Finite automata

M. V. Lawson

Department of Mathematics
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Edinburgh EH14 4AS
Scotland
M.V.Lawson@ma.hw.ac.uk

1 Introduction

The term 'finite automata' describes a class of models of computation that are characterised by having a finite number of states. The use of the word 'automata' harks back to the early days of the subject in the 1950's when they were viewed as abstract models of real circuits. However, the readers of this chapter can also view them as being implemented by programming languages, or as themselves constituting a special class of programming languages. Taking this latter viewpoint, finite automata form a class of programming languages with very restricted operation sets: roughly speaking, programs where there is no recursion or looping. In particular, none of them is a universal programming language; indeed, it is possible to prove that some quite straightforward operations cannot be programmed even with the most general automata I discuss. Why, then, should anyone be interested in programming with, as it were, one hand tied behind their back? The answer is that automata turn out to be useful — so they are not merely mathematical curiosities. In addition, because they are restricted in what they can do, we can actually say more about them, which in turn helps us manipulate them.

The most general model I discuss in this chapter is that of a finite transducer, in Section 3.3, but I build up to this model in Sections 2, 3.1, and 3.2 by discussing, in increasing generality: finite acceptors, finite purely sequential transducers, and finite sequential transducers. The finite acceptors form the foundation of the whole enterprise through their intimate link with regular expressions and, in addition, they form the pattern after which the more general theories are modelled.

What then are the advantages of the various kinds of finite transducers considered in this chapter? There are two main ones: the speed with which data can be processed by such a device, and the algorithms that enable one to make the devices more efficient. The fact that finite transducers of vari-

ous kinds have turned out to be useful in natural language processing is a testament to both of these advantages [23]. I discuss the advantages of finite transducers in a little more detail in Section 4.

To read this chapter, I have assumed that you have been exposed to a first course in discrete math(s); you need to know a little about sets, functions, and relations, but not much more. My goal has been to describe the core of the theory in the belief that once the basic ideas have been grasped, the business of adding various bells and whistles can easily be carried out according to taste.

Other reading There are two classic books outlining the theory of finite transducers: Berstel [4] and Eilenberg [12]. Of these, I find Berstel's 1979 book the more accessible. However, the theory has moved on since 1979, and in the course of this chapter I refer to recent papers that take the subject up to the present day. In particular, the paper [24] contains a modern, mathematical approach to the basic theory of finite transducers. The book by Jacques Sakarovitch [30] is a recent account of automata theory that is likely to become a standard reference. The chapters by Berstel and Perrin [6], on algorithms on words, and by Laporte [21], on symbolic natural language processing, both to be found in the forthcoming book by M. Lothaire, are excellent introductions to finite automata and their applications. The articles [25] and [35] are interesting in themselves and useful for their lengthy bibliographies. My chapter deals entirely with finite strings — for the theory of infinite strings see [26]. Finally, finite transducers are merely a part of theoretical computer science; for the big picture, see [17].

Terminology This has not been entirely standardised so readers should be on their guard when reading papers and books on finite automata. Throughout this chapter I have adopted the following terminology introduced by Jacques Sakarovitch and suggested to me by Jean-Eric Pin: a 'purely sequential function' is what is frequently referred to in the literature as a 'sequential function'; whereas a 'sequential function' is what is frequently referred to as a 'subsequential function'. The new terminology is more logical than the old, and signals more clearly the role of sequential functions (in the new sense).

2 Finite Acceptors

The automata in this section might initially not seem very useful: their response to an input is to output either a 'yes' or a 'no'. However, the concepts and ideas introduced here provide the foundation for the rest of the chapter, and a model for the sorts of things that finite automata can do.

2.1 Alphabets, strings, and languages

Information is encoded by means of sequences of symbols. Any finite set A used to encode information will be called an *alphabet*, and any finite sequence whose components are drawn from A is called a *string over* A or simply a *string*, and sometimes a *word*. We call the elements of an alphabet *symbols*, *letters*, or *tokens*. The symbols in an alphabet do not have to be especially simple; an alphabet could consist of pictures, or each element of an alphabet could itself be a sequence of symbols. A string is formally written using brackets and commas to separate components. Thus (now, is, the, winter, of, our, discontent) is a string over the alphabet whose symbols are the words in an English dictionary. The string () is the empty string. However, we shall write strings without brackets and commas and so, for instance, we write 01110 rather than $(0, 1, 1, 1, 0)$. The empty string needs to be recorded in some way and we denote it by ε. The set of all strings over the alphabet A is denoted by A^*, read 'A star'. If w is a string then $|w|$ denotes the total number of symbols appearing in w and is called the *length of* w. Observe that $|\varepsilon| = 0$. It is worth noting that two strings u and v over an alphabet A are *equal* if they contain the same symbols in the same order.

Given two strings $x, y \in A^*$, we can form a new string $x \cdot y$, called the *concatenation of* x *and* y, by simply adjoining the symbols in y to those in x. We shall usually denote the concatenation of x and y by xy rather than $x \cdot y$. The string ε has a special property with respect to concatenation: for each string $x \in A^*$ we clearly have that $\varepsilon x = x = x\varepsilon$. It is important to emphasise that the order in which strings are concatenated is important: thus xy is generally different from yx.

There are many definitions concerned with strings, but for this chapter I just need two. Let $x, y \in A^*$. If $u = xy$ then x is called a *prefix* of u, and y is called a *suffix* of u.

Alphabets and strings are needed to define the key concept of this section: that of a language. Before formally defining this term, here is a motivating example.

Example 1. Let A be the alphabet that consists of all words in an English dictionary; so we regard each English word as being a single symbol. The set A^* consists of all possible finite sequences of words. Define the subset L of A^* to consist of all sequences of words that form grammatically correct English sentences. Thus

$$\text{to be or not to be} \in L$$

whereas

$$\text{be be to to or not } \notin L.$$

Someone who wants to understand English has to learn the rules for deciding when a string of words belongs to the set L. We can therefore think of L as being the English language.

For *any* alphabet A, *any* subset of A^* is called an *A-language* or a *language over A* or simply a *language*. Languages are usually infinite; the question we shall address in Section 2.2 is to find a finite way of describing (some) infinite languages.

There are a number of ways of combining languages to make new ones. If L and M are languages over A so are $L \cap M$, $L \cup M$, and L': respectively, the intersection of L and M, the union of L and M, and the complement of L. These are called the *Boolean operations* and come from set theory. Recall that '$x \in L \cup M$' means '$x \in L$ or $x \in M$ or both.' In automata theory, we usually write $L + M$ rather than $L \cup M$ when dealing with languages. There are two further operations on languages that are peculiar to automata theory and extremely important: the product and the Kleene star. Let L and M be languages. Then

$$L \cdot M = \{xy \colon x \in L \text{ and } y \in M\}$$

is called the *product of L and M*. We usually write LM rather than $L \cdot M$. A string belongs to LM if it can be written as a string in L *followed by* a string in M. For a language L, we define $L^0 = \{\varepsilon\}$, and $L^{n+1} = L^n \cdot L$. For $n > 0$, the language L^n consists of all strings u of the form $u = x_1 \dots x_n$ where $x_i \in L$. The *Kleene star* of a language L, denoted L^*, is defined to be

$$L^* = L^0 + L^1 + L^2 + \dots .$$

2.2 Finite acceptors

An information-processing device transforms inputs into outputs. In general, there are two alphabets associated with such a device: an *input alphabet A* for communicating with it, and an *output alphabet B* for receiving answers. For example, consider a device that takes as input sentences in English and outputs the corresponding sentence in Russian. In later sections, I shall describe mathematical models of such devices of increasing generality. In this section, I shall look at a special case: there is an input alphabet A, but each input string causes the device to output either 'yes' or 'no' once the whole input has been processed. Those input strings from A^* that cause the machine to output 'yes' are said to be *accepted* by the machine, and those that cause it to output 'no' are said to be *rejected*. In this way, A^* is partitioned into two subsets: the 'yes' subset we call the *language accepted by the machine*, and the 'no' subset we call the *language rejected by the machine*. A device that operates in this way is called an *acceptor*. We shall describe a mathematical model of a special class of acceptors. Our goal is *to describe potentially infinite languages by finite means.*

A *finite (deterministic) acceptor* $\mathbf{A}$ is specified by five pieces of information:

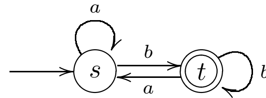
$$\mathbf{A} = (S, A, i, \delta, T)\,,$$

where S is a finite set called the *set of states*, A is the finite *input alphabet*, i is a fixed element of S called the *initial state*, δ is a function $\delta \colon S \times A \to S$,

called the *transition function*, and T is a subset of S (possibly empty!) called the set of *terminal* or *final states*.

