CITS2211 Discrete Structures
Finite State Machines

Unit coordinator and Lecturer: Débora Corrêa

What is a computer?

Studying simple machines allows us to study ideas such as memory, information, nondeterminism, and complexity.

Last week we learned that there are more "problems" than there are "programs" to solve them since the latter set is countably infinite and the former set is uncountable.

Over the next few weeks we will study models of computation in order to understand its properties and the fundamental limits.

Why (study finite state machines)?

- What is a computer? What is computation?
- To study computers we need a computational model
- Any model is accurate in some ways but not in others
- We shall start with the simplest model of computation: a Finite State Machine (FSM)

Why (study finite state machines)?

Finite State Machines (FSMs) are (essentially) computers with very small memory.

FSMs are widely used in practice for simple mechanisms: automatic doors, lifts, microwave or washing machine controllers, and many other electromechanical devices.

FSMs are used also to specify communication protocols, user interfaces, web forms, and workflows.

FSMs are the simplest model in a hierarchy that we will use to understand the fundamental question of what is computable.

Reading

Introduction to the Theory of Computation by Michael Sipser

Chapter 1: Regular Languages

Sections 1.1 Finite Automata and 1.2 Nondeterminism

Lecture Outlines

This week we will study

- FSMs and FSM diagrams
- FSM formal definition
- How to read a FSM
- FSM computations
- Accepting strings and recognizing languages

- How to design a FSM
- nondeterministic FSMs
- NFSM formal definition
- The surprising and useful equivalence of NFSMs and FSMs

Finite State Machines

FSM Diagrams

A simple model of computation

- Q: What is a computer?
- A: A laptop, PC, phone, tablet, super-computer, ticket machine, ATM ?
- All of these systems are **reactive**: their actions depend on the inputs they receive
- Their response to a particular stimulus (a signal, or a piece of input) is **not the same** on every occasion.

Example

- For example, in the case of a parking ticket machine, it will not print a ticket when you press the button unless you have already inserted some money.
- The response to the print button depends on the previous history of the use of the system: its **memory**.

Inputs

What **stimuli (inputs)** does a ticket machine take account of ?

Inputs

What **stimuli (inputs)** does a ticket machine take account of ?

- insert some money m,
- press the print ticket button t,
- press the cancel button for refunds r

The **alphabet** of inputs is a set: $\Sigma = \{m, t, r\}$

States

The machine's *memory* is represented by a set of **states**.

For example

1=awaiting coins, 2=ready to print, 3=finished printing

The set of possible states for a given machine is written
$Q = \{1, 2, 3\}$

States are drawn as circles in FSM diagrams.

There are only a **finite** number of possible states allowed for a Finite State Machine.

Transitions

How does computation occur?

The machine has **transitions** from one state to another depending on the stimulus (input) provided.

The **transition function** is of type:

$$Q \times \Sigma \to Q$$

Transitions are drawn as edges between the states in FSM diagrams. Edges are labelled with the input symbol for the transition. For every state, symbol pair there must be a transition to some other state.

To simplify FSM diagrams, we sometimes do *not show* transitions for illegal inputs.

Outputs

In this unit we will **not** study FSMs with an **explicit output alphabet**.

Outputs (if needed) can be implicit in the states (eg a light being on while machine is in some state) or the transitions (eg use a unique symbol for each input-output pair).

If explicit outputs are needed, then they could be written on the transition edges. A machine with outputs is called a **transducer**. You may see such models in some texts.

Starting and Stopping

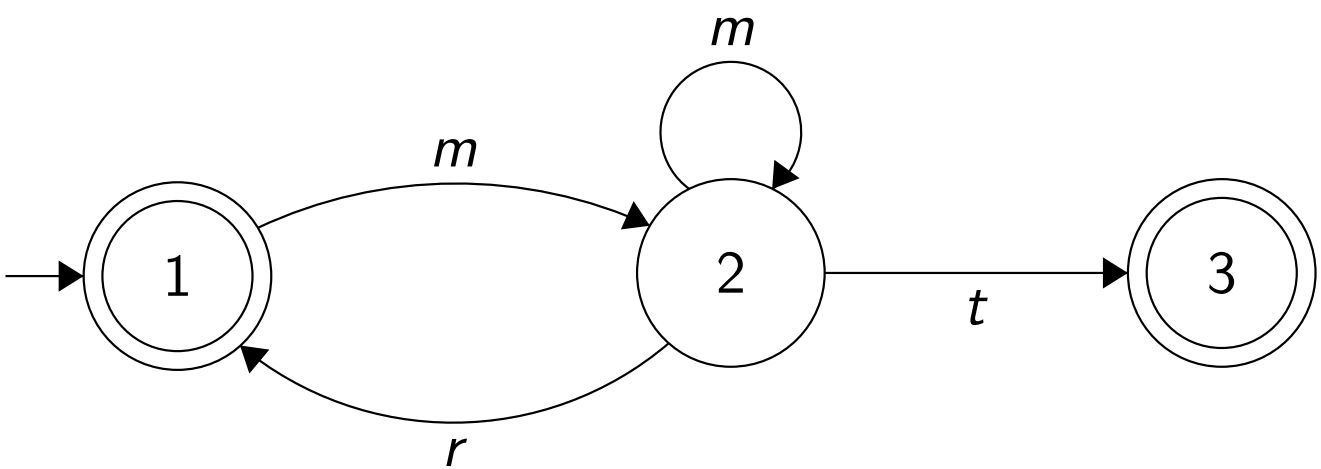
One state from Q is identified as the **starting state**. Think of this as the initial state of the machine before any inputs are received.

