# 3 Automata

## INTRODUCTION

Finite state machines are the most basic model of machines, organisms and processes in technology, nature, society, the universe and philosophy, a model that captures the essence of finite systems and allows us to learn, demonstrate and utilize their power.

On a theoretical level, finite state machines represent the very basic model of automata to start with in designing, learning, analysing and demonstrating components, principles and power of real and idealized computers and also a variety of basic computation modes.

On a practical level, finite state machines approximate real machines, systems and processes closely enough. That is why the aim of applied research and development in computing is often to reduce idealized concepts and methods to those realizable by finite state machines.

Finite state automata are also a good model for demonstrating how finite devices working in discrete time can be used to process infinite or continuous objects.

## LEARNING OBJECTIVES

The aim of the chapter is to demonstrate

1. the fundamental concept of finite state machine;
2. basic concepts, properties and algorithms concerning finite automata, their minimization and main decision problems;
3. basic concepts, properties and algorithms concerning regular expressions, regular languages and their closure properties;
4. finite transducers and their power and properties;
5. weighted finite automata and transducers and their use for image generation, transformation and compression;
6. how to use discrete finite automata to process infinite and continuous objects;
7. various modifications of finite automata: nondeterministic, probabilistic, two-way, multihead and linearly bounded automata and their power.

> The fact is, that civilization requires slaves. The Greeks were quite right there. Unless there are slaves to do the ugly, horrible, uninteresting work, culture and contemplation become almost impossible. Human slavery is wrong, insecure, and demoralizing. On mechanical slavery, on the slavery of the machine, the future of the world depends.
>
> *Oscar Wilde, 1895*

The concept of finite state devices is one of the most basic in modern science, technology and philosophy; one that in a strikingly simple way captures the essence of the most fundamental principle of how machines, nature and society work. The whole process of the development of a deterministic and mechanistic view of the world, initiated by R. Descartes whose thinking was revolutionary for its time, culminated in a very simple, powerful model of finite state machines, due to McCulloch and Pitts (1943), obtained from an observation of principles of neural activities.[1]

In this chapter we present, analyse and illustrate several models of automata, as well as some of their (also surprising) applications. The most basic model is that of a finite state machine, which is an abstraction of a real machine (and therefore of fixed size and finite memory machines), functioning in discrete time steps.

Finite state machines are building blocks, in a variety of ways, for other models of computing, generating and recognizing devices, both sequential and parallel, deterministic and randomized. This lies behind their fundamental role in the theory and practice of computing. Because of their simplicity, efficiency and well worked out theory, it is often a good practice to simplify sophisticated computational concepts and methods to such an extent that they can be realized by (co-operating) finite state machines.

Basic theoretical concepts and results concerning finite state machines are presented in the first part of this chapter. In the second part several applications are introduced, showing the surprising power and usefulness of the basic concepts concerning finite state machines: for example, for image generation, transformation and compression. Finally, various modifications of the basic model of finite state machines are considered. Some of them do not increase the power of finite state machines, but again show how robust the basic model is. Others turn out to be more powerful. This results in a variety of models filling the gap between finite state machines and universal computers discussed in the following chapter.

It will also be demonstrated that though such machines are finite and work in discrete steps, they can process, in a reasonable sense, infinite and continuous objects. For example, they can be seen as processing infinite words and computing (even very weird) continuous functions.

## 3.1 Finite State Devices

The finite state machine model of a device abstracts from the technology on which the device is based. Attention is paid only to a finite number of clearly distinguished **states** that the device can be in and

[1] Automata and automatization have for a long time been among the most exciting ideas for humankind, not only because they offer ways to get rid of dull work, but also because they offer means by which humankind can overcome their physical and intellectual limitations. The first large wave of fascination with automata came in the middle of the nineteenth century, when construction of sophisticated automata, imitating functions considered essential for living and/or intelligent creatures, flourished. The emerging automata industry, see the interesting account in Bailey (1982), played an important role in the history of modern technology. The second wave, apparently less mysterious but much more powerful, came with the advent of universal computers.

a finite number of clearly identified **events**, usually called **external inputs** or **signals**, that may cause the device to change its current state.

A simple finite state model of a digital watch is shown in Figure 3.1a. The model abstracts from what, how and by whom the watch is made, and shows only eight main states, depicted by boxes, the watch can be in, from the user's point of view ('update hours', 'display date', 'display time'), and transitions between the states caused by pushing one of four buttons $a, b, c, d$. Each transition is labelled by the button causing that transition. Having such a simple state transition model of a digital watch, it is easy to follow the sequence of states of the watch when the buttons are pushed in a given sequence. For example, by pushing buttons $a, c, d, c, a, a$, in this order, the watch gets, transition by transition, from the state 'display time' back to the same state.

The finite state model of a watch in Figure 3.1a models watch behaviour as a process that goes on and on (until the watch gets broken or the battery dies). Observe that this process has no other outputs beside the states themselves – various displays. Note also that in some states, for example, 'display watch', it is not specified for all buttons what happens if the button is pressed. (This can be utilized to make a more detailed model of a watch, with more states and actions, for example, to manipulate the stopwatch.) Note also that neither requirements nor restrictions are made on how often a button may be pressed and how much time a state transition takes.

There are many interesting questions one can ask/study about the model in Figure 3.1a. For example, given two states $p$ and $q$, which sequence of buttons should one push in order to get from state $p$ to state $q$?

**Exercise 3.1.1** *Describe the five shortest sequences of buttons that make the watch in Figure 3.1a go from state p to state q if (a) p = 'display alarm', q = 'display hours'; (b) p = 'display time', q = 'display alarm'.*

Two other models of finite automata are depicted in Figures 3.1b, c. In both cases the states are depicted by circles, and transitions by arrows labelled by actions (external symbols or inputs) causing these transitions. These two finite state machines are more abstract. We do not describe what the states mean. Only transitions between states are depicted and states are partitioned into 'yes'- and 'no'-states. For these two models we can also ask the question: which sequences of inputs make the machine change from a given state $p$ to a given state $q$; or a simpler question: which sequences of inputs make the machine go from the starting state to a 'yes'-state. For example, in the case of the model in Figure 3.1b, the sequences of letters 'the', 'thee', 'their' and 'then' have such a property; whereas the sequence 'tha' has not. In the case of the finite state model in Figure 3.1c a sequence of inputs makes the machine go from the initial state into the single 'yes'-state if and only if this sequence contains an even number of $a$'s.

As we shall soon see, the questions as to which inputs make a finite state machine go from one state to another or to a 'yes'-state turn out to be, very important in relation to such an abstract model of finite state machines.

In our model of finite state machines we use a very general concept of a (global) state. A digital device is often composed of a large number of elementary devices, say $n$, such that each of them is always in one of the two binary states. Any combination of these elementary states forms the so-called 'global state'. The overall number of (global) states of the device is $2^n$ in such a case. However, in a simple finite state model of a device, very often only a few of the global states are used.

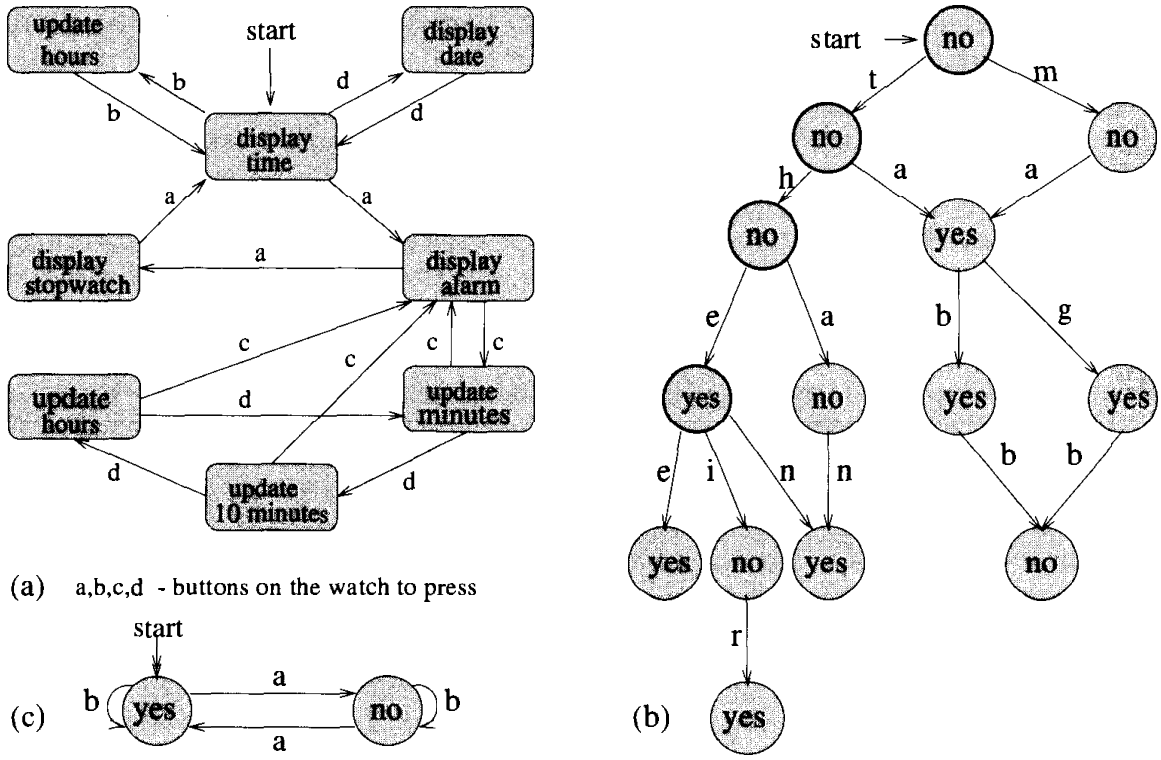



**Figure 3.1** Finite state devices

**Exercise 3.1.2** *Extend the finite state model of the watch in Figure 3.1 to incorporate other functions which a watch usually has.*

**Exercise 3.1.3** *Express in a diagram possible states and transitions for a coffee vending machine that acts as follows. It takes 5, 10 and 20p coins, in any order, until the overall amount is at least 90p. At the moment this happens, the machine stops accepting coins, produces coffee and makes change. (Take into consideration only the money-checking activity of the machine.)*

Four basic types of finite state machines are recognizers, acceptors, transducers and generators (see Figure 3.2). A **recognizer** is a finite state machine $\mathcal{A}$ that always starts in the same initial state. Any input causes a state change (to a different or to the same state) and only a state change – no output is produced. States are partitioned into 'yes'-states (**terminal** states) and 'no'-states (**nonterminal** states). A sequence of inputs is said to be **recognized** (**rejected**) by $\mathcal{A}$ if and only if this sequence of inputs places the machine in a **terminal** state (a **nonterminal** state).

**Example 3.1.4** *The finite state machine in Figure 3.3a recognizes an input sequence* $(a_1,b_1)\ldots(a_{n-1},b_{n-1})(a_n,b_n)$, *with* $(a_1,b_1)$ *as the first symbol, if and only if there is a* $k$, $1 \leq k \leq n$, *such that* $a_k = b_k = 1$. *(Interestingly enough, this is precisely the case if* $\binom{i+j}{i} \bmod 2 = 0$ *for the integers* $i = bin(a_n a_{n-1}\ldots a_1)$ *and* $j = bin(b_n b_{n-1}\ldots b_1)$ *– show that!)*

An **acceptor** is also a finite state machine that always starts in the same initial state. An input either

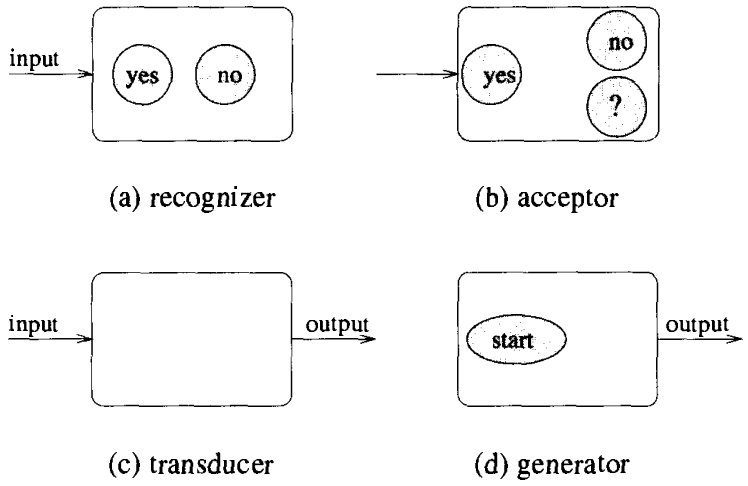



**Figure 3.2** A recognizer, an acceptor, a transducer and a generator

causes a state transition or is not accepted at all, and again no output is produced. A sequence of inputs is said to be accepted if and only if it puts the automaton in a terminal state. (The other possibilities are that a sequence of inputs puts the automaton in a nonterminal state or that its processing is interrupted at some point, because the next transition is not defined.)

**Example 3.1.5** *Figure 3.3d shows an acceptor that accepts exactly the words of the language* $a^*cb^*$.

A **transducer** acts as a recognizer, but for each input an output is produced.

**Example 3.1.6** *The transducer shown in Figure 3.3b produces for each input word* $w = w_1cw_2c \ldots cw_{n-1}cw_n$, $w_i \in \{0,1\}^*$ *the output word* $w' = \phi(w_1)cw_2c\phi(w_3)c \ldots cw_{n-1}\phi(w_n)$ *if* $n$ *is odd and* $w' = \phi(w_1)cw_2c\phi(w_3)c \ldots c\phi(w_{n-1})cw_n$ *if* $n$ *is even, where* $\phi$ *is the morphism defined by* $\phi(c) = c$, $\phi(0) = 01$ *and* $\phi(1) = 10$. *In Figure 3.3b, in each pair 'i,o', used as a transition label, the first component denotes the input symbol, the second the output string.*

A **generator** has no input. It starts in an initial state, moves randomly, from state to state, and at each move an output is produced. For each state transition a probability is given that the transition takes place.

**Example 3.1.7** *The generator depicted in Figure 3.3c has only one state, and all state changes have the same probability, namely* 1/3. *It is easy to see that if a sequence of output symbols* $(x_1,y_1) \ldots (x_n,y_n)$ *is interpreted as a point of the unit square, with the coordinates* $(0.x_1 \ldots x_n, 0.y_1 \ldots y_n)$ *as in Section 2.1.2, then the generator produces the Sierpiński triangle shown in Figure 2.1.*

Is it not remarkable that a one-state generator can produce such a complex fractal structure? This is in no way an exception. As will be seen later, finite state generators can generate very complex images indeed.

## 3.2 Finite Automata

So far we have used the concepts of finite state recognizers and acceptors only intuitively. These concepts will now be formalized, generalized and analysed. The main new idea is the introduction of nondeterminism. In some states behaviour of the automaton does not have to be determined uniquely. We show that such a generalization is fully acceptable and, in addition, sometimes very useful.

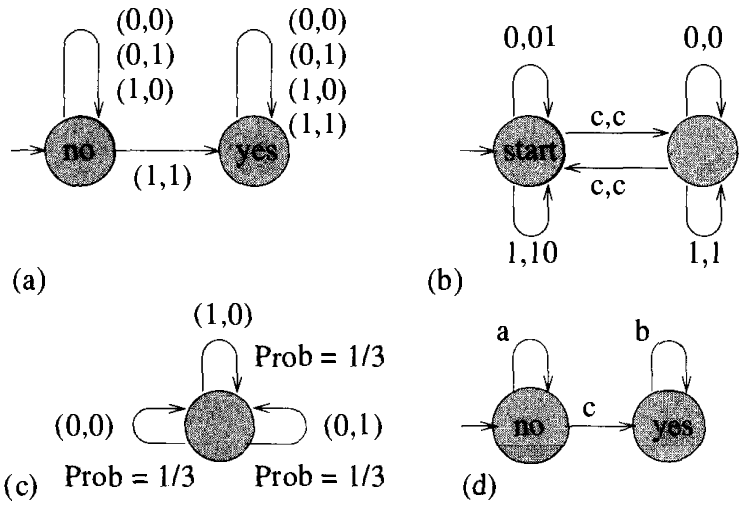



**Figure 3.3** Examples of a recognizer, a transducer, a generator and an acceptor

## 3.2.1 Basic Concepts

**Definition 3.2.1** *A* **(nondeterministic) finite automaton** $\mathcal{A}$ *(for short,* **NFA** *or* **FA***) over the (input)* **alphabet** $\Sigma$ *is specified by a finite* **set of states** $Q$*, a distinct* **(initial) state** $q_0$*, a set* $Q_F \subseteq Q$ *of* **terminal (final) states** *and a* **transition relation** $\delta \subset Q \times \Sigma \times Q$*. Formally,* $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$*.*

*If* $\delta$ *is a function, that is* $\delta : Q \times \Sigma \to Q$*, we also use the notation* $\delta(q,a)$ *to specify the value of* $\delta$ *for arguments* $q,a$*.*

Informally, a computation of $\mathcal{A}$ for an input word $w$ always starts in the initial state $q_0$ and continues by a sequence of **steps** (**moves** or **transitions**), one for each input symbol. In each step the automaton moves from its current state, say $p$, according to the next symbol of the input word, say $a$, into a state $q$ such that $(p,a,q) \in \delta$ – if such a $q$ exists. If there is a unique $q \in Q$ such that $(p,a,q) \in \delta$, then the transition from the state $p$ by the input $a$ is uniquely determined. We usually say that it is deterministic. If there are several $q$ such that $(p,a,q) \in \delta$, then one of the possible transitions is chosen, and all of them are considered as being equally likely. If, for some state $p$ and input $a$, there is no $q$ such that $(p,a,q) \in \delta$, then we say that input $a$ in state $p$ leads to a termination of the computation. A computation ends after the last symbol of $w$ is processed or a termination occurs. We can also say that a computation is performed in discrete time steps and the time instances are ordered $0,1,2,\ldots$ with 0 the time at which each computation starts.

For a formal definition of computation of a FA the concept of configuration is important. A **configuration** $C$ of $\mathcal{A}$ is a pair $(p,w) \in Q \times \Sigma^*$. Informally, the automaton $\mathcal{A}$ is in the configuration $(p,w)$, if it is in the state $p$ and $w$ is the part of the input word yet to be processed. A configuration $(q_0,w)$ is called **initial**, and any configuration $(q,\varepsilon), q \in Q_F$ is called **final**.

A **computational step** of $\mathcal{A}$ is the relation

$$\vdash_{\mathcal{A}} \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$$

between configurations defined for $p,q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$ by

$$(p,aw) \vdash_{\mathcal{A}} (q,w) \Leftrightarrow (p,a,q) \in \delta.$$

Informally, $(p,aw) \vdash_{\mathcal{A}} (q,w)$ means that $\mathcal{A}$ moves from state $p$ after input $a$ to state $q$. A **computation** of $\mathcal{A}$ is the transitive and reflexive closure $\vdash_{\mathcal{A}}^*$ of the relation $\vdash_{\mathcal{A}}$ between configurations: that is, $C \vdash_{\mathcal{A}}^* C'$

(a)

$$\begin{array}{llll} \delta(q_0,0) = \{q_2\}, & \delta(q_0,2) = \{q_1,q_2\}, \\ \delta(q_1,0) = \{q_3\}, & \delta(q_1,1) = \{q_0\}, \\ \delta(q_2,0) = \{q_0\}, & \delta(q_2,2) = \{q_3,q_0\}, \\ \delta(q_3,0) = \{q_1\}, & \delta(q_3,1) = \{q_2\}. \end{array}$$

(b)

| states | inputs | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| $q_0$ | $q_2$ | - | $q_1,q_2$ |
| $q_1$ | $q_3$ | $q_0$ | - |
| $q_2$ | $q_0$ | - | $q_3,q_0$ |
| $q_3$ | $q_1$ | $q_2$ | - |

(c)

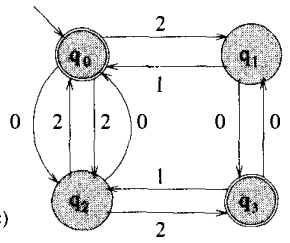



**Figure 3.4** Finite automata representations

for configurations $C$ and $C'$ if and only if there is a sequence of configurations $C_1, \ldots, C_n$ such that $C = C_1$, $C_i \vdash_{\mathcal{A}} C_{i+1}$, for $1 \leq i < n$, and $C_n = C'$.

Instead of $(p,w) \vdash_{\mathcal{A}}^{*} (q,\varepsilon)$, we usually use the notation $p \overset{*}{\underset{w}{\Longrightarrow}} q$. A state $q$ is called reachable in $\mathcal{A}$ if there is an input word $w$ such that $q_0 \overset{*}{\underset{w}{\Longrightarrow}} q$.

**Exercise 3.2.2** *Let $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ be a FA. Let us define a recurrence as follows: $A_0 = \{q_0\}$, $A_i = \{q' \mid (q,a,q') \in \delta$ for some $q \in A_{i-1}, a \in \Sigma\}$, for $i \geq 1$. Show that a state $q$ is reachable in $\mathcal{A}$ if and only if $q \in A_j$ for some $j \leq |Q|$. (This implies that it is easy to compute the set of all reachable states.)*

Three basic ways of representing finite automata are illustrated in Figure 3.4 on the automaton $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$, where $Q = \{q_0,q_1,q_2,q_3\}$, $\Sigma = \{0,1\}$, and $Q_F = \{q_0,q_3\}$: an **enumeration** of transitions (Figure 3.4a), a **transition matrix** (Figure 3.4b) with rows labelled by states and columns by input symbols, and a **state graph** or a **transition diagram** (Figure 3.4c) with states represented by circles, transitions by directed edges labelled by input symbols, the initial state by an ingoing arrow, and final states by double circles. For a finite automaton $\mathcal{A}$ let $G_{\mathcal{A}}$ denote its state graph. Observe that a state $q$ is reachable in the automaton $\mathcal{A}$ if and only if the corresponding node is reachable in the graph $G_{\mathcal{A}}$ from its starting vertex.

To every finite automaton $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ and every $q \in Q$, we associate the language $L(q)$ of those words that make $\mathcal{A}$ move from state $q$ to a final state. More formally,

$$L(q) = \{w \in \Sigma^* \mid q \overset{*}{\underset{w}{\Longrightarrow}} p \in Q_F\}.$$

$L(\mathcal{A}) = L(q_0)$ is then the **language recognized** by $\mathcal{A}$. A language $L$ is called a **regular language** if there is a finite automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$. The **family of languages recognizable** by finite automata, or the **family of regular languages**, is denoted by

$$\mathcal{L}(FA) = \{L(\mathcal{A}) \mid \mathcal{A} \text{ is a finite automaton}\}.$$

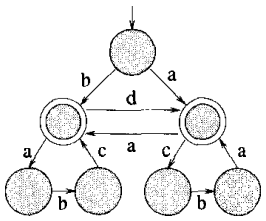



**Figure 3.5** Finite automaton

**Exercise 3.2.3** *Let* $L_n = \{uv \mid uv \in \{0,1\}^*, |u| = |v| = n, u \neq v\}$. *Design a FA accepting the language (a)* $L_2$*; (b)* $L_3$*; (c)* $L_4$.

**Exercise 3.2.4** *Describe the language accepted by the FA depicted in Figure 3.5.*

Another way to define the language recognized by a finite automaton $\mathcal{A}$ is in terms of its state graph $G_{\mathcal{A}}$. A **path** in $G_{\mathcal{A}}$ is a sequence of triples $(p_1,a_1,p_2)(p_2,a_2,p_3)\ldots(p_n,a_n,p_{n+1})$ such that $(p_i,a_i,p_{i+1}) \in \delta$, for $1 \leq i \leq n$. The word $a_1 \ldots a_n$ is the label of such a path, $p_1$ its origin and $p_{n+1}$ its terminus. A word $w \in \Sigma^*$ is recognizable by $\mathcal{A}$ if $w$ is the label of a path with $q_0$ as its origin and a final state as its terminus. $L(\mathcal{A})$ is then the set of all words recognized by $\mathcal{A}$.

The language recognized by a finite automaton $\mathcal{A}$ can be seen as the computational process that $\mathcal{A}$ represents. This is why two finite automata $\mathcal{A}_1, \mathcal{A}_2$ are called **equivalent** if $L(\mathcal{A}_1) = L(\mathcal{A}_2)$; that is, if the corresponding languages (computational processes they represent) are equal.

**Exercise 3.2.5** *A natural generalization is to consider finite automata* $\mathcal{A} = \langle \Sigma, Q, Q_I, Q_F, \sigma \rangle$ *with a set* $Q_I$ *of initial states, where computation and recognition are defined similarly. Show that to each such finite automaton* $\mathcal{A}$ *we can easily construct an equivalent ordinary finite automaton.*

If two FA are equivalent, that is, if they are 'the same' insofar as the computational processes (languages) they represent are the same, they can nevertheless look very different, and can also have a different number of states. A stronger requirement for similarity is that they are **isomorphic** – they differ only in the way their states are denoted.

**Definition 3.2.6** *Two FA* $\mathcal{A}_i = \langle \Sigma, Q_i, q_{0,i}, Q_{F,i}, \delta_i \rangle$, $i = 1,2$ *are isomorphic if there is a bijection* $\mu : Q_1 \to Q_2$ *such that* $\mu(q_{0,1}) = q_{0,2}$, $q \in Q_{F,1}$ *if and only if* $\mu(q) \in Q_{F,2}$, *and for any* $q, q' \in Q_1$, $a \in \Sigma$ *we have* $(q,a,q') \in \delta_1$ *if and only if* $(\mu(q),a,\mu(q')) \in \delta_2$.

**Exercise 3.2.7** *Design a finite automaton that accepts those binary words that represent integers (with the most significant bit as the first) divisible by three.*

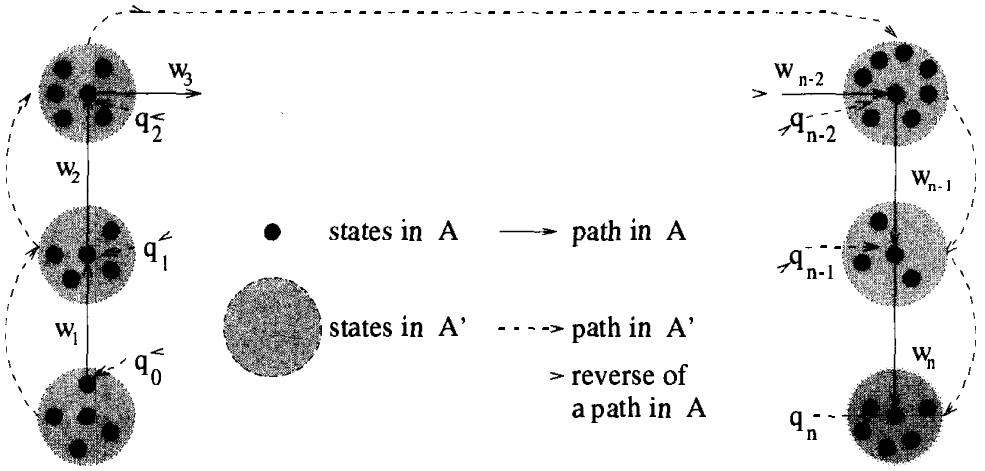



**Figure 3.6** A path in a NFA and in an equivalent DFA obtained by the subset construction

## 3.2.2 Nondeterministic versus Deterministic Finite Automata

The formal definition of a FA (on page 158) allows it to have two properties that contradict our intuition: a state transition, for a given input, does not have to be unique and does not have to be defined. Our intuition seems to prefer that a FA is deterministic and complete in the following sense.

A finite automaton is a **deterministic finite automaton** if its transition relation is a partial function: that is, for each state $p \in Q$ and input $a \in \Sigma$ there is at most one $q \in Q$ such that $(p,a,q) \in \delta$. A finite automaton is called **complete** if for any $p \in Q, a \in \Sigma$ there is at least one $q$ such that $(p,a,q) \in \delta$. In the following the notation DFA will be used for a deterministic and complete FA.

