Chapter 1

Introduction and Basic Definitions

This chapter introduces the concept of a finite automaton, which is perhaps the simplest form of abstract computing device. Although finite automata theory is concerned with relatively simple machines, it is an important foundation of a large number of concrete and abstract applications. The finite-state control of a finite automaton is also at the heart of more complex computing devices such as finite-state transducers (Chapter 7), pushdown automata (Chapter 10), and Turing machines (Chapter 11).

Applications for finite automata can be found in the algorithms used for string matching in text editors and spelling checkers and in the lexical analyzers used by assemblers and compilers. In fact, the best known string matching algorithms are based on finite automata. Although finite automata are generally thought of as abstract computing devices, other non-computer applications are possible. These applications include traffic signals and vending machines or any device in which there are a finite set of inputs and a finite set of things that must be “remembered” by the device.

Briefly, a deterministic finite automaton, also called a recognizer or acceptor, is a mathematical model of a finite-state computing device that recognizes a set of words over some alphabet; this set of words is called the language accepted by the automaton. For each word over the alphabet of the automaton, there is a unique path through the automaton; if the path ends in what is called a final or accepting state, then the word traversing this path is in the language accepted by the automaton.

Finite automata represent one attempt at employing a finite description to rigorously define a (possibly) infinite set of words (that is, a language). Given such a description, the criterion for membership in the language is straightforward and well-defined; there are simple algorithms for ascertaining whether a given word belongs to the set. In this respect, such devices model one of the behaviors we require of a compiler: recognizing syntactically correct programs. Actually, finite automata have inherent limitations that make them unsuitable for modeling the compilers of modern programming languages, but they serve as an instructive first approximation. Compilers must also be capable of producing object code from source code, and a model of a simple translation device is presented in Chapter 7 and enhanced in later chapters.

Logic circuitry can easily be devised to implement these automata in hardware. With appropriate data structures, these devices can likewise be modeled with software. An example is the highly interactive Turing’s World©, developed at Stanford University by Jon Barwise and John Etchemendy. This Apple® Macintosh graphics package and the accompanying tutorial are particularly useful in experimenting with many forms of automata. Both hardware and software approaches will be explored in this chapter. We begin our formal treatment with some fundamental definitions.
1.1 Alphabets and Words

The devices we will consider are meant to react to and manipulate symbols. Different applications may employ different character sets, and we will therefore take care to explicitly mention the alphabet under consideration.

**Definition 1.1** \( \Sigma \) is an alphabet \( \iff \) \( \Sigma \) is a finite nonempty set of symbols.

An element of an alphabet is often called a letter, although there is no reason to restrict symbols in an alphabet to consist solely of single characters. Some familiar examples of alphabets are the 26-letter English alphabet and the ASCII character set, which represents a standard set of computer codes. In this text we will usually make use of shorter, simpler alphabets, like those given in Example 1.1.

**Example 1.1**

i. \( \{0,1\} \)

ii. \( \{a,b,c\} \)

iii. \( \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\} \)

It is important to emphasize that the elements (letters) of an alphabet are not restricted to single characters. In example (iii.) above, the alphabet is composed of the ordered pairs in \( \{0,1\} \times \{0,1\} \). Such an alphabet will be utilized in Chapter 7 when we use sequential machines to construct a simple binary adder.

Based on the definition of an alphabet, we can define composite entities called words or strings, which are finite sequences of symbols from the alphabet.

**Definition 1.2** For a given alphabet \( \Sigma \) and a natural number \( n \), a sequence of symbols \( a_1 a_2 \ldots a_n \) is a word (or string) over the alphabet \( \Sigma \) of length \( n \) \( \iff \) for each \( i = 1,2,\ldots,n \), \( a_i \in \Sigma \).

As formally specified in Definition 1.5, the order in which the symbols of the word occur will be deemed significant, and therefore a word of length 3 can be identified with an ordered triple belonging to \( \Sigma \times \Sigma \times \Sigma \). Indeed, one may view the three-letter word \( bca \) as a convenient shorthand for the ordered triple \( \langle b,c,a\rangle \). A word over an alphabet is thus an ordered string of symbols, where each symbol in the string is an element of the given alphabet. An obvious example of words is what you are reading right now, which are words (or strings) over the standard English alphabet. In some contexts, these strings of symbols are occasionally called sentences.

**Example 1.2**

Let \( \Sigma = \{0,1,2,3,4,5,6,7,8,9\} \); some examples of words over this alphabet are

i. \( 42 \)

ii. \( 242342 \)

Even though only three different members of \( \Sigma \) occur in the second example, the length of \( 242342 \) is 6, as each symbol is counted each time it occurs. To easily and succinctly express these concepts, the absolute value notation will be employed to denote the length of a string. Thus, \( |42| = 2 \), \( |242342| = 6 \), and \( |a_1 a_2 a_3 a_4| = 4 \).
Definition 1.3  For a given alphabet $\Sigma$ and a word $x = a_1a_2 \ldots a_n$ over $\Sigma$, $|x|$ denotes the length of $x$. That is, $|a_1a_2 \ldots a_n| = n$.

It is possible to join together two strings to form a composite word; this process is called concatenation. The concatenation of two strings of symbols produces one longer string of symbols, which is made up of the characters in the first string, followed immediately by the symbols of the second string.

Definition 1.4  Given an alphabet $\Sigma$, let $x = a_1 \ldots a_n$ and $y = b_1 \ldots b_m$ be strings where each $a_i \in \Sigma$ and each $b_j \in \Sigma$. The concatenation of the strings $x$ and $y$, denoted by $x \cdot y$, is the juxtaposition of $x$ and $y$; that is, $x \cdot y = a_1 \ldots a_n b_1 \ldots b_m$.

Note in Definition 1.4 that $|x \cdot y| = n + m = |x| + |y|$. Some examples of string concatenation are

i. $aaa \cdot bbb = aaabb$

ii. $\text{home} \cdot \text{run} = \text{homerun}$

iii. $a^2 \cdot b^3 = aabb$  

Example (iii) illustrates a shorthand for denoting strings. Placing a superscript after a symbol means that this entity is a string made by concatenating it to itself the specified number of times. In a similar fashion, $(ac)^3$ is meant to express $acacac$. Note that an equal sign was used in the above examples. Formally, two strings are equal if they have the same number of symbols and these symbols match, character for character.

Definition 1.5  Given an alphabet $\Sigma$, let $x = a_1 \ldots a_n$ and $y = b_1 \ldots b_m$ be strings over $\Sigma$. $x$ and $y$ are equal iff $n = m$ and for each $i = 1, 2, \ldots, n$, $a_i = b_i$.

The operation of concatenation has certain algebraic properties: it is associative, and it is not commutative. That is,

i. $(\forall x \in \Sigma^*)(\forall y \in \Sigma^*)(\forall z \in \Sigma^*)x \cdot (y \cdot z) = (x \cdot y) \cdot z$.

ii. For most strings $x$ and $y$, $x \cdot y \neq y \cdot x$.

When the operation of concatenation is clear from the context, we will adopt the convention of omitting the symbol for the operator (as is done in arithmetic with the multiplication operator). Thus $xyz$ refers to $x \cdot y \cdot z$. In fact, in Chapter 6 it will be seen that the operation of concatenation has many algebraic properties that are similar to those of arithmetic multiplication.

It is often necessary to count the number of occurrences of a given symbol within a word. The notation described in the next definition will be an especially useful shorthand in many contexts.

Definition 1.6  Given an alphabet $\Sigma$, and some $b \in \Sigma$, the length of a word $w$ with respect to $b$, denoted $|w|_b$, is the number of occurrences of the letter $b$ within that word.
Example 1.3

i. $|abb|_b = 2$

ii. $|abb|_c = 0$

iii. $|10000000111888188888881000000001118881888888|_1 = 5$

Definition 1.7  Given an alphabet $\Sigma$, the empty word, denoted by $\lambda$, is defined to be the (unique) word consisting of zero letters.

The empty word is often denoted by $\epsilon$ in many formal language texts. The empty string serves as the identity element for concatenation. That is, for all strings $x$,

$$x \cdot \lambda = \lambda \cdot x = x$$

Even though the empty word is represented by a single character, $\lambda$ is a string but is not a member of any alphabet: $\lambda \not\in \Sigma$. (Symbols in an alphabet all have length one; $\lambda$ has length zero.)

A particular string $x$ can be divided into substrings in several ways. If we choose to break $x$ up into three substrings $u$, $v$, and $w$, there are many ways to accomplish this. For example, if $x = abccdbc$, it could be written as $ab \cdot ccd \cdot bc$; that is, $x = uvw$, where $u = ab$, $v = ccd$, and $w = bc$. This $x$ could also be written as $abc \cdot \lambda \cdot cdbc$, where $u = abc$, $v = \lambda$, and $w = cdbc$. In this second case, $|x| = 7 = 3 + 0 + 4 = |u| + |v| + |w|$.

A fundamental structure in formal languages involves sets of words. A simple example of such a set is $\Sigma^k$, the collection of all words of exactly length $k$ (for some $k \in \mathbb{N}$) that can be constructed from the letters of $\Sigma$.

Definition 1.8  Given an alphabet $\Sigma$ and a nonnegative integer $k \in \mathbb{N}$, we define

$$\Sigma^k = \{x \mid x \text{ is a word over } \Sigma \text{ and } |x| = k\}$$

Example 1.4

If

$$\Sigma = \{0, 1\}$$

then

$$\Sigma^0 = \{\lambda\}$$
$$\Sigma^1 = \{0, 1\}$$
$$\Sigma^2 = \{00, 01, 10, 11\}$$
$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

$\lambda$ is the only element of $\Sigma^0$, the set of all words containing zero letters from $\Sigma$. There is no difficulty in letting $\lambda$ be an element (and the only element) of $\Sigma^0$, since each $\Sigma^k$ is not necessarily an alphabet, but is instead a set of words; $\lambda$, according to the definition, is indeed a word consisting of zero letters.
Definition 1.9  Given an alphabet $\Sigma$, define

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$$

and

$$\Sigma^+ = \bigcup_{k=1}^{\infty} \Sigma^k = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$$

$\Sigma^*$ is the set of all words that may be constructed from the letters of an alphabet $\Sigma$. $\Sigma^+$ is the set of all nonempty words that may be constructed from $\Sigma$.