The start state is identified by an arrow pointing to it, but not coming from any other state.

A machine can stop in any state: input may cease, or there may be no matching transition to take.

Zero, one or more states from Q may be identified as **accepting states**. These are *good* places to stop. In diagrams, accepting states are denoted by a double circle.

Ticket Machine example



FSM Formal Definition

Formal defn of FSMs

Definition: A finite state machine (FSM) is defined to be a 5-tuple $(Q, q_0, \Sigma, F, \delta)$ where

- Q is a finite set of *states*;
- $q_0 \in Q$ is the start state;
- Σ is a finite alphabet of *input symbols*;
- $F \subseteq Q$ is a set of accepting states (F may be the empty set);
- $\delta\colon Q \times \Sigma \to Q$ is the *state transition function*

(Note: You will find slight variations of this definition in some texts. We will follow Sipser)

Computations of a FSM

Behaviour and Traces

How do we describe the behaviour of a particular FSM?

Definition: A **trace** of an FSM M is a finite sequence of states and transition labels, starting and ending with a state, such that

- the trace is written $s_0, i_1, s_1, i_2, s_2, i_3 \ldots, s_{n-1}, i_n, s_n$
- the first state, s_0 must be the start state of M
- every triple, (s_j, i_{j+1}, s_{j+1}) must be from the state transition relation δ of M

We say $s_0, i_1, s_1, i_2, \ldots, i_n, s_n$ is a trace on the **input string** $i_1 i_2 \ldots i_n$

Example:

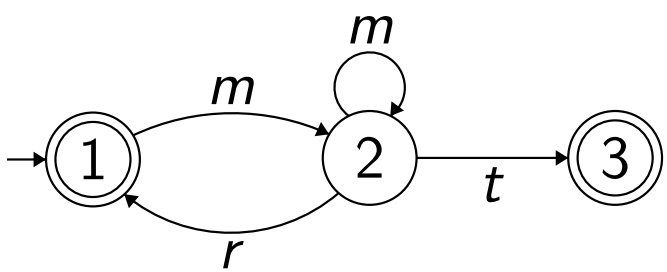
The ticket machine has traces $1, m, 2, m, 2, m, 2, t, 3$ and $1, m, 2, m, 2, m, 2$ for input strings *mmmt* and *mmm* (respectively)

Accepting a String

We say that an FSM **accepts** an **input string** if there is a trace for that input string which leaves the FSM in an accepting state.

Notation: Given an alphabet Σ, the set of all strings formed from that alphabet is denoted by Σ^*.

The ticket machine FSM accepts the strings *mmmt* and *mt* and *mr* but it does not accept *mmm* and it does not accept *t*