The following theorem shows that our definition of finite automata, which allows 'strange' nondeterminism, has not increased the recognition power of DFA.

**Theorem 3.2.8** *To every finite automaton there is an equivalent deterministic and complete finite automaton.*

**Proof:** Given a FA $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$, an equivalent DFA $\mathcal{A}'$ can be constructed, by the **subset construction**, as

$$\mathcal{A}' = \langle \Sigma, 2^Q, \{q_0\}, \{B \mid B \in 2^Q, B \cap Q_F \neq \emptyset\}, \delta' \rangle,$$

where the new transition relation $\delta'$ is defined as follows:

$$(A,a,B) \in \delta' \text{ if and only if } B = \{q \mid \exists p \in A, (p,a,q) \in \delta\}.$$

The states of $\mathcal{A}'$ are therefore sets of the states of $\mathcal{A}$. There is a transition in $\mathcal{A}'$ from a state $S$, a set of states of $\mathcal{A}$, to another state $S_1$, again a set of states of $\mathcal{A}$, under an input $a$ if and only if to each state in $S_1$ there is a transition in $\mathcal{A}$ from some state in $S$ under the input $a$.

$\mathcal{A}$ is clearly deterministic and complete. To show that $\mathcal{A}$ and $\mathcal{A}'$ are equivalent, consider the state graphs $G_\mathcal{A}$ and $G_{\mathcal{A}'}$. For any path in $G_\mathcal{A}$, from the initial state to a final state, labelled by a word $w = w_1 \ldots w_s$, there is a unique path in $G_{\mathcal{A}'}$, labelled also by $w$, from the initial state to a final state (see Figure 3.6). The corresponding states of $\mathcal{A}'$, as the sets of states of $\mathcal{A}$, can be determined, step by step, using the transition function $\delta'$, from the initial state of $\mathcal{A}'$ and $w$. The state of $\mathcal{A}'$ reached by the path labelled by a prefix of $w$ has to contain exactly the states of $\mathcal{A}$ reached, in $G_\mathcal{A}$, by the path labelled by

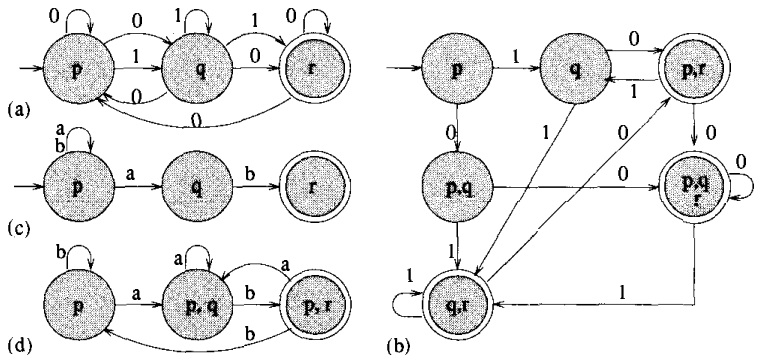



**Figure 3.7** FA and equivalent DFA obtained by the subset construction

the same prefix of $w$. Similarly, to any path in $\mathcal{A}'$, from the initial to a final state, labelled by a word $w$, there is a path in $G_{\mathcal{A}}$ from the initial to a final state. The states on this path can be taken from the corresponding states of the path labelled by $w$ in $G_{\mathcal{A}'}$ in such a way that they form a path in $G_{\mathcal{A}}$. To design it, one has to start in the last state (of $\mathcal{A}'$) of the path in $G_{\mathcal{A}'}$, to pick up a state $q_n$ (terminal in $\mathcal{A}$), from this state of $\mathcal{A}'$ and go backwards to pick up states $q_{n-1}, q_{n-2}, \ldots, q_1$. This is possible, because whenever $A \underset{a}{\Longrightarrow} B$ in $\mathcal{A}'$, then for any $q' \in B$ there is a $q \in A$ such that $q \underset{a}{\Longrightarrow} q'$ in $\mathcal{A}$. □

**Example 3.2.9** *Let $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ be the nondeterministic finite automaton depicted in Figure 3.7a. The finite automaton obtained by the subset construction has the following transition function $\delta'$:*

$$
\begin{array}{rclrcl}
\delta'(\emptyset,0) & = & \emptyset; & \delta'(\emptyset,1) & = & \emptyset;\\
\delta'(\{p\},0) & = & \{p,q\}; & \delta'(\{p\},1) & = & \{q\};\\
\delta'(\{q\},0) & = & \{p,r\}; & \delta'(\{q\},1) & = & \{q,r\};\\
\delta'(\{r\},0) & = & \{p,r\}; & \delta'(\{r\},1) & = & \emptyset;\\
\delta'(\{p,q\},0) & = & \{p,q,r\}; & \delta'(\{p,q\},1) & = & \{q,r\};\\
\delta'(\{p,r\},0) & = & \{p,q,r\}; & \delta'(\{p,r\},1) & = & \{q\};\\
\delta'(\{q,r\},0) & = & \{p,r\}; & \delta'(\{q,r\},1) & = & \{q,r\};\\
\delta'(\{p,q,r\},0) & = & \{p,q,r\}; & \delta'(\{p,q,r\},1) & = & \{q,r\}.
\end{array}
$$

The states $\{r\}$ and $\emptyset$ are not reachable from the initial state $\{p\}$; therefore they are not included in the state graph $G_{\mathcal{A}'}$ of $\mathcal{A}'$ in Figure 3.7b. The subset construction applied to the FA in Figure 3.7c provides the DFA shown in Figure 3.7d. Other states created by the subset construction are not reachable in this case.

**Exercise 3.2.10** *Design a DFA equivalent to NFA in (a) Figure 3.8a; (b) Figure 3.8b.*

Since nondeterministic and incomplete FA conform less to our intuition of what a finite state machine is and, are not more powerful than DFA, it is natural to ask why they should be considered at all.

There are two reasons, both of which concern efficiency. The first concerns design efficiency. It is quite often easier, even significantly easier, to design a NFA accepting a given regular language than an equivalent DFA. For example, it is straightforward to design a NFA recognizing the language

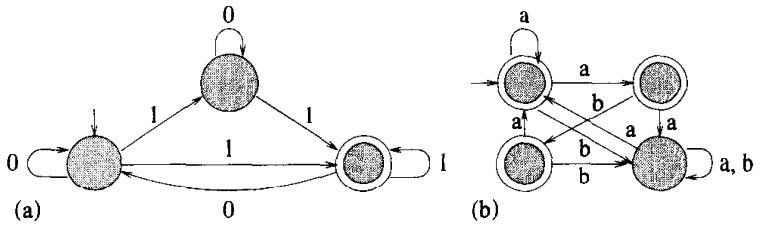



**Figure 3.8** Examples of a NFA

$\{a,b\}^*a\{a,b\}^n$ (see Figure 3.9a for the general case and Figure 3.9b for $n = 2$). On the other hand, it is much more difficult to design a DFA for this language (see the one in Figure 3.9c for $n = 2$). The second reason concerns size efficiency, and this is even 'provably important'.

The number of states of a FA $\mathcal{A}$, in short $state(\mathcal{A})$, is its **state complexity**. In the case of the NFA in Figure 3.7c the subset construction does not provide a DFA with a larger number of states. On the other hand, the subset construction applied to the NFA in Figure 3.7a, has significantly increased the number of states. In general, the subset construction applied to a NFA with $n$ states provides a DFA with $2^n$ states. This is the number of subsets of each set of $n$ elements and indicates that the subset construction can produce exponentially more states. However, some of these states may not be reachable, as the example above shows. Moreover, it is not yet clear whether some other method could not provide a DFA with fewer states but still equivalent to the given NFA.

In order to express exactly how much more economical a NFA may be, compared with an equivalent DFA, the following economy function is introduced:

$$Economy_{\text{NFA}}^{\text{DFA}}(n) = \max\{\min\{state(\mathcal{B}) | \mathcal{B} \text{ is a DFA equivalent to } \mathcal{A}\} | \mathcal{A} \text{ is NFA}, state(\mathcal{A}) = n\}.$$

The following result shows that a DFA can be, provably, exponentially larger than an equivalent NFA.

**Theorem 3.2.11** $Economy_{\text{NFA}}^{\text{DFA}}(n) = 2^n$.

**Proof idea:** The inequality $\text{Economy}_{\text{NFA}}^{\text{DFA}}(n) \leq 2^n$ follows from the subset construction. In order to prove the opposite inequality, it is sufficient to show, which can be done, that the minimum DFA equivalent to the one shown in Figure 3.9d must have $2^n$ states. □

A simpler example, though not so perfect, of the exponential growth of states provided by the subset construction, is shown in Figure 3.9. The minimum DFA equivalent to the NFA shown in Figure 3.9a must have $2^{n-1}$ states. This is easy to see, because the automaton has to remember the last $n-1$ symbols. For $n = 2$ the equivalent DFA is shown in Figure 3.9c.

**Corollary 3.2.12** *Nondeterminism of a NFA does not increase its computational power, but can essentially (exponentially) decrease the number of states (and thereby also increase the design efficiency).*

**Exercise 3.2.13** *Design a DFA equivalent to the one in Figure 3.9d for (a) $n = 4$; (b) $n = 5$.*

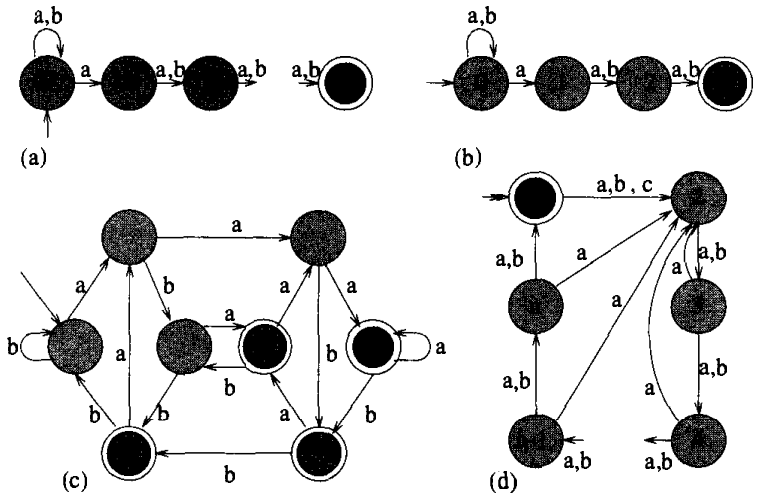



**Figure 3.9** Examples showing that the subset construction can yield an exponential growth of states

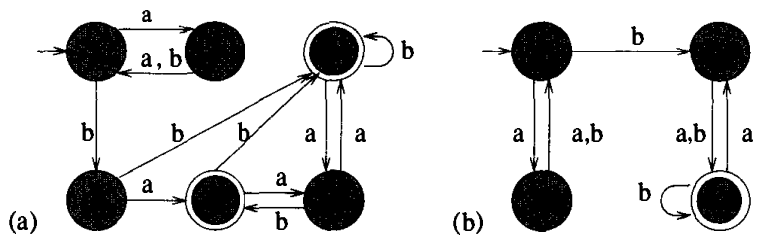



**Figure 3.10** Two equivalent DFA

### 3.2.3 Minimization of Deterministic Finite Automata

Once we have the task of designing a DFA that recognizes a given regular language $L$, it is natural to try to find a 'minimal' DFA, with respect to the number of states, for $L$. Figure 3.10 shows that two equivalent DFA may have different numbers of states.

The following questions therefore arise naturally:

- How many different but equivalent minimal DFA can exist for a given FA?
- How can a minimal DFA equivalent to a given DFA be designed?
- How fast can one construct a minimal DFA?

In order to answer these questions, new concepts have to be introduced. Two states $p, q$ of a FA $\mathcal{A}$ are called **equivalent**; in short $p \equiv_{\mathcal{A}} q$, if $L(p) = L(q)$ in $\mathcal{A}$. A FA $\mathcal{A}$ is called **reduced** if no two different states of $\mathcal{A}$ are equivalent. A DFA $\mathcal{A}$ is called **minimal** if there is no DFA equivalent to $\mathcal{A}$ and with fewer states.

We show two simple methods for minimizing finite automata. Both are based on the result, shown later, that if a DFA is reduced, then it is minimal.

1. **Minimization of DFA using the operations of reversal and subset construction.** The first method is based on two operations with finite automata. The operation of **reversal** assigns to a DFA $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ the finite automaton $\rho(\mathcal{A}) = \langle \Sigma, Q, Q_F, \{q_0\}, \rho(\delta) \rangle$, that is, the initial and final

states are exchanged, and $q \in \rho\langle\delta\rangle(q',a)$ if and only if $\delta(q,a) = q'$. The operation of **subset construction** assigns to any FA $\mathcal{A} = \langle\Sigma, Q, Q_I, Q_F, \delta\rangle$, with a set $Q_I$ of initial states, a DFA $\pi(\mathcal{A})$ obtained from $\mathcal{A}$ by the subset construction (and containing only reachable states).

**Theorem 3.2.14** *Let $\mathcal{A}$ be a finite automaton, then $\mathcal{A}' = \pi(\rho(\pi(\rho(\mathcal{A}))))$ is a reduced DFA equivalent to $\mathcal{A}$.*

**Proof:** Clearly $\mathcal{A}'$ is a DFA equivalent to $\mathcal{A}$. It is therefore sufficient to prove that $\pi(\rho(D))$ is reduced whenever $D = \langle\Sigma, Q', q_0', Q_F', \delta'\rangle$ is a FA and each of its states is reachable. Let $Q_1 \subseteq Q'$, $Q_2 \subseteq Q'$ be two equivalent states of $\pi(\rho(D))$. Since each state of $D$ is reachable, for each $q_1 \in Q_1$ there is a $w \in \Sigma^*$ such that $q_1 = \delta'(q_0', w)$. Thus $q_0' \in \rho(\delta')(Q_1, w)$. As $Q_1$ and $Q_2$ are equivalent, we also have $q_0' \in \rho(\delta')(Q_2, w)$, and therefore $q_2 = \delta'(q_0', w)$ for some $q_2 \in Q_2$. Since $\delta'$ is a mapping, we get $q_1 = q_2$, and therefore $Q_1 \subseteq Q_2$. By symmetry, $Q_1 = Q_2$. □

Unfortunately, there is a DFA $\mathcal{A}$ with $n$ states such that $\pi(\rho(\mathcal{A}))$ has $2^n$ states (see the one in Figure 3.9d). The time complexity of the above algorithm is therefore exponential in the worst case.

2. **Minimization of DFA through equivalence automata.** The second way of designing a reduced DFA $\mathcal{A}'$ equivalent to a given DFA $\mathcal{A}$ is also quite simple, and leads to a much more efficient algorithm. In the state graph $G_{\mathcal{A}}$ identify nodes corresponding to the equivalent states and then identify multiple edges with the same label between the same nodes. The resulting state graph is that of a reduced DFA. More formally,

**Definition 3.2.15** *Let $\mathcal{A} = \langle\Sigma, Q, q_0, Q_F, \delta\rangle$ be a DFA. For any state $q \in Q$ let $[q]$ be the equivalence class on $Q$ with respect to the relation $\equiv_{\mathcal{A}}$. The* **equivalence automaton** *$\mathcal{A}'$ for $\mathcal{A}$ is defined by*

*$\mathcal{A}' = \langle\Sigma, Q', [q_0], Q_F', \delta'\rangle$, where $Q' = \{[q] \mid q \in Q\}$, $Q_F' = \{[q] \mid q \in Q_F\}$, and $\delta' = \{([q_1], a, [q_2]) \mid (q_1', a, q_2') \in \delta$ for some $q_1' \in [q_1], q_2' \in [q_2]\}$.*

Minimization of DFA is now based on the following result.

**Theorem 3.2.16** *(1) The equivalence automaton $\mathcal{A}'$ of a DFA $\mathcal{A}$ is well defined, reduced and equivalent to $\mathcal{A}$.*

*(2) State($\mathcal{B}$) $\geq$ state($\mathcal{A}'$) for any DFA $\mathcal{B}$ equivalent to a DFA $\mathcal{A}$.*

*(3) Any minimal DFA $\mathcal{B}$ equivalent to a DFA $\mathcal{A}$ is isomorphic with $\mathcal{A}'$.*

**Proof:** (1) If $q \equiv_{\mathcal{A}} q'$, then either both $q$ and $q'$ are in $Q_F$, or both are not in $Q_F$. Final states of $\mathcal{A}'$ are therefore well defined. Moreover, if $L(q) = L(q')$ for some $q, q' \in Q$, then for any $a \in \Sigma$, $L(\delta(q,a)) = L(\delta(q',a))$, and therefore all transitions of $\mathcal{A}'$ are well defined. If $w = w_1 \ldots w_n \in \Sigma^*$ and $q_i = \delta(q_0, w_1 \ldots w_i)$, then $[q_i] = \delta'([q_0], w_1 \ldots w_i)$. This implies that $L(\mathcal{A}) = L(\mathcal{A}')$. The condition of $\mathcal{A}'$ being reduced is trivially fulfilled due to the construction of $\mathcal{A}'$.

(2) It is sufficient to prove (2) assuming that all states of $\mathcal{B}$ are reachable from the initial state. Let $\mathcal{B} = \langle\Sigma, Q'', q_0'', Q_F'', \delta''\rangle$ be a DFA equivalent to $\mathcal{A}$. Consider the mapping $g : Q'' \to Q'$ defined as follows: since all states of $\mathcal{B}$ are reachable, for any $q'' \in Q''$ there is a $w_{q''} \in \Sigma^*$ such that $\delta''(q_0'', w_{q''}) = q''$. Define now $g(q'') = \delta''([q_0], w_{q''})$. From the minimality of $\mathcal{A}'$ and its equivalence with $\mathcal{B}$, it follows that this mapping is well defined and surjective.

(3) In the case of minimality of $\mathcal{B}$ it is easy to verify that the mapping $g$ defined in (2) is actually an isomorphism. □

**Corollary 3.2.17** *If a DFA is reduced, then it is minimal.*

The task of constructing a minimal DFA equivalent to a given DFA $\mathcal{A}$ has therefore been reduced to that of determining which pairs of states of $\mathcal{A}$ are equivalent, or nonequivalent, which seems to be easier. This can be done as follows.

Let us call two states $q, q'$ of $\mathcal{A}$

1. 0-**nonequivalent**, if one of them is a final state and the other is not;

2. $i$-**nonequivalent**, for $i > 0$, if they are either $(i-1)$-nonequivalent or there is an $a \in \Sigma$ such that $\delta(q,a)$ and $\delta(q',a)$ are $(i-1)$-nonequivalent.

Let $A_i$ be the set of pairs of $i$-nonequivalent states, $i \geq 0$. Clearly, $A_i \subseteq A_{i+1}$, for all $i \geq 0$, and one can show that $A_n = A_{n+k}$ for any $k \geq 0$ if $n = \text{state}(\mathcal{A})$.

Two states $q, q'$ are not equivalent if and only if there is a $w = w_1 \ldots w_m \in \Sigma^*$ such that $\delta(q,w) \in Q_F$ and $\delta(q',w) \notin Q_F$. This implies that states $\delta(q, w_1 \ldots w_{m-i})$ and $\delta(q', w_1 \ldots w_{m-i})$ are $i$-nonequivalent. Hence, if $q$ and $q'$ are not equivalent, they are $n$-nonequivalent.

The recurrent definition of the sets $A_i$ actually specifies an $\mathcal{O}(n^2m)$ algorithm, $m = |\Sigma|$, to determine equivalent states, and thereby the minimal DFA.

**Example 3.2.18** *The construction of i-nonequivalent states for the DFA in Figure 3.10a yields* $A_0 = \{(1,5),(1,6),(2,5),(2,6),(3,5),(3,6),(4,5),(4,6)\}$, $A_1 = A_0 \cup \{(1,2),(1,4),(2,3),(3,4)\}$, $A_2 = A_1 \cup \{(1,3)\}$, $A_3 = A_2$. *The resulting minimal DFA is depicted in Figure 3.10b.*

It can be shown, by using a more efficient algorithm to determine the equivalence, that one can construct the minimal DFA in sequential time $\mathcal{O}(mn \lg n)$, where $m$ is the size of the alphabet and $n$ is the number of states of the given DFA (see references).

**Exercise 3.2.19** *Design the minimal DFA accepting the language (a) of all words over the alphabet* $\{a,b\}$ *that contain the subword 'abba' and end with the subword 'aaa'; (b) of all words over the alphabet* $\{0,1\}$ *that contain at least two occurrences of the subword* '111';
*(c)* $L = \{w \mid \#_a w \equiv \#_b w \pmod 3\} \subseteq \{a,b\}^*$.

## 3.2.4 Decision Problems

To decide whether two DFA, $\mathcal{A}_1$ and $\mathcal{A}_2$, are equivalent, it suffices to construct the minimal equivalent DFA $\mathcal{A}_1'$ to $\mathcal{A}_1$ and the minimal DFA $\mathcal{A}_2'$ to $\mathcal{A}_2$. $\mathcal{A}_1$ and $\mathcal{A}_2$ are then equivalent if and only if $\mathcal{A}_1'$ and $\mathcal{A}_2'$ are isomorphic. If $n = \max\{state(\mathcal{A}_1), state(\mathcal{A}_2)\}$ and $m$ is the size of the alphabet, then minimization can be done in $\mathcal{O}(mn \lg n)$ sequential time, and the isomorphism can be checked in $\mathcal{O}(nm)$ sequential time.

One way to decide the equivalence of two NFA $\mathcal{A}_1$ and $\mathcal{A}_2$ is to design DFA equivalent to $\mathcal{A}_1$ and $\mathcal{A}_2$ and then minimize these DFA. If the resulting DFA are isomorphic, the original NFA are equivalent; otherwise not. However, this may take exponential time. It seems that there is no essentially better method, because the equivalence problem for NFA is a **PSPACE**-complete problem (see Section 5.11.2).

Two other basic decision problems for FA $\mathcal{A}$ are the **emptiness problem** – is $L(\mathcal{A})$ empty? – and the **finiteness problem** – is $L(\mathcal{A})$ finite? It follows from the next theorem that these two problems are decidable; one has only to check whether there is a $w \in L(\mathcal{A})$ such that $|w| < n$ in the first case and $n \leq |w| < 2n$ in the second case.

**Theorem 3.2.20** *Let* $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ *be a DFA and* $|Q| = n$.
*(1)* $L(\mathcal{A}) \neq \emptyset$ *if and only if there is a* $w \in L(\mathcal{A})$ *such that* $|w| \leq n$.
*(2)* $L(\mathcal{A})$ *is infinite if and only if there is a* $w \in L(\mathcal{A})$ *such that* $n < |w| < 2n$.

Theorem 3.2.20 is actually a corollary of the following basic result.

**Lemma 3.2.21 (Pumping lemma for regular languages)** *If $\mathcal{A}$ is a FA and there is a $w \in L(\mathcal{A})$, $|w| > n = state(\mathcal{A})$, then there are $x,y,z \in \Sigma^*$ such that $w = xyz$, $|xz| \leq n, 0 < |y| \leq n$, and $xy^iz \in L(\mathcal{A})$, for all $i \geq 0$.*

**Proof:** Let $w$ be the shortest word in $L(\mathcal{A})$ with $|w| > n$ and $w = w_1 \ldots w_k, w_i \in \Sigma$. Consider the following sequence of states:

$$q_i = \delta(q_0, w_1 \ldots w_i), 0 \leq i \leq k.$$

Let us now take $i_1$ and $i_2$ such that $0 \leq i_1 < i_2 \leq k, q_{i_1} = q_{i_2}$, and $i_2 - i_1$ is as small as possible. Such $i_1, i_2$ must exist (pigeonhole principle), and clearly $i_2 - i_1 \leq n$. Denote $x = w_1 \ldots w_{i_1}, y = w_{i_1+1} \ldots w_{i_2}$, $z = w_{i_2+1} \ldots w_k$. Then $\delta(q_0, xy^i) = q_{i_1} = q_{i_2}$ for all $i \geq 0$, and therefore also $xy^iz \in L(\mathcal{A})$. Because of the minimality of $w$ we get $|xz| \leq n$. □

**Exercise 3.2.22** *Show the following modification of the pumping lemma for regular languages. Let $L$ be a regular language. There exists an $N_L \in \mathbf{N}$ such that if for some strings $x_1, x_2, x_3, x_1x_2x_3 \in L$ and $|x_2| \geq N_L$, then there exist strings $u, v$ and $w$ such that $x_2 = uvw$, $v \neq \varepsilon$, $|uv| \leq N_L$ and $x_1uv^iwx_3 \in L$ for all $i \geq 0$.*

**Exercise 3.2.23** *Show, using one of the pumping lemmas for regular languages, that the language $\{wcw \mid w \in \{a,b\}^*\}$ is not regular.*

### 3.2.5 String Matching with Finite Automata

Finding all occurrences of a **pattern** in a text is a problem that arises in a large variety of applications, for example, in text editing, DNA sequence searching, and so on. This problem can be solved elegantly and efficiently using finite automata.

**String matching problem**. Given a string (called a **pattern**) $x \in \Sigma^*$, $|x| = m$, design an algorithm to determine, for an arbitrary $y \in \Sigma^*, y = y_1 \ldots y_n, y_j \in \Sigma$ for $1 \leq j \leq n$, all integers $1 \leq i \leq n$ such that $x$ is a suffix of the string $y_1 \ldots y_i$.

A naïve string matching algorithm, which checks in $m$ steps, for all $m \leq i \leq n$, whether $x$ is a suffix of $y_1 \ldots y_i$, clearly requires $\mathcal{O}(mn)$ steps.

The problem can be reduced to that of designing, for a given $x$, a finite automaton $\mathcal{A}_x$ capable of deciding for a given word $y \in \Sigma^*$ whether $y \in \Sigma^*x$.

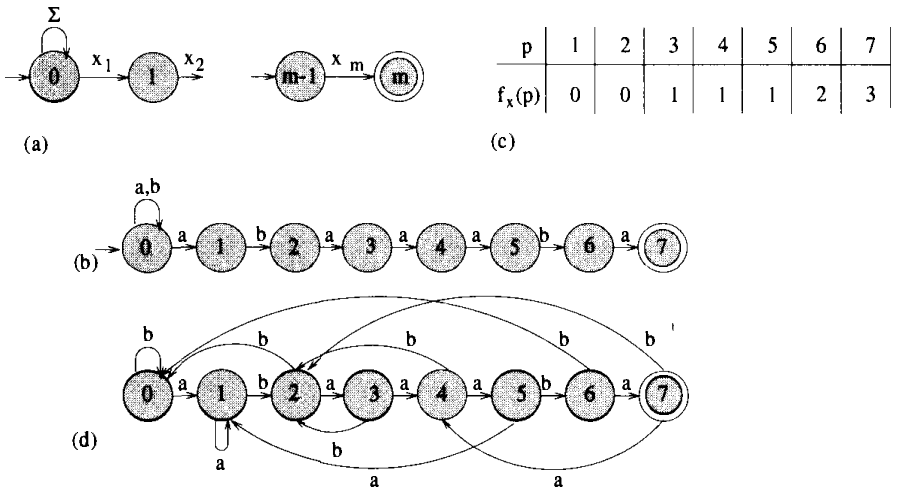
If $x = x_1 \ldots x_m, x_i \in \Sigma$, then the NFA $\mathcal{A}_x$ shown for an arbitrary $x$ in Figure 3.11a and for $x = abaaaba$ in Figure 3.11b accepts $\Sigma^*x$. $\mathcal{A}_x$ has $m+1$ states that can be identified with the elements of the set $P_x$ of prefixes of $x$ – that is, with the set

$$P_x = \{\varepsilon, x_1, x_1x_2, \ldots, x_1x_2 \ldots x_m\}$$

or with the integers from 0 to $m$, with $i$ standing for $x_1 \ldots x_i$.

