cdot z\}.$$  
   The strings in $A$ include the equal sign and the multiplication sign.

19. Let $L$ be a regular language. Define $\text{Remove-One-Char}(L)$, or $\text{ROC}(L)$ for short, to be the language containing those strings that can be obtained by removing a single character from a string in $L$; more formally:
   $$\text{ROC}(L) = \{su \mid sau \in L, \text{ where } s, u \in \Sigma^*, a \in \Sigma\}.$$  
   Show that $\text{ROC}(L)$ is also regular. It may be helpful to illustrate your construction with a diagram, but you should provide a reasonably precise explanation so that it is completely clear how your construction works. Remember to give a brief justification of why it works, also.

20. Let $C = \text{shuffle}(A, B)$ denote the shuffle $C$ of two languages $A$ and $B$; it consists of all strings $w$ of the form $w = a_1b_1a_2b_2\cdots a_kb_k$, for $k > 0$, with $a_1a_2\cdots a_k \in A$ and $b_1b_2\cdots b_k \in B$. Show that if $A$ and $B$ are regular then so is $C = \text{shuffle}(A, B)$.

21. Let $w = w_1w_2\cdots w_k$ where each $w_i \in \Sigma$, $1 \leq i \leq k$. And let $h$ be a function from the alphabet $\Sigma$ into the set of strings $\Gamma^*$, so that $h(a) \in \Gamma^*$ for each $a \in \Sigma$.
   Define $\text{FullSubst}(w, h)$, or $\text{FS}(w, a)$ for short, to be $h(w_1)h(w_2)\cdots h(w_k)$. Note that $\text{FullSubst}(\lambda, a) = \lambda$.
   e.g. if $h(a) = cc$, $h(b) = de$, then $\text{FS}(aba) = ccdecc$.
   For $L \subseteq \Sigma^*$, define $\text{FS}(L, h) = \{\text{FS}(w, h) \mid w \in L\}$. 
   Show that if $L$ is regular then so is $\text{FS}(L, h)$.

22. Let $w = w_1w_2\cdots w_k$ where each $w_i \in \Sigma$, $1 \leq i \leq k$. And let $r$ be a function mapping characters in $\Sigma$ into regular expressions over the alphabet $\Gamma$.
   Define $\text{SubstRegExpr}(w, r)$, or $\text{SRE}(w, r)$ for short, to be $r(w_1)r(w_2)\cdots r(w_k)$. Note that $\text{SRE}(\lambda, r) = \lambda$.
   e.g. if $r(a) = c^*$, $r(b) = \lambda \cup d$, then $r(ab) = c^*(\lambda \cup d)$.
   For $L \subseteq \Sigma^*$, define $\text{SRE}(L, r) = \{\text{SRE}(w, r) \mid w \in L\}$. 
   Show that if $L$ is regular then so is $\text{SRE}(L, r)$.
23. Define $\text{RemoveOneSymbol}(w, a)$, or $\text{ROS}(w, a)$ for short, to be the string $w$ with each occurrence of $a$ deleted. 
\[ \text{e.g. } \text{ROS}(abac, a) = bc, \text{ROS}(bc, a) = bc. \]
Let $\text{ROS}(L, a) = \{ \text{ROS}(w, a) \mid w \in L \}$.
Show that if $L$ is regular then so is $\text{ROS}(L, a)$.

24. Define $\text{ReplaceOneSymbol}(w, a, b)$, or $\text{RpOS}(w, a, b)$ for short, to be the collection of strings obtained from $w$ by replacing exactly one occurrence of $a$ with a $b$.
\[ \text{e.g. } \text{RpOS}(bb, a, b) = \{ \}; \text{RpOS}(acca, a, b) = \{ bcca, accb \}. \]
Let $\text{RpOS}(L, a, b) = \{ \text{RpOS}(w, a, b) \mid w \in L \}$.
Show that if $L$ is regular then so is $\text{RpOS}(L)$.

25. i. Let $\frac{1}{2}-L = \{ w \mid \exists x \text{ with } |x| = |w| \text{ and } wx \in L \}$.
Suppose that $L$ is regular; then show that $\frac{1}{2}-L$ is also regular.
Hint. Think nondeterministically. You will need to “guess” $x$. Getting its length right can be done only as $w$ is being read. What else do you need to guess, and what needs to be checked?

ii. Let $\text{Square}-L = \{ w \mid \exists x \text{ with } |x| = |w|^2 \text{ and } x \in L \}$.
Show that if $L$ is regular then so is $\text{Square}-L$.

iii. Let $\text{Power}-L = \{ w \mid \exists x \text{ with } |x| = 2^{|w|} \text{ and } x \in L \}$.
Show that if $L$ is regular then so is $\text{Power}-L$.

26. Consider the following variant of a DFA, called a Mealy-Moore machine, that writes an output string as it is processing its input. Viewed as a graph, it is like a DFA but each edge is labeled with both an input character $a \in \Sigma$, as is standard, and an output character $b \in \Gamma$ or the empty string $\lambda$, where $\Gamma$ is the output alphabet (possibly $\Sigma = \Gamma$). The output $M(x)$ produced by $M$ on input $x$ is simply the concatenation of the output labels on the path followed by $M$ on reading input $x$. The output language produced by $M$, $O(M)$, is defined to be
\[ O(M) = \{ M(x) \mid x \in L(M) \}, \]
the output strings obtained on inputs $x$ that $M$ recognizes.
Show that $O(M)$ is regular.

27. A 2wayDFA is a variant of a DFA in which it is possible to go back and forth over the input, with no limit on how often the reading direction is reversed. This can be formalized as follows. The input $x \in \Sigma^*$ to the DFA is sandwiched between symbols $\$ and $\$, so the DFA can be viewed as reading string $x_0x_1x_2\cdots x_{n}x_{n+1}$, where $x = x_1x_2\cdots x_{n}$, $x_0 = \$,$ and $x_{n+1} = \$$. The DFA is equipped with a read head which will always be over the next symbol to be read. At the start of the computation, the read head is over the symbol $x_1$, and the DFA is at its start vertex.
In general, the DFA will be at some vertex \( v \), with its read head over character \( x_i \) for some \( i, 0 \leq i \leq n+1 \). On its next move the DFA will follow the edge leaving \( v \) labeled \( x_i \). This edge will also carry a label L or R indicating whether the read head moves left (so that it will be over symbol \( x_{i-1} \)), or right (so that it will be over \( x_{i+1} \)). The latter is the only possible move for a standard DFA. There are two constraints: when reading \( x^a \) only moves to the right are allowed and when reading \( x^24 \) only moves to the left are allowed. If there is no move, the computation ends.

A 2wayDFA \( M \) recognizes an input \( x \) if it is at a final vertex when the computation ends. Show that the language recognized by a 2wayDFA is regular.

Hint. Consider the sequence of vertices \( M \) is at when its read head is over character \( x_i \). Observe that in a recognizing computation there can be no repetitions in such a sequence. Now create a standard NFA \( N \) to simulate \( M \). A state or supervertex of \( N \) will record the sequence of vertices that \( M \) occupies when its read head is over the current input character. The constraints on \( N \) are that its moves must be between consistent supervertices (you need to elaborate on what this means). Also, you need to specify what are the final supervertices of \( N \), and you need to argue that \( M \) and \( N \) recognize the same language.