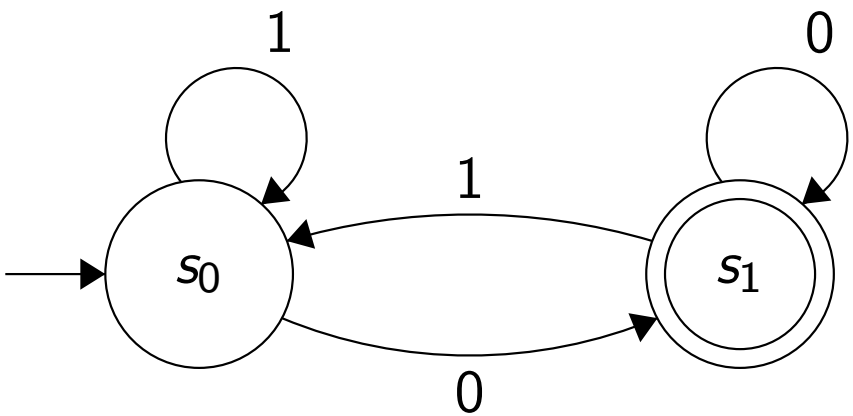


Reading Finite State Machines

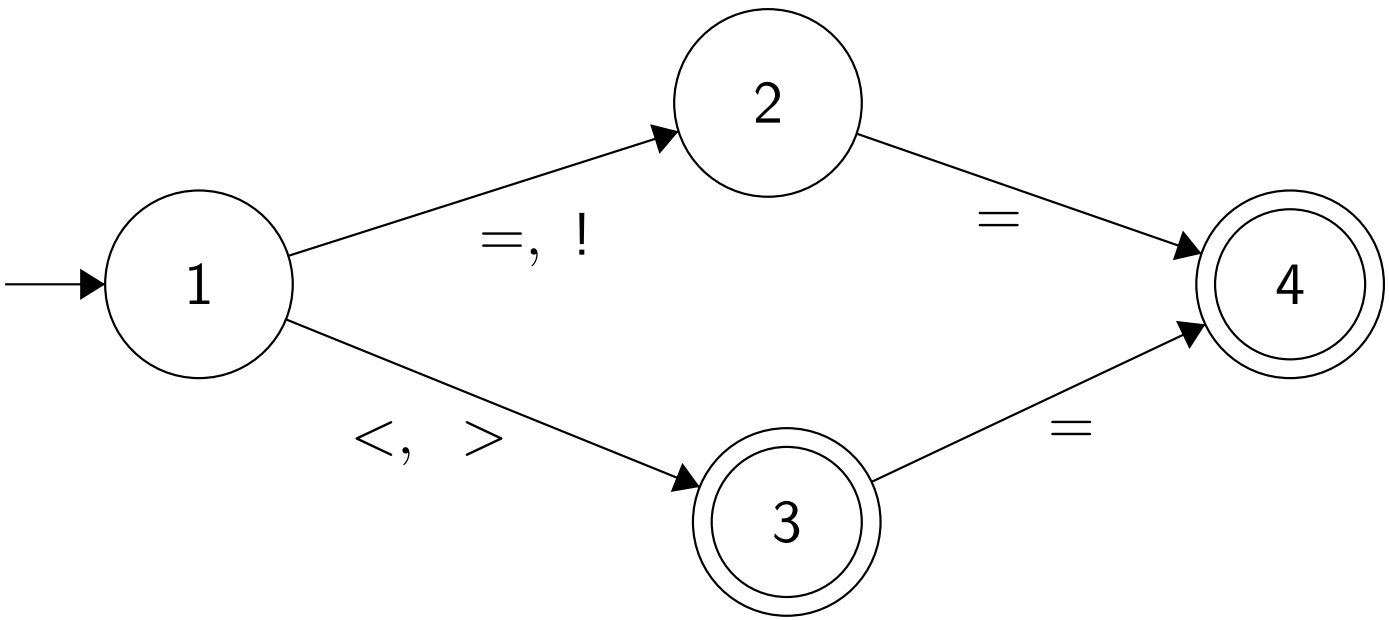
Example FSM M_1

Name the five parts of this FSM?

List some strings it accepts?



Example FSM M_2



The Language Recognised by a FSM

Languages and Recognisers

The set of strings that are accepted by a FSM is known as the **language** that is **recognised** by the FSM.

Given a finite state machine M, the collection of all strings that are accepted by M is called *the language recognised by M* and is denoted $L(M)$.

$$L(M) \subseteq \Sigma^*$$

That is the language recognised by M is a subset of all possible strings of its input alphabet.

Languages and Recognisers

Example:

For the FSM M_1 on slide 23

$L(M_1) = \{w \mid w \in \{0,1\}^* \text{ and } w \text{ ends with a } 0\}$.