$\Sigma^*$, like the set of natural numbers, is an infinite set. Although $\Sigma^*$ is infinite, each word in $\Sigma^*$ is of finite length. This property follows from the definition of $\Sigma^*$ and a property of natural numbers: any $k \in \mathbb{N}$ must by definition be a finite number. $\Sigma^*$ is defined to be the union of all $\Sigma^k$, $k \in \mathbb{N}$. Since each such $k$ is a finite number and every word in $\Sigma^k$ is of length $k$, then every word in $\Sigma^k$ must be of finite length. Furthermore, since $\Sigma^*$ is the union of all such $\Sigma^k$, every word in $\Sigma^*$ must also be of finite length. While $\Sigma^*$ can contain arbitrarily long words, each of these words must be finite, just as every number in $\mathbb{N}$ is finite.

Since $\Sigma^*$ is the union of all $\Sigma^k$ for $k \in \mathbb{N}$, $\Sigma^*$ must also contain $\Sigma^0$. In other words, besides containing all words that can be constructed from one or more letters of $\Sigma$, $\Sigma^*$ also contains the empty word $\lambda$. While $\lambda \notin \Sigma$, $\lambda \in \Sigma^*$. $\lambda$ represents a string and not a symbol, and thus the empty string cannot be in the alphabet $\Sigma$. However, $\lambda$ is included in $\Sigma^*$, since $\Sigma^*$ is not just an alphabet, but a collection of words over the alphabet $\Sigma$. Note, however, that $\Sigma^+$ is $\Sigma^* - \{\lambda\}$; $\Sigma^+$ specifically excludes $\lambda$.

1.2 Definition of a Finite Automaton

We now have the building blocks necessary to define deterministic finite automata. A deterministic finite automaton is a mathematical model of a machine that accepts a particular set of words over some alphabet $\Sigma$.

A useful visualization of this concept might be referred to as the black box model. This conceptualization is built around a black box that houses the finite-state control. This control reacts to the information provided by the read head, which extracts data from the input tape. The control also governs the operation of the output indicator, often depicted as an acceptance light, as shown in Figure 1.1.

There is no limit to the number of symbols that can be on the tape (although each individual word must be of finite length). As the input tape is read by the machine, state transitions, which alter the current state of the automaton, take place within the black box. Depending on the word contained on the input tape, the light bulb either lights or remains dark when the end of the input string is reached, indicating acceptance or rejection of the word, respectively. We assume that the input head can sense when it has passed the last symbol on the tape. In some sense, a personal computer fits the finite-state control model; it reacts to each keystroke entered from the keyboard according to the current state of the CPU and its own internal memory. However, the number of possible bit patterns that even a small computer can assume is so astronomically large that it is totally impractical to model a computer in this fashion. Finite-state machines can be profitably used to describe portions of a computer (such as parts of the arithmetic/logic unit, as discussed in Chapter 7, Example 7.15) and other devices that assume a reasonable number of states.
Although finite automata are usually thought of as processing strings of letters over some alphabet, the input can conceptually be elements from any finite set. A useful example is the “brain” of a vending machine, which, say, dispenses 30¢ candy bars.

Example 1.5

The input to the vending machine is the set of coins {nickel, dime, quarter}, represented by \( n, d, \) and \( q \) in Figure 1.2. The machine may only “remember” a finite number of things; in this case, it will keep track of the amount of money that has been dropped into the machine. Thus, the machine may be in the “state” of remembering that no money has yet been deposited (denoted in this example by \(<0¢>\)), or that a single nickel has been inserted (the state labeled \(<5¢>\), or that either a dime or two nickels have been deposited \(<10¢>\), and so on. Note that from state \(<0¢>\) there is an arrow labeled by the dime token \( d \) pointing to the state \(<10¢>\), indicating that, at a time when the machine “believes” that no money has been deposited, the insertion of a dime causes the machine to transfer to the state that remembers that ten cents has been deposited. From the \(<0¢>\) state, the arrows in the diagram show that if two nickels \((n)\) are input the machine moves through the \(<5¢>\) state and likewise ends in the state labeled \(<10¢>\).

The vending machine thus counts the amount of change dropped into the machine (up to 50¢). The machine begins in the state labeled \(<0¢>\) and follows the arrows to higher-numbered states as coins are inserted. For example, depositing a nickel, a dime, and then a quarter would move the machine to the states \(<5¢>, <15¢>, <40¢>\), and then \(<50¢>\). The states labeled 30¢ and above are doubly encircled to indicate that enough money has been deposited; if 30¢ or more has been deposited, then the machine “accepts,” indicating that a candy bar may be selected.

Finite automata are appropriate whenever there are a finite number of inputs and only a finite number of situations must be distinguished by the machine. Other applications include traffic signals and elevators (as discussed in Chapter 7). We now present a formal mathematical definition of a finite-state machine.

Definition 1.10 A deterministic finite automaton or deterministic finite acceptor (DFA) is a quintuple \( \langle \Sigma, S, s_0, \delta, F \rangle \), where

i. \( \Sigma \) is the input alphabet \( (\text{a finite nonempty set of symbols}) \).

ii. \( S \) is a finite nonempty set of states.
iii. $s_0$ is the start (or initial) state, an element of $S$.

iv. $\delta$ is the state transition function; $\delta: S \times \Sigma \rightarrow S$.

v. $F$ is the set of final (or accepting) states, a (possibly empty) subset of $S$.

The input alphabet, $\Sigma$, for any deterministic finite automaton $A$, is the set of symbols that can appear on the input tape. Each successive symbol in a word will cause a transition from the present state to another state in the machine. As specified by the $\delta$ function, there is exactly one such state transition for each combination of a symbol $a \in \Sigma$ and a state $s \in S$. This is the origin of the word “deterministic” in the phrase “deterministic finite automaton.”

The various states represent the memory of the machine. Since the number of states in the machine is finite, the number of distinguishable situations that can be remembered by the machine is also finite. This limitation of the device’s ability to store its past history is the origin of the word “finite” in the phrase “deterministic finite automaton.” At any given time during processing, if the previous history of the machine is considered to be the reactions of the DFA to the letters that have already been read, then the current state represents all that is known about the history of the machine.

The start state of the machine is the state in which the machine always begins processing a string. From this state, successive input symbols from $\Sigma$ are used by the $\delta$ function to arrive at successive states in the machine. Processing stops when the string of symbols is exhausted. The state in which the machine is left can either be a final state, in which case the word is accepted, or it can be any one of the other states of $S$, in which case the word is rejected.

To produce a formal description of the concepts defined above, it is necessary to enumerate each part of the quintuple that comprises the DFA. $\Sigma$, $S$, $s_0$, and $F$ are easily enumerated, but the function $\delta$ can often be tedious to describe. One device used to display the mapping $\delta$ is the state transition diagram. Besides graphically displaying the transitions of the $\delta$ function, the state transition diagram for a deterministic finite automaton also illustrates the other four parts of the quintuple.

A finite automaton state transition diagram is a directed graph. The states of the machine represent the vertices of the graph, while the mapping of the $\delta$ function describes the edges. Final states are de-
Figure 1.3: The DFA described in Example 1.6

```
type
  Sigma = 'a'..'c';
  State = (s0,s1,s2);
var
  TransitionTable=array[State,Sigma] of State;
```

```
function Delta(S:State;A:Sigma):State;
begin
  Delta := TransitionTable [S, A]
end; {Delta}
```

Figure 1.4: A Pascal implementation of a state transition function

noted by a doubly encircled state, and the start state is identified by a straight incoming arrow. Each domain element of the transition function corresponds to an edge in the directed graph. We formally define a finite automaton state transition diagram for \( \langle \Sigma, S, s_0, \delta, F \rangle \) as a directed graph \( G = \langle V, E \rangle \), as follows:

i. \( V = S \),

ii. \( E = \{ (s, t, a) \mid s, t \in S, a \in \Sigma \land \delta(s, a) = t \} \),

where \( V \) is the set of vertices of the graph, and \( E \) is the set of edges connecting these vertices. Each element of \( E \) is an ordered triple, \( (s, t, a) \), such that \( s \) is the origin vertex, \( t \) is the terminus, and \( a \) is the letter from \( \Sigma \) labeling the edge. Thus, for any vertex there is exactly one edge leaving that vertex for each element of \( \Sigma \).

**Example 1.6**

In the DFA shown in Figure 1.3, the set of edges \( E \) of the graph \( G \) is given by \( E = \{ (s_0, s_1, a), (s_0, s_2, b), (s_1, s_1, a), (s_1, s_2, b), (s_2, s_1, a), (s_2, s_0, b) \} \). The figure also shows that \( s_0 \) is the designated start state and that \( s_1 \) is the only final state. The state transition function for a finite automaton is often represented in the form of a *state transition table*. A state transition table is a matrix with the rows of the matrix labeled and
indexed by the states of the machine, and the columns of the matrix labeled and indexed by the elements of the input alphabet; the entries in the table are the states to which the DFA will move. Formally, let $T$ be a state transition table for some deterministic finite automaton $A = \langle \Sigma, S, s_0, \delta, F \rangle$, and let $s \in S$ and $a \in \Sigma$. Then the value of each matrix entry is given by the equation

$$\forall s \in S \forall a \in \Sigma T_{sa} = \delta(s, a)$$

For the automaton in Example 1.6, the state transition table is

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_1$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_1$</td>
<td>$s_0$</td>
</tr>
</tbody>
</table>

This table represents the following transitions:

$$\delta(s_0, a) = s_1 \quad \delta(s_0, b) = s_2$$
$$\delta(s_1, a) = s_1 \quad \delta(s_1, b) = s_2$$
$$\delta(s_2, a) = s_1 \quad \delta(s_2, b) = s_0$$

State transition tables are the most common method of representing the basic structure of an automaton within a computer. When represented as an array in the memory of the computer, access is very fast and the structure lends itself easily to manipulation by the computer. Techniques such as depth-first search are easily and efficiently implemented when the state transition diagram is represented as a table. Figure 1.4 illustrates an implementation of the $\delta$ function via transition tables in Pascal.

With $\delta$, we can describe the state in which we will find ourselves after processing a single letter. We also want to be able to describe the state at which we will arrive after processing an entire string. We will extend the $\delta$ function to cover entire strings rather than just single letters; $\overline{\delta}(s, x)$ will be the state we wind up at when starting at $s$ and processing, in order, all the letters of the string $x$. While this is a relatively easy concept to (vaguely) state in English, it is somewhat awkward to formally define. To facilitate formal proofs concerning DFAs, we use the following recursive definition.

