Chapter 2 introduces the concept of a Finite Automata (or FA). An FA has several properties:

*It is theoretical. It allows computer scientists to talk about general mathematical properties of computers.*

*It is tied to what are called formal languages. A formal language is also theoretical, and is used to think mathematically about programming languages (like Java).*

*It can be quite useful in real life applications. We will look at one such application shortly.*
Basics of Finite Automata

Let’s start with an example FA drawn using a state-transition diagram.

- States are represented as numbered circles.
- There is always one initial state.
- There must be one or more accepting states (also called final states).
- Arrows between states represent state-transition. They are labeled with an input token from the problem alphabet (0 and 1 in this case).
- A ? Means an incomplete piece – have to fill it in sooner or later.

Figure 2.2  A possible state-transition diagram
“Running” a FA

Make the initial state the current state

While current state is not an accepting state or a failure state do the following

\{
    Read the next input (must be from alphabet; 0 or 1 in this example).
    Choose the arc that corresponds to the token read and follow it.
    Make the resulting state the new current state.
\}

If current state is an accepting state, print “ACCEPT” else print “REJECT”.

![Diagram](image-url)
An FA used as a language recognizer

You can feed an FA a series of “sentences” from an alphabet. A sentence is a combination of tokens from the alphabet.

Given an alphabet of 0 and 1, here are four sentences:

0, 1, 0110110, 1010

I hope you see that there are an infinite number of sentences for any non-empty alphabet.

A language is defined as some subset of all sentences. So for any given sentence, we can say that it is either in the language or not in the language.

Our goal is to use an FA to make this “in or not in” decision. Thus, the FA will accept or not accept a sentence depending on whether it is in the language.

Before going further, here is some useful notation for describing sentences:

A * (star), left paren and right paren are special symbols that cannot be in the alphabet. They allow you to define a repeating sequence. For example:

(b)* means zero or more occurrences of the token b (assumes b is in the alphabet).

(aba)* means zero or more occurrences of the sequence aba.

ab(a)*b means the sequence ab followed by zero or more occurrences of a followed by a single b.
The problem the book sets up is to “reverse engineer” a FA that will accept strings in some language $L_A$.

You are given some example strings above.

Any ideas on how to characterize the language using $(...)^*$ notation? See any patterns?

How about $01(001)^*01$?

Why might this be a wrong guess?

Well, maybe $0100100100100101$ is not in the language. Hmmmm.

Let’s try to build an FA for $01(001)^*01$ in any case.
First cut at an FA

In this case, we find ourselves beset with doubts. For example, the following words are accepted by the black box:

- 0101
- 0100101
- 0100100101
- 0100100100101

Let’s try on board. You can use a question mark for places you are not sure about.

**Figure 2.2** A possible state-transition diagram
So have we found the answer?

Is this the only FA that will accept these strings?
Before we go over end-of-chapter problems ...

Let’s look at another way to view an FA, i.e., as a state-transition *table* (as opposed to a state-transition diagram).

![State-transition diagram](image)

*Figure 2.2  A possible state-transition diagram*

<table>
<thead>
<tr>
<th>state/input</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ok, now let’s try Problem 1b (page 12)

Define an FA that accepts the language 0+1.
Definition: + means OR. So will accept either 0 or 1.
Let’s draw it first, then do table.

<table>
<thead>
<tr>
<th>state/input</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>reject</td>
<td>reject</td>
</tr>
<tr>
<td>2</td>
<td>reject</td>
<td>reject</td>
</tr>
</tbody>
</table>
Finally, let’s look at practical use

Chapter 2 lays out a theoretical problem: given a black-box, figure out the language that it accepts. Use an FA to represent a parser of the language.

Let’s look at an alternative, *forward engineering*, approach. Define an FA for some problem we have. Use it to define the black box.
A soda machine

- All drinks cost 50 cents. (Yeah, right.)
- The machine will accept quarters and half-dollars.
- The user must select the soda she wants.
- The machine never runs out of soda.
- The machine has a coin-return button.

Let’s think about the alphabet first:
{Deposit-25, Deposit-50, select, coin-return}
A soda machine

Now let’s draw the FA:
A soda machine

How about the table version:

<table>
<thead>
<tr>
<th>State/input</th>
<th>25</th>
<th>50</th>
<th>CR</th>
<th>Sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>Rej</td>
<td>Rej</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>?</td>
<td>2</td>
<td>Rej</td>
</tr>
<tr>
<td>2</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
</tr>
<tr>
<td>3</td>
<td>Rej</td>
<td>Rej</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
</tr>
</tbody>
</table>
A soda machine

Practical issues:
Error messages on most rejects?
Should states 2 and 4 wrap to state 0?
How do we show actions, e.g., give change, dispense soda?

<table>
<thead>
<tr>
<th>State/input</th>
<th>25</th>
<th>50</th>
<th>CR</th>
<th>Sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>Rej</td>
<td>Rej</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>?</td>
<td>2</td>
<td>Rej</td>
</tr>
<tr>
<td>2</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
</tr>
<tr>
<td>3</td>
<td>Rej</td>
<td>Rej</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
</tr>
</tbody>
</table>
State Machines are what we want

When trying to model a problem, we can use a state machine to help us think about behavior, before we code.

Different from FA as follows:

- No real notion of sentence recognition.
- Alphabet is set of events as opposed to tokens.
- Typically no “acceptance” state, but instead infinite loop.
- Add actions to arcs.
Let’s try programming it

public static void main( String [] args ){
    int current_state = 0;
    while( true ){
        int event = readSide();
        if( current_state == 0 && event == 0 )  //row 0
            current_state = 1;
        else if (current_state == 0 && event == 1 )
            current_state = 3;
        else if( current_state == 0 && event == 2 )
            printMsg("Put in money first!");
        else if( current_state == 0 && event == 3 )
            printMsg("Put in money first!");
        else if( current_state == 1 && event == 0 ) //row 1
            current_state = 3;
        else break;  //must be stop event
    }  //while
}  //main

<table>
<thead>
<tr>
<th>State/input</th>
<th>25 (0)</th>
<th>50 (1)</th>
<th>CR (2)</th>
<th>Sel (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rej (0)</td>
<td>Rej (0)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>?</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Rej</td>
<td>Rej</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Rej</td>
<td>Rej</td>
<td>Rej</td>
</tr>
</tbody>
</table>