It is easy to see that the DFA $\mathcal{A}'_x$, which can be obtained from $\mathcal{A}_x$ by the subset construction, has also only $m+1$ states. Indeed, those states of $\mathcal{A}'_x$ that are reachable from the initial state by a word $y$ form exactly the set of those elements of $P_x$ that are suffixes of $y$. This set is uniquely determined by the longest of its elements, say $p$, since the others are those suffixes of $p$ that are in $P_x$. Hence, the states of $\mathcal{A}'$ can also be identified with integers from 0 to $m$. (See $\mathcal{A}'_x$ for $x = abaaaba$ in Figure 3.11d.)

Let $f_x : P_x \to P_x$ be the **failure function** that assigns to each $p \in P_x - \{\varepsilon\}$ the longest proper suffix of $p$ that is in $P_x$. (For $x = abaaaba$ $f_x$ is shown in Figure 3.11c, as a mapping from $\{0, \ldots, 7\}$ to $\{0, \ldots, 7\}$.)




| p | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $f_x(p)$ | 0 | 0 | 1 | 1 | 1 | 2 | 3 |

**Figure 3.11** String matching automata and a failure function

Then the state of $\mathcal{A}'_x$ corresponding to the longest suffix $p$ contains those states of $\mathcal{A}_x$ that correspond to the prefixes

$$p, f_x(p), f_x^2(p), \ldots, \varepsilon.$$

To compute $f_x$, for an $x \in \Sigma^*$, we can use the following recursive rule: $f_x(x_1) = \varepsilon$, and for all $p$, $pa \in P_x - \{\varepsilon\}$:

$$f_x(pa) = \begin{cases} f_x(p)a, & \text{if } f_x(p)a \in P_x; \\ f_x(f_x(p)a), & \text{otherwise.} \end{cases}$$

Once $f_x$ is known, the transition function $\delta_x$ of $\mathcal{A}'$, for $p \in P_x$ and $a \in \Sigma$, has the following form:

$$\delta_x(p,a) = \begin{cases} pa, & \text{if } pa \in P_x; \\ \delta_x(f_x(p),a), & \text{otherwise.} \end{cases}$$

This means that we actually do not need to store $\delta$. Indeed, we can simulate $\mathcal{A}'_x$ on any input word $y$ by the following algorithm, one of the pearls of algorithm design, with the input $x, f_x, y$.

**Algorithm 3.2.24 (Knuth–Morris–Pratt's string matching algorithm)**

$m \leftarrow |x|; n \leftarrow |y|, q \leftarrow 0;$
**for** $i \leftarrow 1$ **to** $n$ **do while** $0 < q < m$ and $x_{q+1} \neq y_i$ **do** $q \leftarrow f_x(q)$ **od**;
**if** $q < m$ and $x_{q+1} = y_i$ **then** $q \leftarrow q + 1$;
**if** $q = m$ **then** print 'pattern found starting with $(i-m)$-th symbol';
$q \leftarrow f_x(q)$
**od**

$\mathcal{O}(m)$ steps are needed to compute $f_x$, and since $q$ can get increased at most by 1 in an $i$- cycle, the overall time of Knuth–Morris–Pratt's algorithm is $\mathcal{O}(m+n)$. (Quite an improvement compared to $\mathcal{O}(mn)$ for the naïve algorithm.)

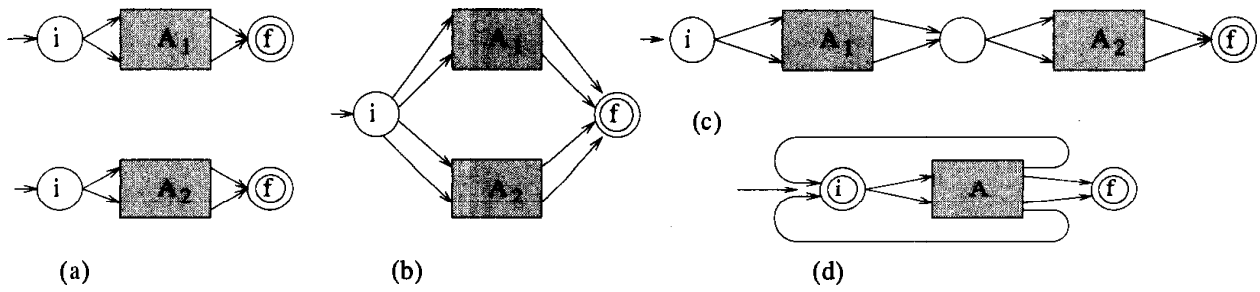



**Figure 3.12** Closure of regular languages under union, concatenation and iteration

**Exercise 3.2.25** *Compute the failure function for the patterns (a) aabaaabaaaab; (b) aabbaaabbb.*

**Exercise 3.2.26** *Show in detail why the overall time complexity of Knuth–Morris–Pratt's algorithm is* $O(m+n)$.

## 3.3 Regular Languages

Regular languages are one of the cornerstones of formal language theory, and they have many interesting and important properties.

### 3.3.1 Closure Properties

The family of regular languages is closed under all basic language operations. This fact can be utilized in a variety of ways, especially to simplify the design of FA recognizing given regular languages.

**Theorem 3.3.1** *The family of regular languages is closed under the operations*

1. *union, concatenation, iteration, complementation and difference;*
2. *substitution, morphism and inverse morphism.*

**Proof:** To simplify the proof, we assume, in some parts of the proof, that the state graphs $G_{\mathcal{A}}$ of those FA we consider are in the normal form shown in Figure 3.12a: namely, there is no edge entering the input state $i$, and there is a single final state $f$ with no outgoing edge. Given a FA $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ accepting a regular language that does not contain the empty word, it is easy to construct an equivalent FA in the above normal form. Indeed, it is enough to add two new states, $i$ – a new input state – and $f$ – a new terminal state – and the following sets of state transitions:

- $\{(i,a,q) \mid (q_0,a,q) \in \delta\}$;
- $\{(p,a,f) \mid (p,a,q) \in \delta, q \in Q_F\}$;
- $\{(i,a,f) \mid (q_0,a,q) \in \delta, q \in Q_F\}$.

To simplify the proof of the theorem we assume, in addition, that languages we consider do not contain the empty word. The adjustments needed to prove the theorem in full generality are minor.

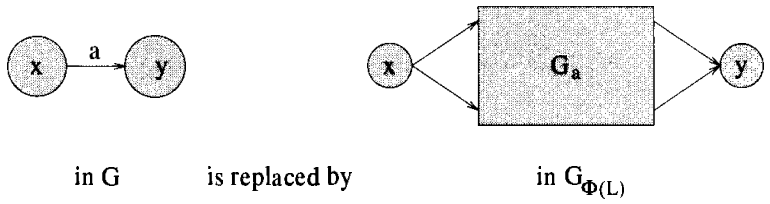



**Figure 3.13** Closure of regular languages under substitution

For example, by taking the state $i$ in Figure 3.12a as an additional terminal state, we add $\varepsilon$ to the language.

Figures 3.12b, c, d show how to design a FA accepting the **union, concatenation** and **iteration** of regular languages provided that FA in the above normal form are given for these languages. (In the case of union, transitions from the new initial state lead exactly to those states to which transitions from the initial states of the two automata go. In the case of iteration, each transition to the final state is doubled, to go also to the initial state.)

**Complementation.** If $\mathcal{A}$ is a DFA over the alphabet $\Sigma$ accepting a regular language $L$, then by exchanging final and nonfinal states in $\mathcal{A}$ we get a DFA accepting the complement of $L$ – the language $L^c$. More formally, if $L = L(\mathcal{A})$, $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$, then $L^c = L(\mathcal{A}')$, where $\mathcal{A}' = \langle \Sigma, Q, q_0, Q - Q_F, \delta \rangle$.

**Intersection.** Let $L_1 = L(\mathcal{A}_1)$, $L_2 = L(\mathcal{A}_2)$, where $\mathcal{A}_1 = \langle \Sigma, Q_1, q_{1,0}, Q_{1,F}, \delta_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, q_{2,0}, Q_{2,F}, \delta_2 \rangle$ are DFA. The intersection $L_1 \cap L_2$ is clearly the language accepted by the DFA

$$\langle \Sigma, Q_1 \times Q_2, (q_{1,0}, q_{2,0}), Q_{1,F} \times Q_{2,F}, \delta \rangle,$$

where $\delta((p,q),a) = (\delta_1(p,a), \delta_2(q,a))$ for any $p \in Q_1$, $q \in Q_2$ and $a \in \Sigma$.

**Difference.** Since $L_1 - L_2 = L_1 \cap L_2^c$ the closure of regular languages under difference follows from their closure under complementation and intersection.

**Substitution.** Let $\phi : \Sigma \to 2^{\Sigma_1^*}$ be a substitution such that $\phi(a)$ is a regular language for each $a \in \Sigma$. Let $L$ be a regular language over $\Sigma$, and $L = L(\mathcal{A})$, $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$. For each $a \in \Sigma$ let $G_a$ be the state graph in the normal form for the language $\phi(a)$. To get the state graph for a FA accepting the language $\phi(L)$ from the state graph $G_{\mathcal{A}}$, it suffices to replace in $G_{\mathcal{A}}$ any edge labelled by an $a \in \Sigma$ by the state graph $G_a$ in the way shown in Figure 3.13.

The closure of regular languages under **morphism** follows from the closure under substitution.

**Inverse morphism.** Let $\phi : \Sigma \to \Sigma_1^*$ be a morphism, $L \subset \Sigma_1^*$ a regular language, $L = L(\mathcal{A})$ for a FA $\mathcal{A}$. As defined in Section 2.5.1,

$$\phi^{-1}(L) = \{w \in \Sigma^* \mid \phi(w) \in L\}.$$

Let $G_{\mathcal{A}} = \langle V, E \rangle$ be the state graph for $\mathcal{A}$. The state graph $G_{\phi^{-1}(L)}$ for a FA recognizing the language $\phi^{-1}(L)$ will have the same set of nodes (states) as $G_{\mathcal{A}}$ and the same set of final nodes (states). For any $a \in \Sigma$, $q \in V$ there will be an edge $(p,a,q)$ in $G_{\phi^{-1}(L)}$ if and only if $p \overset{*}{\underset{\phi(a)}{\Longrightarrow}} q$ in $\mathcal{A}$. Clearly, $w$ is a label of a path in $G_{\phi^{-1}(L)}$, from the initial to a final node, if and only if $\phi(w) \in L$. □

Using the results of Theorem 3.3.1 it is now easy to see that regular languages form a Kleene algebra. Actually, regular languages were the original motivation for the introduction and study of Kleene algebras.

**Exercise 3.3.2** *Show that if $L \subseteq \Sigma^*$ is a regular language, then so are the languages (a) $L^R = \{w \mid w^R \in L\}$; (b) $\{u\}^{-1}L$, where $u \in \Sigma^*$.*

### 3.3.2 Regular Expressions

There are various formal systems that can be used to describe exactly regular languages. (That is, each language they describe is regular, and they can be used to describe any regular language.) The most important is that of **regular expressions.** They will now be defined inductively, together with their semantics (interpretation), a mapping that assigns a regular language to any regular expression.

**Definition 3.3.3** *A regular expression $E$, over an alphabet $\Sigma$, is an expression formed using the following rules, and represents the language $L(E)$ defined as follows:*

1. *$\emptyset$ is a regular expression, and $L(\emptyset) = \emptyset$.*
2. *$a$ is a regular expression for any $a \in \Sigma \cup \{\varepsilon\}$ and $L(a) = \{a\}$.*
3. *If $E_1, E_2$ are regular expressions, then so are*

$$(E_1 + E_2), (E_1 \cdot E_2), (E_1^*)$$

*and*

$$L((E_1 + E_2)) = L(E_1) \cup L(E_2),\ L((E_1 \cdot E_2)) = L(E_1) \cdot L(E_2),\ L((E_1^*)) = L(E_1^*),$$

*respectively.*

4. *There are no other regular expressions over $\Sigma$.*

**Remark 3.3.4** Several conventions are used to simplify regular expressions. First, the following priority of operators is assumed: $*, \cdot, +$. Second, the operators of concatenation are usually omitted. This allows us to omit most of the parentheses. For example, a regular expression describing a word $w = a_1 \ldots a_n$ is usually written as $a_1 a_2 \ldots a_n$ and not $(\ldots((a_1.a_2).a_3)\ldots a_n)$. Finally, the expression $\{w_1, \ldots, w_n\}$ is used to denote the finite language containing the words $w_1, \ldots, w_n$.

**Example 3.3.5** *$\{0,1\}^*$, $\{0,1\}^*000\{0,1\}^*$ and $\{a,b\}^*c\{a,b\}^*c\{a,b\}^*$ are regular expressions.*

**Regular expressions and finite automata**

The following theorem, identifying languages accepted by finite automata and described by regular expressions, is one of the cornerstones of formal language and automata theory, as well as of their applications.

**Theorem 3.3.6 (Kleene's theorem)** *A language $L$ is regular if and only if there is a regular expression $E$ such that $L = L(E)$.*

**Proof:** It follows from Theorem 3.3.1 that each language described by a regular expression is regular. To finish the proof of the theorem, it is therefore sufficient to show how to design, given a DFA $\mathcal{A}$, a regular expression $E_{\mathcal{A}}$ such that $L(E_{\mathcal{A}}) = L(\mathcal{A})$. This is quite a straightforward task once a proper notation is introduced.

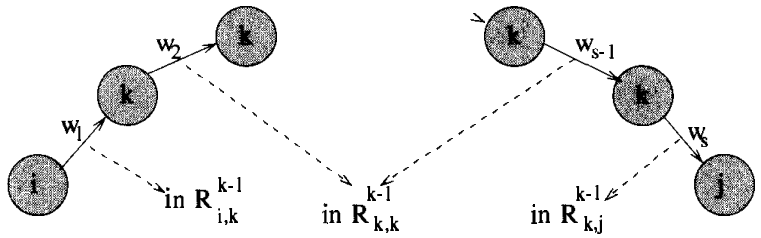



**Figure 3.14** Decomposition of a computational path for a word $w \in R^k_{i,j}$, $w = w_1 w_2 \ldots w_s$

Let $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$. Without loss of generality we assume that $Q = \{0, 1, \ldots, n\}$, $q_0 = 0$ and consider, for $0 \leq i, j \leq n$, $-1 \leq k \leq n$, the set $R^k_{ij}$ defined by

$$R^k_{i,j} = \{w \in \Sigma^* \mid \delta(i,w) = j \text{ and } \delta(i,u) \leq k \text{ for any proper prefix } u \text{ of } w\}.$$

In other words, the set $R^k_{i,j}$ contains those strings that make $\mathcal{A}$ go from state $i$ to state $j$ passing only through states in the set $\{0,1,\ldots,k\}$. Clearly,

$$R^{-1}_{i,j} = \{a \mid \delta(i,a) = j\} \cup \{\varepsilon \mid \text{if } i = j\} \subseteq \Sigma \cup \{\varepsilon\},$$

and therefore there is a regular expression representing $R^{-1}_{i,j}$. Since

$$L(A) = \bigcup_{j \in Q_F} R^n_{0,j},$$

in order to prove the theorem, it suffices to show the validity of the recurrence (3.1) for any $R^k_{i,j}$:

$$R^k_{i,j} = \begin{cases} \emptyset, & \text{if } k = -1, i \neq j, \delta(i,a) \neq j; \\ \{a\}, & \text{if } k = -1, i \neq j, \delta(i,a) = j; \\ \{\varepsilon\}, & \text{if } k = -1, i = j, \delta(i,a) \neq j; \\ \{\varepsilon, a\}, & \text{if } k = -1, i = j, \delta(i,a) = j; \\ R^{k-1}_{i,j} \cup R^{k-1}_{i,k} (R^{k-1}_{k,k})^* R^{k-1}_{k,j}, & \text{if } k \geq 0. \end{cases} \tag{3.1}$$

Once this is done, it is straightforward to show by induction that for each language $R^k_{i,j}$ there is a regular expression representing it.

However, the validity of the recurrence (3.1) is actually easy to see. Any path in $G_{\mathcal{A}}$ from a state $i$ to a state $j$ that passes only through states $\{0,1,\ldots,k\}$ can be decomposed into subpaths that may contain the state $k$ only at the beginning or the end of the path (if the state $k$ occurs on the path at all). See Figure 3.14 for the case that the state $k$ occurs several times. These subpaths belong to one of the subsets $R^{k-1}_{i,k}$ or $R^{k-1}_{k,k}$ or $R^{k-1}_{k,j}$. □

Two regular expressions $E_1$ and $E_2$ are said to be equivalent if they describe the same language. Some of the most basic pairs of equivalent regular expressions are listed in the right-hand column of Table 2.1.

**Exercise 3.3.7** *Determine which of the following equalities between regular languages are valid:*
*(a)* $(011 + (10)^*1 + 0)^* = 011(011 + (10)^*1 + 0)^*$;
*(b)* $((1+0)^*100(1+0)^*)^* = ((1+0)100(1+0)^*100)^*$.

### Design of finite automata from regular expressions

One of the advantages of regular expressions is that they specify a regular language in a natural way in a linear form, by a string. Regular expressions are therefore very convenient as a specification language for regular languages, especially for processing on computers. An important practical problem is to design, given a regular expression $E$, a FA (or a DFA) that recognizes the language $L(E)$. An elegant way of doing this, by using the **derivatives of regular expressions**, will now be described. However, in order to do so, a proper notation has to be introduced.

For a regular expression $E$ let $\rho(E)$ be a regular expression that is equal to $\varepsilon$ if $\varepsilon \in L(E)$, and to $\emptyset$ otherwise. (That is, $\rho(E)F$ equals $F$ if the empty word is in $L(E)$, and $\emptyset$ otherwise.) To compute $\rho(E)$ for a regular expression $E$, we can use the following inductive definition of $\rho$ (where $a \in \Sigma$):

$$\begin{array}{llllll} \rho(\emptyset) = \emptyset, & \rho(\varepsilon) = \varepsilon, & \rho(E+F) = \rho(E)+\rho(F), \\ \rho(a) = \emptyset, & \rho(E^*) = \varepsilon, & \rho(E\cdot F) = \rho(E)\cdot\rho(F). \end{array}$$

**Definition 3.3.8** *The derivative of a regular expression $E$ by a symbol $a$, notation $a^{-1}E$, is a regular expression defined recursively by $a^{-1}\emptyset = \emptyset$ and*

$$\begin{array}{llllll} a^{-1}\varepsilon = \emptyset, & a^{-1}b = \emptyset, \text{ if } a \neq b, & a^{-1}(E^*) = (a^{-1}E)\cdot E^*, \\ a^{-1}a = \varepsilon, & a^{-1}(E+F) = a^{-1}E + a^{-1}F, & a^{-1}(E\cdot F) = (a^{-1}E)F + \rho(E)a^{-1}F. \end{array}$$

The extension from the derivatives by symbols to derivatives by words is defined by $\varepsilon^{-1}E = E$ and $(wa)^{-1}E = a^{-1}(w^{-1}E)$.

It can be shown that with respect to any regular expression $E$ the set $\Sigma^*$ is partitioned into the equivalence classes with respect to the relation $w_1 \equiv w_2$ if $w_1^{-1}E = w_2^{-1}E$. $S_E = \{F \mid \exists w \in \Sigma^* : F = w^{-1}E\}$ has only finitely many equivalence classes with respect to the equivalence of regular expressions.

The following method can be used to design, given a regular expression $E$, a DFA $\mathcal{A}_E$ recognizing the language $L(E)$:

1. The state set of $\mathcal{A}_E$ is given by the set of equivalence classes with respect to the relation $\equiv$. We write $w^{-1}$ for $[w]$.

2. For any state $[w]$ and any symbol $a \in \Sigma$ there will be a single transition from $w^{-1}E$: namely, that into the state $[wa]$.

3. The equivalence class for the whole expression $E$ is the initial state. A state $w^{-1}E$ is final if and only if $\varepsilon \in L(w^{-1}E)$.

To illustrate the method, let us consider the regular expression $E = \{a,b\}^*a\{a,b\}\{a,b\}$. Using the notation $S = \{a,b\}$ we have $E = S^*aSS$. For derivatives we get:

$$\begin{array}{llll} a^{-1}E = E+SS, & b^{-1}E = E, \\ (aa)^{-1}E = E+SS+S, & (ab)^{-1}E = E+S, \\ (aaa)^{-1}E = E+SS+S+\{\varepsilon\}, & (aab)^{-1}E = E+S+\{\varepsilon\}, \\ (aba)^{-1}E = E+SS+\{\varepsilon\}, & (abb)^{-1}E = E+\{\varepsilon\}. \end{array}$$

It is easy to verify that no two of these regular expressions are equivalent, and that further derivations do not provide new regular expressions. The resulting state graph is shown in Figure 3.15. Observe that it is the same state diagram as the one in Figure 3.9c, the state diagram obtained from the one in Figure 3.9b by the subset construction. The algorithm just presented for designing a DFA accepting the language described by a regular expression always provides the minimal DFA with this property.

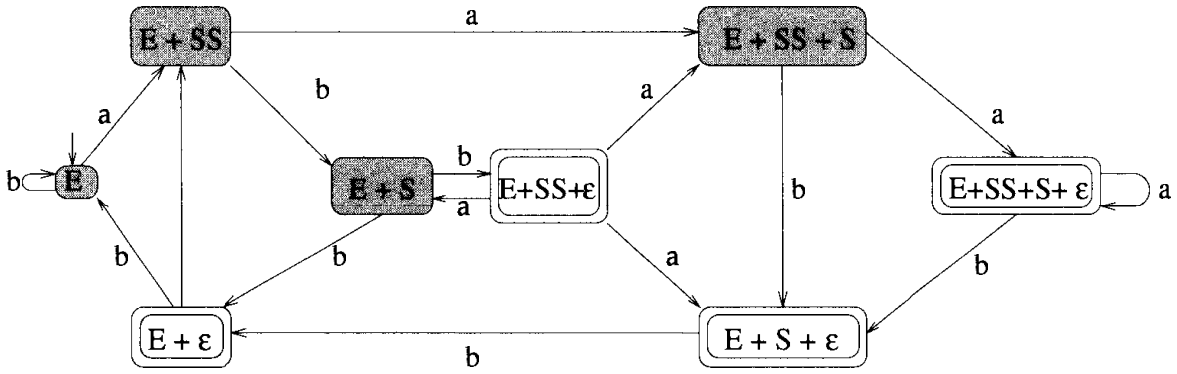



**Figure 3.15** A DFA accepting the language represented by the regular expression $\{a,b\}^*a\{a,b\}\{a,b\}$ and designed using derivatives of regular expressions

**Exercise 3.3.9** *Use the method of derivatives to design a DFA equivalent to the following regular expressions: (a)* $\{a,b\}^*aba\{a,b\}^*$*; (b)* $\{a,b\}^*\{ab,ba\}\{a,b\}^*$.

**Exercise 3.3.10** *Show that the method of derivatives always creates the minimal DFA for a given language. (Hint: show that for each string* $w$*,* $w^{-1}E$ *is the state of the minimal DFA that is reached from the initial state by the input* $w$*.)*

On the other hand, using the ideas presented in the proof of Theorem 3.3.1, given a regular expression, one can design a NFA accepting the same language in linear time.

**Exercise 3.3.11** *Design a NFA describing the same language as the following regular expressions: (a)* $\{\{a,b\}^*aaa\{a,b\}^*\}^*$*; (b)* $aaa\{ab,ba\}^* + aaa\{aa,bb\}^*$.

## 3.3.3 Decision Problems

Finite automata and regular expressions can be seen as two different specification tools for describing regular languages. But how different are they really? One way of understanding the difference is to compare the computational complexity of some main decision problems for DFA and regular expressions.

**The membership problem.** This is the problem of deciding whether, given a regular expression $E$, over an alphabet $\Sigma$, and a $w \in \Sigma^*$, $w \in L(E)$. This can be done in time $\mathcal{O}(|w||E|^2)$. Indeed, in time $\mathcal{O}(|E|)$ one can design a NFA $\mathcal{A}_E$ accepting the same language as $E$, and then, for each symbol of $w$ in time $\mathcal{O}(|E|^2)$, calculate the potential states when simulating acceptance of $w$ on $\mathcal{A}_E$.

**Exercise 3.3.12** *Design in detail an algorithm that decides, given a NFA $\mathcal{A}$ and a word $w$, whether $w \in L(\mathcal{A})$.*

**The emptiness problem** and the **finiteness problem** are, on the other hand, very easy (here we assume that the symbol for the empty language is not used within the expression). They require time proportional to the length of regular expressions. Indeed, if $E$ contains a symbol from $\Sigma$, then the language $L(E)$ is nonempty. Similarly, if $E$ contains a symbol from $\Sigma$ in the scope of an iteration operator, then the language is infinite, and only in such a case.

**Exercise 3.3.13** *Show how to decide, given a FA $\mathcal{A}$ over the alphabet $\Sigma$, whether $L(\mathcal{A}) = \Sigma^*$.*

**The equivalence problem** for regular expressions is, as for NFA, **PSPACE**-complete.

In the rest of this section we discuss the equivalence problem for generalized regular expressions in order to illustrate how the computational complexity of a problem can be altered by a seemingly inessential change in the language used to describe input data. (We shall come to this problem again in Section 5.11.)

The idea of considering some generalized regular expressions is reasonable. We have seen in Section 3.3.1 that the family of regular languages is closed under a variety of operations. Therefore, in principle, we could enhance the language of regular expressions with all these operations: for example, with complementation and intersection, to be very modest. This would in no way increase the descriptional power of such expressions, if measured solely by the family of languages they describe.

Such generalizations of regular expressions look very natural. However, there are good reasons for not using them, unless there are special contra-indications. **Complementation and intersection have enormous descriptive power.** They can be used to describe succinctly various 'complex' regular languages. This in turn can make working with them enormously difficult. One can see this from the following surprising result concerning the equivalence problem for generalized regular expressions.

**Theorem 3.3.14** *The following lower bound holds for the sequential time complexity $T(n)$ of any algorithm that can decide whether two generalized regular expressions with the operations of union, concatenation and complementation, of length $n$, are equivalent:*

$$T(n) = \Omega\left(2^{2^{\cdot^{\cdot^{2^n}}}} \Big\}\log n \text{ times}\right)$$

An even higher lower bound has been obtained for algorithms deciding the equivalence of regular expressions when the iteration operation is also allowed.

Why is this? There is a simple explanation. Using the operation of complementation, one can enormously shorten the description of some regular expressions. Since the time for deciding equivalence is measured with respect to the length of the input (and regular expressions with operations of negation can be very short), the resulting time can be very large indeed.

**Example 3.3.15** *If $\Sigma$ is an alphabet, $|\Sigma| \geq 2$, $x \neq y \in \Sigma^*$, then $\overline{\{x\}}$ and $\overline{(\Sigma^* x \Sigma^*)} \cap \overline{(\Sigma^* y \Sigma^*)}$ are simple examples of generalized regular expressions for which the corresponding regular expressions are much more complex.*

**Exercise 3.3.16** *Give regular expressions that describe the same language as the following generalized regular expressions for* $\Sigma = \{a,b,c\}$: *(a)* $\overline{(\Sigma^* \cdot \{abc\} \cdot \Sigma^*)}$; *(b)* $\overline{(\Sigma^* \cdot \{aba\} \cdot \Sigma^*)} \cap \overline{(\Sigma^* \cdot \{bcb\} \cdot \Sigma^*)}$.

### 3.3.4 Other Characterizations of Regular Languages

In addition to FA and regular expressions, there are many other ways in which regular languages can be described and characterized. In this section we deal with three of them. The first concerns syntactical monoids (see Section 2.6.2).