**Definition 1.11** Given a DFA $A = \langle \Sigma, S, s_0, \delta, F \rangle$, the extended state transition function for $A$, denoted $\overline{\delta}$, is a function $\overline{\delta} : S \times \Sigma^* \rightarrow S$ defined recursively as follows:

i. $\forall s \in S \forall a \in \Sigma \overline{\delta}(s, a) = \delta(s, a)$

ii. $\forall s \in S \overline{\delta}(s, \lambda) = s$

iii. $\forall s \in S \forall x \in \Sigma^* \forall a \in \Sigma \overline{\delta}(s, ax) = \overline{\delta}(\delta(s, a), x)$

The $\overline{\delta}$ function extends the $\delta$ function from single letters to words. Whereas the $\delta$ function maps pairs of states and letters to other states, the $\overline{\delta}$ function maps pairs of states and words to other states. (i.) is the observation that $\delta$ and $\overline{\delta}$ treat single letters the same; this fact is not really essential to the definition of $\overline{\delta}$, since it can be deduced from (ii.) and (iii.) (see the exercises).

The $\overline{\delta}$ function maps the current state $s$ and the first letter $a_1$, of a word $w = a_1 \ldots a_n$ via the $\delta$ function to some other state $t$. It is then recursively applied with the new state $t$ and the remainder of the word, that is, with $a_2 \ldots a_n$. The recursion stops when the remainder of the word is the empty word $\lambda$. See Examples 1.7 and 1.11 for illustrations of computations using this recursive definition.
Since the recursion of the $\overline{\delta}$ function all takes place at the end of the string, $\overline{\delta}$ is called *tail recursive*. Tail recursion is easily transformed into iteration by applying the $\delta$ to successive letters of the input word and using the result of the previous application of $\overline{\delta}$ as an input to the current application.

Figure 1.5 gives an implementation of the $\overline{\delta}$ function in Pascal. Recursion has been replaced by iteration, and previous function results are saved in an auxiliary variable T. The function Delta, the input alphabet Sigma, and the state set State agree with the definitions given in Figure 1.4.

It stands to reason that if we start in state $s$ and word $y$ takes us to state $r$, and if we start in state $r$ and word $x$ takes us to state $t$, then the word $yx$ should take us from state $s$ all the way to $t$. That is, if $\overline{\delta}(s, y) = r$ and $\overline{\delta}(r, x) = t$, then $\overline{\delta}(s, yx)$ should equal $t$, also. We can indeed prove this, as shown with the following theorem.

**Theorem 1.1** Let $A = \langle \Sigma, S, s_0, \delta, F \rangle$ be a DFA. Then

$$(\forall x \in \Sigma^+)(\forall y \in \Sigma^+)(\forall s \in S)(\overline{\delta}(s, yx) = \overline{\delta}(\overline{\delta}(s, y), x))$$

Proof. Define $P(n)$ by

$$(\forall x \in \Sigma^+)(\forall y \in \Sigma^n)(\forall s \in S)(\overline{\delta}(s, yx) = \overline{\delta}(\overline{\delta}(s, y), x))$$

Basis step: $P(0)$: Let $y \in \Sigma^0 (\Rightarrow y = \lambda)$.

$\overline{\delta}(\overline{\delta}(s, y), x)$ (since $y = \lambda$)

$= \overline{\delta}(\overline{\delta}(s, \lambda), x)$ (by Definition 1.11ii)

$= \overline{\delta}(s, x)$ (since $x = \lambda \cdot x$)

$= \overline{\delta}(s, \lambda x)$ (since $y = \lambda$)

$= \overline{\delta}(s, yx)$

Inductive step: Assume $P(m)$:

$$(\forall x \in \Sigma^+)(\forall y \in \Sigma^m)(\forall s \in S)(\overline{\delta}(s, yx) = \overline{\delta}(\overline{\delta}(s, y), x)),$$

For any $z \in \Sigma^{m+1}$, $(\exists a \in \Sigma^1)(\exists y \in \Sigma^m) \exists z = a y$. Then

$\overline{\delta}(s, zx)$ (by definition of $z$)

$= \overline{\delta}(s, ax, yx)$ (by Definition 1.11iii)

$= \overline{\delta}(\delta(s, a), yx)$ (since $(\exists t \in S) \exists (s, a) = t$)

$= \overline{\delta}(t, yx)$ (by the induction assumption)

$= \overline{\delta}(\overline{\delta}(t, y), x)$ (by definition of $t$)

$= \overline{\delta}(\overline{\delta}(s, a), y, x)$ (by Definition 1.11iii)

$= \overline{\delta}(\overline{\delta}(s, ay), x)$ (by definition of $z$)

$= \overline{\delta}(\overline{\delta}(s, z), x)$

Therefore, $P(m) \Rightarrow P(m + 1)$, and since this implication holds for any nonnegative integer $m$, by the principle of mathematical induction we can say that $P(n)$ is true for all $n \in \mathbb{N}$. Since the statement therefore holds for any string $y$ of any length, the assertion is indeed true for all $y$ in $\Sigma^*$. This completes the proof of the theorem.

Note that the statement of Theorem 1.1 is very similar to the rule iii of the recursive definition of the extended state transition function (Definition 1.11) with the string $y$ replacing the single letter $a$. We will see a remarkable number of situations like this, where a recursive rule defined for a single symbol extends in a natural manner to a similar rule for arbitrary strings.
As alluded to earlier, the state in which a string terminates is significant; in particular, it is important to determine whether the terminal state for a string happens to be one of the states that was designated to be a final state.

**Definition 1.12** Given a DFA $A = \langle \Sigma, S, s_0, \delta, F \rangle$, $A$ accepts a word $w \in \Sigma^*$ iff $\overline{\delta}(s_0, w) \in F$.

We say a word $w$ is accepted by a machine $A = \langle \Sigma, S, s_0, \delta, F \rangle$ iff the extended state transition function $\delta$ associated with $A$ maps to a final state from $s_0$ when processing the word $w$. This means that the path from the start state ultimately leads to a final state when the word $w$ is presented to the machine. We will occasionally say that $A$ recognizes $w$; a DFA is sometimes referred to as a recognizer.

**Definition 1.13** Given a DFA $A = \langle \Sigma, S, s_0, \delta, F \rangle$, $A$ rejects a word $w \in \Sigma^*$ iff $\overline{\delta}(s_0, w) \notin F$.

In other words, a word $w$ is rejected by a machine $A = \langle \Sigma, S, s_0, \delta, F \rangle$ iff the $\overline{\delta}$ function associated with $A$ maps to a nonfinal state from $s_0$ when processing the word $w$.

**Example 1.7**

Let

$$A = \langle \Sigma, S, s_0, \delta, F \rangle$$

where

```pascal
const
MaxWordLength = 255; {an arbitrary constraint}
type
  Word = record
    Length : 0..MaxWordLength;
    Letters: packed array [0..MaxWordLength] of Sigma
  end; {Word}
function DeltaBar(S:State; W:Word) : State;
{uses the function Delta defined previously}
var
  T:State;
  I: 0..MaxWordLength;
begin
  T := S;
  if W.Length > 0
    then
      for I := 1 to W.Length do
        T := Delta(T, W.Letters[I]);
      DeltaBar := T
    end; {DeltaBar}
end; {DeltaBar}
```

Figure 1.5: A Pascal implementation of the extended state transition function
Σ = \{0, 1\}
S = \{q_0, q_1\}
s_0 = q_0
F = \{q_1\}

and δ is given by the transition table

<table>
<thead>
<tr>
<th>δ</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_0</td>
<td>q_0</td>
<td>q_1</td>
</tr>
<tr>
<td>q_1</td>
<td>q_1</td>
<td>q_0</td>
</tr>
</tbody>
</table>

The structure of this automaton is shown in Figure 1.6.

To see how some of the above definitions apply, let \(x = 0100\):

\[
\delta(q_0, x) = \delta(q_0, 0100) \\
= \delta(\delta(q_0, 0), 100) \\
= \delta(q_0, 100) \\
= \delta(\delta(q_0, 1), 00) \\
= \delta(q_1, 00) \\
= \delta(\delta(q_1, 0), 0) \\
= \delta(q_1, 0) \\
= \delta(q_1) \\
= q_1
\]

Thus, \(\delta(q_0, x) = q_1 \in F\), which means that \(x\) is accepted by \(A\); \(A\) recognizes \(x\).

Now let \(y = 1100\):

\[
\delta(q_0, y) = \delta(q_0, 1100) \\
= \delta(\delta(q_0, 1), 100) \\
= \delta(q_1, 100) \\
= \delta(\delta(q_1, 1), 00) \\
= \delta(q_0, 00) \\
= \delta(\delta(q_0, 0), 0) \\
= \delta(q_0, 0) \\
= \delta(q_0) \\
= q_0
\]

Therefore, \(\delta(q_0, y) = q_0 \notin F\), which means that \(y\) is not accepted by \(A\).

Following the Pascal conventions defined in the previous programming fragments, the function Accept defined in Figure 1.7 tests for acceptance of a string by consulting a FinalState set and using DeltaBar to refer to the TransitionTable.

The functions Delta, DeltaBar, and Accept can be combined to form a Pascal program that models a DFA. The sample fragments given in Figures 1.4, 1.5, and 1.7 rightly pass the candidate string as a
parameter. A full program would be complicated by several constraints, including the awkward way in which strings must be handled in Pascal. To highlight the correspondence between the code modules and the automata definitions, the program given in Figure 1.8 handles input at the character level rather than at the word level. The definitions in the procedure Initialize reflect the structure of the DFA shown in Figure 1.9. Invoking this program will produce a response to a single input word. For example, a typical exchange would be

```
  cba
Rejected
```

Running this program again might produce

```
  cccc
Accepted
```

This behavior is essentially the same as that of the C program shown in Figure 1.10. The succinct coding clearly shows the relationship between the components of the quintuple for the DFA and the corresponding code.

**Definition 1.14** Given an alphabet Σ, L is a language over the alphabet Σ iff \( L \subseteq \Sigma^* \).

A language is a collection of words over some alphabet. If the alphabet is denoted by Σ, then a language \( L \) over Σ is a subset of \( \Sigma^* \). Since \( L \subseteq \Sigma^* \), \( L \) may be finite or infinite. Clearly, the words used in the English language are a subset of words over the Roman alphabet and this collection is therefore a language according to our definition. Note that a language \( L \), in this context, is simply a list of words; neither syntax nor semantics are involved in the specification of \( L \). Thus, a language as defined by Definition 1.14 has little of the structure or relationships one would normally expect of either a natural language (like English) or a programming language (like Pascal).