There are two ways of providing the five pieces of information needed to specify an acceptor: transition diagrams and transition tables. A *transition diagram* is a special kind of directed labelled graph: the vertices are labelled by the states S of $\mathbf{A}$; there is an arrow labelled a from the vertex labelled s to the vertex labelled t precisely when $\delta(s, a) = t$ in $\mathbf{A}$. That is to say, the input a causes the acceptor $\mathbf{A}$ to change from state s to state t. Finally, the initial state and terminal states are distinguished in some way: we mark the initial state by an inward-pointing arrow, $\longrightarrow (i)$, and the terminal states by double circles $((t))$. A *transition table* is just a way of describing the transition function δ in tabular form and making clear in some way the initial and terminal states. The table has rows labelled by the states and columns labelled by the input letters. At the intersection of row s and column a we put the element $\delta(s, a)$. The states labelling the rows are marked to indicate the initial state and the terminal states.

Example 2. Here is a simple example of a transition diagram of a finite acceptor.



We can easily read off the five ingredients that specify an acceptor from this diagram:

- The set of states is $S = \{s, t\}$.
- The input alphabet is $A = \{a, b\}$.
- The initial state is s.
- The set of terminal states is $\{t\}$.

Finally, the transition function $\delta\colon S \times A \to S$ is given by

$$\delta(s, a) = s, \quad \delta(s, b) = t, \quad \delta(t, a) = s, \text{ and } \delta(t, b) = t.$$

Here is the transition table of our acceptor

	a	b
$\rightarrow s$	s	t
$\leftarrow t$	s	t

We designate the initial state by an inward-pointing arrow $\rightarrow$ and the terminal states by outward-pointing arrows $\leftarrow$. If a state is both initial and terminal, then the inward- and outward-pointing arrows will be written as a single double-headed arrow $\leftrightarrow$.

To avoid too many arrows cluttering up a transition diagram, the following convention is used: if the letters $a_1, \ldots, a_m$ label m transitions from the state s to the state t, then we simply draw *one* arrow from s to t labelled $a_1, \ldots, a_m$ rather than m arrows labelled a_1 to a_m, respectively.

Let $\mathbf{A}$ be a finite acceptor with input alphabet A and initial state i. For each state q of $\mathbf{A}$ and for each input string x, there is a unique path in $\mathbf{A}$ that begins at q and is labelled by the symbols in x in turn. This path ends at a state we denote by $q \cdot x$. We say that x is *accepted* by $\mathbf{A}$ if $i \cdot x$ is a terminal state. That is, x labels a path in $\mathbf{A}$ that begins at the initial state and ends at a terminal state. Define the *language accepted* or *recognised* by $\mathbf{A}$, denoted $L(\mathbf{A})$, to be the set of all strings in the input alphabet that are accepted by $\mathbf{A}$. A language is said to be *recognisable* if it is accepted by some finite automaton. Observe that the empty string is accepted by an automaton if and only if the initial state is also terminal.

Example 3. We describe the language recognised by our acceptor in Example 2. We have to find all those strings in $(a+b)^*$ that label paths starting at s and finishing at t. First, any string x ending in a 'b' will be accepted. To see why, let $x = x'b$ where $x' \in A^*$. If x' leads the acceptor to state s, then the b will lead the acceptor to state t; and if x' leads the acceptor to state t, then the b will keep it there. Second, a string x ending in 'a' will not be accepted. To see why, let $x = x'a$ where $x' \in A^*$. If x' leads the acceptor to state s, then the a will keep it there; and if x' leads the acceptor to state t, then the a will send it to state s. We conclude that $L(\mathbf{A}) = A^*\{b\}$.

Here are some further examples of recognisable languages. I leave it as an exercise to the reader to construct suitable finite acceptors.

Example 4. Let $A = \{a, b\}$.

(i) The empty set $\emptyset$ is recognisable.
(ii) The language $\{\varepsilon\}$ is recognisable.
(iii) The languages $\{a\}$ and $\{b\}$ are recognisable.

It is worth pointing out that not all languages are recognisable. For example, the language consisting of those strings of a's and b's having an equal number of a's and b's is not recognisable.

One very important feature of finite (deterministic) acceptors needs to be highlighted, since it has great practical importance. The time taken for a finite acceptor to determine whether a string is accepted or rejected is a linear function of the length of the string; once a string has been completely read, we will have our answer.

The classic account of the theory of finite acceptors and their languages is contained in the first three chapters of [16]. The first two chapters of my book [22] describe the basics of finite acceptors at a more elementary level.

2.3 Non-deterministic ε-acceptors

The task of constructing a finite acceptor to recognise a given language can be a frustrating one. The chief reason for the difficulty is that finite acceptors are quite rigid: they have *one* initial state, and for each input letter and each state exactly *one* transition. Our first step, then, will be to relax these two conditions.

A *finite non-deterministic acceptor* $\mathbf{A}$ is determined by five pieces of information:

$$\mathbf{A} = (S, A, I, \delta, T),$$

where S is a finite set of states, A is the input alphabet, I is a set of initial states, $\delta\colon S \times A \to \mathsf{P}(S)$ is the transition function, where $\mathsf{P}(S)$ is the set of all subsets of S, and T is a set of terminal states. In addition to allowing any number of initial states, the key feature of this definition is that $\delta(s, a)$ is now a subset of S, possibly empty. The transition diagrams and transition tables we defined for deterministic acceptors can easily be adapted to describe non-deterministic ones. If q is a state and x a string, then the set of all states q' for which there is a path in $\mathbf{A}$ beginning at q, ending at q', and labelled by x is denoted by $q \cdot x$. The language $L(\mathbf{A})$ recognised by a non-deterministic acceptor consists of all those strings in A^* that label a path in $\mathbf{A}$ from at least one of the initial states to at least one of the terminal states.

It might be thought that, because there is a degree of choice available, non-deterministic acceptors might be able to recognise languages that deterministic ones could not. In fact, this is not so.

Theorem 1. *Let $\mathbf{A}$ be a finite non-deterministic acceptor. Then there is an algorithm for constructing a deterministic acceptor, $\mathbf{A}^d$, such that $L(\mathbf{A}^d) = L(\mathbf{A})$.*

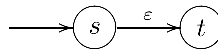
We now introduce a further measure of flexibility in constructing acceptors. In both deterministic and non-deterministic acceptors, transitions may only be labelled with elements of the input alphabet; no edge may be labelled with the empty string ε. We shall now waive this restriction. A *finite non-deterministic acceptor with ε-transitions* or, more simply, a *finite ε-acceptor*, is a 5-tuple,

$$\mathbf{A} = (S, A, I, \delta, T),$$

where all the symbols have the same meanings as in the non-deterministic case except that now

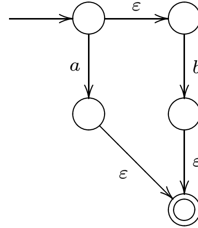
$$\delta\colon S \times (A \cup \{\varepsilon\}) \to \mathsf{P}(S).$$

The only difference between such acceptors and non-deterministic ones is that we allow transitions, called *ε-transitions*, of the form



A path in an ε-acceptor is a sequence of states each labelled by an element of the set $A \cup \{\varepsilon\}$. The string corresponding to this path is the *concatenation* of these labels in order; it is important to remember at this point that for every string x we have that $\varepsilon x = x = x\varepsilon$. We say that a string x is accepted by an ε-automaton if there is a path from an initial state to a terminal state the *concatenation of whose labels is* x.