Let $L \subseteq \Sigma^*$ be a language. The **context** of a string $w \in \Sigma^*$, with respect to $L$, is defined by

$$C_L(w) = \{(u,v) \mid uwv \in L\}.$$

The relation $\equiv_L$ on $\Sigma^* \times \Sigma^*$, defined by

$$x \equiv_L y \iff C_L(x) = C_L(y),$$

is clearly an equivalence relation (the so-called **syntactical equivalence**). In addition, it is a congruence in the free monoid $(\Sigma^*, \cdot)$, because

$$x_1 \equiv_L y_1, x_2 \equiv_L y_2 \Rightarrow x_1x_2 \equiv_L y_1y_2.$$

This implies that the set of equivalence classes $[w]_L$, $w \in \Sigma^*$, with respect to the relation $\equiv_L$ and the operation $[w_1]_L.[w_2]_L = [w_1w_2]_L$ forms a monoid, the **syntactical monoid** of $L$.

**Theorem 3.3.17 (Myhill's theorem)** *A language $L$ is regular if and only if its syntactical monoid is finite.*

**Proof:** Let $L$ be a regular language and $L = L(\mathcal{A})$ for a DFA $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$. For any $p \in Q$ let $l(p) = \{w \mid \delta(q_0,w) = p\}$. Moreover, for any $w \in \Sigma^*$ let $S_w = \{(p,q) \mid \delta(p,w) = q\}$. Clearly,

$$C_L(w) = \bigcup_{(p,q) \in S_w} l(p) \times L(q).$$

Since the number of different sets $S_w$ is finite, so is the number of contexts $C_L(w)$; therefore the syntactical monoid of $L$ is finite.

Now let us assume that the syntactical monoid $\mathcal{M}$ for a language $L \subseteq \Sigma^*$ is finite. We design a DFA recognizing $L$ as follows. Elements of $\mathcal{M}$, that is the equivalence classes with respect to the relation $\equiv_L$, will be the states, with $[\varepsilon]_L$ the initial state. States of the form $[w]_L, w \in L$ will be the final states. For a state $[w]$ and a symbol $a \in \Sigma$ there will be a transition $([w]_L, a, [wa]_L)$. Clearly, the resulting DFA recognizes $L$. □

The second characterization is in terms of the **prefix equivalence** $\equiv_L^p$ defined for a language $L \in \Sigma^*$ by

$$w_1 \equiv_L^p w_2 \iff \forall u \in \Sigma^* (w_1u \in L \Leftrightarrow w_2u \in L).$$

**Theorem 3.3.18 (Nerode's theorem)** *A language $L$ is regular if and only if its prefix equivalence has finitely many equivalence classes. If a language $L$ is regular, then the number of its prefix equivalence classes equals the number of states of the minimal DFA for $L$.*

**Proof:** (1) If $L$ is regular, then by Myhill's theorem the set of syntactical equivalence classes of $L$ is finite. Since $u \equiv_L w \Rightarrow u \equiv_L^p w$, the set of prefix equivalence classes of $L$ has to be finite also.

(2) Let the number of prefix equivalence classes $[w]_L^p$ be finite. These classes will be the states of a DFA $\mathcal{A}$ that will recognize $L$, and is defined as follows. $[\varepsilon]_L^p$ is the initial state and $\{[w]_L^p \mid w \in L\}$ are final states. The transition function $\delta$ is defined by $\delta([w]_L^p, a) = [wa]_L^p$. Since $w_1 \equiv_L^p w_2 \Rightarrow w_1 a \equiv_L^p w_2 a$ for all $w_1, w_2$ and $a$, $\delta$ is therefore well defined, and $\mathcal{A}$ clearly recognizes $L$.

The resulting DFA has to be minimal, because no two of its states are equivalent; this follows from the definition of prefix equivalence classes. □

**Exercise 3.3.19** *Show that syntactical monoids of the following languages are infinite (and therefore these languages are not regular): (a)* $\{a^n b^n \mid n > 0\}$; *(b)* $\{a^i \mid i \text{ is prime}\}$.

**Exercise 3.3.20** *Determine the syntax equivalence and prefix equivalence classes for the following languages: (a)* $\{a,b\}^* aa\{a,b\}^*$; *(b)* $\{a^i b^j \mid i,j \geq 1\}$.

Nerode's theorem can also be used to derive lower bounds on the number of states of DFA for certain regular languages.

**Example 3.3.21** *Consider the language* $L_n = \{a,b\}^* a \{a,b\}^{n-1}$. *Let* $x, y$ *be two different strings in* $\{a,b\}^n$, *and let them differ in the i-th left-most symbol. Clearly,* $xb^{i-1} \in L \iff yb^{i-1} \notin L$, *because one of the strings* $xb^{i-1}$ *and* $yb^{i-1}$ *has a and the second b in the n-th position from the right. This implies that* $L_n$ *has at least* $2^n$ *prefix equivalence classes, and therefore each DFA for* $L_n$ *has to have at least* $2^n$ *states.*

**Exercise 3.3.22** *Design an* $n+1$ *state NFA for the language* $L_n$ *from Example 3.3.21 (and show in this way that for* $L_n$ *there is an exponential difference between the minimal number of states of NFA and DFA recognizing* $L_n$*).*

**Exercise 3.3.23** *Show that the minimal deterministic FA to accept the language* $L = \{w \mid \#_a w \bmod k = 0\} \subseteq \{a,b\}^*$ *has k states, and that no NFA with less than k states can recognize L.*

**Example 3.3.24 (Recognition of regular languages in logarithmic time)** *We show now how to use the syntactical monoid of a regular language L to design an infinite balanced-tree network of processors (see Figure 3.16) recognizing L in parallel logarithmic time.*

Since the number of syntactical equivalence classes of a regular language is finite, they can be represented by symbols of a finite alphabet. This will be used in the following design of a tree network of processors.

Each processor of the tree network has one external input. For a symbol $a \in \Sigma$ on its external input the processor produces as an output symbol representing the (syntactical equivalence) class $[a]_L$. For the input #, a special marker, on its external input the processor produces as the output symbol representing the class $[\varepsilon]_L$.

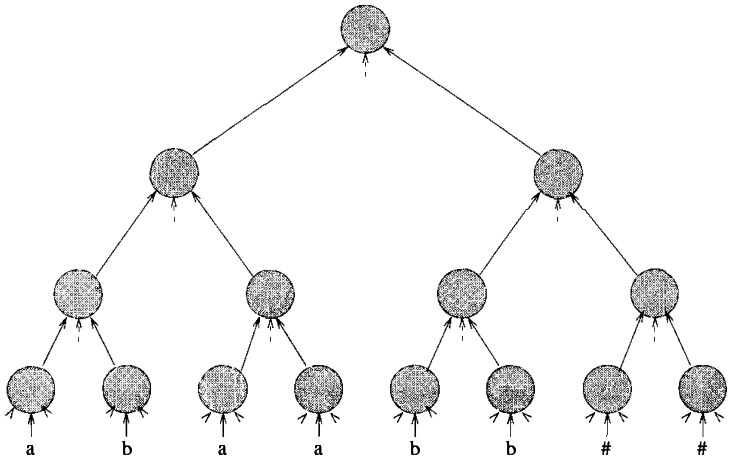



**Figure 3.16** Tree automaton recognizing a regular language

The tree automaton works as follows. An input word $w = a_1 \ldots a_n \in \Sigma^*$ is given, one symbol per processor, to the external inputs of the left-most processors of the topmost level of processors that has at least $|w|$ processors. The remaining processors at that level receive, at their external inputs, the marker # (see Figure 3.16 for $n = 6$). All processors of the input level process their inputs simultaneously, and send their results to their parents. (Processors of all larger levels are 'cut off' in such a computation.) Processing in the network then goes on, synchronized, from one level to another, until the root processor is reached. All processors of these levels process only internal inputs; no external inputs are provided. An input word $w$ is accepted if and only if at the end of this processing the root processor produces a symbol from the set $\{[w]_L \mid w \in L\}$.

It is clear that such a network of memory-less processors accepts the language $L$. It is a simple and fast network; it works in logarithmic time, and therefore much faster than a DFA. However, there is a price to pay for this. It can be shown that in some cases for a regular language accepted by a NFA with $n$ states, the corresponding syntactical monoid may have up to $n^n$ elements. The price to be paid for recognition of regular languages in logarithmic time by a binary tree network of processors can therefore be very high in terms of the size of the processors (they need to process a large class of inputs), and it can also be shown that in some cases there is no way to avoid paying such a price.

**Exercise 3.3.25** *Design a tree automaton that recognizes the language (a)* $\{a^{2^n} \mid n \geq 0\}$ *(note that this language is not regular); (b)* $\{w \mid w \in \{a\}^*\{b\}^*, |w| = 2^k, k \geq 1\}$.

## 3.4 Finite Transducers

Deterministic finite automata are recognizers. However, they can also be seen as computing characteristic functions of regular languages – the output of a DFA $\mathcal{A}$ is 1 (0) for a given input $w$ if $\mathcal{A}$ comes to a terminal (nonterminal) state on the input $w$. In this section several models of finite state machines computing other functions, or even relations, are considered.

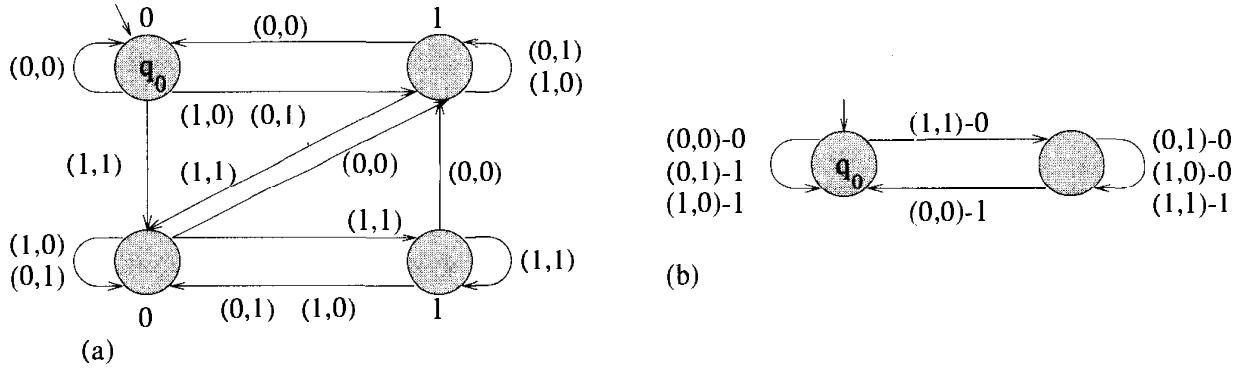



Figure 3.17 Moore and Mealy machines for serial addition

### 3.4.1 Mealy and Moore Machines

Two basic models of finite transducers, as models of finite state machines computing functions, are called the **Moore machine** and the **Mealy machine**. They formalize an intuitive idea of an input–output mapping realized by a finite state machine in two slightly different ways.

**Definition 3.4.1** *In a* **Moore machine** $\mathcal{M} = \langle \Sigma, Q, q_0, \delta, \rho, \Delta \rangle$, *the symbols* $\Sigma, Q, q_0$ *and* $\delta$ *have the same meaning as for DFA,* $\Delta$ *is an* **output alphabet**, *and* $\rho : Q \to \Delta$ *an* **output function**.

For an input word $w = w_1 \ldots w_n$, $w_i \in \Sigma$, $\rho(q_0)\rho(q_1)\ldots\rho(q_n)$ is the corresponding output word, where $q_i = \delta(q_0, w_1 \ldots w_i), 1 \leq i \leq n$. In a Moore machine the outputs are therefore 'produced by states'. Figure 3.17a shows a Moore machine for a serial addition of two binary numbers. (It is assumed that both numbers are represented by binary strings of the same length (leading zeros are appended if necessary), and for numbers $x_n \ldots x_1$, $y_n \ldots y_1$ the input is a sequence of pairs $(x_1, y_1), \ldots, (x_n, y_n)$ in this order. Observe also that the output always starts with one 0 which is then followed by output bits of $bin^{-1}(bin(x_n, \ldots, x_1) \times bin(y_n \ldots y_1)$.

**Definition 3.4.2** *In a* **Mealy machine** $\mathcal{M} = \langle \Sigma, Q, q_0, \delta, \rho, \Delta \rangle$, *symbols* $\Sigma, Q, q_0, \delta, \Delta$ *have the same meaning as in a Moore machine, and* $\rho : Q \times \Sigma \to \Delta$ *is an* **output function**.

For an input word $w = w_1 \ldots w_n$, $w_i \in \Sigma$, $\rho(q_0, w_1) \ldots \rho(q_{n-1}, w_n)$ is the corresponding output word, where $q_i = \delta(q_0, w_1 \ldots w_i)$. Outputs are therefore produced by transitions. Figure 3.17b shows a Mealy machine for the serial addition of two binary numbers $x = x_1 \ldots x_n$, $y = y_1 \ldots y_n$, with inputs presented as above.

Let us now denote by $T_{\mathcal{M}}(w)$ the output produced by a Moore or a Mealy machine $\mathcal{M}$ for the input $w$. For a Moore machine $|T_{\mathcal{M}}(w)| = |w| + 1$ and for a Mealy machine $|T_{\mathcal{M}}(w)| = |w|$. A Moore machine can therefore never be fully equivalent to a Mealy machine. However, it is easy to see that for any Moore machine there is a Mealy machine (and vice versa) such that they are equivalent in the following slightly weaker sense.

**Theorem 3.4.3** *For every Mealy machine* $\mathcal{M}$ *over an alphabet* $\Sigma$ *there is a Moore machine* $\mathcal{M}'$ *over* $\Sigma$ *(and vice versa) such that* $\rho(q_0)T_{\mathcal{M}}(w) = T_{\mathcal{M}'}(w)$, *for every input* $w \in \Sigma^*$, *where* $\rho$ *is the output function of* $\mathcal{M}'$.

**Exercise 3.4.4** *Design (a) a Moore machine (b) a Mealy machine, such that given an integer $x$ in binary form, the machine produces $\lfloor \frac{x}{3} \rfloor$.*

**Exercise 3.4.5*** *Design (a) a Mealy machine (b) a Moore machine that transforms a Fibonacci representation of a number into its normal form.*

**Exercise 3.4.6** *Design a Mealy machine $\mathcal{M}$ that realizes a 3-step delay. (That is, $\mathcal{M}$ outputs at time $t$ its input at time $t-3$.)*

## 3.4.2 Finite State Transducers

The concept of a Mealy machine will now be generalized to that of a finite state transducer. One new idea is added: nondeterminism.

**Definition 3.4.7** *A* **finite (state) transducer** *(FT for short) $\mathcal{T}$ is described by a finite set of* **states** $Q$, *a finite* **input alphabet** $\Sigma$, *a finite* **output alphabet** $\Delta$, *the initial state $q_0$ and a finite* **transition relation** $\rho \subseteq Q \times \Sigma^* \times \Delta^* \times Q$. *For short, $\mathcal{T} = \langle Q, \Sigma, \Delta, q_0, \rho \rangle$.*

A FT $\mathcal{T}$ can also be represented by a graph, $G_{\mathcal{T}}$, with states from $Q$ as vertices. There is an edge in $G_{\mathcal{T}}$ from a state $p$ to a state $q$, labelled by $(u,v)$, if and only if $(p,u,v,q) \in \rho$. Such an edge is interpreted as follows: the input $u$ makes $\mathcal{T}$ transfer from state $p$ to state $q$ and produces $v$ as the output.

Each finite transducer $\mathcal{T}$ defines a relation

$$\begin{aligned} R_{\mathcal{T}} \quad = \quad & \{(u,v) \mid \exists (q_0,u_0,v_0,q_1),(q_1,u_1,v_1,q_2),\ldots,(q_n,u_n,v_n,q_{n+1}), \\ & \text{where } (q_i,u_i,v_i,q_{i+1}) \in \rho, \text{for } 0 \leq i \leq n, \text{ and } u = u_0 \ldots u_n, v = v_0 \ldots v_n\}. \end{aligned}$$

The relation $R_{\mathcal{T}}$ can also be seen as a mapping from subsets of $\Sigma^*$ into subsets of $\Delta^*$ such that for $L \subseteq \Sigma^*$ $R_{\mathcal{T}}(L) = \{v \mid \exists u \in L, (u,v) \in R_{\mathcal{T}}\}$.

Perhaps the most important fact about finite transducers is that they map regular languages into regular languages.

**Theorem 3.4.8** *Let $\mathcal{T} = \langle Q, \Sigma, \Delta, q_0, \rho \rangle$ be a finite transducer. If $L \subseteq \Sigma^*$ is a regular language, then so is $R_{\mathcal{T}}(L)$.*

**Proof:** Let $\Delta' = \Delta \cup \{\#\}$ be a new alphabet with $\#$ as a new symbol not in $\Delta$. From the relation $\rho$ we first design a finite subset $A_\rho \subset Q \times \Sigma^* \times \Delta'^* \times Q$ and then take $A_\rho$ as a new alphabet. $A_\rho$ is designed by a decomposition of productions of $\rho$. We start with $A_\rho$ being empty, and for each production of $\rho$ we add to $A_\rho$ symbols defined according to the following rules:

1. If $(p,u,v,q) \in \rho$, $|u| \leq 1$, then $(p,u,v,q)$ is taken into $A_\rho$.
2. If $r = (p,u,v,q) \in \rho$, $|u| > 1$, $u = u_1 \ldots u_k$, $1 \leq i \leq k$, $u_i \in \Sigma$, then new symbols $t_1^r, \ldots, t_{k-1}^r$ are chosen, and all quadruples

$$(p,u_1,\#,t_1^r),(t_1^r,u_2,\#,t_2^r),\ldots,(t_{k-2}^r,u_{k-1},\#,t_{k-1}^r),(t_{k-1}^r,u_k,v,q)$$

are taken into $A_\rho$.

Now let $Q_L$ be the subset of $A_\rho^*$ consisting of strings of the form

$$(q_0, u_0, v_0, q_1)(q_1, u_1, v_1, q_2) \ldots (q_s, u_s, v_s, q_{s+1}) \tag{3.2}$$

such that $v_s \neq \#$ and $u_0 u_1 \ldots u_s \in L$. That is, $Q_L$ consists of strings that describe a computation of $\mathcal{T}$ for an input $u = u_0 u_1 \ldots u_s \in L$. Finally, let $\tau : A_\rho \mapsto \Delta'^*$ be the morphism defined by

$$\tau((p,u,v,q)) = \begin{cases} v, & \text{if } v \neq \#; \\ \varepsilon, & \text{otherwise.} \end{cases}$$

From the way $\mathcal{T}$ and $Q_L$ are constructed it is readily seen that $\tau(Q_L) = R_{\mathcal{T}}(L)$.

It is also straightforward to see that if $L$ is regular, then $Q_L$ is regular too. Indeed, a FA $\mathcal{A}$ recognizing $Q_L$ can be designed as follows. A FA recognizing $L$ is used to check whether the second components of symbols of a given word $w$ form a word in $L$. In parallel, a check is made on whether $w$ represents a computation of $\mathcal{T}$ ending with a state in $Q$. To verify this, the automaton needs always to remember only one of the previous symbols of $w$; this can be done by a finite automaton.

As shown in Theorem 3.3.1, the family of regular languages is closed under morphisms. This implies that the language $R_{\mathcal{T}}(L)$ is regular. □

Mealy machines are a special case of finite transducers, as are the following generalizations of Mealy machines.

**Definition 3.4.9** *In a* **generalized sequential machine** $\mathcal{M} = (Q, \Sigma, \Delta, q_0, \delta, \rho)$, *symbols* $Q, \Sigma, \Delta$ *and* $q_0$ *have the same meaning as for finite transducers,* $\delta : Q \times \Sigma \to Q$ *is a transition mapping, and* $\rho : Q \times \Sigma \to \Delta^*$ *is an output mapping.*

Computation on a generalized sequential machine is defined exactly as for a Mealy machine. Let $f_{\mathcal{M}} : \Sigma^* \to \Delta^*$ be the function defined by $\mathcal{M}$. For $L \subseteq \Sigma^*$ and $L' \subseteq \Delta^*$ we therefore consider $f_{\mathcal{M}}(L)$ and define $f_{\mathcal{M}}^{-1}(L') = \{u \mid u \in \Sigma^*, f_{\mathcal{M}}(u) \in L'\}$.

It follows from Theorem 3.4.8 that if $\mathcal{M}$ is a generalized sequential machine with the input alphabet $\Sigma$ and the output alphabet $\Delta$ and $L \subseteq \Sigma^*$ is a regular language, then so is $f_{\mathcal{M}}(L)$. We show now that a reverse claim also holds: if $L' \subseteq \Delta^*$ is a regular language, then so is $f_{\mathcal{M}}^{-1}(L')$.

Indeed, let $\mathcal{M} = (Q, \Sigma, \Delta, q_0, \delta, \rho)$. Consider the finite transducer $\mathcal{T} = (Q, \Delta, \Sigma, q_0, \delta')$ with $\delta' = \{(p,u,v,q) \mid \delta(p,u) = q, \rho(p,u) = v\} \cup \{(p,\varepsilon,\varepsilon,q)\}$. Clearly, $f_{\mathcal{M}}^{-1}(L') = R_{\mathcal{T}}(L')$ and, by Theorem 3.4.8, $f_{\mathcal{M}}^{-1}(L')$ is regular. Hence

**Theorem 3.4.10** *If* $\mathcal{M}$ *is a generalized sequential machine, then mappings* $f_{\mathcal{M}}$ *and* $f_{\mathcal{M}}^{-1}$ *both preserve regular languages.*

In Section 3.3 we have seen automata-independent characterizations of languages recognized by FA. There exists also a machine-independent characterization of mappings defined by generalized sequential machines.

**Theorem 3.4.11** *For a mapping* $f : \Sigma^* \to \Delta^*$, *there exists a generalized sequential machine* $\mathcal{M}$ *such that* $f = f_{\mathcal{M}}$, *if and only if* $f$ *satisfies the following conditions:*

1. $f$ *preserves prefixes; that is, if* $u$ *is a prefix of* $v$, *then* $f(u)$ *is a prefix of* $f(v)$.
2. $f$ *has a bounded output; that is, there exists an integer* $k$ *such that* $|f(wa)| - |f(w)| \leq k$ *for any* $w \in \Sigma^*, a \in \Sigma$.
3. $f(\varepsilon) = \varepsilon$.

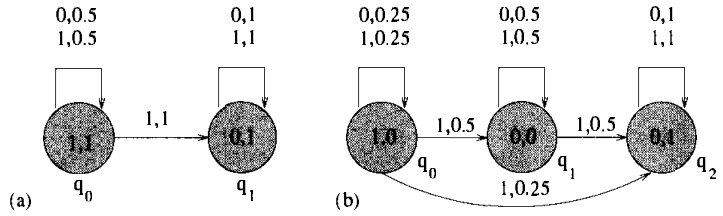



**Figure 3.18** Two WFA computing functions on rationals and reals

4. *$f^{-1}(L)$ is a regular language if $L$ is regular.*

**Exercise 3.4.12** *Let $f$ be the function defined by $f(a) = b, f(b) = a$ and $f(x) = x$ for $x \in \{a,b\}^* - \{a,b\}$. (a) Does $f$ preserve regular languages? (b) Can $f$ be realized by a generalized sequential machine?*

**Exercise 3.4.13*** *Show how to design, given a regular language $R$, a finite transducer $\mathcal{T}_R$ such that $\mathcal{T}_R(L) = L \diamond R$ (where $\diamond$ denotes the shuffle operation introduced in Section 2.5.1).*

## 3.5 Weighted Finite Automata and Transducers

A seemingly minor modification of the concepts of finite automata and transducers, an assignment of weights to transitions and states, results in finite state devices with unexpected computational power and importance for image processing. In addition, the weighted finite automata and the transducers introduced in this section illustrate a well-known experience that one often obtains powerful practical tools by slightly modifying and 'twisting' theoretical concepts.

### 3.5.1 Basic Concepts

The concept of a weighted finite automaton is both very simple and tricky at the same time. Let us therefore start with its informal interpretation for the case in which it is used to generate images. Each state $p$ determines a function that assigns a greyness value to each pixel, represented by an input word $w$, and therefore it represents an image. This image is computed as follows: to each path starting in $p$ and labelled by $w$ a value (of greyness) is computed by multiplying the weights of all transitions along the path and, in addition, the so-called terminal weight of the final state of the path. These values are then added for all paths from $p$ labelled by $w$. The initial weights of all nodes are then used to form a linear combination of these functions to get the final image-generating function. More formally,

**Definition 3.5.1** *A* **weighted finite automaton** *(for short WFA) $\mathcal{A}$ is described by a finite set of* **input symbols** *$\Sigma$, a finite set of* **states** *$Q$, an* **initial distribution** *$i : Q \to \mathbf{R}$, and a* **terminal distribution** *$t : Q \to \mathbf{R}$ of states, as well as a* **weighted transition function** *$w : Q \times \Sigma \times Q \to \mathbf{R}$. In short, $\mathcal{A} = (\Sigma, Q, i, t, w)$.*

To each WFA $\mathcal{A}$ we first associate the following distribution function $\delta_{\mathcal{A}} : Q \times \Sigma^* \to \mathbf{R}$:

$$\delta_{\mathcal{A}}(p,\varepsilon) = t(p); \tag{3.3}$$

$$\delta_{\mathcal{A}}(p,au) = \sum_{q \in Q} w(p,a,q)\delta_{\mathcal{A}}(q,u) \text{ for each } p \in Q, a \in \Sigma, u \in \Sigma^*. \tag{3.4}$$

A WFA $\mathcal{T}$ can be represented by a graph $G_{\mathcal{T}}$ (see Figures 3.18a, b) with states as vertices and transitions as edges. A vertex representing a state $q$ is labelled by the pair $(i(q), t(q))$. If $w(p,a,q) = r$ is nonzero, then there is, in $G_{\mathcal{T}}$, a directed edge from $p$ to $q$ labelled by the pair $(a,r)$.

A WFA $\mathcal{A}$ can now be seen as computing a function $f_{\mathcal{A}} : \Sigma^* \to \mathbf{R}$ defined by

$$f_{\mathcal{A}}(u) = \sum_{p \in Q} i(p)\delta_{\mathcal{A}}(p,u).$$

Informally, $\delta_{\mathcal{A}}(p,u)$ is the sum of all 'final weights' of all paths starting in $p$ and labelled by $u$. The final weight of each path is obtained by multiplying the weights of all transitions on the path and also the final weight of the last node of the path. $f_{\mathcal{A}}(u)$ is then obtained by taking a linear combination of all $\delta_{\mathcal{A}}(p,u)$ defined by the initial distribution $i$.

**Example 3.5.2** *For the WFA $\mathcal{A}_1$ in Figure 3.18a we get*

$$\delta_{\mathcal{A}_1}(q_0,011) = 0.5\cdot 0.5\cdot 0.5\cdot 1 + 0.5\cdot 0.5\cdot 1\cdot 1 + 0.5\cdot 1\cdot 1\cdot 1 = 0.875; \delta_{\mathcal{A}_1}(q_1,011) = 1\cdot 1\cdot 1\cdot 1 = 1,$$

*and therefore* $f_{\mathcal{A}_1}(011) = 1\cdot 0.875 + 0\cdot 1 = 0.875$. *Similarly,* $\delta_{\mathcal{A}_1}(q_0,0101) = 0.625$ *and* $f_{\mathcal{A}_1}(0101) = 0.625$. *For the WFA $\mathcal{A}_2$ in Figure 3.18b we get, for example,*

$$\delta_{\mathcal{A}_2}(q_0,0101) = 0.25\cdot 0.5\cdot 0.5\cdot 1\cdot 1 + 0.25\cdot 0.25\cdot 1\cdot 1\cdot 1 = 0.125,$$

*and therefore also* $f_{\mathcal{A}_2}(0101) = 0.125$.