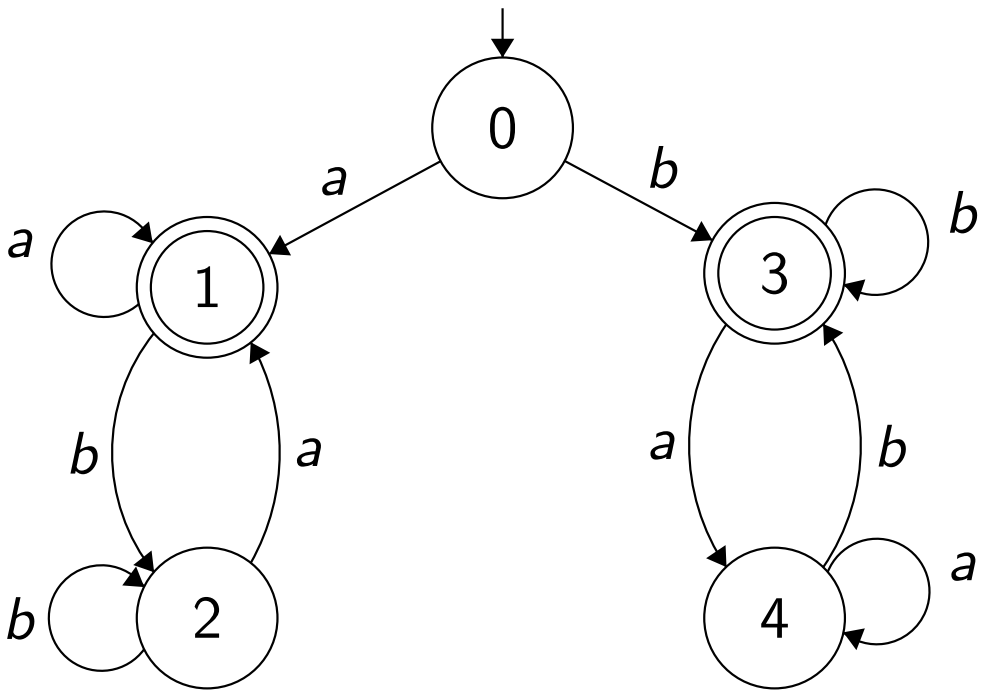
Although accept and recognise have similar meanings in English, we will use *accept* for individual strings and *recognise* for the language which is the set of all strings accepted by an FSM.

An interesting question is, what languages can be recognised by finite state machines? And are there languages that can not be?

Language example

What is the language recognised by this FSM?

Hint: Generate some examples of accepted strings.



Designing an FSM

Steps for designing an FSM

So far we have studied FSMs that already exist. What if you need to write your own?

Design is a creative process so there is no recipe for it. But here are some hints to help you.

Steps for designing an FSM

- Pretend that you are the machine.
- Generate some example strings: both strings that are accepted by the machine and those that are not
- Step through your example strings one symbol at a time and think about how the machine would decide to accept or not
- What do you need to remember about the string as you are reading it? The things you have to remember will become the states of the machine.
- Work out the transitions between states for each state and symbol combination.
- Set the start state as the state you will be in before you have seen any symbols
- Set the accepting state(s) as the states(s) where you want to accept a string.

Designing an FSM

Example: Define an FSM that accepts only strings of alternating 0s and 1s, starting with a 0 and ending with a 1.

Designing an FSM (1)

Think of some example strings

accept: 0101 and 01010101

do not accept: 10100 nor 011 nor 1010

Step through the examples working out what to do at each step

0101: see 0 next need 1, next need another 0 (must alternate), then need another 1, accept this alternating string ending in 1

10101010: see 1 and do not accept because this does not start with a 0

0110: see 0, then 1, then do not accept the next 1 because not alternating

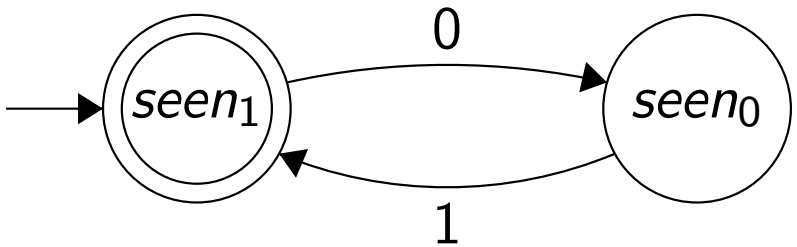
Designing an FSM (2)

What did we need to remember?
Whether we had just seen a 0 (then next must be 1), and whether we had seen a 1 (then next must be 0)
So make 2 states: seen0 and seen1
Transitions: from seen0 next need a 1, and vice versa

Starting state? the first symbol must be a 0, which means you are in seen1 state.