Example 5. Consider the following finite ε-acceptor:

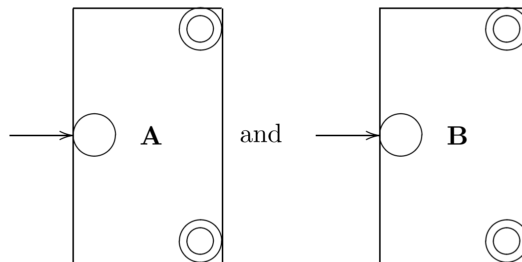


The language it recognises is $\{a, b\}$. The letter a is recognised because $a\varepsilon$ labels a path from the initial to the terminal state, and the letter b is recognised because $\varepsilon b \varepsilon$ labels a path from the initial to the terminal state.

The existence of ε-transitions introduces a further measure of flexibility in building acceptors but, as the following theorem shows, we can convert such acceptors to non-deterministic automata without changing the language recognised.

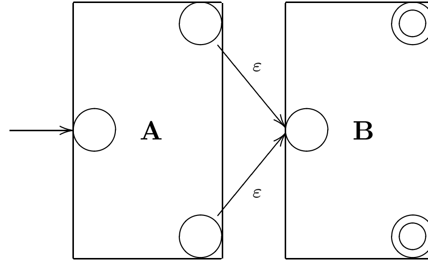
Theorem 2. *Let* $\mathbf{A}$ *be a finite* ε*-acceptor. Then there is an algorithm that constructs a non-deterministic acceptor without* ε*-transitions,* $\mathbf{A}^s$*, such that* $L(\mathbf{A}^s) = L(\mathbf{A})$.

Example 6. We can use ε-acceptors to prove that if L and M are recognisable languages, then so is LM. By assumption, we are given two acceptors $\mathbf{A}$ and $\mathbf{B}$ such that $L(\mathbf{A}) = L$ and $L(\mathbf{B}) = M$. We picture $\mathbf{A}$ and $\mathbf{B}$ schematically as follows:



Now construct the following ε-acceptor: from each terminal state of $\mathbf{A}$ draw an ε-transition to the initial state of $\mathbf{B}$. Make each of the terminal states of

A ordinary states and make the initial state of **B** an ordinary state. Call the resulting acceptor **C**. This can be pictured as follows:



It is easy to see that this ε-acceptor recognises LM. By Theorem 2, we can construct a non-deterministic acceptor recognising LM, and by Theorem 1 we can convert this non-deterministic acceptor into a deterministic acceptor recognising LM. We have therefore proved that LM is recognisable.

Using the idea of Example 6, the following can easily be proved.

Theorem 3. *Let A be an alphabet and L and M be languages over A.*

(i) *If L and M are recognisable then $L + M$ is recognisable.*
(ii) *If L and M are recognisable then LM is recognisable.*
(iii) *If L is recognisable then L^* is recognisable.*

It is worth mentioning that the recognisable languages are closed under all the Boolean operations: thus if L and M are recognisable so too are $L \cap M$, $L + M$, and L'. Furthermore, given finite deterministic acceptors for L and M, it is easy to construct directly finite deterministic acceptors for $L \cap M$, $L + M$, and L'. The proof of this can be found in Chapter 2 of [22].

The explicit algorithms for constructing deterministic acceptors from non-deterministic ones ('the subset construction'), and non-deterministic acceptors from ε-acceptors are described in most books on automata theory; see [16], and Chapters 3 and 4 of [22], for example.

2.4 Kleene's theorem

This is now a good opportunity to reflect on which languages we can now prove are recognisable. I want to pick out four main results. Let $A = \{a_1, \ldots, a_n\}$. Then from Example 4 and Theorem 3, we have the following:

- Each of the languages $\emptyset$, $\{\varepsilon\}$, and $\{a_i\}$ is recognisable.
- The union of two recognisable languages is recognisable.
- The product of two recognisable languages is recognisable.
- The Kleene star of a recognisable language is recognisable.

Call a language over an alphabet *basic* if it is either empty, consists of the empty string alone, or consists of a single symbol from the alphabet. Then what we have proved is the following: a language that can be constructed from the basic languages by using only the operations $+$, $\cdot$, and $*$ a finite number of times must be recognisable. Such a language is said to be *regular* or *rational*.

Example 7. Consider the language L over the alphabet $\{a, b\}$ that consists of all strings of even length. We shall show that this is a regular language. A string of even length is either empty, or can be written as a product of strings of length 2. Conversely every string that can be written as a product of strings of length 2 has even length. It follows that

$$L = (((\{a\}\{a\} + \{a\}\{b\}) + \{b\}\{a\}) + \{b\}\{b\})^*.$$

Thus L is regular.

In the example above, we would much rather write

$$L = (aa + ab + ba + bb)^*$$

for clarity. How to do this in general is formalised in the notion of a 'regular expression.' Let $A = \{a_1, \ldots, a_n\}$ be an alphabet. A *regular expression (over A)* or *rational expression (over A)* is a sequence of symbols formed by repeated application of the following rules:

(R1) $\emptyset$ is a regular expression.
(R2) ε is a regular expression.
(R3) $a_1, \ldots, a_n$ are each regular expressions.
(R4) If s and t are regular expressions then so is $(s + t)$.
(R5) If s and t are regular expressions then so is $(s \cdot t)$.
(R6) If s is a regular expression then so is (s^*).
(R7) Every regular expression arises by a finite number of applications of the rules (R1) through (R6).

As usual, we will write st rather than $s \cdot t$. Each regular expression s describes a regular language, denoted by $L(s)$. This language is calculated by means of the following rules. Simply put, they tell us how to 'insert the curly brackets.'

(D1) $L(\emptyset) = \emptyset$.
(D2) $L(\varepsilon) = \{\varepsilon\}$.
(D3) $L(a_i) = \{a_i\}$.
(D4) $L(s + t) = L(s) + L(t)$.
(D5) $L(s \cdot t) = L(s) \cdot L(t)$.
(D6) $L(s^*) = L(s)^*$.

It is also possible to get rid of many of the left and right brackets that occur in a regular expression by making conventions about the precedence of the regular operators. When this is done, regular expressions form a useful notation for

describing regular languages. However, if a language is described in some other way, it may be necessary to carry out some work to find a regular expression that describes it; Example 7 illustrates this point.

The first major result in automata theory is the following, known as Kleene's theorem.

Theorem 4. *A language is recognisable if and only if it is regular. In particular, there are algorithms that accept as input a regular expression r, and output a finite acceptor $\mathbf{A}$ such that $L(\mathbf{A}) = L(r)$; and there are algorithms that accept as input a finite acceptor $\mathbf{A}$, and output a regular expression r such that $L(r) = L(\mathbf{A})$.*

This theorem is significant for two reasons: first, it tells us that there is an algorithm that enables us to construct an acceptor recognising a language from a suitable description of that language; second, it tells us that there is an algorithm that will produce a description of the language recognised by an acceptor.

A number of different proofs of Kleene's theorem may be found in Chapter 5 of [22]. For further references on how to convert regular expressions into finite acceptors, see [5] and [7]. Regular expressions as I have defined them are useful for proving Kleene's theorem but hardly provide a convenient tool for describing regular languages over realistic alphabets containing large numbers of symbols. The practical side of regular expressions is described by Friedl [14] who shows how to use regular expressions to search texts.

2.5 Minimal automata

In this section, I shall describe an important feature of finite acceptors that makes them particularly useful in applications: the fact that they can be minimised. I have chosen to take the simplest approach in describing this property, but at the end of this section, I describe a more sophisticated one needed in generalisations.

Given a recognisable language L, there will be many finite acceptors that recognise L. All things being equal, we would usually want to pick the smallest such acceptor: namely, one having the smallest number of states. It is conceivable that there could be two different acceptors $\mathbf{A}_1$ and $\mathbf{A}_2$ both recognising L, both having the same number of states, and sharing the additional property that there is no acceptor with fewer states recognising L. In this section, I shall explain why this cannot happen. This result has an important consequence: every recognisable language is accepted by an essentially unique smallest acceptor. To show that this is true, we begin by showing how an acceptor may be reduced in size without changing the language recognised. There are two methods that can be applied, each dealing with a different kind of inefficiency.