**Exercise 3.5.3** *Determine, for the WFA $\mathcal{A}_1$ in Figure 3.18a and for $\mathcal{A}_2$ in Figure 3.18b:*
*(a)* $\delta_{\mathcal{A}_1}(q_0,10101), f_{\mathcal{A}_1}(10101)$; *(b)* $\delta_{\mathcal{A}_2}(q_0,10101), f_{\mathcal{A}_1}(10101)$.

**Exercise 3.5.4** *Determine $f_{\mathcal{A}_3}(x)$ and $f_{\mathcal{A}_4}(x)$ for the WFA $\mathcal{A}_3$ and $\mathcal{A}_4$ obtained from $\mathcal{A}_1$ in Figure 3.18a by changing the initial and terminal distributions as follows:*
*(a)* $i(q_0) = 1$, $i(q_1) = 0$, $t(q_0) = 0$, *and* $t(q_1) = 1$; *(b)* $i(q_0) = i(q_1) = 1$, *and* $t(q_0) = t(q_1) = 1$.

**Exercise 3.5.5** *(a) Show that $f_{\mathcal{T}_1}(x) = 2bre(x) + 2^{-|x|}$ for the WFT $\mathcal{T}_1$ depicted in Figure 3.18.*
*(b) determine functions computed by WFA obtained from the one in Figure 3.18a by considering several other initial and terminal distributions.*

If $\Sigma = \{0,1\}$ is the input alphabet of a WFA $\mathcal{A}$, then we can extend $f_{\mathcal{A}} : \Sigma^* \to \mathbf{R}$ to a (partial) **real function** $f^{\omega}_{\mathcal{A}} : [0,1] \to \mathbf{R}$ defined as follows: for $x \in [0,1]$ let $bre^{-1}(x) \in \Sigma^{\omega}$ be the unique binary representation of $x$ (see page 81). Then

$$f^{\omega}_{\mathcal{A}}(x) = \lim_{n\to\infty} f_{\mathcal{A}}(Prefix_n(bre^{-1}(x))),$$

provided the limit exists; otherwise $f^{\omega}_{\mathcal{A}}(x)$ is undefined.

For the rest of this section, to simplify the presentation, a binary string $x_1 \ldots x_n$, $x_i \in \{0,1\}$ and an $\omega$-string $y = y_1 y_2 \ldots$ over the alphabet $\{0,1\}$ will be interpreted, depending on the context, either as strings $x_1 \ldots x_n$ and $y_1 y_2 \ldots$ or as reals $0.x_1 \ldots x_n$ and $0.y_1 y_2 \ldots$. Instead of $bin(x)$ and $bre(y)$, we shall often write simply $x$ or $y$ and take them as strings or numbers.

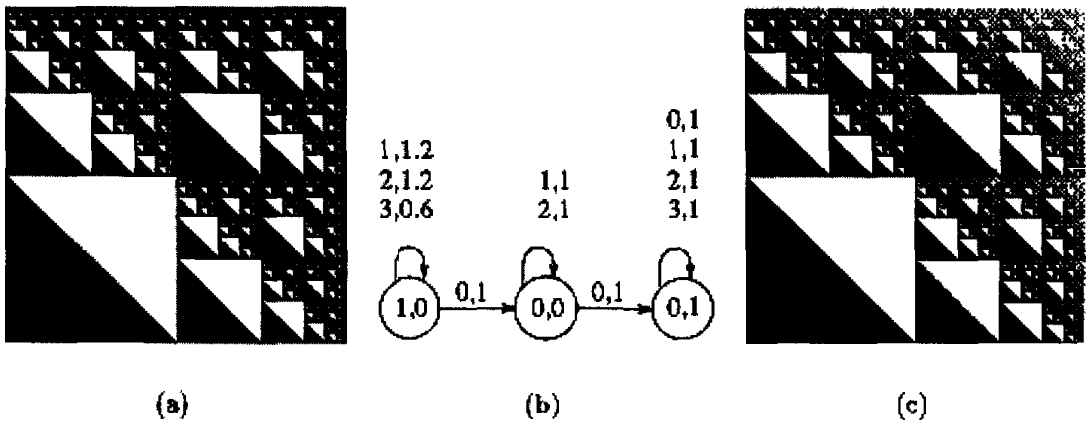



**Figure 3.19** Generation of a fractal image

**Exercise 3.5.6** *Show, for the WFA $\mathcal{A}_1$ in Figure 3.18a, that (a) if $x \in \Sigma^*$, then $f_{\mathcal{A}_1}(x0^n) = 2bre(x) + 2^{-(n+|x|)}$; (b) $f_{\mathcal{A}_1}(x) = 2bre(x)$.*

**Exercise 3.5.7** *Show that $f_{\mathcal{A}}(x)^{\omega}(x) = x^2$ for the WFA $\mathcal{A}_2$ in Figure 3.18b.*

**Exercise 3.5.8** *Determine $f^{\omega}_{\mathcal{A}_3}(x)$ for the WFA obtained from WFA $\mathcal{A}_2$ by taking other combinations of values for the initial and final distributions.*

Of special importance are WFA over the alphabet $P = \{0,1,2,3\}$. As shown in Section 2.5.3, a word over $P$ can be seen as a pixel in the square $[0,1] \times [0,1]$. A function $f_{\mathcal{A}} : P^* \to \mathbf{R}$ is then considered as a multi-resolution image with $f_{\mathcal{A}}(u)$ being the greyness of the pixel specified by $u$. In order to have compatibility of different resolutions, it is usually required that $f_{\mathcal{A}}$ is **average-preserving**. That is, it holds that

$$f_{\mathcal{A}}(u) = \frac{1}{4}[f_{\mathcal{A}}(u0) + f_{\mathcal{A}}(u1) + f_{\mathcal{A}}(u2) + f_{\mathcal{A}}(u3)].$$

In other words, the greyness of a pixel is the average of the greynesses of its four main subpixels. (One can also say that images in different resolutions look similar if $f_{\mathcal{A}}$ is average-preserving – multi-resolution images contain only more details.)

It is easy to see that with the pixel representation of words over the alphabet $P$ the language $L = \{1,2,3\}^*0\{1,2\}^*0\{0,1,2,3\}^*$ represents the image shown in Figure 3.19a (see also Exercise 2.5.17). At the same time $L$ is the set of words $w$ such that $f_{\mathcal{A}}(w) = 1$ for the WFA obtained from the one in Figure 3.19b by replacing all weights by 1. Now it is easy to see that the average-preserving WFA shown in Figure 3.19b generates the grey-scale image from Figure 3.19c.

The concept of a WFA will now be generalized to a **weighted finite transducer** (for short, WFT).

**Definition 3.5.9** *In a WFT $\mathcal{T} = \langle \Sigma_1, \Sigma_2, Q, i, t, w \rangle$, $\Sigma_1$ and $\Sigma_2$ are input alphabets; $Q$, $i$ and $t$ have the same meaning as for a WFA; and $w : Q \times (\Sigma_1 \cup \{\varepsilon\}) \times (\Sigma_2 \cup \{\varepsilon\}) \times Q \to \mathbf{R}$ is a* **weighted transition function**.

We can associate to a WFT $\mathcal{T}$ the state graph $G_{\mathcal{T}}$, with $Q$ being the set of nodes and with an edge from a node $p$ to a node $q$ with the label $(a_1, a_2 : r)$ if $w(p, a_1, a_2, q) = r$.

A WFT $\mathcal{T}$ specifies a **weighted relation** $R_{\mathcal{T}} : \Sigma_1^* \times \Sigma_2^* \to \mathbf{R}$ defined as follows. For $p,q \in Q$, $u \in \Sigma_1^*$ and $v \in \Sigma_2^*$, let $A_{p,q}(u,v)$ be the sum of the weights of all paths $(p_1,a_1,b_1,p_2)(p_1,a_2,b_2,p_2)\ldots(p_n,a_n,b_n,p_{n+1})$ from the state $p = p_1$ to the state $p_{n+1} = q$ that are labelled by $u = a_1 \ldots a_n$ and $v = b_1 \ldots b_n$. Moreover, we define

$$R_{\mathcal{T}}(u,v) = \sum_{p,q \in Q} i(p) A_{p,q}(u,v) t(q).$$

That is, only the paths from an initial to a final state are taken into account. In this way $R_{\mathcal{T}}$ relates some pairs $(u,v)$, namely, those for which $R_{\mathcal{T}}(u,v) \neq 0$, and assigns some weight to the relational pair $(u,v)$.

Observe that $A_{p,q}(u,v)$ does not have to be defined. Indeed, for some $p$, $q$, $u$ and $v$, it can happen that $A_{p,q}(u,v)$ is infinite. This is due to the fact that if a transition is labelled by $(a_1,a_2 : r)$, then it may happen that either $a = \varepsilon$ or $a_2 = \varepsilon$ or $a_1 = a_2 = \varepsilon$. Therefore there may be infinitely many paths between $p$ and $q$ labelled by $u$ and $v$. To overcome this problem, we restrict ourselves to those WFT which have the property that if the product of the weights of a cycle is nonzero, then either not all first labels or not all second labels on the edges of the path are $\varepsilon$.

The concept of a weighted relation may seem artificial. However, its application to functions has turned out to be a powerful tool. In image-processing applications, weighted relations represent an elegant and powerful way to transform images.

**Definition 3.5.10** *Let $\rho : \Sigma_1^* \times \Sigma_2^* \to \mathbf{R}$ be a weighted relation and $f : \Sigma_1^* \to \mathbf{R}$ a function. An application of $\rho$ on $f$, in short $g = \rho \circ f = \rho(f) : \Sigma_2^* \to \mathbf{R}$, is defined by*

$$g(v) = \sum_{u \in \Sigma_1^*} \rho(u,v) f(u),$$

*for $v \in \Sigma_2^*$, if the sum, which can be infinite, converges; otherwise $g(u)$ is undefined. (The order of summation is given by a strict ordering on $\Sigma_1^*$.)*

Informally, an application of $\rho$ on $f$ produces a new function $g$. The value of this function for an argument $v$ is obtained by taking $f$-values of all $u \in \Sigma^*$ and multiplying each $f(u)$ by the weight of the paths that stand for the pair $(u,v)$. This simply defined concept is very powerful. The concept itself, as well as its power, can best be illustrated by examples.

**Exercise 3.5.11** *Describe the image transformation defined by the WFT shown in Figure 3.20a which produces, for example, the image shown in Figure 3.20c from the image depicted in Figure 3.20b.*

**Example 3.5.12 (Derivation)** *The WFT $\mathcal{T}_3$ in Figure 3.21a defines a weighted relation $R_{\mathcal{T}_3}$ such that for any function $f : \{0,1\}^* \to \mathbf{R}$, interpreted as a function on fractions, we get*

$$R_{\mathcal{T}_3} \circ f(x) = \frac{df(x)}{dx}$$

*(and therefore $\mathcal{T}_3$ acts as a functional), in the following sense: for any fixed $n$ and any function $f : \Sigma_1^n \to \mathbf{R}$, $R_{\mathcal{T}_3} \circ f(x) = \frac{f(x+h)-f(x)}{h}$, where $h = \frac{1}{2^n}$. (This means that if $x$ is chosen to have $n$ bits, then even the least*

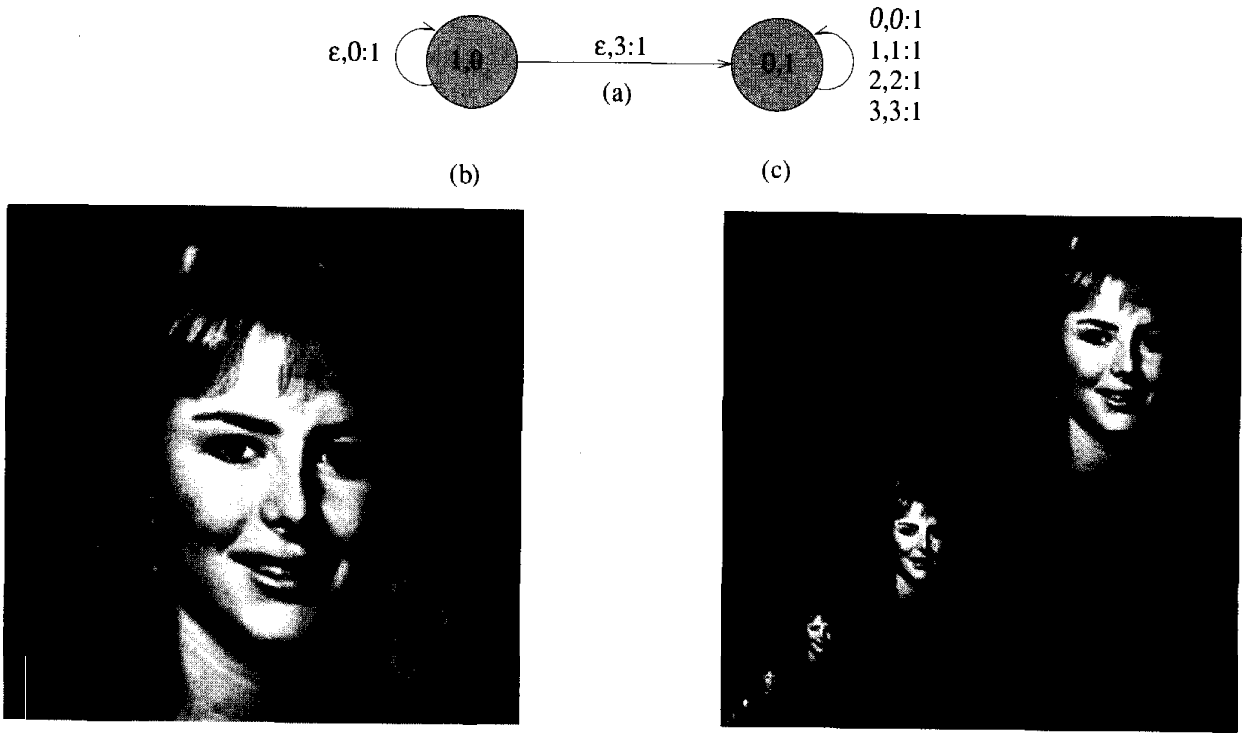



**Figure 3.20** Image transformation

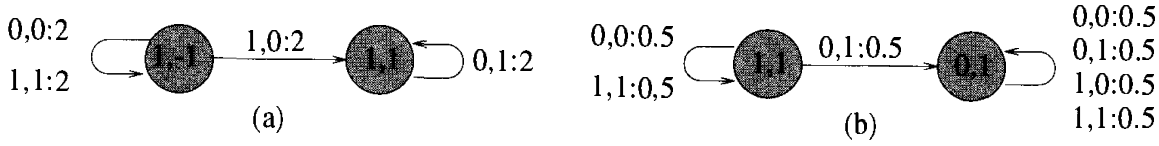



**Figure 3.21** WFT for derivation and integration

*significant* 0, *in a binary representation of* $x$, *matters.) Indeed,* $R_{\mathcal{T}_3}(x,y) \neq 0$, *for* $x,y \in \{0,1\}^*$ *if and only if either* $x = y$ *and then* $R_{\mathcal{T}_3}(x,y) = -2^{|x|}$ *or* $x = x_1 10^k$, $y = x_1 01^k$, *for some* $k$, *and in such a case* $R_{\mathcal{T}_3}(x,y) = 2^{|x|}$. *Hence* $R_{\mathcal{T}_3} \circ f(x) = R_{\mathcal{T}_3}(x,x)f(x) + R_{\mathcal{T}_3}(x + \frac{1}{2^{|x|}}, x)f(x + \frac{1}{2^{|x|}}) = -2^{|x|}f(x) + 2^{|x|}f(x + \frac{1}{2^{|x|}})$. *Take now* $n = |x|$, $h = \frac{1}{2^n}$.

**Example 3.5.13 (Integration)** *The WFT* $\mathcal{T}_4$ *in Figure 3.21b determines a weighted relation* $R_{\mathcal{T}_4}$ *such that for any function* $f : \Sigma^* \to \mathbf{R}$

$$R_{\mathcal{T}_4} \circ f(x) = \int_0^x f(t)\,dt$$

*in the following sense:* $R_{\mathcal{T}_4} \circ f$ *computes* $h(f(0) + f(h) + f(2h) + \ldots + f(x))$ *(for any fixed resolution* $h = \frac{1}{2^k}$, *for some* $k$, *and all* $x \in \{0,1\}^k$).

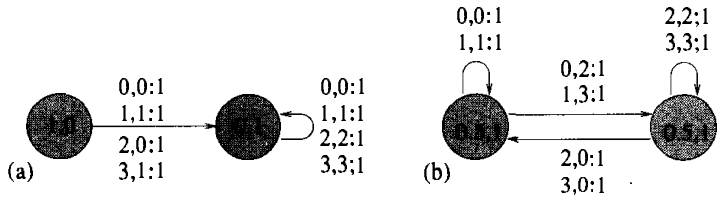



**Figure 3.22** Two WFT

**Exercise 3.5.14** *Explain in detail how the WFT in Figure 3.21b determines a functional for integration.*

**Exercise 3.5.15*** *Design a WFT for a partial derivation of functions of two variables with respect: (a) to the first variable; (b) to the second variable.*

The following theorem shows that the family of functions computed by WFA is closed under the weighted relations realized by WFT.

**Theorem 3.5.16** *Let $\mathcal{A}_1 = \langle \Sigma_1, Q_1, i_1, t_1, w_1 \rangle$ be a WFA and $\mathcal{A}_2 = \langle \Sigma_2, Q_2, i_2, t_2, w_2 \rangle$ be an $\varepsilon$-loop free WFT. Then there exists a WFA $\mathcal{A}$ such that $f_{\mathcal{A}} = R_{\mathcal{A}_2} \circ f_{\mathcal{A}_1}$.*

This result actually means that to any WFA $\mathcal{A}$ over the alphabet $\{0,1\}$ two WFA $\mathcal{A}'$ and $\mathcal{A}''$ can be designed such that for any $x \in \Sigma^*, f_{\mathcal{A}'}(x) = \frac{df_{\mathcal{A}}(x)}{dx}$ and $f_{\mathcal{A}''}(x) = \int_0^x f_{\mathcal{A}}(x)dx$.

**Exercise 3.5.17** *Construct a WFT to perform (a)* a rotation by 45 degrees clockwise; (b) a circular left shift by one pixel in two dimensions.*

**Exercise 3.5.18** *Describe the image transformations realized by WFT in: (a) Figure 3.22a; (b) Figure 3.22b.*

**Exercise 3.5.19*** *Prove Theorem 3.5.16.*

## 3.5.2 Functions Computed by WFA

For a WFA $\mathcal{A}$ over the alphabet $\{0,1\}$, the real function $f^{\omega}_{\mathcal{A}} : [0,1] \to \mathbf{R}$ does not have to be total. However, it is always total for a special type of WFT introduced in Definition 3.5.20. As will be seen later, even such simple WFT have unexpected power.

**Definition 3.5.20** *A WFA $\mathcal{A} = \langle \Sigma, Q, i, t, w \rangle$ is called a* **level weighted finite automaton** *(for short, LWFA) if*

1. *all weights are between 0 and 1;*

2. *the only cycles are self-loops;*

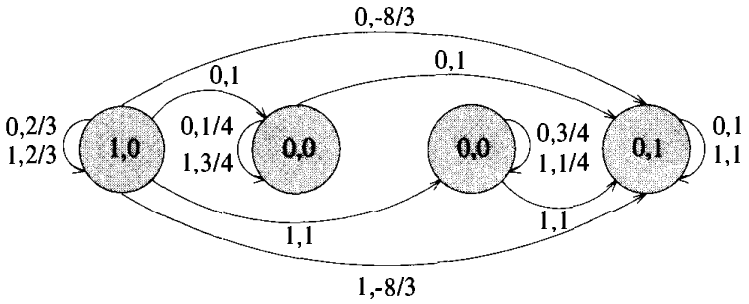



**Figure 3.23** A LWFA that computes a function that is everywhere continuous and nowhere has a derivative

3. *if the weight of a self-loop is 1, then it must be a self-loop of a node that has no other outgoing edges than self-loops.*

For example, the WFA in Figure 3.18b is a LWFA; the one in Figure 3.18a is not. LWFA have unexpected properties summarized in the following theorem.

**Theorem 3.5.21** *LWFA have the following properties:*

1. *It is decidable, given a LWFA, whether the real function it computes is continuous. It is also decidable, given two LWFA, whether the real functions they compute are identical.*
2. *Any polynomial of one variable with rational coefficients is computable by a LWFA. In addition, for any integer n there is a fixed, up to the initial distribution, LWFA $\mathcal{A}_n$ that can compute any polynomial of one variable and degree at most n. (To compute different polynomials, only different initial distributions are needed.)*
3. *If arbitrary negative weights are allowed, then there exists a simple LWFA (see Figure 3.23) computing a real function that is everywhere continuous and has no derivatives at any point of the interval* $[0,1]$.

**Exercise 3.5.22*** *Design a LWFA computing all polynomials of one variable of degree 3, and show how to fix the initial and terminal distributions to compute a particular polynomial of degree 3.*

## 3.5.3 Image Generation and Transformation by WFA and WFT

As already mentioned, an average-preserving mapping $f : P^* \to \mathbf{R}$ can be considered as a multi-resolution image. There is a simple way to ensure that a WFA on $P$ defines an average-preserving mapping and thereby a multi-resolution image.

**Definition 3.5.23** *A WFA $\mathcal{A} = \langle P, Q, i, t, w \rangle$ is average-preserving if for all $p \in Q$*

$$\sum_{a \in \Sigma, q \in Q} w(p,a,q)t(q) = 4t(p).$$

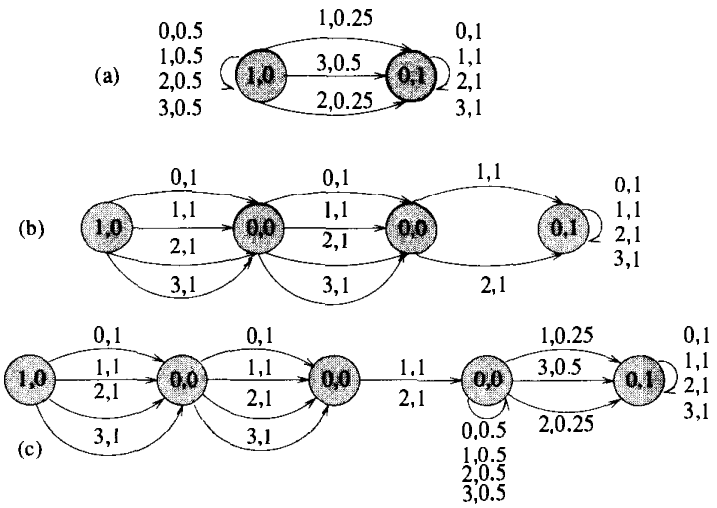



**Figure 3.24** WFA generating two images and their concatenation

Indeed, we have

**Theorem 3.5.24** *Let $\mathcal{A}$ be a WFA on $P$. If $\mathcal{A}$ is average-preserving, then so is $f_{\mathcal{A}}$.*

**Proof:** Let $u \in P^*, a \in P$. Since

$$f_{\mathcal{A}}(ua) = \sum_{q \in Q} \delta(q, ua)t(q) \tag{3.5}$$

$$= \sum_{p,q \in Q} \delta(p,u)w(p,a,q)t(q), \tag{3.6}$$

we have

$$\sum_{a \in P} f_{\mathcal{A}}(ua) = \sum_{p \in Q} \delta(p,u) \sum_{a \in P, q \in Q} w(p,a,q)t(q) \tag{3.7}$$

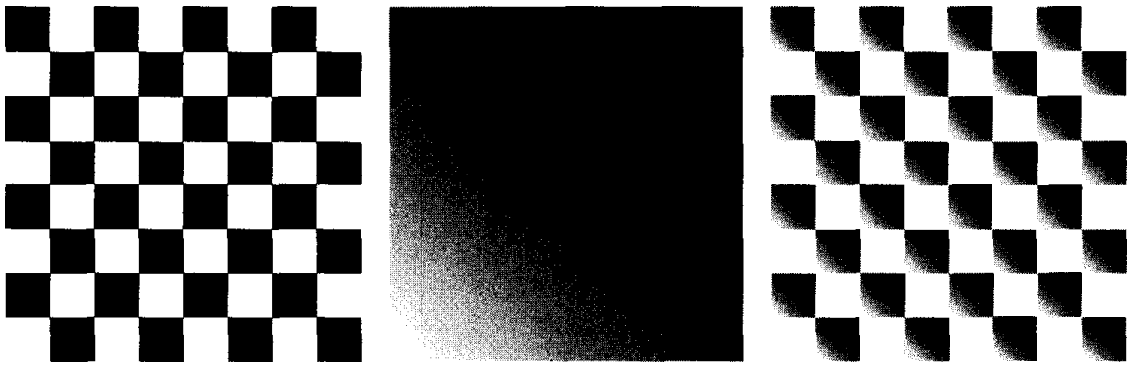
$$= \sum_{p \in Q} \delta(p,u)4t(p) = 4f_{\mathcal{A}}(u). \tag{3.8}$$

□

The family of multi-resolution images generated by a WFA is closed under various operations such as addition, multiplication by constants, Cartesian product, concatenation, iteration, various affine transformations, zooming, rotating, derivation, integration, filtering and so on. Concatenation of WFA (see also Section 2.5.3) is defined as follows.

**Definition 3.5.25** *Let $\mathcal{A}_1, \mathcal{A}_2$ be WFA over $P$ and $f_{\mathcal{A}_1}, f_{\mathcal{A}_2}$ multi-resolution images defined by $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Their concatenation $\mathcal{A}_1\mathcal{A}_2$ is defined as*

$$f_{\mathcal{A}_1\mathcal{A}_2}(u) = \sum_{u_1u_2 = u} f_{\mathcal{A}_1}(u_1)f_{\mathcal{A}_2}(u_2).$$



(a) (b) (c)

**Figure 3.25** Concatenation of two images generated by WFA

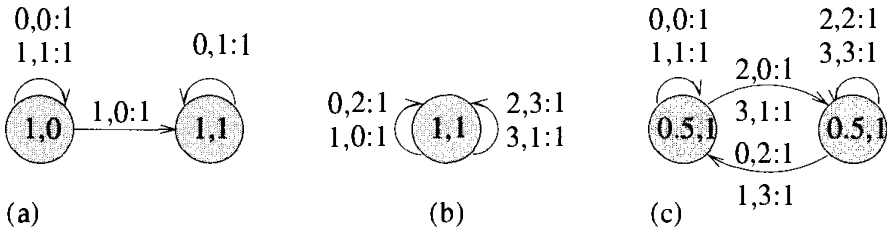



**Figure 3.26** Image transformations defined by WFT: (a) circular shift left, (b) rotation, (c) vertical squeezing

**Exercise 3.5.26** *(a) Show that the WFA in Figure 3.24b generates the chess board shown in Figure 3.25a; (b) that the WFA in Figure 3.24a generates the linear slope shown in Figure 3.25b; (c) that concatenation of the two images in Figures 3.24a, b (see the result in Figure 3.25c) generates the WFA in Figure 3.24c.*

Observe that several of the WFA we have considered, for example, the one in Figure 3.24b, are nondeterministic in the sense that if the weights are discarded, a nondeterministic FA is obtained. It can be shown that nondeterministic WFA generate more images than deterministic ones. For example, there is no deterministic WFA that generates the same linear slope as does the WFA in Figure 3.24a.

## 3.5.4 Image Compression

We have seen several examples of WFT generating images. From the application point of view, it is the inverse problem that is of special importance: given an image, how to design a WFT generating that image. Indeed, to store a multi-resolution image directly, a lot of memory is needed. A WFT generating the same image usually requires much less memory. There is a simple-to-formulate algorithm that can do image compression.

