**CS2303 - Winter 2005**
**Notes on Finite State Machines**

Although finite state machines are covered in the course textbook, the way we define them in this course will be slightly different, so the presentation in these notes should be followed instead of the text.

**Motivation:** Suppose we want to solve a problem like the following:

Given a string of zeroes and ones, we want to recognize when the input contains the substring 001 but does not start with 11.

We're going to design a simple, theoretical "machine" that will be able to solve pattern recognition problems like this one.

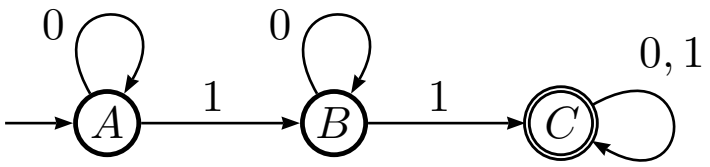
A finite state machine (or finite automaton) $M$ is an abstract model of a very simple computer (with limited memory). It has the following components:

- a finite set $Q$ of *states*
- a finite set $\Sigma$ of *input symbols*
- an *initial state* $q_0 \in Q$
- a *transition function* $\delta : Q \times \Sigma \to Q$
- a set $A \subseteq Q$ of *accepting states*

The machine starts in the initial state. It then receives input, one symbol at a time. Upon receiving each input symbol, the transition function determines the next state of the machine. If it was in state $q$ and it receives input symbol $a$, then $\delta(q, a)$ tells us the next state of the machine. When the end of an input string is reached, the machine will be in some state; let's call it $q_f$. If $q_f$ is a member of the set $A$ of accepting states, then we say that the input string is *accepted* by the machine. Otherwise, the input string is not accepted.

**Example 1:**

The finite state machine described below will accept all strings of zeroes and ones that contain at least two ones. Instead of describing $Q$, $\Sigma$, $q_0$, $\delta$ and $A$, we almost always represent a finite state machine with a diagram like the following:




The diagram above is a nice, graphical way of representing all the information that would be in the official, formal description of a finite state machine. The components of the finite state machine above are the following:

- The set of states is $\{A, B, C\}$.
- The set of input symbols is $\{0, 1\}$.
- The initial state is $A$. (This is indicated by the unlabelled arrow going into $A$.)
- The transition function $\delta$ can be described by the following *transition table*:

| | 0 | 1 |
|---|---|---|
| A | A | B |
| B | B | C |
| C | C | C |

  The top row tells us that, if we're in state $A$, an input of 0 causes us to stay in state $A$, while an input of 1 causes us to move to state $B$.

- The set of accepting states (indicated by a double circle) is $\{C\}$.

If the sequence of symbols in an input string will take us to state $C$, then that string is accepted by the machine; otherwise, it is not accepted.

Let's consider the example input string `1011000` and trace how the finite state machine would process it.

- The machine starts in state $A$.
- The first input symbol is 1. When we're in state $A$ and we see an input of 1, the diagram (or the transition table) tells us to move to state $B$.
- The second input symbol is 0. When we're in state $B$ and we see an input of 0, we stay in state $B$.
- The third input symbol is 1. This causes us to move to state $C$. Note that, in this example, once we're in state $C$, there is no way to leave. We will always stay in state $C$, no matter how much additional input we see. Therefore, in this case, the rest of the input (1000) doesn't really matter. We know that we will end up staying in state $C$ and that the string will be accepted.
- This is the right decision. The string *does* contain two or more ones, and so we *should* accept it.

Let's trace a second example through our machine: 0010.

- The machine starts in state $A$.
- The first input symbol is 0. When we're in state $A$ and we see an input of 0, the diagram (or the transition table) tells us to stay in state $A$.
- This happens again upon seeing the second 0 in the input.
- The third input symbol is 1. This causes us to move to state $B$.
- Finally, the fourth input symbol is 0 and we stay in state $B$. Because the input is now finished, and because we did not end up in an accepting state, this string is not accepted.
- Again, this is the right decision. The string does not have two or more ones, and so it should not be accepted.

It is worthwhile at this point to think about why our machine had three states. How would we have designed the machine from scratch, given the problem we were trying to solve? Given the description of the problem, we know that the only thing we need to remember about a string is the number of ones we have seen so far. This is what the three different states represent. Any time we're in state $A$, it means that we have not seen any ones yet. In state $B$, we have seen 1 one so far. In state $C$, we have seen 2 or more ones.