**Example 1.8**

Some other examples of valid languages are

i. \( \emptyset \)

ii. \( \{ w \in \{0,1\}^* \mid |w| > 5 \} \)
iii. \{\lambda\}

iv. \{\lambda, \text{bilbo}, \text{frodo}, \text{samwise}\}

v. \{x \in \{a, b\}^* \mid |x|_a = |x|_b\}

The empty language, denoted by \emptyset or \{ \}, is different from \{\lambda\}, the language consisting of only the empty word \lambda. Whereas the empty language consists of zero words, the language consisting of \lambda contains one word (which contains zero letters). The distinction is analogous to an example involving sets of numbers: the set \{0\}, containing only the integer 0, is still a larger set than the empty set.

Every DFA differentiates between words that do not reach final states and words that do. In this sense, each automaton defines a language.

**Definition 1.15** Given a DFA \( A = \langle \Sigma, S, s_0, \delta, F \rangle \), the language accepted by \( A \), denoted \( L(A) \), is defined to be

\[
L(A) = \{ w \in \Sigma^* \mid \vec{\delta}(s_0, w) \in F \}
\]

\( L(A) \), the language accepted by a finite automaton \( A \), is the set of all words \( w \) from \( \Sigma^* \) for which \( \vec{\delta}(s_0, w) \in F \). In order for a word \( w \) to be contained in \( L(B) \), the path through the finite automaton \( B \), as determined by the letters in \( w \), must lead from the start state to one of the final states.

For deterministic finite automata, the path for a given word \( w \) is unique: there is only one path since, at any given state in the automaton, there is exactly one transition for each \( a \in \Sigma \). This is not necessarily the case for another variety of finite automaton, the nondeterministic finite automaton, as will be seen in Chapter 4.

**Definition 1.16** Given an alphabet \( \Sigma \), a language \( L \subseteq \Sigma^* \) is finite automaton definable (FAD) iff there exists some DFA \( B = \langle \Sigma, S, s_0, \delta, F \rangle \), such that \( L = L(B) \).

The set of all words over \( \{0, 1\} \) that contain an odd number of 1s is finite automaton definable, as evidenced by the automaton in Example 1.7, which accepts exactly this set of words.

#### 1.3 Examples of Finite Automata

This section illustrates the definitions of the quintuples and the state transition diagrams for some non-trivial automata. The following example and Example 1.11 deal with the recognition of tokens, an important issue in the construction of compilers.
program DFA(input, output);
{This program tests whether input strings are accepted by the }
{automaton displayed in Figure 1.9. The program expects input from}
{the keyboard, delimited by a carriage return. No error checking }
{is done; letters outside [‘a’ .. ‘c’] cause a range error. }
{type
   Sigma = ’a’..'c’;
   State = (s0, s1, s2);
var
   TransitionTable : array [State, Sigma] of State;
   FinalState : set of State;
function Delta(s : State; c : Sigma) : State;
begin
   Delta := TransitionTable[s,c]
end; { Delta }
function DeltaBar(s : State) : State;
var
   t : State;
begin
   t := s;
   { Step through the keyboard input one letter at a time. }
   while not eoln(input) do
      begin
         t := Delta(t, input);
         get(input)
      end;
   DeltaBar := t
end; { DeltaBar }
function Accept : boolean;
begin
   Accept := DeltaBar(s0) in FinalState
end; { Accept }
procedure Initialize;
begin
   FinalState := [s2];
   { Set up the state transition table. }
   TransitionTable [s0,’a’] := s1; TransitionTable [s0,’b’] := s0;
   TransitionTable [s0,’c’] := s2; TransitionTable [s1,’a’] := s2;
   TransitionTable [s1,’b’] := s0; TransitionTable [s1,’c’] := s0;
   TransitionTable [s2,’a’] := s0; TransitionTable [s2,’b’] := s0;
   TransitionTable [s2,’c’] := s1;
end; { Initialize }
begin { DFA }
   Initialize;
   if Accept then
      writeln(output, ’Accepted’)
   else
      writeln(output, ’Rejected’)       39
end. { DFA }

Figure 1.8: A Pascal program that emulates the DFA shown in Figure 1.9
Example 1.9

The set of FORTRAN identifiers is a finite automaton definable language. This statement can be proved by verifying that the following machine accepts the set of all valid FORTRAN 66 identifiers. These identifiers, which represent variable, subroutine, and array names, can contain from 1 to 6 (nonblank) characters, must begin with an alphabetic character, can be followed by up to 5 letters or digits, and may have embedded blanks. In this example, we have ignored the difference between capital and lowercase letters, and \( \diamond \) represents a blank.

\[
\Sigma = \text{ASCII} \\
\Gamma = \text{ASCII} - \{a, b, c, \ldots, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \diamond\} \\
S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\} \\
s_0 = s_0
\]

\[
\begin{array}{c|cccccccccc}
\delta & a & b & c \ldots & y & z & 0 & 1 \ldots & 8 & 9 & \diamond & \Gamma \\
\hline
s_0 & s_1 & s_1 & s_1 \ldots & s_1 & s_1 & s_7 & s_7 \ldots & s_7 & s_7 & s_7 & s_7 \\
s_1 & s_2 & s_2 & s_2 \ldots & s_2 & s_2 & s_2 \ldots & s_2 & s_2 & s_2 & s_2 & s_2 \\
s_2 & s_3 & s_3 & s_3 \ldots & s_3 & s_3 & s_3 \ldots & s_3 & s_3 & s_3 & s_3 & s_3 \\
s_3 & s_4 & s_4 & s_4 \ldots & s_4 & s_4 & s_4 \ldots & s_4 & s_4 & s_4 & s_4 & s_7 \\
s_4 & s_5 & s_5 & s_5 \ldots & s_5 & s_5 & s_5 \ldots & s_5 & s_5 & s_5 & s_7 & s_7 \\
s_5 & s_6 & s_6 & s_6 \ldots & s_6 & s_6 & s_6 \ldots & s_6 & s_6 & s_6 & s_7 & s_7 \\
s_6 & s_7 & s_7 & s_7 \ldots & s_7 & s_7 & s_7 \ldots & s_7 & s_7 & s_7 & s_7 & s_7 \\
s_7 & s_7 & s_7 & s_7 \ldots & s_7 & s_7 & s_7 \ldots & s_7 & s_7 & s_7 & s_7 & s_7 \\
\end{array}
\]

\( F = \{s_1, s_2, s_3, s_4, s_5, s_6\} \)

The entries under the column labeled \( \Gamma \) show the transitions taken for each member of the set \( \Gamma \). The state transition diagram of the machine corresponding to this quintuple is displayed in Figure 1.11. Note that, while each of the 26 letters transition from \( s_0 \) to \( s_1 \), a single arrow labeled \( a \cdot z \) is sufficient to denote all these transitions. Similarly, the transition labeled \( \Sigma \) from \( s_7 \) indicates that every element of the alphabet follows the same path.
Figure 1.10: A C program that emulates the DFA shown in Figure 1.9
Figure 1.11: A DFA that recognizes valid FORTRAN identifiers

Figure 1.12: The DFA $M$ discussed in Example 1.10
Example 1.10

The DFA $M$ shown in Figure 1.12 accepts only those strings that have an even number of $b$s and an even number of $a$s. Thus

$$L(M) = \{x \in \{a, b\}^* \mid |x|_a = 0 \mod 2 \land |x|_b = 0 \mod 2\}$$

The corresponding quintuple for $M = \langle \Sigma, S, s_0, \delta, F \rangle$ has the following components:

- $\Sigma = \{a, b\}$
- $S = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle\}$
- $s_0 = \langle 0, 0 \rangle$

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle 0, 0 \rangle$</td>
<td>$\langle 1, 0 \rangle$</td>
<td>$\langle 0, 1 \rangle$</td>
</tr>
<tr>
<td>$\langle 0, 1 \rangle$</td>
<td>$\langle 1, 1 \rangle$</td>
<td>$\langle 0, 0 \rangle$</td>
</tr>
<tr>
<td>$\langle 1, 0 \rangle$</td>
<td>$\langle 0, 0 \rangle$</td>
<td>$\langle 1, 1 \rangle$</td>
</tr>
<tr>
<td>$\langle 1, 1 \rangle$</td>
<td>$\langle 0, 1 \rangle$</td>
<td>$\langle 1, 0 \rangle$</td>
</tr>
</tbody>
</table>

$F = \{\langle 0, 0 \rangle\}$

Note that the transition function can be succinctly specified by

$$\delta(\langle i, j \rangle, a) = \langle 1 - i, j \rangle \text{ and } \delta(\langle i, j \rangle, b) = \langle i, 1 - j \rangle \text{ for all } i, j \in \{0, 1\}$$

See the exercises for some other problems involving congruence modulo 2.

Example 1.11

Consider a typical set of all real number constants in modified scientific notation format described by the BNF in Table 1.1.