The first method removes states that cannot play any role in deciding whether a string is accepted. Let $\mathbf{A} = (S, A, i, \delta, T)$ be a finite acceptor. We

say that a state $s \in S$ is *accessible* if there is a string $x \in A^*$ such that $i \cdot x = s$. Observe that the initial state itself is always accessible because $i \cdot \varepsilon = i$. A state that is not accessible is said to be *inaccessible*. An acceptor is said to be *accessible* if every state is accessible. It is clear that the inaccessible states of an automaton can play no role in accepting strings; consequently, we expect that they could be removed without the language being changed. This turns out to be the case.

Theorem 5. *Let* $\mathbf{A}$ *be a finite acceptor. Then there is an algorithm that constructs an accessible acceptor,* $\mathbf{A}^a$*, such that* $L(\mathbf{A}^a) = L(\mathbf{A})$*.*

The second method identifies states that 'do the same job.' Let $\mathbf{A} = (S, A, i, \delta, T)$ be an acceptor. Two states $s, t \in S$ are said to be *distinguishable* if there exists $x \in A^*$ such that

$$(s \cdot x, t \cdot x) \in (T \times T') \cup (T' \times T),$$

where T' is the set of non-terminal states. In other words, for some string x, one of the states $s \cdot x$ and $t \cdot x$ is terminal and the other non-terminal. The states s and t are said to be *indistinguishable* if they are not distinguishable. This means that for each $x \in A^*$ we have that

$$s \cdot x \in T \Leftrightarrow t \cdot x \in T.$$

Define the relation $\simeq_{\mathbf{A}}$ on the set of states S by

$$s \simeq_{\mathbf{A}} t \Leftrightarrow \; s \text{ and } t \text{ are indistinguishable.}$$

We call $\simeq_{\mathbf{A}}$ the *indistinguishability relation*, and it is an equivalence relation. It can happen, of course, that each pair of states in an acceptor is distinguishable: in other words, the relation $\simeq_{\mathbf{A}}$ is equality. We say that such an acceptor is *reduced*.

Theorem 6. *Let* $\mathbf{A}$ *be a finite acceptor. Then there is an algorithm that constructs a reduced acceptor,* $\mathbf{A}^r$*, such that* $L(\mathbf{A}^r) = L(\mathbf{A})$*.*

Our two methods can be applied to an acceptor $\mathbf{A}$ in turn yielding an acceptor $\mathbf{A}^{ar} = (\mathbf{A}^a)^r$ that is both accessible and reduced. The reader may wonder at this point whether there are other methods for removing states. We shall see that there are not.

We now come to a fundamental definition. Let L be a recognisable language. A finite deterministic acceptor $\mathbf{A}$ is said to be *minimal (for L)* if $L(\mathbf{A}) = L$, and if $\mathbf{B}$ is any finite acceptor such that $L(\mathbf{B}) = L$, then the number of states of $\mathbf{A}$ is less than or equal to the number of states of $\mathbf{B}$. Minimal acceptors for a language L certainly exist. The problem is to find a way of constructing them. Our search is narrowed down by the following observation whose simple proof is left as an exercise: if $\mathbf{A}$ is minimal for L, then $\mathbf{A}$ is both accessible and reduced.

In order to realise the main goal of this section, we need to have a precise mathematical definition of when two acceptors are essentially the same: one that we can check in a systematic way however large the automata involved. The definition below provides the answer to this question.

Let $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (Q, A, q_0, \gamma, G)$ be two acceptors with the same input alphabet A. An *isomorphism* θ *from* $\mathbf{A}$ *to* $\mathbf{B}$ is a function $\theta\colon S \to Q$ satisfying the following four conditions:

(IM1) The function θ is bijective.
(IM2) $\theta(s_0) = q_0$.
(IM3) $s \in F \Leftrightarrow \theta(s) \in G$.
(IM4) $\theta(\delta(s, a)) = \gamma(\theta(s), a)$ for each $s \in S$ and $a \in A$.

If there is an isomorphism from $\mathbf{A}$ to $\mathbf{B}$ we say that $\mathbf{A}$ is *isomorphic* to $\mathbf{B}$. Isomorphic acceptors may differ in their state labelling and may look different when drawn as directed graphs, but by suitable relabelling, and by moving states and bending transitions, they can be made to look identical. Thus isomorphic automata are 'essentially the same' meaning that they differ in only trivial ways.

Theorem 7. *Let L be a recognisable language. Then L has a minimal acceptor, and any two minimal acceptors for L are isomorphic. A reduced accessible acceptor recognising L is a minimal acceptor for L.*

Remark It is worth reflecting on the significance of this theorem, particularly since in the generalisations considered later in this chapter, a rather more subtle notion of 'minimal automaton' has to be used. Theorem 7 tells us that if by some means we can find an acceptor for a language, then by applying a couple of algorithms, we can convert it into the smallest possible acceptor for that language. This should be contrasted with the situation for arbitrary problems where, if we find a solution, there are no general methods for making it more efficient, and where the concept of a smallest solution does not even make sense. The above theorem is therefore one of the benefits of working with a restricted class of operations.

The approach I have adopted to describing a minimal acceptor can be generalised in a straightforward fashion to the Moore and Mealy machines I describe in Section 3.1. However, when I come to the sequential transducers of Section 3.2, this naive approach breaks down. In this case, it is indeed possible to have two sequential transducers that do the same job, are as small as possible, but which are not isomorphic. A specific example of this phenomenon can be found in [27]. However, it transpires that we can still pick out a 'canonical machine' that also has the smallest possible number of states. The construction of this canonical machine needs slightly more sophisticated mathematics; I shall outline how this approach can be carried out for finite acceptors.

The finite acceptors I have defined are technically speaking the 'complete finite acceptors.' An *incomplete* finite acceptor is defined in the same way as a complete one except that we allow the transition function to be a partial

function. Clearly, we can convert an incomplete acceptor into a complete one by adjoining an extra state and defining appropriate transitions. However, there is no need to do this: incomplete acceptors bear the same relationship to complete ones as partial functions do to (globally defined) functions, and in computer science it is the partial functions that are the natural functions to consider. For the rest of this paragraph, 'acceptor' will mean one that could be incomplete. One way of simplifying an acceptor is to remove the inaccessible states. Another way of simplifying an acceptor is to remove those states s for which there is no string x such that $s \cdot x$ is terminal. An acceptor with the property that for each state s there is a string x such that $s \cdot x$ is terminal is said to be *coaccessible*. Clearly, if we prune an acceptor of those states that are not coaccessible, the resulting acceptor is coaccessible. The reader should observe that if this procedure is carried out on a complete acceptor, then the resulting acceptor could well be incomplete. This is why I did not define this notion earlier. Acceptors that are both accessible and coaccessible are said to be *trim*. It is possible to define what we mean by a 'morphism' between acceptors; I shall not make a formal definition here, but I will explain how they can be used. Let L be a recognisable language, and consider all the trim acceptors recognising L. If $\mathbf{A}$ and $\mathbf{B}$ are two such acceptors, it can be proved that there is *at most one* morphism from $\mathbf{A}$ to $\mathbf{B}$. If there is a morphism from $\mathbf{A}$ to $\mathbf{B}$, and from $\mathbf{B}$ to $\mathbf{A}$, then $\mathbf{A}$ and $\mathbf{B}$ are said to be 'isomorphic'; this has the same significance as my earlier definition of isomorphic. The key point now is this:

> *there is a distinguished trim acceptor $\mathbf{A}_L$ recognising L characterised by the property that for each trim acceptor $\mathbf{A}$ recognising L there is a, necessarily unique, morphism from $\mathbf{A}$ to $\mathbf{A}_L$.*

It turns out that $\mathbf{A}_L$ can be obtained from $\mathbf{A}$ by a slight generalisation of the reduction process I described earlier. By definition, $\mathbf{A}_L$ is called the *minimal acceptor for L*. It is a consequence of the defining property of $\mathbf{A}_L$ that $\mathbf{A}_L$ has the smallest number of states amongst all the trim acceptors recognising L. The reader may feel that this description of the minimal acceptor merely complicates my earlier, more straightforward, description. However, the important point is this: the characterisation of the minimal acceptor in the terms I have highlighted above generalises, whereas its characterisation in terms of having the smallest possible number of states does not. A full mathematical justification of the claims made in this paragraph can be found in Chapter III of [12].