As we look at more complicated problems, the role of the states will always be the same. As we process an input string, we will want to remember one or more pieces of information about the part of the string we've seen so far. Each state will represent a different possible case that could arise.

For example, if we wanted to recognize strings of letters `a-z` that contained at least two `e`'s, at least three `s`'s, and no `n`'s, we would need to read input symbols from the string and keep track of:

1) How many `e`'s have we seen so far (0, 1, at least 2)?
2) How many `s`'s have we seen so far (0, 1, 2, at least 3)?
3) How many `n`'s have we seen so far (0, at least 1)?

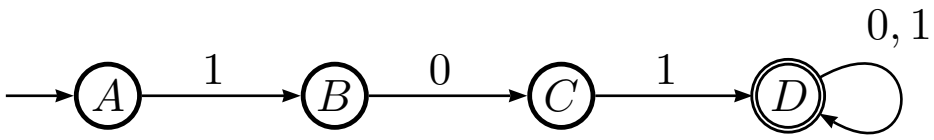
We would need 24 different states to keep track of all possible combinations of answers to questions 1, 2, 3:

| State name | `e`'s seen so far | `s`'s seen so far | `n`'s seen so far |
|---|---|---|---|
| $S_0$ | 0 | 0 | 0 |
| $S_1$ | 0 | 0 | $\geq 1$ |
| $S_2$ | 0 | 1 | 0 |
| $S_3$ | 0 | 1 | $\geq 1$ |
| $S_4$ | 0 | 2 | 0 |
| $S_5$ | 0 | 2 | $\geq 1$ |
| $S_6$ | 0 | $\geq 3$ | 0 |
| $S_7$ | 0 | $\geq 3$ | $\geq 1$ |
| $S_8$ | 1 | 0 | 0 |
| $S_9$ | 1 | 0 | $\geq 1$ |
| $S_{10}$ | 1 | 1 | 0 |
| $S_{11}$ | 1 | 1 | $\geq 1$ |
| $S_{12}$ | 1 | 2 | 0 |
| $S_{13}$ | 1 | 2 | $\geq 1$ |
| $S_{14}$ | 1 | $\geq 3$ | 0 |
| $S_{15}$ | 1 | $\geq 3$ | $\geq 1$ |
| $S_{16}$ | $\geq 2$ | 0 | 0 |
| $S_{17}$ | $\geq 2$ | 0 | $\geq 1$ |
| $S_{18}$ | $\geq 2$ | 1 | 0 |
| $S_{19}$ | $\geq 2$ | 1 | $\geq 1$ |
| $S_{20}$ | $\geq 2$ | 2 | 0 |
| $S_{21}$ | $\geq 2$ | 2 | $\geq 1$ |
| $S_{22}$ | $\geq 2$ | $\geq 3$ | 0 |
| $S_{23}$ | $\geq 2$ | $\geq 3$ | $\geq 1$ |

So, when asked to design a finite state machine to accept a particular class of strings, ask yourself the questions: *What information do we have to keep track of? What possible cases could arise?* This will tell you what states you must have in your machine.

**Example 2:**
Design a finite state machine that will accept all strings of zeroes and ones that contain the substring 101.

In this case, what do we want to remember? If we have already seen the substring 101, then we should be in an accepting state that we will never leave. In particular, if we see 101 at the very beginning of a string, we should move from the initial state (A) to an accepting state (D) by following three transitions: A to B when we see the first one, B to C when we see the 0, and then C to D when we see the second one.




However, this is not a complete finite state machine. For a finite state machine to be complete (and for it to work correctly on all possible inputs), we must define a transition from *every* state on *every* possible input symbol.

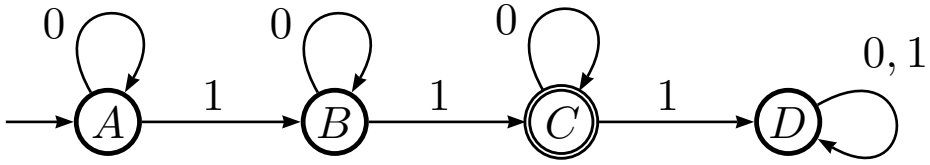
In our example, what should happen in state C if we see a zero? What should happen in state B if we see a one? What should happen in state A if we see a zero?