This set of productions defines real number constants like $+192.$, since

$$<\text{real constant}> \Rightarrow <\text{integer}>.$$

$$<\text{integer}> \Rightarrow <\text{sign}> <\text{natural}>.$$

$$<\text{sign}> <\text{natural}> \Rightarrow + <\text{natural}>.$$

$$+ <\text{natural}> \Rightarrow + <\text{digit}> <\text{natural}>.$$

$$+ <\text{digit}> <\text{natural}> \Rightarrow + 1 <\text{natural}>.$$

$$+ 1 <\text{natural}> \Rightarrow + 1 <\text{digit}> <\text{natural}>.$$

$$+ 1 <\text{digit}> <\text{digit}> \Rightarrow + 1 <\text{digit}> 2.$$

$$+ 1 <\text{digit}> 2. \Rightarrow + 192.$$

Other possibilities are

1
3.1415
2.718281828
27.
42.42
1.0E−32

43
Table 1.1: For Example 1.11

<table>
<thead>
<tr>
<th>production</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;sign&gt;</td>
<td>::= +</td>
</tr>
<tr>
<td>&lt;digit&gt;</td>
<td>::= 0</td>
</tr>
<tr>
<td>&lt;natural&gt;</td>
<td>::= &lt;digit&gt;</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>::= &lt;natural&gt;</td>
</tr>
<tr>
<td>&lt;real constant&gt;</td>
<td>::= &lt;integer&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

while the following strings do not qualify:

- .01
- 1.+1
- 8.E−10

The set of all real number constants that can be derived from the productions given in Table 1.1 is a FAD language. Let R be the deterministic finite automaton defined below. The corresponding state transition diagram is given in Figure 1.13.

\[ \Sigma = \{0,1,2,3,4,5,6,7,8,9,+,−,E,\} \]
\[ S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\} \]
\[ s_0 = s_0 \]

<table>
<thead>
<tr>
<th>δ</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>+</th>
<th>−</th>
<th>E</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_0</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_1</td>
<td>s_1</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
</tr>
<tr>
<td>s_1</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
</tr>
<tr>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_2</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
</tr>
<tr>
<td>s_3</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
</tr>
<tr>
<td>s_4</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
</tr>
<tr>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
</tr>
<tr>
<td>s_6</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
<td>s_5</td>
</tr>
<tr>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
<td>s_7</td>
</tr>
<tr>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_8</td>
<td>s_7</td>
<td>s_7</td>
<td>s_4</td>
<td>s_7</td>
<td>s_7</td>
</tr>
</tbody>
</table>

\[ F = \{s_2, s_3, s_5, s_8\} \]

The language accepted by R, that is \(L(R)\), is exactly the set of all real number constants in modified scientific notation format described by the BNF in Table 1.1.
Figure 1.13: A DFA that recognizes real number constants

For example, let $x = 3.1415$:

$$
\delta(s_0, x) = \delta(s_0, 3.1415)
\begin{align*}
&= \delta(\delta(s_0, 3), 1415) \\
&= \delta(s_2, 1415) \\
&= \delta(\delta(s_2, .), 1415) \\
&= \delta(s_3, 1415) \\
&= \delta(\delta(s_3, .), 415) \\
&= \delta(s_4, 415) \\
&= \delta(\delta(s_4, .), 15) \\
&= \delta(s_5, 15) \\
&= \delta(\delta(s_5, .), 5) \\
&= \delta(s_6, 5) \\
&= \delta(s_7, 5) \\
&= s_8
\end{align*}
$$

$s_8 \in F$, and therefore $3.1415 \in L(R)$.

While many important classes of strings such as numerical constants (Example 1.11) and identifiers (Example 1.9) are FAD, not all languages that can be described by BNF can be recognized by DFAs. These
limitations will be investigated in Chapters 8 and 9, and a more capable type of automaton will be defined in Chapter 10.

1.4 Circuit Implementation of Finite Automata

Now that we have described the mathematical nature of deterministic finite automata, let us turn to the physical implementation of such devices. We will investigate the sort of physical components that actually go into the “brain” of, say, a vending machine. Recall that the basic building blocks of digital logic circuits are logic gates; using 0 or False to represent a low voltage (ground) and 1 or True to represent a higher voltage (often +5 volts), the basic gates have the truth tables shown in Figure 1.14.

Since our DFA will examine one letter at a time, we will generally need some type of timing mechanism, which will be regulated by a clock pulse; we will read one letter per pulse and allow enough interim time for transient signals to propagate through our network as we change states and move to the next letter on the input tape. The clock pulse will alternate between high and low voltages, as shown in Figure 1.15. For applications such as vending machines, the periodic clock pulse would be replaced by a device that pulsed whenever a new input (such as the insertion of a coin) was detected.

We need to retain the present status of the network (current state, letter, and so forth) as we move on to the next input symbol. This is achieved through the use of a D flip-flop (D stands for data or delay), which uses NAND gates and the clock signal to store the current value of, say, \( p \), between clock pulses. The symbol for a D flip-flop (sometimes called a latch) is shown in Figure 1.16, along with the actual gates that comprise the circuit.

The output, \( p \) and \( \neg p \), will reflect the value of the input signal \( p' \) only after the high clock pulse is received and will retain that value after the clock drops to low (even if \( p' \) subsequently changes) until the next clock pulse comes along, at which time the output will reflect the new current value of \( p' \). This is best illustrated by referring to the NAND truth table and tracing the changes in the circuit. Begin with clock = \( p = p' = 0 \) and \( \neg p = 1 \), and verify that the circuit is stable. Now assume that \( p' \) changes to 1, and note that, although some internal values may change, \( p \) and \( \neg p \) remain at 0 and 1, respectively; the old value of \( p' \) has been “remembered” by the D flip-flop. Contrast this with the behavior when we strobe the clock: assume that the clock now also changes to 1 so that we now have clock = \( p = p' = 1 \), and \( p = 0 \). When the signal propagates through the network, we find that \( p \) and \( \neg p \) have changed to reflect the new value of \( p' \); clock = \( p = p' = 1 \), and \( \neg p = 0 \).

We will also have to represent the letters of our input alphabet by high and low voltages (that is, combinations of 0s and 1s). The ASCII alphabet, for example, is quite naturally represented by 8 bits,
Figure 1.15: A typical clock pulse pattern for latched circuits

\[a_1a_2a_3a_4a_5a_6a_7a_8, \text{ where } B, \text{ for example, has the bit pattern } 01000010 \text{ (binary 66). One of these bit patterns should be reserved for indicating the end of our input string } <\text{EOS}>. \text{ Our convention will be to reserve binary zero for this role, which means our ASCII end of string symbol would be } 00000000 \text{ (or NULL). In actual applications using the ASCII alphabet, however, a more appropriate choice for } <\text{EOS}> \text{ might be } 00011100 \text{ (a carriage return) or } 00010100 \text{ (a line feed) or } 00101000 \text{ (a space).}

Our alphabets are likely to be far smaller than the ASCII character set, and we will hence need fewer than 8 bits of information to encode our letters. For example, if \( \Sigma = \{b, c\} \), 2 bits, \( a_1 \) and \( a_2 \), will suffice. Our choice of encoding could be \( 00 = <\text{EOS}>, \ 01 = b, \ 10 = c, \text{ and } 11 \text{ is unused.}

In a similar fashion, we must encode state names. A machine with \( S = \{r_0, r_1, r_2, r_3, r_4, r_5\} \) would need 3 bits (denoted by \( t_1, t_2, \) and \( t_3 \)) to represent the six states. The most natural encoding would be \( r_0 = 000, \ r_1 = 001, \ r_2 = 010, \ r_3 = 011, \ r_4 = 100, \text{ and } r_5 = 101, \text{ with the combinations } 110 \text{ and } 111 \text{ left unused.}

Finally, a mechanism for differentiating between final and nonfinal states must be implemented (although this need not be engaged until the } <\text{EOS}> \text{ symbol is encountered). Recall that we must illuminate the “acceptance light” if the machine terminates in a final state and leave it unlit if the string on the input tape is instead rejected by the DFA. A second “rejection light” can be added to the physical model, and exactly one of the two will light when } <\text{EOS}> \text{ is scanned by the input head.

\textbf{Example 1.12}

When building a logical circuit from the definition of a DFA, we will find it convenient to treat } <\text{EOS}> \text{ as an input symbol, and define the state transition function for it by } (\forall s \in S)(\delta(s_1, <\text{EOS}>) = s). \text{ Thus, the DFA in Figure 1.17a should be thought of as shown in Figure 1.17b. As we have only two states, a single state bit will suffice, representing } s_0 \text{ by } t_1 = 0 \text{ and } s_1 \text{ by } t_1 = 1. \text{ Since } \Sigma = \{b, c\}, \text{ we will again use 2 bits, } a_1 \text{ and } a_2, \text{ to represent the input symbols. As before, } 00 = <\text{EOS}>, \ 01 = b, \ 10 = c, \text{ and } 11 \text{ is unused.}

Determining the state transition function will require knowledge of the current state (represented by the status of } t_1 \text{) and the current input symbol (represented by the pair of bits } a_1 \text{ and } a_2). \text{ These three
Figure 1.16: (a) A data flip-flop or latch (b) The circuitry for a D flip-flop

Figure 1.17: (a) The DFA discussed in Example 1.12 (b) The expanded state transition diagram for the DFA implemented in Figure 1.18
input values will allow the next state $t'_1$ to be calculated. From the $\delta$ function, we know that

$$
\delta(s_0, b) = s_0 \\
\delta(s_0, c) = s_1 \\
\delta(s_1, b) = s_0 \\
\delta(s_1, c) = s_0
$$

These specifications correspond to the following four rows of the truth table for $t'_1$:

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$t'_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

which represents

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$t'_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>0</td>
<td>1</td>
<td>$b$</td>
</tr>
<tr>
<td>$s_0$</td>
<td>1</td>
<td>0</td>
<td>$c$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>0</td>
<td>1</td>
<td>$b$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>1</td>
<td>0</td>
<td>$c$</td>
</tr>
</tbody>
</table>

Adding the state transitions for <EOS> and using * to represent the outcome for the two rows corresponding to the unused combination $a_1a_2 = 11$ fills out the eight rows of the complete truth table, as shown in Table 1.2.

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$t'_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 1.2:

If we arbitrarily assume that the two don’t-care combinations (*) are zero, the principle disjunctive normal form of $t'_1$ contains just two terms: $(\neg t_1 \land a_1 \land \neg a_2) \lor (t_1 \land \neg a_1 \land \neg a_2)$. It is profitable to reassign the don’t-care value in the fourth row to 1, since the expression can then be shortened to $(\neg t_1 \land a_1) \lor (t_1 \land \neg a_1 \land \neg a_2)$ by applying standard techniques for minimizing Boolean functions. Incorporating this into a feedback loop with a D flip-flop provides the heart of the digital logic circuit representing the DFA, as shown in Figure 1.18.