A simple algorithm for minimising an acceptor and an algorithm for constructing a minimal acceptor from a regular expression are described in Chapter 7 of [22]. For an introduction to implementing finite acceptors and their associated algorithms, see [34].

3 Finite Transducers

In Section 2, I outlined the theory of finite acceptors. This theory tells us about devices where the response to an input is simply a 'yes' or a 'no'. In this section, I describe devices that generalise acceptors but generate outputs that provide more illuminating answers to questions. Section 3.1 describes how to modify acceptors so that they generate output. It turns out that there are two ways to do this: either to associate outputs with states, or to associate outputs with transitions. The latter approach is the one adopted for generalisations. Section 3.2 describes the most general way of generating output in a sequential fashion, and Section 3.3 describes the most general model of 'finite state devices.'

3.1 Finite purely sequential transducers

I shall begin this section by explaining how finite acceptors can be adapted to generate outputs.

A language L is defined to be a subset of some A^*, where A is any alphabet. Subsets of sets can also be defined in terms of functions. To see how, let X be a set, and let $Y \subseteq X$ be any subset. Define a function

$$\chi_Y \colon X \to \mathbf{2} = \{0, 1\}$$

by

$$\chi_Y(x) = \begin{cases} 1 \text{ if } x \in Y \\ 0 \text{ if } x \notin Y. \end{cases}$$

The function χ_Y, which contains all the information about which elements belong to the subset Y, is called the *characteristic function* of the subset Y. More generally, *any* function

$$\chi \colon X \to \mathbf{2}$$

defines a subset of X: namely, the set of all $x \in X$ such that $\chi(x) = 1$. It is not hard to see that subsets of X and characteristic functions on X are equivalent ways of describing the same thing. It follows that languages over A can be described by functions $\chi \colon A^* \to \mathbf{2}$, and vice versa.

Suppose now that L is a language recognised by the acceptor $\mathbf{A} = (Q, A, i, \delta, T)$. We should like to regard $\mathbf{A}$ as calculating the characteristic function χ_L of L. To do this, we need to make some minor alterations to $\mathbf{A}$. Rather than labelling a state as terminal, we shall instead add the label '1' to the state; thus if the state q is terminal, we shall relabel it as $q/1$. If a state q is not terminal, we shall relabel it as $q/0$. Clearly with this labelling, we can dispense with the set T since it can be recovered as those states q labelled '1'. What we have done is define a function $\lambda \colon Q \to \mathbf{2}$. Our 'automaton' is now described by the following information: $\mathbf{B} = (Q, A, \mathbf{2}, q_0, \delta, \lambda)$. To see how this automaton computes the characteristic function, we need an auxilliary function $\omega_{\mathbf{B}} \colon A^* \to (0 + 1)^*$, which is defined as follows. Let $x = x_1 \ldots x_n$ be a

string of length n over A, and let the states $\mathbf{B}$ passes through when processing x be $q_0, q_1, \ldots, q_n$. Thus

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \ldots \xrightarrow{x_n} q_n.$$

Define the string

$$\omega_{\mathbf{B}}(x) = \lambda(q_0)\lambda(q_1)\ldots\lambda(q_n).$$

Thus $\omega_{\mathbf{B}}\colon A^* \to (0+1)^*$ is a function such that

$$\omega_{\mathbf{B}}(\varepsilon) = \lambda(q_0) \text{ and } |\omega_{\mathbf{B}}(x)| = |x| + 1.$$

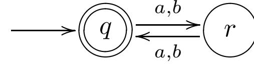
The characteristic function of the language $L(\mathbf{A})$ is the function $\rho\omega_{\mathbf{B}}\colon A^* \to \mathbf{2}$, where ρ is the function that outputs the rightmost letter of a non-empty string.

For the automaton $\mathbf{B}$, I have defined two functions: $\omega_{\mathbf{B}}\colon A^* \to (0+1)^*$, which I shall call the *output response function* of the automaton $\mathbf{B}$, and $\chi_{\mathbf{B}}\colon A^* \to \mathbf{2}$, which I shall call the *characteristic function* of the automaton $\mathbf{B}$. I shall usually just write ω and χ when the automaton in question is clear. We have noted already that $\chi = \rho\omega$. On the other hand,

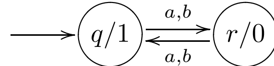
$$\omega(x_1 \ldots x_n) = \chi(\varepsilon)\chi(x_1)\chi(x_1x_2)\ldots\chi(x_1 \ldots x_n).$$

Thus knowledge of either one of ω and χ is enough to determine the other; both are legitimate output functions, and which one we use will be decided by the applications we have in mind. To make these ideas more concrete, here is an example.

Example 8. Consider the finite acceptor $\mathbf{A}$ below



The language $L(\mathbf{A})$ consists of all those strings in $(a+b)^*$ of even length. We now convert it into the automaton $\mathbf{B}$ described above

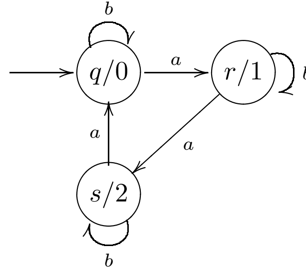


We can calculate the value of the output response function $\omega\colon (a+b)^* \to (0+1)^*$ on the string aba by observing that in processing this string we pass through the four states: q, r, q, and r. Thus $\omega(aba) = 1010$. By definition, $\chi(aba) = 0$.

There is nothing sacrosanct about the set $\mathbf{2}$ having two elements. We could just as well replace it by any alphabet B, and so view an automaton as computing functions from A^* to B. This way of generating output from an automaton leads to the following definition.

A finite *Moore machine*, $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda)$, consists of the following ingredients: Q is a finite set of states, A is the *input alphabet*, B is the *output alphabet*, q_0 is the initial state, $\delta\colon Q \times A \to Q$ is the transition function, and $\lambda\colon Q \to B$ tells us the output associated with each state. As in our special case where $B = \mathbf{2}$, we can define the *output response function* $\omega_{\mathbf{A}}\colon A^* \to B^*$ and the *characteristic function* $\chi_{\mathbf{A}}\colon A^* \to B$; as before, knowing one of these functions means we know the other. When drawing transition diagrams of Moore machines the function λ is specified on the state q by labelling this state $q/\lambda(q)$. The same idea can be used if the Moore machine is specified by a transition table.

Example 9. Here is an example of a finite Moore machine where the output alphabet has more than two letters.



Here the input alphabet $A = \{a, b\}$ and the output alphabet is $B = \{0, 1, 2\}$. In the table below, I have calculated the values of $\omega(x)$ and $\chi(x)$ for various input strings x.

x	$\omega(x)$	$\chi(x)$
ε	0	0
a	01	1
b	00	0
aa	012	2
ab	011	1
ba	001	1
bb	000	0
aaa	0120	0
aab	0122	2
aba	0112	2
abb	0111	1
baa	0012	2
bab	0011	1
bba	0001	1
bbb	0000	0

It is natural to ask under what circumstances a function $f\colon A^* \to B$ is the characteristic function of some finite Moore machine. One answer to this question is provided by the theorem below, which can be viewed as an application of Kleene's theorem. For a proof see Theorem XI.6.1 of [12].