**Algorithm 3.5.27 (Image compression)** *Assume as input an image I given by a function* $\phi : P^* \to \mathbf{R}$. *(It can also be a digitalized photo.)*

1. *Assign the initial state* $q_0$ *to the image represented by the empty word, that is, to the whole image I, and define* $i(q_0) = 1, t(q_0) = \phi(\varepsilon)$, *the average greyness of the image I.*
2. *Recursively, for a state q assigned to a square specified by a string u, consider four subsquares specified by strings* $u0, u1, u2, u3$. *Denote the image in the square ua by* $I_{ua}$. *If this image is everywhere 0, then there will be no transition from the state q with the label a. If the image* $I_{ua}$ *can be expressed as a linear combination of the images* $I_{v_i}$ *corresponding to the states* $p_1, \ldots, p_k$ *– that is,* $I_{ua} = \sum_{i=1}^{k} c_i I_{v_i}$ *– add a new edge from q to each* $p_i$ *with label a and with weight* $w(q, a, p_i) = c_i (i = 1, \ldots, k)$. *Otherwise, assign a new state r to the pixel ua and define* $w(q, a, r) = 1, t(r) = \phi(I_{ua})$ *– the average greyness of the image in the pixel ua.*
3. *Repeat step 3 for each new state, and stop if no new state is created.*

Since any real image has a finite resolution, the algorithm has to stop in practice. If this algorithm is applied to the picture shown in Figure 3.19a, we get a WFA like the one shown in Figure 3.19b but with all weights equal 1. Using the above 'theoretical algorithm' a compression of 5–10 times can be obtained. However, when a more elaborate 'recursive algorithm' is used, a larger compression, 50–60 times for grey-scale images and 100–150 times for colour images (and still providing pictures of good quality), has been obtained.

Of practical importance also are WFT. They can perform most of the basic image transformations, such as changing the contrast, shifts, shrinking, rotation, vertical squeezing, zooming, filters, mixing images, creating regular patterns of images and so on.

**Exercise 3.5.28** *Show that the WFT in Figure 3.26a performs a circular shift left.*

**Exercise 3.5.29** *Show that the WFT in Figure 3.26b performs a rotation by 90 degrees counterclockwise.*

**Exercise 3.5.30** *Show that the WFT in Figure 3.26c performs vertical squeezing, defined as the sum of two affine transformations:* $x_1 = \frac{x}{2}, y_1 = y$ *and* $x_2 = \frac{x+1}{2}, y_2 = y$ *– making two copies of the original image and putting them next to each other in the unit square.*

## 3.6 Finite Automata on Infinite Words

A natural generalization of the concept of finite automata recognizing/accepting finite words and languages of finite words is that of finite automata recognizing $\omega$-words and $\omega$-languages. These concepts also have applications in many areas of computing. Many processes modelled by finite state devices (for instance, the watch in Section 3.1) are potentially infinite. Therefore it is most appropriate to see their inputs as $\omega$-words. Two types of FA play the basic role here.

### 3.6.1 Büchi and Muller Automata

**Definition 3.6.1** *A* **Büchi automaton** $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ *is formally defined exactly like a FA, but it is used only to process* $\omega$*-words, and acceptance is defined in a special way. An* $\omega$*-word* $w = w_0 w_1 w_2 \ldots \in \Sigma^\omega, w_i \in \Sigma$, *is accepted by* $\mathcal{A}$ *if there is an infinite sequence of states* $q_0, q_1, q_2, \ldots$ *such that* $(q_i, w_i, q_{i+1}) \in \delta$, *for all* $i \geq 0$,

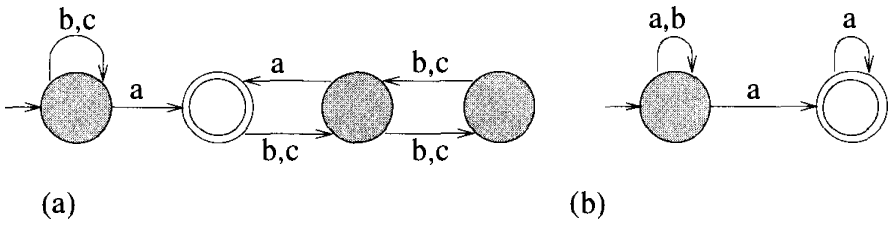



**Figure 3.27** Büchi automata

*and a state in $Q_F$ occurs infinitely often in this sequence. Let $L^\omega(\mathcal{A})$ denote the set of all $\omega$-words accepted by $\mathcal{A}$.*

An $\omega$-language $L$ is called regular if there is a Büchi automaton accepting $L$.

**Example 3.6.2** *Figure 3.27a shows a Büchi automaton accepting the $\omega$-language over the alphabet $\{a,b,c\}$ consisting of $\omega$-words that contain infinitely many a's and between any two occurrences of a there is an odd number of occurrences of b and c. Figure 3.27b shows a Büchi automaton recognizing the language $\{a,b\}^*a^\omega$.*

**Exercise 3.6.3** *Construct a Büchi automaton accepting the language $L \subseteq \{a,b,c\}^*$ defined as follows: (a) $w \in L$ if and only if after any occurrence of the symbol a there is some occurrence of the symbol b in w; (b) $w \in L$ if and only if between any two occurrences of the symbol a there is a multiple of four occurrences of b's or c's.*

The following theorem summarizes those properties of $\omega$-regular languages and Büchi automata that are similar to those of regular languages and FA. Except for the closure under complementation, they are easy to show.

**Theorem 3.6.4** *(1) The family of regular $\omega$-languages is closed under the operations of union, intersection and complementation.*

*(2) An $\omega$-language L is regular if and only if there are regular languages $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ such that $L = A_1B_1^\omega \cup \ldots \cup A_nB_n^\omega$.*

*(3) The emptiness and equivalence problems are decidable for Büchi automata.*

**Exercise 3.6.5** *Show that (a) if L is a regular language, then $L^\omega$ is a regular $\omega$-language; (b) if $L_1$ and $L_2$ are regular $\omega$-languages, then so are $L_1 \cup L_2$ and $L_1 \cap L_2$; (c)** the emptiness problem is decidable for Büchi automata.*

The result stated in point (2) of Theorem 3.6.4 shows how to define regular $\omega$- expressions in such a way that they define exactly regular $\omega$-languages.

One of the properties of FA not shared by Büchi automata concerns the power of nondeterminism. Nondeterministic Büchi automata are more powerful than deterministic ones. This follows easily from

the fact that languages accepted by deterministic Büchi automata can be nicely characterized using regular languages. To show this is the task of the next exercise.

**Exercise 3.6.6** *Show that an $\omega$-language $L \subseteq \Sigma^\omega$ is accepted by a deterministic Büchi automaton if and only if*

$$L = \overrightarrow{W} = \{w \in \Sigma^\omega \mid Prefix_n(w) \in W, \text{ for infinitely many } n\},$$

*for some regular language $W$.*

**Exercise 3.6.7*** *Show that the language $\{a,b\}^\omega - \overrightarrow{(b^*a)^*}$ is accepted by a nondeterministic Büchi automaton but not by a deterministic Büchi automaton.*

There is, however, a modification of deterministic Büchi automata, with a different acceptance mode, the so-called **Muller automata**, that are deterministic and recognize all regular $\omega$-languages.

**Definition 3.6.8** *In a Muller automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \mathcal{F}, \delta \rangle$, where $\Sigma$, $Q$, $q_0$ and $\delta$ have the same meaning as for DFA, but $\mathcal{F} \subseteq 2^Q$ is a family of sets of final states. $\mathcal{A}$ recognizes an $\omega$-word $w = w_0w_1w_2 \ldots$ if and only if the set of states that occur infinitely often in the sequence of states $\{q_i\}_{i=0}^{\infty}$, $q_i = \delta(q_0, w_0w_1w_2 \ldots w_i)$, is an element of $\mathcal{F}$. (That is, the set of those states which the automaton $\mathcal{A}$ takes infinitely often when processing $w$ is an element of $\mathcal{F}$.)*

**Exercise 3.6.9*** *Show the so-called McNaughton theorem: Muller automata accept exactly regular $\omega$-languages.*

**Exercise 3.6.10** *Show, for the regular $\omega$-language $L = \{0,1\}^*\{0\}^\omega$ (that is, not a deterministic regular $\omega$-language), that there are five non-isomorphic minimal (with respect to the number of states) Muller automata for $L$. (This indicates that the minimization problem has different features for Muller automata than it does for DFA.)*

## 3.6.2 Finite State Control of Reactive Systems*

In many areas of computing, for example, in operating systems, communication protocols, control systems, robotics and so on, the appropriate view of computation is that of a nonstop interaction between two agents or processes. They will be called **controller** and **disturber** or **plant** (see Figure 3..28). Each of them is supposed to be able to perform at each moment one of finitely many actions. Programs or automata representing such agents are called **reactive**; their actions are modelled by symbols from finite alphabets, and their continuous interactions are modelled by $\omega$-words .

In this section we illustrate, as a case study, that (regular) $\omega$-languages and $\omega$-words constitute a proper framework for stating precisely and solving satisfactorily basic problems concerning such reactive systems. A detailed treatment of the subject and methods currently being worked out is beyond the scope of this book.

A **desirable interaction** of such agents can be specified through an $\omega$-language $L \subseteq (\Sigma\Delta)^\omega$, where $\Sigma$ and $\Delta$ are disjoint alphabets. An $\omega$-word $w$ from $L$ has therefore the form $W = c_1d_1c_2d_2 \ldots$, where

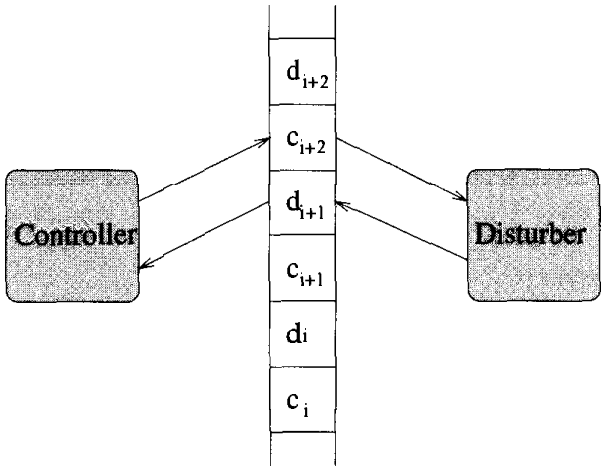



**Figure 3.28** Controller and disturber

$c_i \in \Sigma$ ($d_i \in \Delta$). The symbol $c_i$ ($d_i$) denotes the $i$th action that the controller (disturber) performs. The idea is that the controller tries to respond to the actions of the disturber in such a way that these actions make the disturber 'behave accordingly'.

Three basic problems arise when such a desirable behaviour is specified by an $\omega$-language. The **verification problem** is to decide, given a controller, whether it is able to interact with the disturber in such a way that the resulting $\omega$-word is in the given $\omega$-language. The **solvability problem** is to decide, given an $\omega$-language description, whether there exists a controller of a certain type capable of achieving an interaction with the disturber resulting always in an $\omega$-word from the given $\omega$-language. Finally, the **synthesis problem** is to design a controller from a given specification of an $\omega$-language for the desired interaction with the disturber. Interestingly enough, all these problems are solvable if the $\omega$-language specifying desirable behaviour of the controller–disturber interactions is a regular $\omega$-language.

Problems of verification and synthesis for such reactive automata can be nicely formulated, like many problems in computing, in the framework of games – in this case in the framework of the **Gale–Stewart games** of two players, who are again called **controller** ($C$) and **disturber** ($D$). Their actions are modelled by symbols from alphabets $\Sigma_C$ and $\Sigma_D$, respectively. Let $\Sigma = \Sigma_C \cup \Sigma_D$.

A **Gale–Stewart game** is specified by an $\omega$-language $L \subseteq (\Sigma_C \Sigma_D)^\omega$. A **play** of the game is an $\omega$-word $p \in \Sigma_C(\Sigma_D\Sigma_C)^\omega$. (An interpretation is that $C$ starts an interaction by choosing a symbol from $\Sigma_C$, and then $D$ and $C$ keep choosing, in turn and indefinitely, symbols from their alphabets (depending, of course, on the interactions to that moment).) Player $C$ **wins** the play $p$ if $p \in L$, otherwise $D$ wins. A **strategy** for $C$ is a mapping $s_C : \Sigma_D^* \to \Sigma_C$ specifying a choice of a symbol from $\Sigma_C$ (a move of $C$) for any finite sequence of choices of symbols by $D$ – moves of $D$ to that moment. Any such strategy determines a mapping $\bar{s}_C : \Sigma_D^\omega \to \Sigma_C^\omega$, defined by

$$\bar{s}_C(d_0d_1d_2\ldots) = c_0c_1c_2\ldots, \text{ where } c_i = s_C(d_0d_1\ldots d_{i-1}).$$

If $D$ chooses an infinite sequence $\mu = d_0d_1\ldots$ of events (symbols) to act and $C$ has a strategy $s_C$, then $C$ chooses the infinite sequence $\gamma = \bar{s}_C(\mu)$ to create, together with $D$, the play $p_{\mu,\bar{s}_C} = c_0d_0c_1d_1\ldots$.

The main problem, the **uniform synthesis problem**, can now be described as follows. Given a specification language for a class $\mathcal{L}$ of $\omega$-languages, design an algorithm, if it exists, such that, given any specification of an $\omega$-language $L \in \mathcal{L}$, the algorithm designs a (winning) strategy $s_C$ for $C$ such

that no matter what strategy $D$ chooses, that is, no matter which sequence $\mu$ disturber $D$ chooses, the play $p_{\mu,\bar{s}_C}$ will be in $L$.

In general the following theorem holds.

**Theorem 3.6.11 (Büchi–Landweber's theorem)** *Let $\Sigma_C, \Sigma_D$ be finite alphabets. To any $\omega$-regular language $L \subseteq \Sigma_C(\Sigma_D\Sigma_C)^\omega$ and any Muller automaton recognizing $L$, a Moore machine $\mathcal{A}_L$ with $\Sigma_D$ as the input alphabet and $\Sigma_C$ as the output alphabet can be constructed such that $\mathcal{A}_L$ provides the winning strategy for the controller with respect to the language $L$.*

The proof is quite involved. Moreover, this result has the drawback that the resulting Moore machine may have superexponentially more states than the Muller automaton defining the game. The problem of designing a winning strategy for various types of behaviours is being intensively investigated.

## 3.7 Limitations of Finite State Machines

Once a machine model has been designed and its advantages demonstrated, an additional important task is to determine its limitations.

In general it is not easy to show that a problem is not within the limits of a machine model. However, for finite state machines, especially finite automata, there are several simple and quite powerful methods for showing that a language is not regular. We illustrate some of them.

**Example 3.7.1 (Proof of nonregularity of languages using Nerode's theorem)** *For the language $L_1 = \{a^i b^i \mid i \geq 0\}$ it clearly holds that $a^i \not\equiv_{L_1} a^j$ if $i \neq j$. This implies that the syntactical monoid for the language $L_1$ is infinite. $L_1$ is therefore not recognizable by a FA.*

**Example 3.7.2 (Proof of nonregularity of languages using the pumping lemma)** *Let us assume that the language $L_2 = \{a^p \mid p \text{ prime}\}$ is regular. By the pumping lemma for regular languages, there exist integers $x, y, z, x + z \neq 0, y \neq 0$, such that all words $a^{x+iy+z}, i \geq 0$, are in $L_2$. However, this is impossible because, for example, $x + iy + z$ is not prime for $i = x + z$.*

**Example 3.7.3 (Proof of nonregularity of languages using a descriptional finiteness argument)** *Let us assume that the language $L_3 = \{a^i c b^i \mid i \geq 1\}$ is regular and that $\mathcal{A}$ is a DFA recognizing $L_3$. Clearly, for any state $q$ of $\mathcal{A}$, there is at most one $i \in \mathbf{N}$ such that $b^i \in L(q)$. If such an $i$ exists, we say that $q$ specifies that $i$. Since $a^i c b^i \in L_3$ for each $i$, for any integer $j$ there must exist a state $q_j$ (the one reachable after the input $a^j c$) that specifies $j$. A contradiction, because there are only finitely many states in $\mathcal{A}$.*

**Exercise 3.7.4** *Show that the following languages are not regular: $\{a^i b^{2i} \mid i \geq 0\}$; (b) $\{a^i \mid i \text{ is composite}\}$; (c) $\{a^i \mid i \text{ is a Fibonacci number}\}$; (d) $\{w \in \{0,1\}^* \mid w = w^R\}$.*

**Example 3.7.5** *We now show that neither a Moore nor a Mealy machine can multiply two arbitrary binary integers given the corresponding pairs of bits as the input as in the case of binary adders in Figure 3.17. (To be consistent with the model in Figure 3.17, we assume that if the largest number has $n$ bits, then the most significant pair of bits is followed by additional $n$ pairs $(0,0)$ on the input.)*

*If the numbers $x$ and $y$ to be multiplied are both equal to $2^{2m}$, the $2m+1$-th input symbol will be $(1,1)$ and all others are $(0,0)$. After reading the $(1,1)$ symbol, the machine still has to perform $2m$ steps before producing*

*a* 1 *on the output. However, this is impossible, because during these* $2m$ *steps* $\mathcal{M}$ *has to get into a cycle. (It has only* $m$ *states, and all inputs after the input symbol* $(1,1)$ *are the same –* $(0,0)$*.) This means that either* $\mathcal{M}$ *produces a* 1 *before the* $(4m+1)$*-th step or* $\mathcal{M}$ *never produces a* 1*. But this is a contradiction to the assumption that such a machine exists.*

**Exercise 3.7.6** *Show that there is no finite state machine to compute the function (a)* $f_1(n) =$ *the n-th Fibonacci number; (b)* $f(0^n1^m) = 1^{n \bmod m}$.

**Example 3.7.7** *It follows from Theorem 3.4.11 that no generalized sequential machine can compute the function* $f : \{0,1\}^* \to \{0,1\}^*$ *defined by* $f(w) = w^R$*. Indeed, the prefix condition from that theorem is not fulfilled.*

**Example 3.7.8** *Let* $L \subset \{0,1\}^\omega$ *be a language of* $\omega$*-words* $w$ *for which there is an integer* $k > 1$ *such that* $w$ *has a symbol* 1 *exactly in the positions* $k^n$ *for all integers* $n$*. We claim that* $L$ *is not a regular* $\omega$*-language. Indeed, since the distances between two consecutive* 1*s are getting bigger and bigger, a finite automaton cannot check whether they are correct.*

Concerning weighted finite transducers it has been shown that they can compute neither exponential functions nor trigonometric functions.

## 3.8 From Finite Automata to Universal Computers

Several natural ideas for enhancing the power of finite automata will now be explored. Surprisingly, some of these ideas do not lead to an increase in the computational power of finite automata at all. Some of them, also surprisingly, lead to very large increases. All these models have one thing in common. The only memory they need to process an input is the memory needed to store the input. One of these models illustrates an important new mode of computation – probabilistic finite automata. The importance of others lies mainly in the fact that they can be used to represent, in an isolated form, various techniques for designing of Turing machines, discussed in the next chapter.

### 3.8.1 Transition Systems

A **transition system** $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ is defined similarly to a finite automaton, except that the finite transition relation $\delta$ is a subset of $Q \times \Sigma^* \times Q$ and not of $Q \times \Sigma \times Q$ as for finite automata. In other words, in a transition system, a longer portion of an input word only can cause a single state transition. Computation and acceptance are defined for transition systems in the same way as for finite automata: namely, an input word $w$ is accepted if there is a path from the initial state to a final state labelled by $w$.

Each finite automaton is a transition system. On the other hand, to each transition system $\mathcal{A}$ it is easy to design an equivalent FA which accepts the same language. To show this, we sketch a way to modify the state graph $G_{\mathcal{A}}$ of a transition system $\mathcal{A}$ in order to get a state graph of an equivalent FA.

1. Replace each transition (edge) $p \underset{w}{\Longrightarrow} q, w = w_1w_2 \ldots w_k$, $w_i \in \Sigma$, $k > 1$ by $k$ transitions $p \underset{w_1}{\Longrightarrow} p_1 \underset{w_2}{\Longrightarrow} p_2 \cdots p_{k-2} \underset{w_{k-1}}{\Longrightarrow} p_{k-1} \underset{w_k}{\Longrightarrow} q$, where $p_1, \ldots, p_{k-1}$ are newly created states (see the step from Figure 3.29a to 3.29b).

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | c | e | g | i | k | m | o | r | t | v | |
| | b | d | f | h | j | l | n | p | s | u | w | |

(a) one tape with two tracks

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | c | e | g | i | k | m | o | r | t | v | |
| | b | d | f | h | j | l | n | p | s | u | w | |

(b) one tape with one track

**Figure 3.29** Derivation of a complete FA from a transition system

2. Remove $\varepsilon$-transitions. This is a slightly more involved task. One needs first to compute the transitive closure of the relation $\underset{\varepsilon}{\Longrightarrow}$ between states. Then for any triple of states $p,q,q'$ and each $a \in \Sigma$ such that $p \overset{*}{\underset{\varepsilon}{\Longrightarrow}} q \underset{a}{\Longrightarrow} q'$, the transition $p \underset{a}{\Longrightarrow} q'$ is added. If, after such modifications, $q' \overset{*}{\underset{\varepsilon}{\Longrightarrow}} q$ for some $q' \in Q$ and $q \in Q_F$, add $q'$ to the set of final states, and remove all $\varepsilon$-transitions and unreachable states (see the step from Figure 3.29b to 3.29c).
3. If we require the resulting automaton to be complete, we add a new 'sink state' to which all missing transitions are added and directed (see the step from Figure 3.29c to 3.29d). By this construction we have shown the following theorem.

**Theorem 3.8.1** *The family of languages accepted by transition systems is exactly the family of regular languages.*

The main advantage of transition systems is that they may have much shorter descriptions and smaller numbers of states than any equivalent FA. Indeed, for any integer $n$ a FA accepting the one-word language $\{a^n\}$ must have $n-1$ states, but there is a two-state transition system that can do it.

**Exercise 3.8.2** *Design a transition system with as few states as possible that accepts those words over the alphabet $\{a,b,c\}$ that either begin or end with the string 'baac', or contain the substring 'abca'. Then use the above method to design an equivalent FA.*

**Exercise 3.8.3** *Design a minimal, with respect to number of states, transition system accepting the language $L = (a^4b^3)^* \cup (a^4b^6)^*$. Then transform its state graph to get a state graph for a FA accepting the same language.*

## 3.8.2 Probabilistic Finite Automata

We have mentioned already the power of randomization. We now explore how much randomization can increase the power of finite automata.

**Definition 3.8.4** *A* **probabilistic finite automaton** $\mathcal{P} = \langle \Sigma, Q, q_0, Q_F, \phi \rangle$ *has an* **input alphabet** $\Sigma$, *a* **set of states** $Q$, *the* **initial state** $q_0$, *a set of* **final states** $Q_F$ *and a* **probability distribution mapping** $\phi$ *that assigns to each $a \in \Sigma$ a $|Q| \times |Q|$ matrix $M_a$ of nonnegative reals with rows and columns of each $M_a$ labelled by states and such that $\sum_{q \in Q} M_a(p,q) = 1$ for any $a \in \Sigma$ and $p \in Q$. Informally, $M_a(p,q)$ determines the probability that the automaton $\mathcal{P}$ goes, under the input $a$, from state $p$ to state $q$; $M_a(p,q) = 0$ means that there is no transition from $p$ to $q$ under the input $a$.*

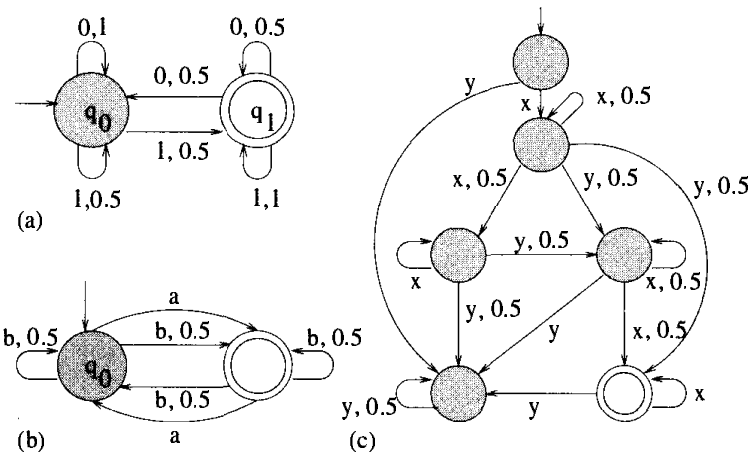



**Figure 3.30** Probabilistic finite automata – missing probabilities are 1

*If $w = w_1 \ldots w_n$, $w_i \in \Sigma$, then the entry $M_w(p,q)$ of the matrix $M_w = M_{w_1} M_{w_2} \ldots M_{w_n}$ is exactly the probability that $\mathcal{P}$ goes, under the input word $w$, from state $p$ to state $q$. Finally, for a $w \in \Sigma^*$, we define*

$$Pr_{\mathcal{P}}(w) = \sum_{q \in Q_F} M_w(q_0, q).$$

*$Pr_{\mathcal{P}}(w)$ is the probability with which $\mathcal{P}$ recognizes $w$.*

There are several ways to define acceptance by a probabilistic finite automaton. The most basic one is very obvious. It is called **acceptance with respect to a cut-point**. For a real number $0 \leq c < 1$ we define a language

$$L_c(\mathcal{P}) = \{u \mid Pr_{\mathcal{P}}(u) > c\}.$$

The language $L_c(\mathcal{P})$ is said to be the language recognized by $\mathcal{P}$ with respect to the **cut-point** c. (Informally, $L_c(\mathcal{P})$ is the set of input strings that can be accepted with a probability larger than $c$.)

**Example 3.8.5** *Let* $\Sigma = \{0,1\}$, $Q = \{q_0, q_1\}$, $Q_F = \{q_1\}$,

$$M_0 = \begin{vmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \end{vmatrix}, \qquad M_1 = \begin{vmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{vmatrix}.$$

Figure 3.30a shows the corresponding probabilistic finite automaton $\mathcal{P}_0$. Each edge is labelled by an input symbol and by the probability that the corresponding transition takes place. By induction it can easily be shown that for any $w = w_1 \ldots w_n \in \Sigma^n$, the matrix $M_w = M_{w_1} M_{w_2} \ldots M_{w_n}$ has in the right upper corner the number $0.w_n \ldots w_1$, expressed in binary notation. (Show that!)

**Exercise 3.8.6** *Determine, for all possible c, the language accepted by the probabilistic automaton in Figure 3.30b with respect to the cut-point c.*

**Exercise 3.8.7** *Determine the language accepted by the probabilistic automaton in Figure 3.30c with respect to the cut-point 0.5. (Don't be surprised if you get a nonregular language.)*

First we show that with this general concept of acceptance with respect to a cut-point, the probabilistic finite automata are more powerful than ordinary FA.

**Theorem 3.8.8** *For the probabilistic finite automaton $\mathcal{P}_0$ in Example 3.8.5 there exists a real $0 \leq c < 1$ such that the language $L_c(\mathcal{P}_0)$ is not regular.*

**Proof:** If $w = w_1 \ldots w_n$, then (as already mentioned above) $Pr_{\mathcal{P}_0}(w) = 0.w_n \ldots w_1$ (because $q_1$ is the single final state). This implies that if $0 \leq c_1 < c_2 < 1$ are arbitrary reals, then $L_{c_1}(\mathcal{P}_0) \subsetneq L_{c_2}(\mathcal{P}_0)$. The family of languages that $\mathcal{P}_0$ recognizes, with different cut-points, is therefore not countable. On the other hand, the set of regular expressions over $\Sigma$ is countable, and so therefore is the set of regular languages over $\Sigma$. Hence there exists an $0 \leq c < 1$ such that $L_c(\mathcal{P}_0)$ is not a regular language. □

The situation is different, however, for acceptance with respect to **isolated cut-points**. A real $0 \leq c < 1$ is an isolated cut-point with respect to a probabilistic FA $\mathcal{P}$ if there is a $\delta > 0$ such that for all $w \in \Sigma^*$

$$|Pr_{\mathcal{P}}(w) - c| > \delta. \tag{3.9}$$

**Theorem 3.8.9** *If $\mathcal{P} = \langle \Sigma, Q, q_0, Q_F, \phi \rangle$ is a probabilistic FA with $c$ as an isolated cut-point, then the language $L_c(\mathcal{P})$ is regular.*

To prove the theorem we shall use the following combinatorial lemma.