Basically, if we're in state C, we are remembering that we have just seen the sequence 10. In case we happen to see a 1 as the next input symbol, we want to be in a position where we can move directly to our accepting state upon seeing a 1. However, if we see a zero instead, what should we do? By reading a zero as the next input symbol, we are undoing all the "good" that the substring of 10 did in moving us towards the accepting state. We must go all the way back to the beginning and, in order to make any progress towards the accepting state, we must start over with a new instance of the substring 101.

*[This example will be completed in class.]*

**Example 3:**
Design a finite state machine that will accept all strings of zeroes and ones that contain *exactly* two ones.

In this case, our machine should be exactly the same as Example 1 (at least two ones), with one exception. In Example 1, as soon as we saw a second one, we knew that we would want to accept the string. Therefore, it was OK to move to state $C$ as soon as we saw a second one and then stay there, regardless of any additional input symbols. In this example, we want to accept a string if it has exactly two ones, but not if it has three or more ones, so we should enter state $C$ when we see a second one, but then *leave* state $C$ (and make sure we never return) as soon as we see a third one.



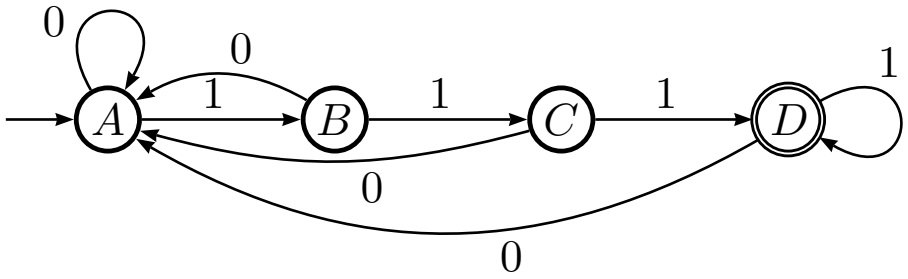

A state such as $D$ in the example above is often referred to as a *dead state*. As soon as we enter a dead state, there is no way to exit it, and we are guaranteed that we will not accept the string, regardless of any additional input.

In addition to examples like the one above, dead states are appropriate states to add to finite state machines in examples like `Design a finite state machine that will accept all strings of zeroes and ones that` **do not** `contain the substring 101.` You can set up a machine to do this by ensuring that, as soon as you see the substring 101, you enter into a dead state and are unable to leave.

**Exercise:** Design a finite state machine that will accept all strings of zeroes and ones that **do not** contain the substring 101.

**Example 4:** Draw a finite state machine that will accept all strings of zeroes and ones that end in 111.

In this case, we want to make sure that we enter an accepting state whenever we have just seen the substring 111. In case that happens to be the end of the string, we want to make sure that we are in a position to accept it. However, while we're in the accepting state, if we see an input symbol of zero, we should leave the accepting state and go back to the very beginning. In order to get back to the accepting state, we would have to see another instance of the substring 111.




On the other hand, seeing another *one* while we're in state D is perfectly fine. We now know that the string ends in 1111, but this is fine; it still ends in 111.

**Exercise:** Draw a finite state machine that will accept all strings of zeroes and ones that *start* with 111.

**Example 5:** Draw a finite state machine that will accept all strings of zeroes and ones that contain the substring 00 **and** contain at least two ones.

In this case, we have to remember two different facts about the string we've seen so far. Have we seen the substring 00 yet (or have we just seen the first 0 in what might end up being a substring of 00)? How many ones have we seen so far?

The states in our finite state machine must capture all the possible combinations of answers to those questions.

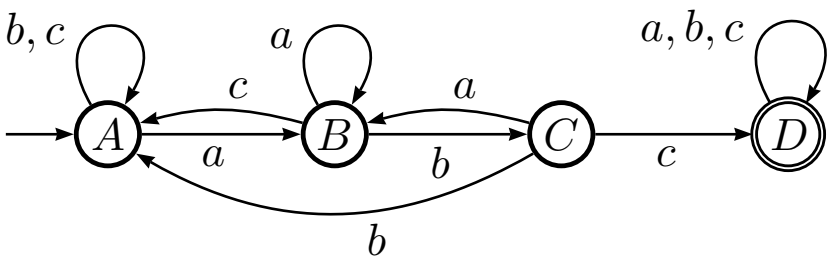
| State name | Have we seen 00 yet? | Number of ones seen so far |
|---|---|---|
| $S_0$ | No | 0 |
| $S_1$ | No | 1 |
| $S_2$ | No | $\geq 2$ |
| $S_3$ | No, but one zero was just seen | 0 |
| $S_4$ | No, but one zero was just seen | 1 |
| $S_5$ | No, but one zero was just seen | $\geq 2$ |
| $S_6$ | Yes | 0 |
| $S_7$ | Yes | 1 |
| $S_8$ | Yes | $\geq 2$ |

Of these states, only $S_8$ should be an accepting state. What should the entire FSM look like?