Theorem 8. *Let $f\colon A^* \to B$ be an arbitrary function. Then there is a finite Moore machine $\mathbf{A}$ with input alphabet A and output alphabet B such that $f = \chi_{\mathbf{A}}$, the characteristic function of $\mathbf{A}$, if and only if for each $b \in B$ the language $f^{-1}(b)$ is regular.*

Moore machines are not the only way in which output can be generated. A *Mealy machine* $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda)$ consists of the following ingredients: Q is a finite set of states, A is the input alphabet, B is the output alphabet, q_0 is the initial state, $\delta\colon Q \times A \to Q$ is the transition function, and $\lambda\colon Q \times A \to B$ associates an output symbol with each transition. The *output response function* $\omega_{\mathbf{A}}\colon A^* \to B^*$ of the Mealy machine $\mathbf{A}$ is defined as follows. Let $x = x_1 \ldots x_n$ be a string of length n over A, and let the states $\mathbf{A}$ passes through when processing x be $q_0, q_1, \ldots, q_n$. Thus

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \ldots \xrightarrow{x_n} q_n.$$

Define

$$\omega_{\mathbf{A}}(x) = \lambda(q_0, x_1)\lambda(q_1, x_2) \ldots \lambda(q_{n-1}, x_n).$$

Thus $\omega_{\mathbf{A}}\colon A^* \to (0 + 1)^*$ is a function such that

$$\omega_{\mathbf{A}}(\varepsilon) = \varepsilon \text{ and } |\omega_{\mathbf{A}}(x)| = |x|.$$

Although Moore machines generate output when a state is entered, and Mealy machines during a transition, the two formalisms have essentially the same power. The simple proofs of the following two results can be found as Theorems 2.6 and 2.7 of [16].

Theorem 9. *Let A and B be finite alphabets.*

(i) *Let $\mathbf{A}$ be a finite Moore machine with input alphabet A and output alphabet B. Then there is a finite Mealy machine $\mathbf{B}$ with the same input and output alphabets and a symbol $a \in A$ such that $\chi_{\mathbf{A}} = a\chi_{\mathbf{B}}$.*
(ii) *Let $\mathbf{A}$ be a finite Mealy machine with input alphabet A and output alphabet B. Then there is a finite Moore machine $\mathbf{B}$ with the same input and output alphabets and a symbol $a \in A$ such that $\chi_{\mathbf{B}} = a\chi_{\mathbf{A}}$.*

A partial function $f\colon A^* \to B^*$ is said to be *prefix-preserving* if for all $x, y \in A^*$ such that $f(xy)$ is defined, the string $f(x)$ is a prefix of $f(xy)$. From Theorems 8 and 9, we may deduce the following characterisation of the output response functions of finite Mealy machines.

Theorem 10. *A function $f\colon A^* \to B^*$ is the output response function of a finite Mealy machine if and only if the following three conditions hold:*

(i) $|f(x)| = |x|$ *for each* $x \in A^*$.
(ii) f *is prefix-preserving.*
(iii) *The set* $f^{-1}(X)$ *is a regular subset of* A^* *for each regular subset* $X \subseteq B^*$.

Both finite Moore machines and finite Mealy machines can be minimised in a way that directly generalises the minimisation of automata described in Section 2.5. The details can be found in Chapter 7 of [9], for example.

Mealy machines provide the most convenient starting point for the further development of the theory of finite automata, so for the remainder of this section I shall concentrate solely on these. There are two ways in which the definition of a finite Mealy machine can be generalised. The first is to allow both δ, the transition function, and λ, the output associated with a transition, to be *partial* functions. This leads to what are termed *incomplete finite Mealy machines*. The second is to define λ: $Q \times A \to B^*$; in other words, we allow an input *symbol* to give rise to an output *string*. If both these generalisations are combined, we get the following definition.

A *finite (left) purely sequential transducer* $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda)$ consists of the following ingredients: Q is a finite set of states, A is the input alphabet, B is the output alphabet, q_0 is the initial state, δ: $Q \times A \to Q$ is a partial function, called the transition function, and λ: $Q \times A \to B^*$ is a partial function that associates an output string with each transition. The *output response function* $\omega_{\mathbf{A}}$: $A^* \to B^*$ of the purely sequential transducer $\mathbf{A}$ is a partial function defined as follows. Let $x = x_1 \dots x_n$ be a string of length n over A, and suppose that x labels a path in $\mathbf{A}$ that starts at the initial state; thus

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \dots \xrightarrow{x_n} q_n.$$

Define the string

$$\omega_{\mathbf{A}}(x) = \lambda(q_0, x_1)\lambda(q_1, x_2) \dots \lambda(q_{n-1}, x_n).$$

I have put the word 'left' in brackets; it refers to the fact that in the definition of δ and λ we read the input string from left to right. If instead we read the input string from right to left, we would have what is known as a *finite right purely sequential transducer*. I shall assume that a finite purely sequential transducer is a left one unless otherwise stated. A partial function f: $A^* \to B^*$ is said to be *(left) purely sequential* if it is the output response function of some finite (left) purely sequential transducer. *Right purely sequential* partial functions are defined analogously. The notions of left and right purely sequential functions are distinct, and there are partial functions that are neither.

The following theorem generalises Theorem 10 and was first proved in [15]. A proof can be found in [4] as Theorem IV.2.8.

Theorem 11. *A partial function* f: $A^* \to B^*$ *is purely sequential if and only if the following three conditions hold:*

(i) *There is a natural number* n *such that if* x *is a string in* A^*, *and* $a \in A$, *and* $f(xa)$ *is defined, then*

$$|f(xa)| - |f(x)| \leq n.$$

(ii) f *is prefix-preserving.*
(iii) *The set* $f^{-1}(X)$ *is a regular subset of* A^* *for each regular subset* $X \subseteq B^*$.

The theory of minimising finite acceptors can be extended to finite purely sequential transducers. See Chapter XII, Section 4 of [12].

The final result of this section is proved as Proposition IV.2.5 of [4].

Theorem 12. *Let* $f\colon A^* \to B^*$ *and* $g\colon B^* \to C^*$ *be left (resp. right) purely sequential partial functions. Then their composition* $g \circ f\colon A^* \to C^*$ *is a left (resp. right) purely sequential partial function.*

3.2 Finite sequential transducers

The theories of recognisable languages and purely sequential partial functions outlined in Sections 2 and 3.1 can be regarded as the classical theory of finite automata. For example, the Mealy and Moore machines discussed in Section 3.1, particularly in their incomplete incarnations, form the theoretical basis for the design of circuits. But although purely sequential functions are useful, they have their limitations. For example, binary addition cannot quite be performed by means of a finite purely sequential transducer (see Example IV.2.4 and Exercise IV.2.1 of [4]). This led Schützenberger [33] to introduce the 'finite sequential transducers' and the corresponding class of 'sequential partial functions.' The definition of a finite sequential transducer looks like a cross between finite acceptors and finite purely sequential transducers.

A *finite sequential transducer*, $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda, \tau, x_i)$, consists of the following ingredients: Q is a finite set of states, A is the input alphabet, B is the output alphabet, q_0 is the initial state, $\delta\colon Q \times A \to Q$ is a transition partial function, $\lambda\colon Q \times A \to B^*$ is an output partial function, $\tau\colon T \to B^*$ is the *termination* function, where T is a subset of Q called the set of terminal states, and $x_i \in B^*$ is the *initialisation value*.

To see how this works, let $x = x_1 \ldots x_n$ be an input string from A^*. We say that x is *successful* if it labels a path from q_0 to a state in T. For those strings $x \in A^*$ that are successful, we define an output string from B^* as follows: the initialisation value x_i is concatenated with the output response string determined by x and the function λ, just as in a finite purely sequential transducer, and then concatenated with the string $\tau(q_0 \cdot x)$. In other words, the output is computed in the same way as in a finite purely sequential transducer except that this is prefixed by a fixed string and suffixed by a final output string determined by the last state. Partial functions from A^* to B^* that can be computed by some finite sequential transducer are called *sequential*

partial functions.[1] Finite sequential transducers can be represented by suitably modified transition diagrams: the initial state is represented by an inward-pointing arrow labelled by x_i, and each terminal state t is marked by an outward-pointing arrow labelled by $\tau(t)$.

Every purely sequential function is sequential, but there are sequential functions that are not purely sequential. Just as with purely sequential transducers, finite sequential transducers can be minimised, although the procedure is necessarily more complex; see [27] and [11] for details and the Remark at the end of Section 2.5; in addition, the composition of sequential functions is sequential. A good introduction to sequential partial functions and to some of their applications is the work in [27].