The accept portion of the circuitry ensures that we do not indicate acceptance when passing through the final state; it is only activated when we are in a final state while scanning the <EOS> symbol. Similarly, the reject circuitry can only be activated when the <EOS> symbol is encountered. When there are several final states, this part of the circuitry becomes correspondingly more complex. It is instructive to follow the effect a string such as $bcc$ has on the above circuit. Define $a_i(j)$ as the $j$th value the bit $a_i$ takes on as the string $bcc$ is processed; that is, $a_i(j)$ is the value of $a_i$ during the $j$th clock pulse. We then have
Figure 1.18: The circuitry implementing the DFA discussed in Example 1.12

\[
\begin{align*}
    a_1(1) &= 0 & a_2(1) &= 1 & \Rightarrow b \\
    a_1(2) &= 1 & a_2(2) &= 0 & \Rightarrow c \\
    a_1(3) &= 1 & a_2(3) &= 0 & \Rightarrow c \\
    a_1(4) &= 0 & a_2(4) &= 0 & \Rightarrow <\text{EOS}>
\end{align*}
\]

Trace the circuit through four clock pulses (starting with \( t_1 = 0 \)), and observe the current values that \( t_1 \) assumes, noting that it corresponds to the appropriate state of the machine as each input symbol is scanned.

Note that a six-state machine would require more and substantially larger truth tables. Since a state encoding would now need to specify \( t_1, t_2, \) and \( t_3 \), three different truth tables (for \( t'_1, t'_2, \) and \( t'_3 \)) must be constructed to predict the next state transition. More significantly, the input variables would include \( t_1, t_2, t_3, a_1, \) and \( a_2 \), making each table 32 rows long. Three D flip-flop feedback loops would be necessary to store the three values \( t_1, t_2, \) and \( t_3 \).

Also, physical logic circuits of this type have the disconcerting habit of initializing to some random configuration the first time power is applied to the network. A true working model would thus need a reset circuit to initialize each \( t_i \) to 0 in order to ensure that the machine started in state \( s_0 \). Slightly more complex set-reset flip-flops can be used to provide a hardware solution to this problem. However, a simple algorithmic solution would require the input tape to have a leading start-of-string symbol <SOS>. The definition of the state transition function should be expanded so that scanning the <SOS> symbol from any state will automatically transfer control to \( s_0 \). We will adopt the convention that <SOS> will be represented by the highest binary code; in ASCII, for example, this would be 111111111, while in the preceding example it would be 11. To promote uniformity in the exercises, it is suggested that <SOS> should always be given the highest binary code and <EOS> be represented by binary zero; as in the examples given here, the symbols in \( \Sigma \) should be numbered sequentially according to their natural alphabetical order. In a similar fashion, numbered states should be given their corresponding binary codes. The reader should note, however, that other encodings might result in less complex circuitry.
Example 1.13

As a more complex example of automaton circuitry, consider the DFA displayed in Figure 1.19. Two flip-flops $t_1$ and $t_2$ will be necessary to represent the three states, most naturally encoded as $s_0 = 00$, $s_1 = 01$, $s_2 = 10$, with $s_3 = 11$ unused. Employing both <SOS> and <EOS> encodings yields the DFA in Figure 1.20.

Note that we must account for the possibility that the circuitry might be randomly initialized to $t_1 = 1$ and $t_2 = 1$; we must ensure that scanning the <SOS> symbol moves us back into the “real” part of the machine. Two bits of information ($a_1$ and $a_2$) are also needed to describe the input symbols. Following our conventions, we assign <EOS> = 00, $a = 01$, $b = 10$, and <SOS> = 11. The truth table for both the transition function and the conditions for acceptance is given in Table 1.3.

In the first row, $t_1 = 0$ and $t_2 = 0$ indicate state $s_0$, while $a_1 = 0$ and $a_2 = 0$ denote the <EOS> symbol. Since $\delta(s_0, <EOS>) = s_0$, $t'_1 = 0$ and $t'_2 = 0$. We do not want to accept a string that ends in $s_0$, so accept = 0 also. The remaining rows are determined similarly. The (nonminimized) circuitry for this DFA is shown in Figure 1.21.

1.5 Applications of Finite Automata

In this chapter we have described the simplest form of finite automaton, the DFA. Other forms of automata, such as nondeterministic finite automata, pushdown automata, and Turing machines, are introduced later in the text. We close this chapter with three examples to motivate the material in the succeeding chapters.

When presenting automata in this chapter, we made no effort to construct the minimal machine. A minimal machine for a given language is one that has the least number of states required to accept that language.

Example 1.14

In Example 1.5, the vending machine kept track of the amount of change that had been deposited up to 50¢. Since the candy bars cost only 30¢, there is no need to count up to 50¢. In this sense, the machine is not optimal, since a less complex machine can perform the same task, as shown in Figure 1.22.
Table 1.3:

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$t'_1$</th>
<th>$t'_2$</th>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1.20: The expanded state transition diagram for the DFA implemented in Figure 1.21
corresponding quintuple is \( <\{n, d, q\}, \{s_0, s_5, s_{10}, s_{15}, s_{20}, s_{25}, s_{30}\}, s_0, \delta, \{s_{30}\}> \), where for each state \( s_i \), \( \delta \) is defined by

\[
\begin{align*}
\delta(s_i, n) &= s_{\text{min}(30, i+5)} \\
\delta(s_i, d) &= s_{\text{min}(30, i+10)} \\
\delta(s_i, q) &= s_{\text{min}(30, i+25)}
\end{align*}
\]

Note that the higher-numbered states in Example 1.5 were all effectively “remembering” the same thing, that enough coins had been deposited. These final states have been coalesced into a single final state to produce the more efficient machine in Figure 1.22. In the next two chapters, we develop the theoretical background and algorithms necessary to construct from an arbitrary DFA the minimal machine that accepts the same language.

As another illustration of the utility of concepts relating to finite-state machines, we will consider the formalism used by many text editors to search for a particular target string pattern in a text file. To find \texttt{ababb} in a file, for example, a naive approach might consist of checking whether the first five characters of the file fit this pattern, and next checking characters 2 through 6 to find a match, and so on. This results in examining file characters more than once; it ought to be possible to remember past values, and avoid such duplication. Consider the text string \texttt{aabababb}. By the time the fifth character is scanned, we have matched the first four characters of \texttt{ababb}. Unfortunately, \texttt{a}, the sixth character of \texttt{aabababb}, does not produce the final match; however, since characters 4, 5, and 6 (\texttt{aba}) now match the first three characters of the target string, it does allow for the possibility of characters 4 through 8 matching (as is indeed the case in this example). This leads to a general rule: If we have matched the first four letters of
the target string, and the next character happens to be \( a \) (rather than the desired \( b \)), we must remember that we have now matched the first three letters of the target string.

“Rules” such as these are actually the state transitions in the DFA given in the next example. State \( s_i \) represents having matched the first \( i \) characters of the target string, and the rule developed above is succinctly stated as \( \delta(s_4, a) = s_3 \).

**Example 1.15**

A DFA that accepts all strings that contain \( ababb \) as a substring is displayed in Figure 1.23. The corresponding quintuple is

\[
\langle \{a, b\}, \{s_0, s_1, s_2, s_3, s_4, s_5\}, s_0, \delta, \{s_5\}\rangle,
\]

where \( \delta \) is defined by

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( s_0 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_1 )</td>
<td>( s_2 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_3 )</td>
<td>( s_0 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( s_1 )</td>
<td>( s_4 )</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>( s_3 )</td>
<td>( s_5 )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( s_5 )</td>
<td>( s_5 )</td>
</tr>
</tbody>
</table>

It is a worthwhile exercise to test the operation of this DFA on several text strings and verify that the automaton is indeed in state \( s_i \) exactly when it has matched the first \( i \) characters of the target string. Note
that if we did not care what the third character of the substring was (that is, if we were searching for occurrences of *ababb* or *abb*, a trivial modification of the above machine would allow us to search for both substrings at once, as shown in Figure 1.24. The corresponding quintuple is \( \langle \{a, b\}, \{s_0, s_1, s_2, s_3, s_4, s_5\}, s_0, \delta, \{s_5\} \rangle \), where \( \delta \) is defined by

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( s_0 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_1 )</td>
<td>( s_2 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( s_1 )</td>
<td>( s_4 )</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>( s_3 )</td>
<td>( s_5 )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( s_5 )</td>
<td>( s_5 )</td>
</tr>
</tbody>
</table>

Figure 1.24: A DFA that accepts strings that contain either *ababb* or *abb*

In this case, we required one letter between the initial part of the search string (**ab**) and the terminal part (**bb**). It is possible to modify the machine to accept strings that contain **ab**, followed by any number of letters, followed by **bb**. This type of machine would be useful for identifying *comments* in many programming languages. For example, a Pascal comment is essentially of the form (**/*, followed by most combinations of letters, followed by the first occurrence of */**).

It should be noted that the machine in Example 1.15 is highly specialized and tailored for the specific string *ababb*; other target strings would require completely different recognizers. While it appears to require much thought to generate the appropriate DFA for a given string, we will see how the tools presented in Chapter 4 can be used to automate the entire process.

Example 1.15 indicates how automata can be used to guide the construction of software for matching designated patterns. Finite-state machines are also useful in designing hardware that detects designated sequences. Example 4.7 will explore a communications application, and the following discussion illustrates how these concepts can be applied to help evaluate the performance of computers.

A computer program is essentially a linear list of machine instructions, stored in consecutive memory locations. Each memory location holds a sequence of bits that can be thought of as words comprised of 000s and 111s. Different types of instructions are represented by different patterns of bits. The CPU sequentially *fetches* these instructions and chooses its next action by examining the incoming bit pattern to determine the type of instruction that should be executed. The sequences of bits that encode the instruction type are called *opcodes*.

Various performance advantages can be attained when one part of the CPU *prefetches* the next instruction while another part executes the current instruction. However, computers must have the capability of altering the order in which instructions are executed; *branch* instructions allow the CPU to avoid the anticipated next instruction and instead begin executing the instructions stored in some other
area of memory. When a branch occurs, the prefetched instruction will generally need to be replaced by the proper instruction from the new area of memory. The consequent delay can degrade the speed with which instructions are executed.

Irrespective of prefetching problems, it should be clear that a branch instruction followed immediately by another branch instruction is inefficient. If a CPU is found to be regularly executing two or more consecutive branch instructions, it may be worthwhile to consider replacing such series of branches with a single branch to the ultimate destination [FERR]. Such information would be determined by monitoring the instruction stream and searching for patterns that represented consecutive branch opcodes. This activity is essentially the pattern recognition problem discussed in Example 1.15.

