Finite state machines

Automata

A finite state machine is one type of *automaton* (the plural is *automata*). There are physical automata, made as children's toys or machines to entertain adults. They had mechanical clockwork mechanisms and seemed to mimic animals and birds and even people.

The automata here are theoretical machines, which are defined mathematically, and which 'work' on paper, or in a computer simulation. Because they have mathematical definitions, we can deduce their properties logically, and can prove what they can and cannot do.

Finite state machines

These can be in different *states*. A light switch, for example, has two states, on and off. And they have input. For example, a person clicking the switch. The number of states cannot be infinite, so they are called finite state machines. They are sometimes called finite state automata.

They 'work' step-wise. In each step, there is some input, and the machine switches to a new state (or stays in the same state). The new state depends on the input and the current state. The *transition table* says what the new state is, depending on the current state and input.

For example, the transition table of our light switch is:

current state	input = click
off	on
on	off

We have one row for each state. We have one column for each possible input – but here in our very simple example, there is only one possible input, a click, so just one column.

If the switch is off, a click input changes its state to on. And vice versa.

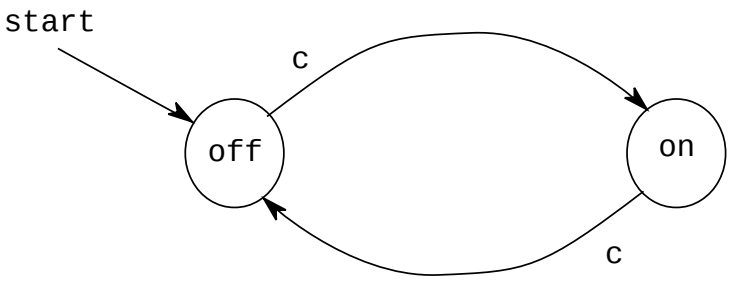
We do not detail *how* the machine works. It might be clockwork, or electronic, or something else. If electronic, it might be logic gates connected up somehow. Those gates might be TTL or CMOS or whatever. All we focus on is the inputs, the states and the transition table. This is so that our model will apply to any type of computer, made out of anything.

The other types of automata are push-down automata and Turing machines.

State transition diagrams

The table defines a FSM. Another way to represent it is in the form of a state transition diagram.

For example our light switch machine is:



Each state is shown in enclosed in a circle, with the start state labelled. An arrow links a state to another, which it transitions to, and the arrow is marked with the input that causes that transition.

Accepting states

As it gets input, the FSM will change state. The sequence of states it goes through will depend on what sequence of input it receives.

Usually an FSM has one or more *accepting states*. Some input sequences end in an accepting state. We say the FSM accepts those input sequences.

For example, in our light switch machine, we might say the on state is the accepting state (since then the light is on). Then this machine accepts an odd number of click inputs.

Formal version

Formally, a FSM has:

A set of *input symbols*, called the alphabet, usually written Σ

A finite set of *states*, Q

A *start state*, $q0 \in Q$

A set of *accepting states*, $F \subset Q$

A *transition function* δ which maps the current state and input to the next state.

δ is a function $Q \times \Sigma \rightarrow Q$

The FSM is the tuple ($Q, \Sigma, \delta, q0, F$)

Web page examples

The web page is at

https://waltermilner.com/FSM

This is a simulation of a FSM coded in JavaScript.

The user can enter a machine configuration, or copy/paste from one of the examples here. The lines must be in the format shown. Lines starting # are comments and are ignored. Apart from comments, lines must be, in order,

- The state labels

- The alphabet of input symbols
- The accept states
- The initial state
- The transition function, with one row for each state, and one column for each input, in the same order as above.

Lists are separated by spaces.

A sequence of input symbols can also be entered. Then click the button to make the machine and set the input sequence.

Then clicking the Step button will execute a single transition, with changes shown in the table below. Or clicking the Accept button will test whether the machine ends in an accept state.

The light switch

A machine configuration file is:

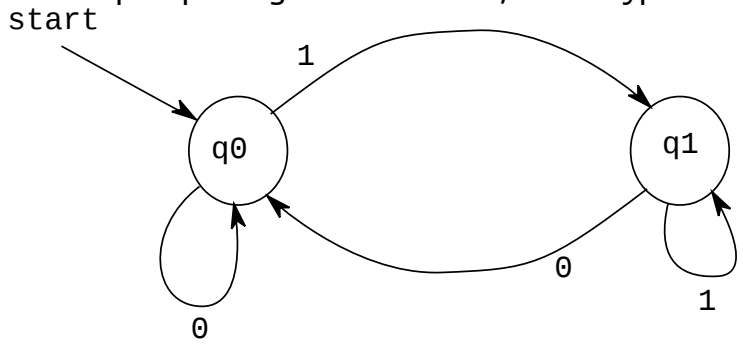
```
# title : light switch
# states
off on
# alphabet
c
# accept states
on
# initial state
off
# transition function – state=row, input = column. Just one column
on
off
```

An accepting input sequence is

```
c c c
```

Flip flop

This models a flip flop in digital electronics, or a D-type latch. It stores one bit.