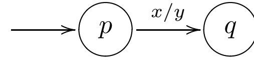
3.3 Finite transducers

In this section, we arrive at our final class of automata, which contains all the automata we have discussed so far as special cases.

A *finite transducer*, $\mathbf{T} = (Q, A, B, q_0, E, F)$, consists of the following ingredients: a finite set of states Q, an input alphabet A, an output alphabet B, an initial state q_0,[2] a set of final or terminal states F, and a set E of transitions where

$$E \subseteq Q \times A^* \times B^* \times Q.$$

A finite transducer can be represented by means of a transition diagram where each transition has the form



where $(p, x, y, q) \in E$. As usual, we indicate the initial state by an inward-pointing arrow and the final states by double circles.

To describe what a finite transducer does, we need to introduce some notation. Let

$$e = (q_1, x_1, y_1, q_1') \dots (q_n, x_n, y_n, q_n')$$

be any sequence of transitions. The state q_1 will be called the *beginning of* e and the state q_n' will be called the *end of* e. The *label* of e is the pair of strings

$$(x_1 \dots x_n, y_1 \dots y_n).$$

If e is the empty string then its label is $(\varepsilon, \varepsilon)$. We say that a sequence of transitions e is *allowable* if it describes an actual path in $\mathbf{T}$; this simply means that for each consecutive pair

$$(q_i, x_i, y_i, q_i')(q_{i+1}, x_{i+1}, y_{i+1}, q_{i+1}')$$

[1] Berstel [4] does not include in his definition the string x_i (alternatively, he assumes that $x_i = \varepsilon$). However, the class of partial functions defined is the same.

[2] Sometimes a *set* of initial states is allowed; this does not change the theory.

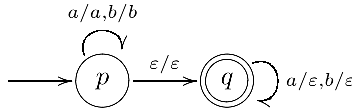
in e we have that $q_i' = q_{i+1}$. An allowable sequence e is said to be *successful* if it begins at the initial state and ends at a terminal state. Define the relation

$$R(\mathbf{T}) = \{(x, y) \in A^* \times B^* \colon (x, y) \text{ is the label of a successful path in } \mathbf{T}\}.$$

We call $R(\mathbf{T})$ the *relation computed by* $\mathbf{T}$.

Observe that in determining the $y \in B^*$ such that $(x, y) \in R(\mathbf{T})$ for a given x, the transducer $\mathbf{T}$ processes the string x in the manner of an ε-acceptor. Thus we need to search for those paths in $\mathbf{T}$ starting at the initial state and ending at a terminal state such that the sequence of labels $(a_1, b_1), \ldots, (a_n, b_n)$ encountered has the property that the concatenation $a_1 \ldots a_n$ is equal to x where some of the a_i may well be ε.

Example 10. The following is a transition diagram of a transducer $\mathbf{T}$



In this case, the input and output alphabets are the same and equal $\{a, b\}$. The relation $R(\mathbf{T}) \subseteq (a + b)^* \times (a + b)^*$ computed by $\mathbf{T}$ is the set of all pairs of strings (x, y) such that y is a prefix of x. This is a relation rather than a function because a non-empty string has more than one prefix.

The relations in $A^* \times B^*$ that can be computed by finite transducers can be described in a way that generalises Kleene's theorem. To see how, we need to define what we mean by a 'regular' or 'rational' subset of $A^* \times B^*$. If $(x_1, y_1), (x_2, y_2) \in A^* \times B^*$, then we define their *product* by

$$(x_1, y_1)(x_2, y_2) = (x_1x_2, y_1y_2).$$

This operation is the analogue of concatenation in A^*. Observe that $(\varepsilon, \varepsilon)$ has the property that $(\varepsilon, \varepsilon)(x, y) = (x, y) = (x, y)(\varepsilon, \varepsilon)$. If $L, M \subseteq A^* \times B^*$, then define LM to be the set of all products of elements in L followed by elements in M. With these preliminaries out of the way, we can define a *regular* or *rational* subset of $A^* \times B^*$ in a way analogous to the definition of a regular subset given in Section 2.4. A regular subset of $A^* \times B^*$ is also called a *regular* or *rational relation* from A^* to B^*.

The following result is another application of Kleene's Theorem; see Theorem III.6.1 of [4] for a proof and [29] for the correct mathematical perspective on finite transducers.

Theorem 13. *A relation $R \subseteq A^* \times B^*$ can be computed by a finite transducer with input alphabet A and output alphabet B if and only if R is a regular relation from A^* to B^*.*

In the Introduction, I indicated that there were simple computations that finite transducers could not do. A good example is that of reversing a string; see Exercise III.3.2 of [4].

The theory of finite transducers is more complex than that of finite acceptors. In what follows, I just touch on some of the key points.

The following is proved as Theorem III.4.4 of [4].

Theorem 14. *Let A, B, C be finite alphabets. Let R be a regular relation from A^* to B^* and let R' be a regular relation from B^* to C^*. Then $R' \circ R$ is a regular relation from A^* to C^*, where $(a, c) \in R' \circ R$ iff there exists $b \in B^*$ such that $(a, b) \in R'$ and $(b, c) \in R$.*

Let R be a regular relation from A^* to B^*. Given a string $x \in A^*$, there may be no strings y such that $(x, y) \in R$; there might be exactly one such string y; or they might be many such strings y. If a relation R from A^* to B^* has the property that for each $x \in A^*$ there is at most one element $y \in B^*$ such that $(x, y) \in R$, then R can be regarded as a partial function from A^* to B^*. Such a function is called a *regular* or *rational partial function*.

It is important to remember that a regular relation that is not a regular partial function is not, in some sense, deficient; there are many situations where it would be unrealistic to expect a partial function. For example, in natural language processing, regular relations that are not partial functions can be used to model ambiguity of various kinds. However, regular partial functions are easier to handle. For example, there is an algorithm that will determine whether two regular partial functions are equal or not (Corollary IV.1.3 [4]), whereas there is no such algorithm for arbitrary regular relations (Theorem III.8.4(iii) [4]). Classes of regular relations sharing some of the advantages of sequential partial functions are described in [1] and [23].

Theorem 15. *There is an algorithm that will determine whether the relation computed by a finite transducer is a partial function or not.*

This was first proved by Schützenberger [32], and a more recent paper [3] also discusses this question.

Both left and right purely sequential functions are examples of regular partial functions, and there is an interesting relationship between arbitrary regular partial functions and the left and right purely sequential ones. The following is proved as Theorem IV.5.2 of [4].

Theorem 16. *Let $f \colon A^* \to B^*$ be a partial function such that $f(\varepsilon) = \varepsilon$. Then f is regular if and only if there is an alphabet C and a left purely sequential partial function $f_L \colon A^* \to C^*$ and a right purely sequential partial function $f_R \colon C^* \to B^*$ such that $f = f_R \circ f_L$.*

The idea behind the theorem is that to compute $f(x)$ we can first process x from left to right and then from right to left. Minimisation of machines that

compute regular partial functions is more complex and less clear-cut than for the sequential ones; see [28] for some work in this direction.

The sequential functions are also regular partial functions. The following definition is needed to characterise them. Let x and y be two strings over the same alphabet. We denote by $x \wedge y$ the *longest common prefix of x and y*. We define the *prefix distance* between x and y by

$$d(x, y) = |x| + |y| - 2|x \wedge y|.$$

In other words, if $x = ux'$ and $y = uy'$, where $u = x \wedge y$, then $d(x, y) = |x'| + |y'|$. A partial function $f \colon A^* \to B^*$ is said to have *bounded variation* if for each natural number n there exists a natural number N such that, for all strings $x, y \in A^*$,

$$\text{if } d(x, y) \leq n \text{ then } d(f(x), f(y)) \leq N.$$

Theorem 17. *Every sequential function is regular. In particular, the sequential functions are precisely the regular partial functions with bounded variation.*

The proof of the second claim in the above theorem was first given by Choffrut [10]. Both proofs can be found in [4] (respectively, Proposition IV.2.4 and Theorem IV.2.7).

Theorem 18. *There is an algorithm that will determine whether a finite transducer computes a sequential function.*

This was first proved by Choffrut [10], and more recent papers that discuss this question are [2] and [3].