It is unwise to try to collect the data representing the contents of the instruction stream on secondary storage so that it can be analyzed later. The volume of information and the speed with which it is generated preclude the collection of a sufficiently large set of data points. Instead, the preferred solution uses a specially tailored piece of hardware to monitor the contents of the CPU opcode register and increment a hardware counter each time the appropriate patterns are detected. The heart of this monitor can be built by transforming the appropriate automaton into the corresponding logic circuitry, as outlined in Section 1.4. Unlike the automaton in Example 1.15, the automaton model for this application would allow transitions out of the final state, so that it may continue to search for successive patterns. The resulting logic circuitry would accept as input the bit patterns currently present in the opcode register, and send a pulse to the counter mechanism each time the accept circuitry was energized.

Note that in this case we would not want to inhibit the accept circuitry by requiring an <EOS> symbol to be scanned. Indeed, we want the light on our conceptual black box to flicker as we process the data, since we are intent on counting the number of times it flickers during the course of our monitoring.

Example 1.16

We close this chapter with an illustration of the manner in which computational algorithms can profitably use the automaton abstraction. Network communications between independent processors are governed by a protocol that implements a finite state control [TANE]. The Kermit protocol, developed at Columbia University, was widely employed to communicate between processors and is still most often used for its original purpose: to transfer files between micros and mainframes [DACR]. During a file transfer, the send portion of Kermit on the source host is responsible for delivering data to the receive portion of the Kermit process on the destination host. The receive portion of Kermit reacts to incoming data in much the same way as the machines presented in this chapter. The receive program starts in a state of waiting for a transfer request (in the form of an initialization packet) to signal the commencement of a file transfer (state R in Figure 1.25). When such a packet is received, Kermit transitions to the RF state, where it awaits a file-header packet (which specifies the name of the file about to be transferred). Upon receipt of the file-header packet, it enters the RD state, where it processes a succession of data packets (which comprise the body of the file being transferred). An EOF packet should arrive after all the data are sent, which can then be followed by another file-header packet (if there is a sequence of files to be transferred) or by a break packet (if the transfer is complete). In the latter case, Kermit reverts to the start state R and awaits the next transfer request. The send portion of the Kermit process on the source host follows the behavior of a slightly more complex automaton. The state transition diagram given in Figure 1.25 succinctly describes the logic of the receive portion of the Kermit protocol; for simplicity, timeouts and error conditions are not reflected in the diagram. The input alphabet is \{B,D,Z,H,S\}, where B represents a break, D is a data packet, Z is EOF, H is a file-header packet, and S is a send-intention packet. The state set is \{A,R,RF,RD\}, where A denotes the abort state, R signifies receive, RF is receive
fileheader, and RD is receive data. Note that unexpected packets (such as a data packet received in the start state R or a break packet received when data packets are expected in state RD) cause a transition to the abort state A.

In actuality, the receive protocol does more than just observe the incoming packets; Kermit sends an acknowledgment (ACK or NAK) of each packet back to the source host. Receipt of the file header should also cause an appropriate file to be created and opened, and each succeeding data packet should be verified and its contents placed sequentially in the new file. A machine model that incorporates actions in response to input is the subject of Chapter 7, where automata with output are explored.

EXERCISES

1.1. Recall how we defined $\overline{\delta}$ in this chapter:

$$(\forall s \in S)(\forall a \in \Sigma) \quad \overline{\delta}_t(s, a) = \delta(s, a)$$

$$(\forall s \in S) \quad \overline{\delta}_t(s, \lambda) = s$$

$$(\forall s \in S)(\forall x \in \Sigma^*)(\forall a \in \Sigma) \quad \overline{\delta}_t(s, a x) = \overline{\delta}_t(\delta(s, a), x)$$

$\overline{\delta}$, here denoted $\overline{\delta}_t$, was tail recursive. Tail recursion means that all recursion takes place at the end of the string. Let us now define an alternative extended transition function, $\overline{\delta}_h$, thusly:

$$(\forall s \in S)(\forall a \in \Sigma) \quad \overline{\delta}_h(s, a) = \delta(s, a)$$

$$(\forall s \in S) \quad \overline{\delta}_h(s, \lambda) = s$$

$$(\forall s \in S)(\forall a \in \Sigma)(\forall x \in \Sigma^*) \quad \overline{\delta}_h(s, a x) = \delta(\overline{\delta}_h(s, a), x)$$
It is clear from the definition of \( \delta_h \) that all the recursion takes place at the head of the string. For this reason, \( \delta_h \) is called head recursive. Show that the two definitions result in the same extension of \( \delta \), that is, prove by mathematical induction that

\[
(\forall s \in S)(\forall x \in \Sigma^*)(\delta_r(s, x) = \delta_h(s, x))
\]

1.2. Consider Example 1.14. The vending machine accepts coins as input, but if you change your mind (or find you do not have enough change), it will not refund your money. Modify this example to have another input, <coin-return>, which is represented by \( r \) and which will conceptually return all your coins.

1.3. (a) Specify the quintuple corresponding to the DFA displayed in Figure 1.26.

(b) Describe the language defined by the DFA displayed in Figure 1.26.

Figure 1.26: The automaton discussed in Exercise 1.3

1.4. Construct a state transition diagram and enumerate all five parts of a deterministic finite automaton \( A = \langle \{a, b, c\}, S, s_0, \delta, F\rangle \) such that

\[
L(A) = \{x \mid |x| \text{ is a multiple of 2 or 3}\}.
\]

1.5. Let \( \Sigma = \{0, 1\} \). Construct deterministic finite automata that will accept each of the following languages, if possible.

(a) \( L_1 = \{x \mid |x| \text{ mod } 7 = 4\} \)

(b) \( L_2 = \Sigma^* - \{w \mid \exists n \geq 1 : w = a_1 \ldots a_n \land a_n = 1\} \)

(c) \( L_3 = \{y \mid |y|_0 = |y|_1\} \)

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

(a) Construct deterministic finite automata \( A_1, A_2, A_3, \) and \( A_4 \) such that:

i. \( L(A_1) = \{x \mid (|x|_a \text{ is odd}) \land (|x|_b \text{ is even})\} \)

ii. \( L(A_2) = \{y \mid (|y|_a \text{ is even}) \lor (|y|_b \text{ is odd})\} \)

iii. \( L(A_3) = \{z \mid (|z|_a \text{ is even}) \lor (|z|_b \text{ is even})\} \) (\( \lor \) represents exclusive-or)

iv. \( L(A_4) = \{z \mid |z|_a \text{ is even}\} \)

(b) How does the structure of each of these machines relate to the one defined in Example 1.10?

1.7. Modify the machine \( M \) defined in Example 1.10 so that the language accepted by the machine consists of strings \( x \in \{a, b\}^* \), where both \( |x|_a \) and \( |x|_b \) are even and \( |x| > 0 \), that is, the new machine should accept \( L(M) - \{\lambda\} \).
1.8. Let $M = \langle \Sigma, S, s_0, \delta, F \rangle$ be an (arbitrary) DFA that accepts the language $L(M)$. Write down a general procedure for modifying this machine so that it will accept $L(M) - \{\lambda\}$. (Specify the five parts of the new machine and justify your statements.) It may be helpful to do this for a specific machine (as in Exercise 1.7) before attempting the general case.

1.9. Let $M = \langle \Sigma, S, s_0, \delta, F \rangle$ be an (arbitrary) DFA that accepts the language $L(M)$. Write down a general procedure for modifying this machine so that it will accept $L(M) \cup \{\lambda\}$. (Specify the five parts of the new machine and justify your statements.)

1.10. Let $M = \langle \Sigma, S, s_0, \delta, F \rangle$ be an (arbitrary) DFA that accepts the language $L(M)$. Write down a general procedure for modifying this machine so that it will accept $L(M) - \{\lambda\}$. (Specify the five parts of the new machine and justify your statements.) It may be helpful to do this for a specific machine (as in Exercise 1.7) before attempting the general case.

1.11. Let $\Sigma = \{a aa, bbb, ccc\}$ and $\Phi = \{x \in \Sigma^* | \text{every b in x is immediately followed by c}\}$. (a) Draw a machine that will accept $\Phi$. (b) Formally specify the five parts of the DFA from part (a).

1.12. Let $\Sigma = \{0 00, 111, 222, 333, 444, 555, 666, 777, 888, 999\}$. Consider the base 10 numbers formed by strings from $\Sigma^*$: $141414$ represents fourteen, the three-digit string $205205205$ represents two hundred and five, and so on. Let $\Omega = \{x \in \Sigma^* | \text{the number represented by x is evenly divisible by 7}\} = \{\lambda, 0 00, 000000, 000000000, \ldots, 777, 070707, 007007007, \ldots, 141414, 212121, 282828, 353535, \ldots\}$. (a) Draw a machine that will accept $\Omega$. (b) Formally specify the five parts of the DFA from part (a).

1.13. Let $\Sigma = \{0 00, 111, 222, 333, 444, 555, 666, 777, 888, 999\}$. Let $\Gamma = \{x \in \Sigma^* | \text{the number represented by x is evenly divisible by 3}\}$. (a) Draw a three-state machine that will accept $\Gamma$. (b) Formally specify the five parts of the DFA from part (a).

1.14. Let $\Sigma = \{0 00, 111, 222, 333, 444, 555, 666, 777, 888, 999\}$. Let $K = \{x \in \Sigma^* | \text{the number represented by x is evenly divisible by 5}\}$. (a) Draw a five-state DFA that accepts $K$. (b) Formally specify the five parts of the DFA from part (a). (c) Draw a two-state DFA that accepts $K$. (d) Formally specify the five parts of the DFA from part (c).

1.15. Let $\Sigma = \{0 00, 111, 222, 333, 444, 555, 666, 777, 888, 999\}$. Draw a DFA that accepts the first eight primes.

1.16. (a) Find all ten combinations of $u, v,$ and $w$ such that $uvw = cabb$ (one such combination is $u = c, v = \lambda, w = ab$).

   (b) In general, if $x$ is of length $n$, and $uvw = x$, how many distinct combinations of $u, v,$ and $w$ will satisfy this constraint?

1.17. Let $\Sigma = \{a, b\}$ and $\Xi = \{x \in \Sigma^* | \text{x contains (at least) two consecutive bs}\}$. Draw a machine that will accept $\Xi$. 59
1.18. The FORTRAN identifier in Example 1.9 recognized all alphabetic words, including those like DO, DATA, END, and STOP, which have different uses in FORTRAN. Modify Figure 1.11 to produce a DFA that will also reject the words DO and DATA while still accepting all other valid FORTRAN identifiers.