**Lemma 3.8.10** *Let $P_n$ be the set of all $n$-dimensional random vectors, that is, $P_n = \{x = (x_1, \ldots, x_n), x_i \geq 0, 1 \leq i \leq n$ - and $\sum_{i=1}^n x_i = 1\}$. Let, for an $\varepsilon > 0$, $U_\varepsilon$ be such a subset of $P_n$ that for any $x, y \in U_\varepsilon$, $x \neq y$ implies $\sum_{i=1}^n |x_i - y_i| \geq \varepsilon$. Then the set $U_\varepsilon$ contains at most $(1 + \frac{2}{\varepsilon})^{n-1}$ vectors.*

**Proof of the theorem:** Assume that $Q = \{q_0, q_1, \ldots, q_{n-1}\}$ and, for simplicity and without loss of generality, that $Q_F = \{q_{n-1}\}$. In this case the probability that $\mathcal{P}$ accepts some $w$ is $Pr_{\mathcal{P}}(w) = M_w(q_0, q_{n-1})$, where $M_w$ is an $n \times n$ matrix defined as on page 198.

Consider now the language $L = L_c(\mathcal{P})$, and assume that we have a set of $k$ words $v_1, \ldots, v_k$ such that no two of them are in the same prefix equivalence class with respect to the relation $\equiv_L^p$. This implies, by the definition of prefix equivalence, that for each pair $i \neq j, 1 \leq i, j \leq k$ there exists a word $y_{ij}$ such that $v_i y_{ij} \in L$ and $v_j y_{ij} \notin L$ - or vice versa.

Now let $(s_1^i, \ldots, s_n^i)$, $1 \leq i \leq k$, be the first row of the matrix $M_{v_i}$, and let $(r_1^{ij}, \ldots, r_n^{ij})$ be the last column of the matrix $M_{y_{ij}}$. Since $M_{v_i y_{ij}} = M_{v_i} M_{y_{ij}}$ and $q_{n-1}$ is the only accepting state, we get $Pr_{\mathcal{P}}(v_i y_{ij}) = s_1^i r_1^{ij} + \ldots + s_n^i r_n^{ij}$ and $Pr_{\mathcal{P}}(v_j y^{ij}) = s_1^j r_1^{ij} + \ldots + s_n^j r_n^{ij}$, and therefore

$$c < s_1^i r_1^{ij} + \ldots + s_n^i r_n^{i,j} \quad \text{and} \quad s_1^j r_1^{ij} + \ldots + s_n^j r_n^{ij} \leq c.$$

If we now use the inequality (3.9), we get

$$\sum_{l=1}^{n} (s_l^i - s_l^j) r_l^{ij} \geq 2\delta. \tag{3.10}$$

In addition, it holds that

$$\begin{aligned} \textstyle\sum_{l=1}^n (s_l^i - s_l^j) r_l^{ij} &\leq (\textstyle\sum_{l=1}^n (s_l^i - s_l^j))^+ \max\{r_l^{ij} \mid 1 \leq l \leq k\} \\ &\quad + (\textstyle\sum_{l=1}^n (s_l^i - s_l^j))^- \min\{r_l^{ij} \mid 1 \leq l \leq k\} \\ &= (\textstyle\sum_{l=1}^n (s_l^i - s_l^j))^+ (\max\{r_l^{ij} \mid 1 \leq l \leq k\} - \min\{r_l^{ij} \mid 1 \leq l \leq k\}) \\ &\leq (\textstyle\sum_{l=1}^n (s_l^i - s_l^j))^+ = \frac{1}{2} \textstyle\sum_{l=1}^n |s_l^i - s_l^j|, \end{aligned}$$

where $(\ldots)^+$ denotes that only the positive numbers in the expression inside the parentheses are taken and, similarly, $(\ldots)^-$ denotes taking only the negative numbers. In deriving these inequalities we have used essentially the fact that $|r_l^{ij}| \leq 1$ for all $l, i, j$.

A combination of the last inequality with the inequality 3.10 yields $\sum_{l=1}^{n} |s_l^i - s_l^j| \geq 4\delta$. An application of Lemma 3.8.10 then gives $k \leq \left(1 + \frac{1}{2\delta}\right)^{n-1}$. Now we can use Myhill's theorem to show that the language $L_c(\mathcal{P})$ must be regular. □

It has been shown that from the point of view of randomized computations, acceptance with respect to an isolated cut-point is very natural. Theorem 3.8.9 is therefore often seen as the main theorem showing the power of probabilistic finite automata. Unfortunately, it is still an open question whether it is decidable, given a probabilistic finite automaton $\mathcal{P}$ with rational probabilities of transitions and a rational $\lambda$, if $\lambda$ is an isolated cut-point of $\mathcal{P}$.

**Exercise 3.8.11*** *A cut-point $\lambda$ is weakly isolated for a probabilistic finite automaton $\mathcal{P}$ if $|Pr_{\mathcal{P}}(w) - \lambda| \geq \varepsilon$ or $Pr_{\mathcal{P}}(w) = \lambda$ for all $w \in \Sigma^*$ and some fixed $\varepsilon$. Prove that if $\lambda$ is a weakly isolated cut-point for $\mathcal{P}$, then the language $L_\lambda(\mathcal{P})$ is regular.*

**Exercise 3.8.12**** *Two probabilistic finite automata $\mathcal{P}_1$ and $\mathcal{P}_2$ are called mutually isolated if $|Pr_{\mathcal{P}_1}(w) - Pr_{\mathcal{P}_2}(w)| \geq \varepsilon$ for all $w \in \Sigma^*$. Prove that if $\mathcal{P}_1$ and $\mathcal{P}_2$ are mutually isolated, then the language $L = \{w \mid Pr_{\mathcal{P}_1}(w) > Pr_{\mathcal{P}_2}(w)\}$ is regular.*

The concept of a probabilistic finite automaton is usually generalized. Instead of a fixed initial state an initial distribution of states is considered, that is, each state is an initial state of a given probability. In order to get the overall probability that a word is accepted, the probability of each path has to be multiplied by the probability that its starting state is initial. Languages accepted by such probabilistic finite automata with respect to a cut-point $c$ are called $c$-stochastic. A language is called (finite state) **stochastic** if there is a probabilistic finite automaton $\mathcal{A}$ and a cut-point $c$ such that $L = L_c(\mathcal{A})$.

**Exercise 3.8.13** *Show that any regular language is $c$-stochastic for any cut-point $0 \leq c \leq 1$.*

**Exercise 3.8.14** *Show that every 0-stochastic language is regular.*

Of special interest are probabilistic finite automata with uniform probability distributions of transitions – for each state $q$ and each input symbol $a$ all transitions from $q$ under $a$ have the same probability. Such probabilistic automata are formally defined exactly like nondeterministic automata; it is therefore natural to ask what is the difference between them. Actually, it is a very big one. Nondeterministic automata are very convenient to deal with, but are completely unrealistic models of computations. By contrast, probabilistic finite automata are very realistic models of computation.

Another way to regard probabilistic finite automata, often very useful for applications, is as defining a probability distribution on the set of inputs.

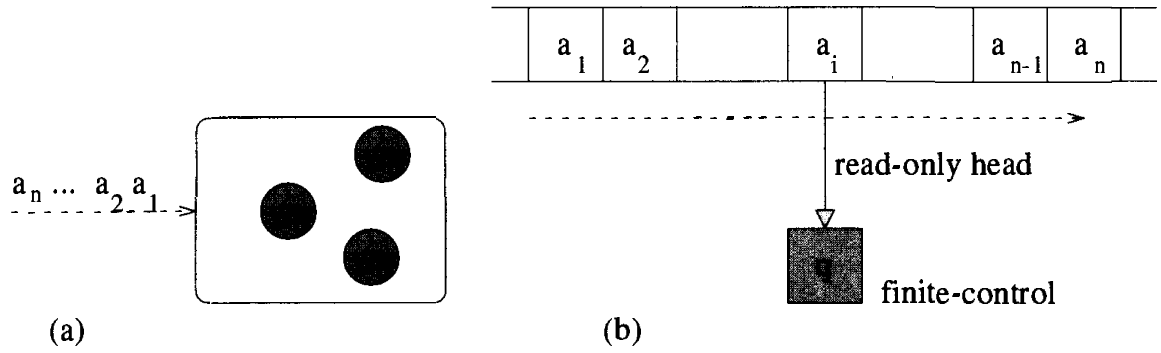



**Figure 3.31** Two ways to see a FA

### 3.8.3 Two-way Finite Automata

In addition to the usual view of a FA as a finite state device (see Figure 3.31a) which processes an external input string symbol by a symbol, there is another view (see Figure 3.31b) that is the basis of several natural generalizations of FA.

As illustrated in Figure 3.31b, a (one-way) deterministic finite automaton $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ can be seen as consisting of a **finite control** (determined by $\delta$) that is always in one of the states in the set $Q$, a **bi-infinite tape** each cell of which may contain a symbol from $\Sigma$ or a special blank symbol $\sqcup$ not in $\Sigma$, and a **read-only head** that always stays on a cell of the tape and can move only to the right, one cell per move (we refer to the two directions on the tape as left and right).

At the beginning of a computation the input word $w$ is written, one symbol per cell, in $|w|$ consecutive cells, and the head is positioned on the cell with the first symbol of $w$. In each computational step $\mathcal{A}$ reads the symbol from the cell the head is on at that moment. Depending on the state of the finite control, $\mathcal{A}$ goes, according to the transition function of the finite control $\delta$, to a new state $q$, and also moves its head to the next cell. If $\mathcal{A}$ reaches a final state at the moment when the head moves over the right end of the input word, then the input word is considered as accepted. In a similar way, nondeterministic finite automata can be defined.

With such a model of a finite automaton the question naturally arises as to whether a more powerful device could be obtained if the head were allowed to move also to the left. Let us explore this idea.

**Definition 3.8.15** *A* **two-way finite automaton** $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ *is defined similarly to an ordinary FA, except that the transition function has the form*

$$\delta : Q \times \Sigma \to Q \times \{\leftarrow, \downarrow, \rightarrow\}$$

*and* $\delta(p,a) = (q,d)$, *where* $d \in \{\leftarrow, \downarrow, \rightarrow\}$, *means that the automaton* $\mathcal{A}$ *in the state* $p$ *moves, under the input* $a$, *to the state* $q$ *and the head moves one cell in the direction* $d$ – *that is, to the right if* $d = \rightarrow$, *to the left if* $d = \leftarrow$, *and does not move at all if* $d = \downarrow$. *The language* $L(\mathcal{A})$, *accepted by* $\mathcal{A}$, *is then defined as follows:*

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ starting with the head on the first symbol of } w \text{ and in the state } q_0 \text{ moves, after a finite number of steps, over the right end of } w \text{ exactly when } \mathcal{A} \text{ comes to a final state}\}.$$

Nondeterministic two-way finite automata can be defined similarly.

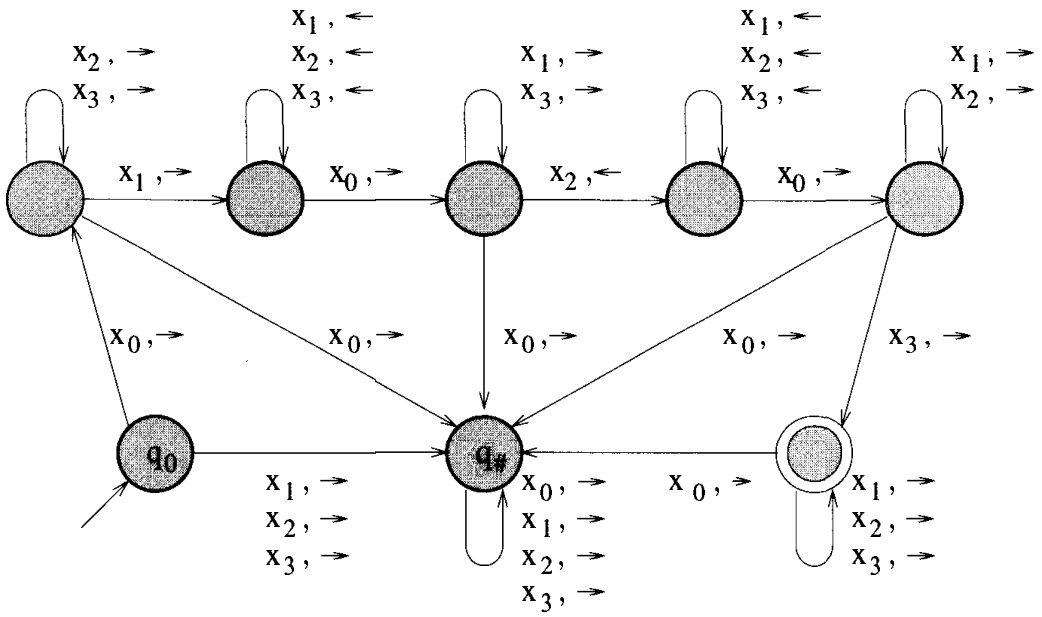



**Figure 3.32** An example of a two-way finite automaton

**Example 3.8.16** *The two-way finite automaton $\mathcal{A}_0$ in Figure 3.32 recognizes the language*

$$L = x_0(\Sigma - \{x_0\})^* \cap \Sigma^* x_1 \Sigma^* \cap \Sigma^* x_2 \Sigma^* \cap \Sigma^* x_3 \Sigma^*$$

*of all words over the alphabet $\Sigma = \{x_0, x_1, x_2, x_3\}$ that begin with $x_0$, do not contain another occurrence of $x_0$ and contain all remaining symbols from $\Sigma$ at least once. Indeed, $\mathcal{A}_0$ first verifies whether the first symbol really is $x_0$. If so, $\mathcal{A}_0$ starts a move to the right to search for an $x_1$. If found, $\mathcal{A}_0$ moves back to the left end and starts a search for an $x_2$. If found, $\mathcal{A}_0$ again moves to the left-most end and starts a search for an $x_3$. If $x_0$ is found twice, or the first symbol is not $x_0$, then $\mathcal{A}_0$ moves to the 'sink' state $q_\#$.*

Two-way finite automata appear to be more powerful than FA. This, however, is misleading.

**Theorem 3.8.17** *Nondeterministic two-way finite automata accept exactly the regular languages.*

**Proof:** We prove the lemma only for the deterministic case. For the nondeterministic case the idea of proof is the same but details are more technical.

The only way a prefix $p$ of an input word $w$ of a two-way FA $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ can influence the behaviour of $\mathcal{A}$ when $\mathcal{A}$ is no longer reading $p$ is through state transitions of $\mathcal{A}$ which $p$ causes. Indeed, the external effect of $p$ is completely determined by a function $T_p : Q \cup \{\#\} \to Q \cup \{\#\}$, that gives, for each state $q \in Q$ in which $\mathcal{A}$ re-enters $p$, the state $\mathcal{A}$ has when leaving $p$ through its rightmost symbol for the next time, or the symbol # if $\mathcal{A}$ leaves $p$ at its leftmost symbol or does not leave it at all. Moreover $T_p(\#)$ is the state in which $\mathcal{A}$ leaves for the first time the rightmost symbol of $p$ when starting on the leftmost symbol in the starting state. The relation $w_1 \equiv w_2$ if and only if $T_{w_1} = T_{w_2}$ is finer than the prefix equivalence for $L(\mathcal{A})$. The number of functions $T_w$, $w \in \Sigma^*$, is finite (actually at most $(|Q|+1)^{|Q|+1}$). Therefore, by Nerode's Theorem, $L(\mathcal{A})$ is a regular language.

Because there are only finitely many of such functions possible for $\mathcal{A}$ and, in addition, from a table $T_p$, transitions of $\mathcal{A}$ and a tape symbol $a$ of $\mathcal{A}$, one can construct table for $T_{pa}$, we can show that there exists a one-way FA $\mathcal{A}'$ that accepts the same language as $\mathcal{A}$. Indeed, $\mathcal{A}'$ will be such that after

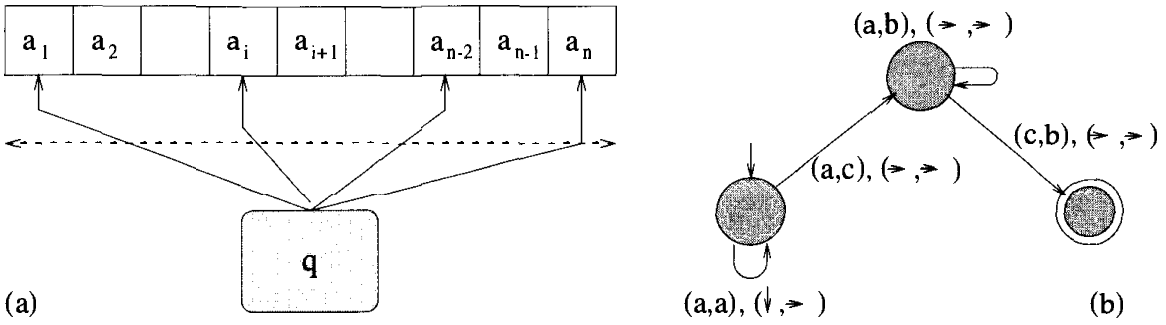



**Figure 3.33** Multi-head finite automata

finishing reading $p$ in a state $(q, T_p)$, where $q$ is the state $\mathcal{A}$ is after the first time $\mathcal{A}$ leaves $p$ and $T_p$ is the corresponding transition table for the prefix $p$. It is now obvious that given the transition function of $\mathcal{A}$ one can easily construct the transition function of $\mathcal{A}'$. If we now define that $\mathcal{A}'$ accepts an input $w$ if and only if $\mathcal{A}'$, after reading $w$, comes into a state $(q', T_w)$, where $q'$ is a final state of $\mathcal{A}$, then $\mathcal{A}$ and $\mathcal{A}'$ accept the same language. □

Informally, the theorem actually says that **multiple readings do not help if writing is not allowed and the machine has only finite memory.**

On the other hand, a two-way finite automaton can be much smaller than an equivalent FA.

**Example 3.8.18** *Let $\Sigma = \{a_0, a_1, \ldots, a_n\}$ and let $L_0$ be the set of all words over $\Sigma$ that start with the symbol $a_0$, do not contain any other occurrence of $a_0$, and contain each of the remaining symbols from $\Sigma$ at least once. Similarly, as in Example 3.8.16, a two-way finite automaton with $2n+2$ states can be constructed to accept $L_0$. On the other hand, it can be shown that any finite automaton recognizing $L_0$ must have at least $2^n$ states.*

One can show that even larger savings in description length can be achieved by using two-way finite automata compared with ordinary FA. It holds, for example, that

$$Economy_{2DFA}^{DFA}(n) = \Omega((n/5)^{n/5}).$$

where the economy function for replacing a DFA with an equivalent 2DFA is defined as on page 163.

**Exercise 3.8.19** *Show, given an integer $n$, how to design a two-way finite automaton accepting the language*

$$L_n = \{10^{i_1}10^{i_2}1 \ldots 10^{i_n}2^k0^{i_k} \mid 1 \leq k \leq n, 1 \leq i_j \leq n, 1 \leq j \leq n\}.$$

## 3.8.4 Multi-head Finite Automata

Another natural idea for enhancing the power of finite automata is to admit a sort of parallelism by allowing the use of several read-only heads (see Figure 3.33a). Two types of multi-head finite automata are obvious: one-way, heads move in one direction only, two-way, heads can move in both directions.

Informally, a $k$-head two-way finite automaton (for short, $k$-2FA) has $k$ heads, and at the beginning of any computation all heads stay on the cell with the first symbol of the input word. Each computation

step is uniquely determined by the current state of the automaton and by the symbols the heads read. A step consists of a state change and moves of heads as specified by the transition function.

More formally, in a $k$-head two-way finite automaton $\mathcal{A} = \langle \Sigma, Q, q_0, Q_F, \delta \rangle$ the symbols $\Sigma, Q, q_0, Q_F$ have the usual meaning, and

$$\delta : Q \times \underbrace{\Sigma \times \ldots \times \Sigma}_{k} \to Q \times \underbrace{\{\leftarrow, \downarrow, \rightarrow\} \times \ldots \times \{\leftarrow, \downarrow, \rightarrow\}}_{k},$$

where $\delta(p, b_1, \ldots, b_k) = (q, d_1, \ldots, d_k)$ means that if $\mathcal{A}$ is in state $p$ and the $i$th head reads $b_i$, for $1 \leq i \leq k$, then $\mathcal{A}$ goes to state $q$ and the $j$th head moves in the direction determined by $d_i$. Nondeterministic $k$-head two-way finite automata are defined similarly, as are one-way multi-head FA.

The language $L(\mathcal{A})$ is defined as the set of words $w$ such that if $\mathcal{A}$ starts with all heads on the first symbol of $w$, then, after some number of steps, $\mathcal{A}$ moves to a final state exactly when one of the heads leaves the cells that are occupied by $w$, at the right end.

**Example 3.8.20** *It is easy to see that the 2-head one-way finite automaton in Figure 3.33b recognizes the language* $\{a^i c b^i \mid i \geq 1\}$ *that is not regular.*

**Exercise 3.8.21** *(a) Design a 2-head FA that will accept the language* $\{a^i b^i c^i \mid n \geq 1\}$*; (b) design a 3-head FA that will accept the language* $\{a^i b^{2i} c^{3i} \mid i \geq 1\}$.

Once we know that even one-way 2-head finite automata are more powerful than 1-head finite automata, it is natural to ask whether any additional increase in the number of heads provides more power. To formulate the result, let us denote by $\mathcal{L}(k\text{-2DFA})$ ($\mathcal{L}(k\text{-2NFA})$) the family of languages accepted by deterministic (nondeterministic) two-way finite automata with $k$ heads. In an analogical way we use notation $\mathcal{L}(k\text{-1DFA})$ and $\mathcal{L}(k\text{-1NFA})$.

**Theorem 3.8.22** *For each* $k \geq 1$ *and* $i = 1,2$, $\mathcal{L}(k\text{-}iDFA) \subsetneq \mathcal{L}((k+1)\text{-}iDFA)$ *and* $\mathcal{L}(k\text{-}iNFA) \subsetneq \mathcal{L}((k+1)\text{-}iNFA)$.

The proofs are quite involved, and represent solutions of long-standing open problems. It follows from Theorem 3.8.22 that $k$-head finite automata, for $k = 1,2,\ldots$, form an infinite hierarchy of more and more powerful machines!

How is this possible? It seems that $k$-head finite automata have again only finitely many states and use only a finite amount of memory. This impression, however, is misleading. The actual state of such a machine is determined not only by the state of its finite control but also by the positions of the heads. If an input word $w$ has length $n$, then the overall number of global states (configurations) a $k$-head FA $\mathcal{A}$ can be in is $|Q|n^k$, where $Q$ is the set of internal states of $\mathcal{A}$. The total number of global states of a $k$-head FA therefore grows polynomially with respect to the length of the input.

The following two closely related families of languages,

$$\bigcup_{k=1}^{\infty} \mathcal{L}(k\text{-2DFA}) \quad \text{and} \quad \bigcup_{k=1}^{\infty} \mathcal{L}(k\text{-2NFA}),$$

play an important role in complexity theory, and are the same as two families of languages defined with respect to space complexity, **L** and **NL**, introduced in Section 5.2.

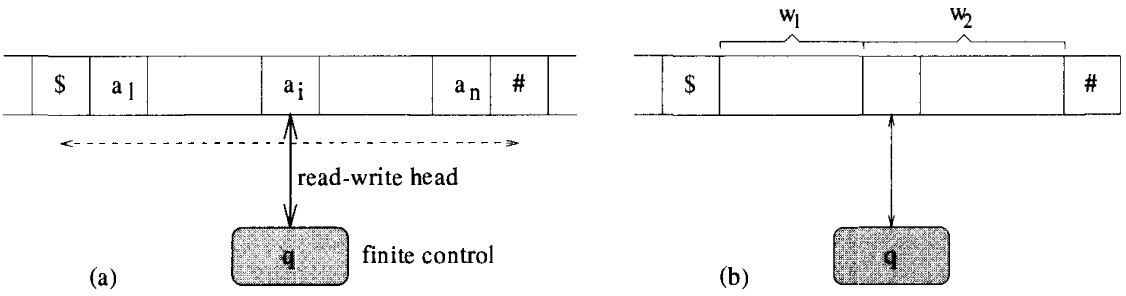



**Figure 3.34** A linearly bounded automaton

### 3.8.5 Linearly Bounded Automata

Another natural generalization of finite automata are linearly bounded automata (LBA for short). The head of a LBA is not only allowed to move in both directions, it may also write (see Figure 3.34a). This is an essentially new and very powerful step in the generalization of the finite automata concept.

Since the head of a LBA can move in both directions and can also write, two markers, \$ and # are used to delimit the beginning and end of the tape section on which the input word is written. A LBA is allowed to move neither left from \$ nor right from #; nor is it allowed to write these markers on the tape or to erase them.

Formally, an LBA $\mathcal{A}$ is specified as $\mathcal{A} = \langle \Sigma, \Delta, Q, q_0, Q_F, \$, \#, \delta \rangle$, where $\Sigma, Q, q_0, Q_F$ have the same meaning as for FA, and

- $\Delta \supseteq \Sigma$ is a **tape alphabet**;
- $\$, \# \in \Delta - \Sigma$ are special **markers**;
- $\delta \subseteq Q \times \Delta \times Q \times \Delta \times \{\leftarrow, \downarrow, \rightarrow\}$ is a **transition relation** satisfying the above-mentioned conditions (that is, if $(p,a,q,b,d) \in \delta$, then $b \notin \{\$, \#\}, a = \$ \Rightarrow d \neq \leftarrow, a = \# \Rightarrow d \neq \rightarrow), a = \$ \Rightarrow b = \$, a = \# \Rightarrow b = \#$.

A LBA $\mathcal{A}$ may perform a transition $(p,a,b,q,d) \in \delta$ when $\mathcal{A}$ is in the state $p$ and the head reads $a$. In this case the finite control of $\mathcal{A}$ goes to state $q$, the head rewrites $a$ by $b$ and moves in the direction $d$. Of course, in general there may be more than one quintuple in $\delta$ starting with the same $p$ and $a$; therefore a move of the head may be nondeterministic. If $\delta$ is a function of its first two arguments, then we have a **deterministic LBA (DLBA** for short).

To describe a **computation** on a LBA $\mathcal{A}$, the concept of **configuration** is again useful. This is a word of the form $w_1 q w_2 \in \Delta^* Q \Delta^*$. A LBA $\mathcal{A}$ is in the configuration $w_1 q w_2$ if $q$ is its current state, $w_1 w_2$ the contents of the tape, and the head is positioned on the first symbol of $w_2$ (see Figure 3.34b). A configuration is **initial** if it has the form $q_0 w$, $w \in \Sigma^*$ (the input alphabet), and **final** if its state is final.

The concept of a configuration is very helpful in formally defining a computation on a LBA. In order to do this, we first introduce the concept of a computation step. A configuration $C'$ is a **direct successor** of a configuration $C$; in short $C \vdash C'$, if $C'$ is a configuration that can be obtained from $C$ by performing a transition, a computation step. A configuration is called **terminating** if there is no configuration that would be its direct successor. (In a terminating computation the LBA 'halts'.) If $C \vdash^* C'$, then $C'$ is called a **successor configuration** of $C$, or a configuration that can be **reached** from $C$. A **computation** of a LBA is a finite or infinite sequence of configurations that starts with the initial configuration, and, for any integer $i > 1$, the $i$th configuration is a direct successor of

the $(i-1)$-th configuration. A **terminating computation** is a finite computation that ends with a terminating configuration.