Finite transducers were introduced in [13] and, as we have seen, form a general framework containing the purely sequential and sequential transducers.

4 Final Remarks

In this section, I would like to touch on some of the practical reasons for using finite transducers. But first, I need to deal with the obvious objection to using them: that they cannot implement all algorithms, because they do not constitute a universal programming language. However, it is the very lack of ambition of finite transducers that leads to their virtues: we can say more about them, and what we can say can be used to help us design programs using them. The manipulation of programs written in universal programming languages, on the other hand, is far more complex. In addition:

- Not all problems require for their solution the full weight of a universal programming language — if we can solve them using finite transducers then the benefits described below will follow.

- Even if the full solution of a problem does fall outside the scope of finite transducers, the cases of the problem that are of practical interest may well be described by finite transducers. Failing that, partial or approximate solutions that can be described by finite transducers may be acceptable for certain purposes.
- It is one of the goals of science to understand the nature of problems. If a problem can be solved by a finite transducer then we have learnt something non-trivial about the nature of that problem.

The benefits of finite transducers are particularly striking in the case of finite acceptors:

- Finite acceptors provide a way to describe potentially infinite languages in finite ways.
- Determining whether a string is accepted or rejected by a deterministic acceptor is linear in the length of the string.
- Acceptors can be both determinised and minimised.

The important point to bear in mind is that languages are interesting because they can be used to encode structures of many kinds. An example from mathematics may be instructive. A *relational structure* is a set equipped with a finite collection of relations of different arities. For example, a set equipped with a single binary relation is just a graph with at most one edge joining any two vertices. We say that a relational structure is *automatic* if the elements of the set can be encoded by means of strings from a regular language, and if each of the n-ary relations of the structure can be encoded by means of an acceptor. Encoding n-ary relations as languages requires a way of encoding n-tuples of strings as single strings, but this is easy and the details are not important; see [19] for the complete definition and [8] for a technical analysis of automatic structures from a logical point of view. Minimisation means that encoded structures can be compressed with no loss of information. Now there are many ways of compressing data, but acceptors provide an additional advantage: they come with a built-in search capability. The benefits of using finite acceptors generalise readily to finite sequential transducers.

There is a final point that is worth noting. Theorem 14 tells us that composing a sequence of finite transducers results in another finite transducer. This can be turned around and used as a design method; rather than trying to construct a finite transducer in one go, we can try to design it as the composition of a sequence of simpler finite transducers.

The books [18, 20, 31] although dealing with natural language processing contain chapters that may well provide inspiration for applications of finite transducers to other domains.

Acknowledgements It is a pleasure to thank a number of people who helped in the preparation of this article.

Dr Karin Haenelt of the Fraunhofer Gesellschaft, Darmstadt, provided considerable insight into the applications of finite transducers in natural language processing which contributed in formulating Section 4.

Dr Jean-Eric Pin, director for research of CNRS, made a number of suggestions including the choice of terminology and the need to clarify what is meant by the notion of minimisation.

Dr Anne Heyworth of the University of Leicester made a number of useful textual comments as well as providing the automata diagrams.

Prof John Fountain and Dr Victoria Gould of the University of York made a number of helpful comments on the text.

My thoughts on the role of finite transducers in information processing were concentrated by a report I wrote for DSTL in July 2003 (Contract number RD026-00403) in collaboration with Peter L. Grainger of DSTL.

Any errors remaining are the sole responsibility of the author.

References

1. C. Allauzen and M. Mohri, Finitely subsequential transducers, *International J. Foundations Comp. Sci.* **14** (2003), 983–994.
2. M.-P. Béal and O. Carton, Determinization of transducers over finite and infinite words, *Theoret. Comput. Sci.* **289** (2002), 225–251.
3. M.-P. Béal, O. Carton, C. Prieur, and J. Sakarovitch, Squaring transducers, *Theoret. Comput. Sci.* **292** (2003), 45–63.
4. J. Berstel, *Transductions and Context-free Languages*, B.G. Teubner, Stuttgart, 1979.
5. J. Berstel and J.-E. Pin, Local languages and the Berry-Sethi algorithm, *Theoret. Comput. Sci.* **155** (1996), 439–446.
6. J. Berstel and D. Perrin, Algorithms on words, in *Applied Combinatorics on Words* (editor M. Lothaire), in preparation, 2004.
7. A. Brüggemann-Klein, Regular expressions into finite automata, *Lecture Notes in Computer Science* **583** (1992), 87–98.
8. A. Blumensath, *Automatic structures*, Diploma Thesis, Rheinisch-Westfälische Technische Hochschule Aachen, Germany, 1999.
9. J. Carroll and D. Long, *Theory of Finite Automata*, Prentice-Hall, Englewood Cliff, NJ, 1989.
10. Ch. Choffrut, Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationelles, *Theoret. Comput. Sci.* **5** (1977), 325–337.
11. Ch. Choffrut, Minimizing subsequential transducers: a survey, *Theoret. Comput. Sci.* **292** (2003), 131–143.
12. S. Eilenberg, *Automata, Languages, and Machines*, Volume A, Academic Press, New York, 1974.
13. C. C. Elgot and J. E. Mezei, On relations defined by generalized finite automata, *IBM J. Res. Develop.* **9** (1965), 47–65.
14. J. E. F. Friedl, *Mastering regular expressions*, O'Reilly, Sebastopol, CA, 2002.
15. S. Ginsburg and G. F. Rose, A characterization of machine mappings, *Can. J. Math.* **18** (1966), 381–388.

16. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.
17. J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd Edition, Addison-Wesley, Reading, MA, 2001.
18. D. Jurafsky and J. H. Martin, *Speech and Language Processing*, Prentice-Hall, Englewood Cliff, NJ, 2000.
19. B. Khoussainov and A. Nerode, Automatic presentations of structures, *Lecture Notes in Computer Science* **960** (1995), 367–392.
20. A. Kornai (editor), *Extended Finite State Models of Language*, Cambridge University Press, London, 1999.
21. E. Laporte, Symbolic natural language processing, in *Applied Combinatorics on Words* (editor M. Lothaire), in preparation, 2004.
22. M. V. Lawson, *Finite Automata*, CRC Press, Boca Raton, FL, 2003.
23. M. Mohri, Finite-state transducers in language and speech processing, *Comput. Linguistics* **23** (1997), 269–311.
24. M. Pelletier and J. Sakarovitch, On the representation of finite deterministic 2-tape automata, *Theoret. Comput. Sci.* **225** (1999), 1–63.
25. D. Perrin, Finite automata, in *Handbook of Theoretical Computer Science*, Volume B (editor J. Van Leeuwen), Elsevier, Amsterdam, 1990, 1–57.
26. D. Perrin and J. E. Pin, *Infinite Words*, Elsevier, Amsterdam, 2004.
27. Ch. Reutenauer, Subsequential functions: characterizations, minimization, examples, *Lecture Notes in Computer Science* **464** (1990), 62–79.
28. Ch. Reutenauer and M.-P. Schützenberger, Minimization of rational word functions, *Siam. J. Comput.* **20** (1991), 669–685.
29. J. Sakarovitch, Kleene's theorem revisited, *Lecture Notes in Computer Science* **281** (1987), 39–50.
30. J. Sakarovitch, *Eléments de Théorie des Automates*, Vuibert, Paris, 2003.
31. E. Roche and Y. Schabes (editors), *Finite-State Language Processing*, The MIT Press, 1997.
32. M.-P. Schützenberger, Sur les relations rationelles, in *Automata theory and formal languages* (editor H. Brakhage), *Lecture Notes in Computer Science* **33** (1975), 209–213.
33. M.-P. Schützenberger, Sur une variante des fonctions séquentielles, *Theoret. Comput. Sci.* **4** (1977), 47–57.
34. B. W. Watson, Implementing and using finite automata toolkits, in *Extended Finite State Models of Language* (editor A. Kornai), Cambridge University Press, London, 1999, 19–36.
35. Sheng Yu, Regular languages, in *Handbook of Formal Languages*, Volume 1 (editors G. Rozenberg and A. Salomaa), Springer-Verlag, Berlin, 1997, 41–110.