1.19. Consider the machine defined in Example 1.11. This machine accepts most real number constants in scientific notation. However, this machine does have some (possibly desirable) limitations. These limitations include requiring that a 0 precede the decimal point when specifying a number with a mantissa less than 1.

(a) Modify Figure 1.13 so that it will accept the set of real-number constants described by the following BNF:

\[
\begin{align*}
\text{<sign>} & ::= + | - \\
\text{<digit>} & ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\text{<natural>} & ::= \text{<digit>} | \text{<digit><natural>} \\
\text{<integer>} & ::= \text{<natural>} | \text{<sign><natural>} \\
\text{<real constant>} & ::= \text{<integer>} | \text{<integer>.} | \text{<integer><natural>}. | \text{<sign><natural>}. | \text{<natural><integer>}. | \text{<integer><natural>E<integer>} | \text{<natural>E<integer>} | \text{<integer>E<integer>} | \text{<integer>E<integer>}
\end{align*}
\]

(b) Write a program in your favorite programming language to implement the automaton derived in part (a). The program should read a line of text and state whether or not the word on that line was accepted.

1.20. Show that part (i) of Definition 1.11 is implied by parts (ii) and (iii) of that definition.

1.21. Develop a more succinct description of the transition function given in Example 1.9 (compare with the description in Example 1.10).

1.22. Let the universal set be \( \{a, b\}^* \). Give an example of

(a) A finite set.
(b) A cofinite set.
(c) A set that is neither finite nor cofinite.

1.23. Consider the DFA given in Figure 1.27.

(a) Specify the quintuple for this machine.
(b) Describe the language defined by this machine.

1.24. Consider the set consisting of the names of everyone in China. Is this set a FAD language?

1.25. Consider the set of all legal infix arithmetic expressions over the alphabet \( \{A, B, +, -, \times, /\} \) without parentheses (assume normal precedence rules apply). Is this set a FAD language? If so, draw the machine. (Assume that \(-\) represents both negation and subtraction.)

(a) What aspect of the machine determines whether $\lambda \in L(M)$?

(b) Specify a condition that would guarantee that $L(M) = \Sigma^*$.

(c) Specify a condition that would guarantee that $L(M) = \emptyset$.

1.27. Construct deterministic finite automata to accept each of the following languages.

(a) $\{ x \in \{a, b, c\}^* \mid \text{abc is a substring of x} \}$

(b) $\{ x \in \{a, b, c\}^* \mid \text{acaba is a substring of x} \}$

1.28. Consider Example 1.14. The vending machine had as input nickels, dimes, and quarters. When 30¢ had been deposited, a candy bar could be selected. Modify this machine to also accept pennies, denoted by $p$, as an additional input. How does this affect the number of states in the machine?

1.29. (a) Describe the language defined by the following quintuple (compare with Figure 1.28).

$$
\begin{align*}
\Sigma &= \{a, b\} \\
S &= \{t_0, t_1\} \\
s_0 &= t_0 \\
F &= \{t_1\}
\end{align*}
$$

(b) Rigorously prove the statement you made in part (a). Hint: First prove the inductive statement

$$
P(n) : (\forall x \in \Sigma^n)(|\delta(t_0, x) = t_0 \iff |x|_b \text{ is even}) \land (|\delta(t_0, x) = t_1 \iff |x|_b \text{ is odd})).
$$

1.30. Consider a vending machine that accepts as input pennies, nickels, dimes, and quarters and dispenses 10¢ candy bars.

(a) Draw a DFA that models this machine.
(b) Define the quintuple for this machine.
(c) How many states are absolutely necessary to build this machine?

1.31. Consider a vending machine that accepts as input nickels, dimes, and quarters and dispenses 10¢ candy bars.
   (a) Draw a DFA that models this machine.
   (b) How many states are absolutely necessary to build this machine?
   (c) Using the standard encoding conventions, draw a circuit diagram for this machine (include <EOS> but not <SOS> in the input alphabet).

1.32. Using the standard encoding conventions, draw a circuit diagram that will implement the machine given in Exercise 1.29, as follows:
   (a) Implements both <EOS> and <SOS>.
   (b) Uses neither <EOS> nor <SOS>.

1.33. Using the standard encoding conventions, draw a circuit diagram that will implement the machine given in Exercise 1.7, as follows:
   (a) Implements both <EOS> and <SOS>.
   (b) Uses neither <EOS> nor <SOS>.

1.34. Modify Example 1.12 so that it correctly handles the <SOS> symbol; draw the new circuit diagram.

1.35. Using the standard encoding conventions, draw a circuit diagram that will implement the machine given in Example 1.6, as follows:
   (a) Implements both <EOS> and <SOS>.
   (b) Uses neither <EOS> nor <SOS>.

1.36. Using the standard encoding conventions, draw a circuit diagram that will implement the machine given in Example 1.10, as follows:
   (a) Implements both <EOS> and <SOS>.
   (b) Uses neither <EOS> nor <SOS>.

1.37. Using the standard encoding conventions, draw a circuit diagram that will implement the machine given in Example 1.14 (include <EOS> but not <SOS> in the input alphabet).
1.38. Using the standard encoding conventions, draw a circuit diagram that will implement the machine given in Example 1.16; include the <SOS> and <EOS> symbols.

1.39. Let $\Sigma = \{a, b, c\}$. Let $L = \{x \in \{a, b, c\}^* \mid |x|_b = 2\}$.

(a) Draw a DFA that accepts $L$.
(b) Formally specify the five parts of a DFA that accepts $L$.

1.40. Draw a DFA accepting $\{x \in \{a, b, c\}^* \mid$ every $b$ in $x$ is eventually followed by $c$\}; that is, $x$ might look like $baabaca$, or $bcacc$, and so on.

1.41. Let $\Sigma = \{a, b\}$. Consider the language consisting of all words that have neither consecutive $a$s nor consecutive $b$s.

(a) Draw a DFA that accepts this language.
(b) Formally specify the five parts of a DFA that accepts $L$.

1.42. Let $\Sigma = \{a, b, c\}$. Let $L = \{x \in \{a, b, c\}^* \mid |x|_a \equiv 0 \text{ mod } 3\}$.

(a) Draw a DFA that accepts $L$.
(b) Formally specify the five parts of a DFA that accepts $L$.

1.43. Let $\Sigma = \{a, b, (, )\}$. Recall that a Pascal comment is essentially of the form: $(*$ followed by most combinations of letters followed by the first occurrence of $*)$. While the appropriate alphabet for Pascal is the ASCII character set, for simplicity we will let $\Sigma = \{a, b, (, )\}$. Note that $(b(b(a)b)*$ is a single valid comment, since all characters prior to the first $*$ (including the second $*$) are considered part of the comment. Consequently, comments cannot be nested.

(a) Draw a DFA that recognizes all strings that contain exactly one valid Pascal comment (and no illegal portions of comments, as in $aa(b(b)*b b)(b a)*$, which should be rejected).
(b) Draw a DFA that recognizes all strings that contain zero or more valid (that is, unnested) Pascal comments. For example, $a(b(b(b)b*b)a)*a a$ and $a(*)a(*)b(*)a b(*)$ is valid.

1.44. (a) Is the set of all postfix expressions over $\{A, B, +, -, \times, /\}$ with two or fewer operators a FAD language? If it is, draw a machine.
(b) Is the set of all postfix expressions over $\{A, B, +, -, \times, /\}$ with four or fewer operators a FAD language? If it is, draw a machine.
(c) Is the set of all postfix expressions over $\{A, B, +, -, \times, /\}$ with eight or fewer operators a FAD language? If it is, draw a machine.
(d) Do you think the set of all postfix expressions over $\{A, B, +, -, \times, /\}$ is a FAD language? Why or why not?

1.45. Let $\Sigma = \{a, b, c\}$. Consider the language consisting of all words that begin and end with different letters.

(a) Draw a DFA that accepts this language.
(b) Formally specify the five parts of a DFA that accepts this language.
1.46. Let $\Sigma = \{a, b, c\}$.

(a) Draw a DFA that rejects all words for which the last two letters match.
(b) Formally specify the five parts of the DFA.

1.47. Let $\Sigma = \{a, b, c\}$.

(a) Draw a DFA that rejects all words for which the first two letters match.
(b) Formally specify the five parts of the DFA.

1.48. Prove that the empty word is unique; that is, using the definition of equality of strings, show that if $x$ and $y$ are empty words then $x = y$.

1.49. For any two strings $x$ and $y$, show that $|xy| = |x| + |y|$.

1.50. (a) Draw the DFA corresponding to $C = \langle \{a, b, c\}, \{t_0, t_1\}, t_0, \delta, \{t_1\} \rangle$, where

$$
\begin{align*}
\delta(t_0, a) &= t_0, & \delta(t_1, a) &= t_0, \\
\delta(t_0, b) &= t_1, & \delta(t_1, b) &= t_1, \\
\delta(t_0, c) &= t_1, & \delta(t_1, c) &= t_0.
\end{align*}
$$

(b) Describe $L(C)$.

(c) Using the standard encoding conventions, draw a circuit diagram for this machine (include <EOS> but not <SOS> in the input alphabet).

1.51. Let $\Sigma = \{I, V, X, L, C, D, M\}$. Recall that $VVVI$ is not considered to be a Roman numeral.

(a) Draw a DFA that recognizes strict-order Roman numerals; that is, 9 must be represented by $VIII$ rather than $IX$, and so on.

(b) Draw a DFA that recognizes the set of all Roman numerals; that is, 9 can be represented by $IX$, 40 by $XL$, and so on.

(c) Write a Pascal program based on your answer to part (b) that recognizes the set of all Roman numerals.

1.52. Describe the set of words accepted by the DFA in Figure 1.9.

1.53. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Let

$L_n = \{x \in \Sigma^* \mid \text{the sum of the digits of } x \text{ is evenly divisible by } n\}$. Thus,

$L_7 = \{\lambda, 0, 7, 00, 07, 16, 25, 34, 43, 52, 59, 61, 68, 70, 77, 86, 95, 000, 007, \ldots\}$.

(a) Draw a machine that will accept $L_7$.

(b) Formally specify the five parts of the DFA given in part (a).

(c) Draw a machine that will accept $L_3$.

(d) Formally specify the five parts of the DFA given in part (c).

(e) Formally specify the five parts of a DFA that will recognize $L_n$.

1.54. Consider the last row of Table 1.3. Unlike the preceding three rows, the outputs in this row are not marked with the don’t-care symbol. Explain.