It has two states. q0 stores a 0, while q1 stores a 1. It has an alphabet of 0 and 1.

It starts in q0. 0 inputs keep it in q0, and a 1 switches to q1

In state q1, 1 inputs keep it in q1, while 0 switches q1 to q0:

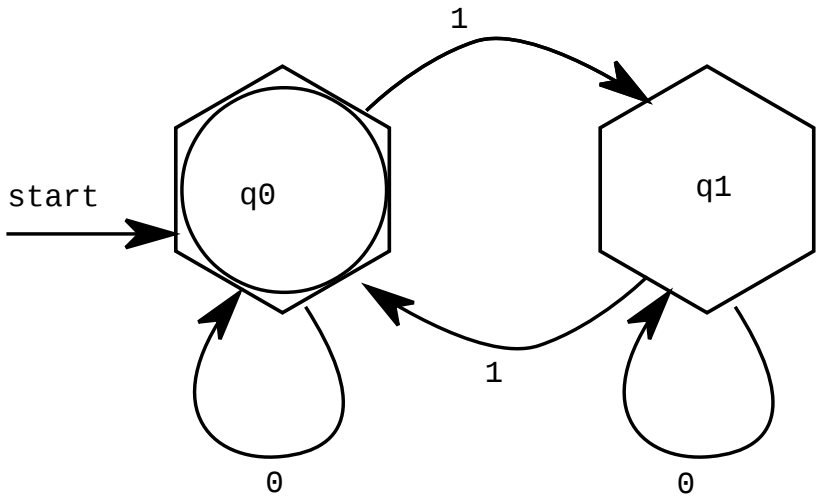
```
# title : flipflop
# states
q0 q1
# alphabet
0 1
# accept states
q1
# initial state
q0
# transition function - state=row
q0 q1
q0 q1
```

An accepting sequence would be

0 0 1 1

or any sequence ending in 1

Parity



This machine accepts even parity input – an even number of 1 inputs.

It starts in q0. 1s flip to the other state, while 0s keep the same state. So it will be in q0 after 0,2,4,6.. numbers of 1 inputs

```
# title : parity - accept even parity
# states
q0 q1
# alphabet
0 1
# accept states
q0
```

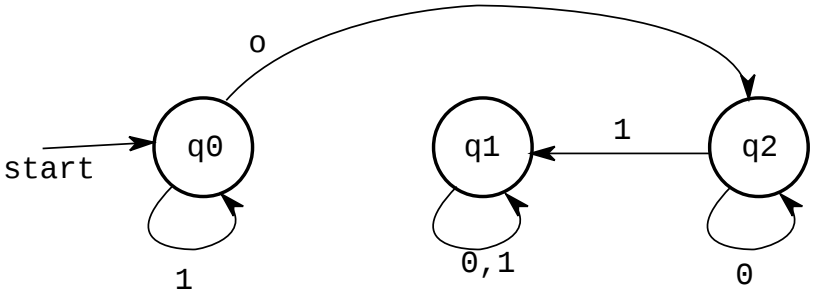
```
# initial state
q0
# transition function - state=row
q0 q1
q1 q0
```

An accepting sequence is 0 0 1 0 1 0 or any even number of 1s.

Contains 01

This is a machine which accepts any string containing 01 (and nothing else). So it accepts, for example, 000001011100, and does not accept 111

It has 3 states, q0 q1 and q2.



q0 is the initial state, and waits for a 0. A 1 keeps it in q0, while a 0 switches to q2

q2 means 'seen a 0, now need a 1'. Another 0 keeps it in q2. A 1 switches to q1.

q1 is the accept state. Any input, 0 or 1, keeps it in q1.

```
# title : 01
# states
q0 q1 q2
# alphabet
0 1
# accept states
q1
# initial state
q0
# transition function - state=row
q2 q0
q1 q1
q2 q1
```

Two digit number

This accepts all two digit decimal numbers:

```
# title : two digit
# states
q0 q1 q2 q3
# alphabet
0-9
```

```
# accept states
q2
# initial state
q0
# transition function - state=row
q1
q2
q3
q3
```

This uses 0-9 as the alphabet. This is just a convenience. We could have listed 0 1 2 3 and so on, and had 10 columns in the transition table. 0-9 is equivalent.

We start in state q0. Any digit moves to q1, and from there, a second digit moves to q2, the accepting state. Anything after that moves to q3, and in q3 anything input stays there. So only precisely 2 digits are accepted.