The language accepted by a LBA $\mathcal{A}$ is defined as follows:

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \exists \text{ computation starting in } q_0 w \text{ and ending in a final configuration}\}.$$

To describe a LBA formally, its transition relation must be specified. To do this in detail may be tedious, but it is basically a straightforward task when a high-level algorithm describing its behaviour is given, as in the following example.

**Example 3.8.23** *We describe the behaviour of a LBA which recognizes the language* $\{a^i b^i \mid i \geq 1\}$.

**begin** Check if the input word has the form $a^i b^i$ – if not, then reject;
**while** there are at least one $a$ and one $b$ on the tape
**do** erase one $a$ and one $b$;
**if** there is still a symbol $a$ or $b$ on the tape **then** reject **else** accept
**end**

**Exercise 3.8.24** *Describe a LBA which accepts the language* $\{a^i b^i c^i \mid i \geq 1\}$.

The above examples show that DLBA can accept languages that are not regular; therefore DLBA are more powerful than finite automata. On the other hand, it is not known whether nondeterminism brings new power in the case of LBA.

**Open problem 3.8.25 (LBA problem)** *Are LBA more powerful as DLBA?*

This is one of the longest standing open problems in foundations of computing.

The next natural question to ask is how powerful are LBA compared with multi-head FA (because multi-head FA have been shown to be more powerful than finite automata). It is in a sense a question as to what provides more power: a possibility to write (and thereby to store immediate results and to make use of memory of a size proportional to the size of the input) or a possibility to use more heads (and thereby parallelism).

Let us denote by $\mathcal{L}(LBA)$ the family of languages accepted by LBA and by $\mathcal{L}(DLBA)$ the family of languages accepted by DLBA. For a reason that will be made clear in Chapter 7, languages from $\mathcal{L}(LBA)$ are called **context-sensitive**, and those from $\mathcal{L}(DLBA)$ are called **deterministic context-sensitive**.

**Theorem 3.8.26** *The following relations hold between the families of languages accepted by multi-head finite automata and LBA:*

$$\bigcup_{k=1}^{\infty} \mathcal{L}(k\text{-2DFA}) \subsetneq \mathcal{L}(\text{DLBA}), \tag{3.11}$$

$$\bigcup_{k=1}^{\infty} \mathcal{L}(k\text{-2NFA}) \subsetneq \mathcal{L}(\text{NLBA}). \tag{3.12}$$

We show here only that each multihead 2DFA can be simulated by a DLBA. Simulation of a multihead 2NFA by a NLBA can be done similarly. The proof that there is a language accepted by a DLBA but not accepted by a multihead 2DFA, and likewise for the nondeterministic case, is beyond the scope of this book.

In order to simulate a $k$-head 2DFA $\mathcal{A}$ by a DLBA $\mathcal{B}$, we need:

(a) to represent a configuration of $\mathcal{A}$ by a configuration of $\mathcal{B}$;
(b) to simulate one transition of $\mathcal{A}$ by a computation on $\mathcal{B}$.

(a) **Representation of configurations.** A configuration of $\mathcal{A}$ is given by a state $q$, a tape content $w = w_1 \ldots w_n$ and the positions of the $k$ heads. In order to represent this information in a configuration of $\mathcal{B}$, the $j$th symbol of $w$, that is, $w_j$, is represented at any moment of a computation by a $(k+2)$-tuple $(q, w_j, s_1, \ldots, s_k)$, where $s_i = 1$ if the $i$th head of $\mathcal{A}$ stays, in the given configuration of $\mathcal{A}$, on the $i$th cell, and $s_i = 0$, otherwise. Moreover, in order to create the representation of the initial configuration of $\mathcal{A}$, $\mathcal{B}$ replaces the symbol $w_1$ in the given input word $w$ by $(q_0, w_1, 1, \ldots, 1)$ and all other $w_i, 1 < i \leq |w|$ by $(q_0, w_i, 0, \ldots, 0)$.

(b) **Simulation of one step of $\mathcal{A}$.** $\mathcal{B}$ reads the whole tape content, and remembers in its finite state control the state of $\mathcal{A}$ and the symbols read by heads in the corresponding configuration of $\mathcal{A}$. This information is enough for $\mathcal{B}$ to simulate a transition of $\mathcal{A}$. $\mathcal{B}$ need only make an additional pass through the tape in order to replace the old state of $\mathcal{A}$ by the new one and update the positions of all heads of $\mathcal{A}$. □

It can happen that a LBA gets into an infinite computation. Indeed, the head can get into a cycle, for example, one step right and one step left, without rewriting the tape. However, in spite of this the following theorem holds.

**Theorem 3.8.27** *The membership problem for LBA is decidable.*

**Proof:** First an observation: the number of configurations of a LBA $\mathcal{A} = \langle \Sigma, \Delta, Q, q_0, Q_F, \$, \#, \delta \rangle$ that can be reached from an initial configuration $q_0 w$ is bounded by $c_w = |Q||\Delta|^{|w|}(|w|+2)$. ($\Delta^{|w|}$ is the number of possible contents of the tape of length $|w|$, $|w|+2$ is the number of cells the head can stand on, and $|Q|$ is the number of possible states.) This implies that if $\mathcal{A}$ is a DLBA, then it is sufficient to simulate $c_w$ steps of $\mathcal{A}$ in order to find out whether there is a terminal configuration reachable from the initial configuration $q_0 w$ – that is, whether $w$ is accepted by $\mathcal{A}$. Indeed, if $\mathcal{A}$ does not terminate in $c_w$ steps, then it must be in an infinite loop. If $\mathcal{A}$ is not deterministic, then configurations reachable from the initial configuration $q_0 w$ form a configuration tree (see Figure 3.35), and in order to find out whether $w \in L(\mathcal{A})$, it is enough to check all configurations of this tree up to the depth $c_w$. □

The fact that a LBA may not halt is unfortunate. This makes it hard to design more complex LBA from simpler ones, for example, by using sequential composition of LBA. The following result is therefore of importance.

**Theorem 3.8.28** *For each LBA there is an equivalent LBA that always terminates.*

To prove this theorem, we apply a new and often useful technique of dividing the tape into more tracks (see Figure 3.36), in this case into two. Informally, each cell of the tape is divided into an upper and a lower subcell. Each of these subcells can contain a symbol and the head can work on the tape in such a way as to read and write only to a subcell of one of the tracks. Formally, this is nothing other than using pairs $\frac{x}{y}$ of symbols as symbols of the tape alphabet, and at each writing changing either none or only one of them or both of them.

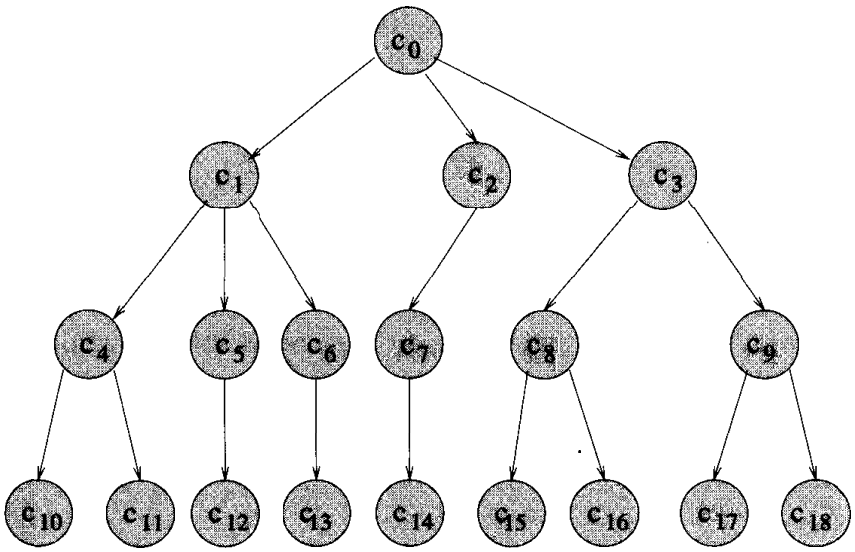



**Figure 3.35** Configuration tree

| | a | c | e | g | i | k | m | o | r | t | v | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | b | d | f | h | j | l | n | p | s | u | w | |

(a) one tape with two tracks

| | a/b | c/d | e/f | g/h | i/j | k/l | m/n | o/p | r/s | t/u | v/w | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

(b) one tape with one track

**Figure 3.36** A tape with two or one tracks

**Proof:** Given an LBA $\mathcal{A}$ with input alphabet $\Sigma$, we design from $\mathcal{A}$ another LBA $\mathcal{B}$ the tape of which consists of two tracks. At the beginning of a computation the input word $w$ is seen as being written in the upper track. $\mathcal{B}$ first computes the number $c_w = |Q||\Delta|^{|w|}(|w|+2)$, the maximum number of possible configurations, and stores this number in the second track. (Such a computation is not a problem with LBA power.) Space is another issue. There is, however, enough space to write $c_w$ on the second track, because $|Q||\Delta|^{|w|}(|w|+2) \leq (2|Q||\Delta|)^{|w|}$. Therefore it is enough to use a number system with a sufficiently large base, for example, $2|Q||\Delta|$, the size of which does not depend on the input word $w$. $\mathcal{B}$ then simulates the computation of $\mathcal{A}$ step by step. Whenever the simulation of a step of $\mathcal{A}$ is finished, $\mathcal{B}$ decreases the number on the second track by 1. If $\mathcal{A}$ accepts before the number on the second track is zero, then $\mathcal{B}$ accepts as well. If $\mathcal{B}$ decreases the number on the second track to zero, then $\mathcal{B}$ moves to a terminating, but not a final, state. Clearly, $\mathcal{B}$ accepts an input word $w$ if and only if $\mathcal{A}$ does. □

The family of context-sensitive languages contains practically all formal languages one has to deal with in practice. It is a rich family, and one of its basic properties is stated in the following theorem.

**Theorem 3.8.29** *Both families $\mathcal{L}(LBA)$ and $\mathcal{L}(DLBA)$ are closed under Boolean operations (union, intersection and complementation).*

**Proof:** Given two LBA (or DLBA) $\mathcal{A}_1, \mathcal{A}_2$ that always terminate, it is easy to design a LBA (or DLBA) that for a given input $w$ simulates first the computation of $\mathcal{A}_1$ on $w$ and then the computation of $\mathcal{A}_2$ on $w$, and accepts $w$ if and only if both $\mathcal{A}_1$ and $\mathcal{A}_2$ accept $w$ (in the case of intersection) or if at least one of them accepts it (in the case of union). This implies closure under union and intersection. To

show closure under complementation is fairly easy for a DLBA $\mathcal{A} = \langle \Sigma, \Delta, Q, q_0, Q_F, \delta, \$, \#, \delta \rangle$, which always terminates. It is enough to take $Q - Q_F$ instead of $Q_F$ as the set of the final states. The proof that the family $\mathcal{L}(LBA)$ is also closed under complementation is much more involved. □

Another natural idea for enhancing the power of finite automata is to allow the head to move everywhere on the tape and to do writing and reading everywhere, not only on cells occupied by the input word. This will be explored in the following chapter and, as we shall see, it leads to the most powerful concept of machines we have.

All the automata we have dealt with in this chapter can be seen as more or less restricted variants of the Turing machines discussed in the next chapter. All the techniques used to design automata in this chapter can be used also as techniques 'to program' Turing machines. This is also one of the reasons why we discussed such models as LBA in detail.

---

**Moral:** Automata, like people, can look very similar and be very different, and can look very different and be very similar. A good rule of thumb in dealing with automata is, as in life, to think twice and explore carefully before making a final judgement.

---

## 3.9 Exercises

1. Let $\mathcal{A}$ be the FA over the alphabet $\{a, b\}$ with the initial state 1, the final state 3, and the transition relation $\delta = \{(1,a,1),(1,b,1),(1,a,2),(2,b,3)\}$. Design an equivalent deterministic and complete FA.

2. Design state graphs for FA which accept the following languages: (a) $L = \{w \mid w \in \{a,b\}^*, aaa$ is not a subword of $w\}$; (b) $L = \{w \mid w \in \{a,b\}^*, w = xbv, |v| = 2\}$; (c) $L = \{w \mid w \in \{a,b\}^*, aaa$ is not a subword of of $w$ and $w = xby, |y| = 2\}$.

3. Design a finite automaton to decide whether a given number $n$ is divided by 3 for the cases: (a) $n$ is given in binary, the most significant digit first; (b) $n$ is given in binary, the least significant digit first; (c) $n$ is given in decimal; (d)* $n$ is given in Fibonacci number representation.

4. Show that if a language $L_1$ can be recognized by a DFA with $n$ states and $L_2$ by a DFA with $m$ states, then there is a DFA with $n2^m$ states that recognizes the language $L_1L_2$ (and in some cases no smaller DFA for $L_1L_2$ exists).

5.* Show that for any $n$-state DFA $\mathcal{A}$ there exists a DFA $\mathcal{A}'$ having at most $2^{n-1} + 2^{n-2}$ states and such that $L(\mathcal{A}') = (L(\mathcal{A}))^*$.

6. Show that a language $L \subseteq \{a\}^*$ over a one-symbol alphabet is regular if and only if there are two finite sets $M_1, M_2 \subset \{a\}^*$ and a $w \in \{a\}^*$ such that $L = M_1 \cup M_2\{w\}^*$.

7. Show that if $R$ is a regular language, then so is the language $R_{half} = \{x \mid \exists y |x| = |y|, xy \in R\}$.

8. Show that the following languages are not regular: (a) $\{ww \mid w \in \{a,b\}^*\}$;
   (b) $\{a^i b^j c^j \mid i,j \geq 1\} \cup b^*c^*$; (c) $L = \{w \mid w \in \{a,b\}^*, w$ contains more $a$'s than $b$'s$\}$.

9. Which of the following languages is regular: (a) UNEQUAL=$\{a^n b^m \mid n,m \in \mathbf{N}, n \neq m\}$; (b) $\{a\}^*$UNEQUAL; (c) $\{b\}^*$UNEQUAL?

10. Show that the following languages are not regular: (a) $\{a^{2^n} \mid n \geq 1\}$; (b) $\{a^{n!} \mid n \geq 1\}$.

11. Let $w$ be a string. How many states has the minimal DFA recognizing the set of all substrings of $w$?

12.* Let $L_n = \{x_1\#x_2\#\ldots x_m\#\#x \mid x_i \in \{a,b\}^n, x = x_j \text{ for some } 1 \leq j \leq m\}$. Show that each DFA accepting $L_n$ must have $2^{2^n}$ states.

13. Let $R_1$ and $R_2$ be regular languages recognized by DFA $\mathcal{A}_1$ and $\mathcal{A}_2$ with $r$ and $s$ states, respectively. Show that the languages $R_1 \cup R_2$ and $R_1 \cap R_2$ can be recognized by DFA with $rs$ states. (Hint: take the Cartesian product of the states of $\mathcal{A}_1$ and $\mathcal{A}_2$ as the new set of states.)

14. Find two languages $L_1, L_2$ such that neither of them is regular but their union and also their intersection are.

15. For two languages $K, L \subset \Sigma^*$ define the right quotient $K/L = \{u \in \Sigma^* \mid \exists v \in L, uv \in K\}$. Show that the family of regular languages is closed under the operation of right quotient.

16.* Let us assume that there exists a morphism $\phi$ from $\Sigma^*$ to a finite monoid $\mathcal{M}$, and let for some $L \subseteq \Sigma^*, \phi^{-1}(\phi(L)) = L$. Prove that $L$ is a regular language.

17.* Show that the regular language $\{10101\}^*$ can be expressed without the operation of iteration, using only the operations of union, concatenation, complementation and intersection.

18. Show that the family of regular languages is closed under the shuffle operation.

19. Show that if a tree automaton has the property that for the acceptance of an input word it does not matter which level of processors is used to start the computation with an input word $w$, provided the level has at least $|w|$ processors and the input is given to the left-most processor, one symbol per processor, then such a tree automaton always accepts a regular language.

20. Let $R \subset \Sigma^*$ be a regular language. Is the language $twist(R)$ defined by the following mapping: $TWIST : \Sigma^* \to \Sigma^*$; $TWIST(x) = x$ if $x \in \Sigma \cup \{\varepsilon\}$, $TWIST(awb) = abTWIST(w)$ if $a, b \in \Sigma$, regular?

21. Design a Moore or a Mealy machine with three states which for the input 00001000100010 produces as output 01010000101001.

22. Show that the family of relations defined by finite transducers is closed under composition.

23.* Let $f$ be a bijection between $\Sigma^*$ and $\Delta^*$ that preserves prefixes, lengths and regular sets. Show that $f$ is realized by a generalized sequential machine.

24. Which image transformations realize the WFT in Figure 3.37?

25. Construct a WFT to perform (a)** a rotation by 90 degrees counterclockwise with linear slope; (b)* stretching defined by the mapping $(x, y) \to (x, \frac{3}{2}y)$.

26.** Show that the function computed by the LWFT depicted in Figure 3.38 is continuous if and only if the following two conditions are satisfied: (1) $\alpha + \beta = 1$, (2) $\delta(1-\alpha) = \gamma(1-\delta)$, $0 \leq \alpha, \beta \leq 1$, $0 \leq \gamma, \delta$.

27.* Show that the polynomial $x^n$ can be computed by a WFA with $n+1$ states $(q_n, \ldots, q_0)$ with the initial distribution $(1, 0, \ldots, 0)$, final distribution $(1, \ldots, 1)$ and the following transitions:

(1) $q_i \xrightarrow{j,2^{-i}} q_i$ for $j = 0, 1, i = 0, 1, \ldots, n$; (2) $q_i \xrightarrow{1,2^{-i}\binom{i}{t}} q_{i-t}$, for $i = 1, \ldots, n$ and $t = 1, \ldots, i$.

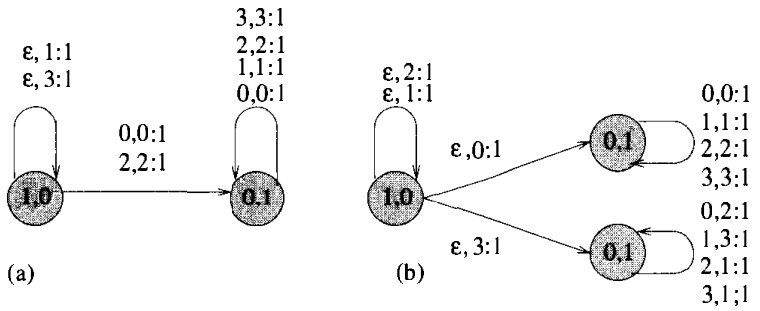



**Figure 3.37** WFA

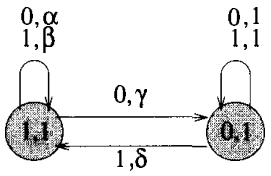



**Figure 3.38** LWFT

28. Prove the second assertion of Theorem 3.6.4.

29. Show that the $\omega$-language $\{a^n b^n c^\omega \mid n \geq 0\}$ is not regular.

30. Design Büchi automata that recognize the following $\omega$-languages: (a) an $\omega$-language consisting of $\omega$-words over the alphabet $\{a,b,c\}$ with infinitely many $a$'s and $b$'s and such that there is an odd number of $c$'s between any two symbols from $\{a,b\}$; (b) an $\omega$-language consisting of all $\omega$-words over $\{a,b,c\}$ with infinitely many $a$'s and $b$'s, but with never more than three $c$'s in a row.

31. Determine whether the following $\omega$-languages are regular: (a) $\{a^i b^i \mid i \geq 0\}^\omega$; (b) $\{a^i b^j \mid i,j \geq 0\}^\omega$; (c) $\{a^i b^j \mid 1 \leq i \leq j\}^\omega$.

32. Show that there is no finite state machine to compute the following functions $f : \mathbf{N} :\rightarrow \mathbf{N}$: (a) $f(n) = n^2$; (b) $f(n) = \lfloor \sqrt{n} \rfloor$.

33. Design a transition system with as few states as possible to recognize the languages (a) $\{a^{3i} b^{4j} \mid i,j \geq 1\}$; (b)* $L_n = \{a^i \mid 1 \leq i \leq n\}$.

34. Let $\mathcal{A} = \langle \Sigma, Q, Q_I, Q_F, \delta \rangle$ be a transition system with the alphabet $\Sigma = \{a,b,c\}$, states $Q = \{1,2,\ldots,7\}$, the initial states $Q_I = \{1,2\}$, the final states $Q_F = \{4,5\}$ and the transitions $\{(1,abc,5),(2,\varepsilon,4),\ (3,b,4),\ (4,a,6),\ (4,c,7),\ (6,c,5)\}$. Transform $\mathcal{A}$, step by step, into an equivalent transition system with the following properties: (a) only one initial state; (b) transitions only on symbols from $\Sigma \cup \{\varepsilon\}$; (c) transitions on all symbols from all states; (d) all states reachable from the initial state; (e) complete and deterministic FA.

35. Show that every stochastic language is $c$-stochastic for any $0 \leq c \leq 1$.

36.* Give an example of a probabilistic finite automaton which accepts a nonregular language with the cut-point $\frac{1}{3}$.

37. Design a multi-head FA that recognizes the languages (a) $\{a^i b^j c^i d^j \mid i,j \geq 1\}$; (b) $\{ww^R \mid w \in \{0,1\}^*\}$.

38. Design LBA that recognize the languages (a) $\{a^i \mid i \text{ is a prime}\}$; (b) $\{ww^R \mid, w \in \{0,1\}^*\}$.

39. Which of the following string-to-string functions over the alphabet $\{0,1\}$ can be realized by a finite transducer: (a) $w \rightarrow w^R$; (b) $w_1 \ldots w_n \rightarrow w_1 w_1 w_2 w_2 \ldots w_n w_n$; (c) $w_1 \ldots w_n \rightarrow w_1 \ldots w_n w_1 \ldots w_n$?

## Questions

1. When does the subset construction yield the empty set of states as a new reachable state?
2. Are minimal nondeterministic finite automata always unique?
3. Is the set of regular languages closed under the shuffle operation?
4. Is the mapping $1^i \rightarrow 1^{F_i}$ realizable by a finite transducer?
5. What is the role of initial and terminal distributions for WFA?
6. How can one define WFA generating three-dimensional images?
7. Weighted finite automata and probabilistic finite automata are defined very similarly. What are the differences?
8. Does the power of two-way finite automata change if we assume that input is put between two end markers?
9. Are LBA with several heads on the tape more powerful than ordinary LBA?
10. What are natural ways to define finite automata on $\omega\omega$- words, and how can one define in a natural way the concept of regular $\omega\omega$- languages?

## 3.10 Historical and Bibliographical References

It is surprising that such a basic and elementary concept as that of finite state machine was discovered only in the middle of this century. The lecture of John von Neumann (1951) can be seen as the initiative to develop a mathematical theory of automata, though the concept of finite automata, as discussed in this chapter, is usually credited to McCulloch and Pitts (1943). Its modern formalization is due to Moore (1956) and Scott (1959). (Dana Scott received the Turing award in 1976.)

Finite automata are the subject of numerous books: for example, Salomaa (1969), Hopcroft and Ullman (1969), Brauer (1984) and Floyd and Beigel (1994). (John E. Hopcroft received the Turing award in 1986 for his contribution to data structures, Robert Floyd in 1978 for his contribution to program correctness.) A very comprehensive but also very special treatment of the subject is due to Eilenberg (1974). See also the survey by Perrin (1990).

Bar-Hillel and his collaborators, see Bar-Hillel (1964), were the first to deal with finite automata in more detail. The concept of NFA and Theorem 3.2.8 are due to Rabin and Scott (1959). The proof that there is a NFA with $n$ states such that each equivalent DFA has $2^n$ states can be found in Trakhtenbrot and Barzdin (1973) and in Lupanov (1963). Minimization of finite automata and Theorem 3.2.16 are due to Huffman (1954) and Moore (1956). The first minimization algorithm, based on two operations, is from Brauer (1988) and credited to Brzozowski (1962). Asymptotically the fastest known minimization algorithm, in time $\mathcal{O}(mn \lg n)$, is due to Hopcroft (1971). The pumping lemma

for regular language has emerged in the course of time; for two variants and detailed discussion see Floyd and Beigel (1994). For string-matching algorithms see Knuth, Morris and Pratt (1977).

The concepts of regular language and regular expression and Theorem 3.3.6 are due to Kleene (1956). The concept of derivatives of regular languages is due to Brzozowski (1964). Very high lower bounds for the inequivalence problem for generalized regular expressions are due to Stockmeyer and Meyer (1973). The characterization of regular languages in terms of syntactical congruences, Theorems 3.3.16 and 3.3.17 are due to Myhill (1957) and Nerode (1958). The recognition of regular languages in logarithmic time using syntactical monoids is due to Culik, Salomaa, and Wood (1984). The existence of regular languages for which each processor of the recognizing tree network of processors has to be huge is due to Gruska, Napoli and Parente (1994).

For two main models of finite state machines see Mealy (1955) and Moore (1956), and for their detailed analysis see Brauer (1984). The results concerning finite transducers and generalized sequential machines, Theorems 3.4.8–11 are due to Ginsburg and Rose (1963, 1966); see also Ginsburg (1966). (Moore and Mealy machines are also called Moore and Mealy automata and in such a case finite automata as defined in Section 3.1 are called Rabin-Scott automata.)

The concept of a weighted finite automaton and a weighted finite transducer are due to Culik and his collaborators: Culik and Kari (1993, 1994,1995); Culik and Friš (1995); Culik and Rajčáni (1996). See also Culik and Kari (1995) and Rajčáni (1995) for a survey. Section 3.4.2 and examples, exercises and images are derived from these and related papers. For a more practical 'recursive image compression algorithm' see Culik and Kari (1994). The idea of using finite automata to compute continuous functions is due to Culik and Karhumäki (1994). The existence of a function that is everywhere continuous, but nowhere has derivatives and is still computable by WFA is due to Derencourt, Karhumäki, Latteux and Terlutte (1994). An interesting and powerful generalization of WFT, the iterative WFT, has been introduced by Culik and Rajčáni (1995).

The idea of finite automata on infinite words is due to Büchi (1960) and McNaughton (1966). Together with the concept of finite automata on infinite trees, due to Rabin (1969), this created the foundations for areas of computing dealing with nonterminating processes. For Muller automata see Muller (1963). A detailed overview of computations on infinite objects is due to Gale and Stewart (1953) and Thomas (1990). For a presentation of problems and results concerning Gale–Stewart (1953) games see Thomas (1995).

The concept of a transition system and Theorem 3.8.1 are due to Myhill (1957). Probabilistic finite automata were introduced by Rabin (1963), Carlyle (1964) and Bucharaev (1964). Theorems 3.8.8 and 3.8.9 are due to Rabin (1963), and the proof of the second theorem presented here is due to Paz (1971). See also Salomaa (1969), Starke (1969) and Bucharaev (1995) for probabilistic finite automata.

Two-way finite automata were introduced early on by Rabin and Scott (1959), who also made a sketch of the proof of Theorem 3.8.17. A simpler proof is due to Shepherdson (1959); see also Hopcroft and Ullman (1969). Example 3.8.16 is due to Barnes (1971) and Brauer (1984). For results concerning the economy of description of regular languages with two-way FA see Meyer and Fischer (1971). Multi-head finite automata were introduced by Rosenberg (1966), and the existence of infinite hierarchies was shown by Yao and Rivest (1978) for the one-way case and Monien (1980) for two-way $k$-head finite automata.

Deterministic linearly bounded automata were introduced by Myhill (1960), nondeterministic ones by Kuroda (1964). The closure of DLBA under intersection and complementation was shown by Landweber (1963), and the closure of NLBA under complementation independently by Szelepcsényi (1987) and Immerman (1988).