*[This will be completed in class.]*

**Example 6:** This example is provided just to show that we can construct finite state machines when we have more than two possible input symbols.

Draw a finite state machine that accepts strings of `a`'s, `b`'s and `c`'s that contain the substring `abc`.




**Example 7:** Something we haven't seen yet is an example of a finite state machine that contains more than one accepting state. There is nothing in the definition of a finite state machine that says that we can't have several accepting states.

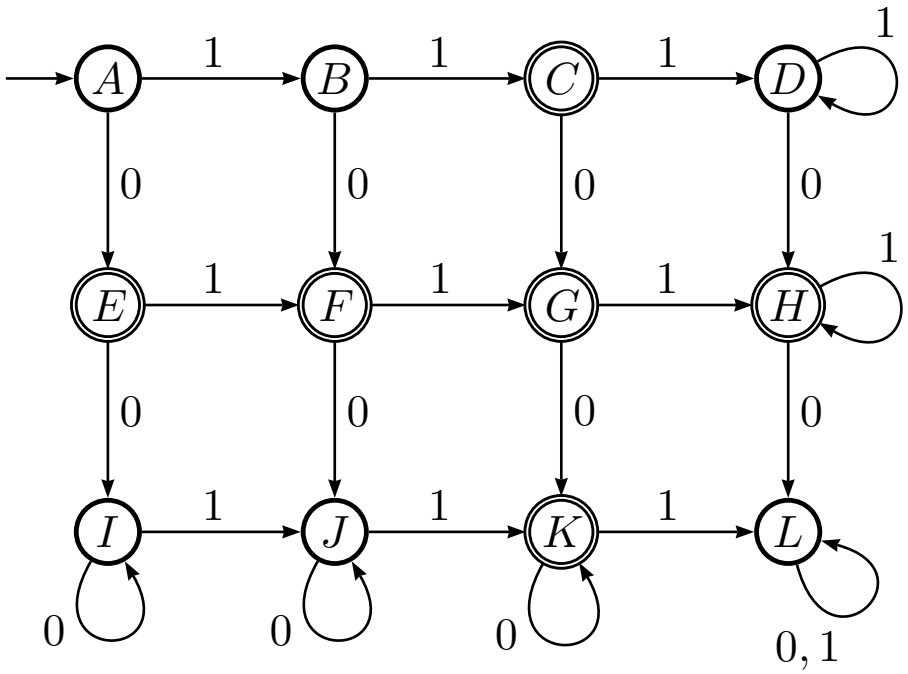
Draw a finite state machine that will accept strings of zeroes and ones that contain exactly two ones **or** exactly one zero.

Again, we can create states that capture all the possible cases that might arise, if we were to count the number of ones and the number of zeroes seen so far.

| State name | Number of zeroes seen | Number of ones seen |
|---|---|---|
| A | 0 | 0 |
| B | 0 | 1 |
| C | 0 | 2 |
| D | 0 | $\geq 3$ |
| E | 1 | 0 |
| F | 1 | 1 |
| G | 1 | 2 |
| H | 1 | $\geq 3$ |
| I | $\geq 2$ | 0 |
| J | $\geq 2$ | 1 |
| K | $\geq 2$ | 2 |
| L | $\geq 2$ | $\geq 3$ |

One of the big differences between this example and ones we've seen before is that the description in this case contains the word **or**. As long as we have *either* exactly two ones *or* exactly one zero, we want to accept the string. So, in our list of states above, we should accept a string if we are in states $E, F, G$ or $H$ (because those states represent situations in which the number of zeroes is exactly one) or if we are in states $C, G$ or $K$ (because those states represent situations in which the number of ones is exactly two). Our accepting states should be $C, E, F, G, H$ and $K$.

The finite state machine would look like this:




**Describing finite state machines**

In these notes, we have looked only at the idea of designing finite state machines given a description of the strings that we want to accept. We will also talk in class about looking at a finite state machine and describing the strings that are accepted by the given machine.