Accepting state(s)? when have just seen a 1 which is the seen1 state. This machine has only one accepting state

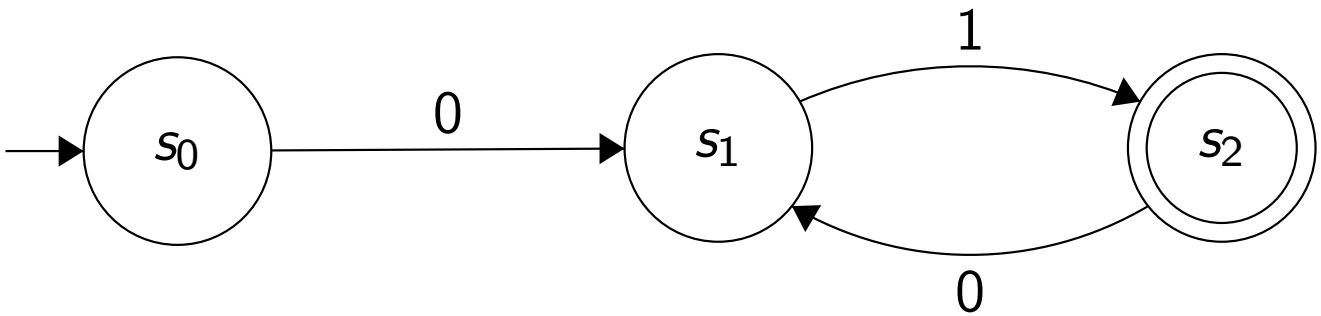
Designing an FSM (3)



Alternative

Here is another possible FSM for accepting exactly strings of alternating 0s and 1s, starting with a 0 and ending with a 1.

How do they differ? Is one any better than the other?



Failure

Failure to accept a sequence of inputs can arise in two ways:

- We can construct trace(s) corresponding to the sequence, but the final state of each trace is not an accepting state.

  e.g. 010 in the above example

- We cannot construct a trace for the given sequence (e.g. 0110 in the above example)

  Note: The missing sequences are implicitly in the transition function, but not shown in the diagram.

More design practice

Example: 1. Design an FSM to accept all binary strings starting with 111

Example: 2. Design an FSM to accept all binary strings with even parity. (That is, with an even number of 1s.)
e.g. 010111 and 1011001 but not 100

Example: 3. Design an FSM to accept all binary strings with the same number of 0s and 1s, but in any order.
e.g. 0101 and 1011100010 but not 010

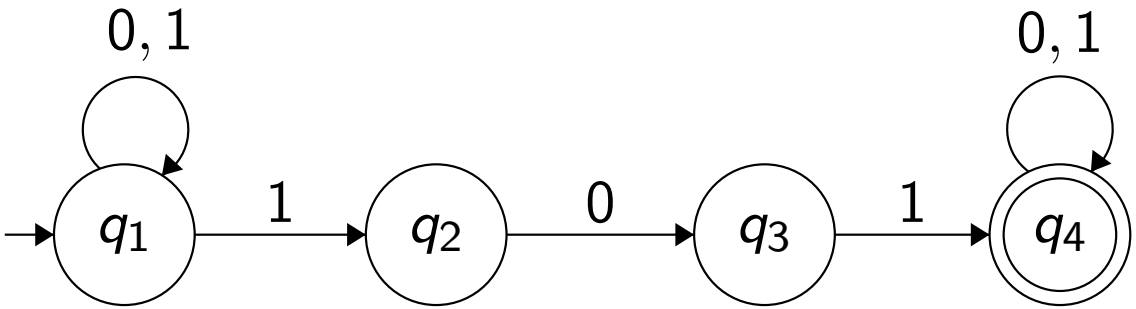
More design practice

Maybe, one of those exercises on the previous pages is impossible.

More on that later.

Nondeterministic FSMS

Nondeterministic FSM



- In an NFSM a state may have **zero, one or many** exiting arrows for each alphabet symbol.
- Recall that a (deterministic) FSM has exactly one transition for each state, symbol pair: *transition function*

NFSM computations

- Suppose we run an NFSM on an input string and come to a state with multiple choices for the current input symbol.
- After reading the symbol the machine splits into multiple copies of itself and follows **all** the possibilities in parallel
- If there are further choices the machine splits again
- Finally, if **any one of the copies** of the machine is in an accept state at the end of the input, the NFSM accepts the input string.

These multiple possible output choices can be thought of as a transition to a **set of states** (rather than a single state).

NFSM formal definition

Formal defn of FSMs

Definition: A nondeterministic finite state machine (NFSM) is defined to be a 5-tuple $(Q, q_0, \Sigma, F, \delta)$ where

- Q is a finite set of *states*;
- $q_0 \in Q$ is the start state;
- Σ is a finite alphabet of *input symbols*;
- $F \subseteq Q$ is a set of accepting states (F may be the empty set);
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the *state transition function*

The transition function in an NFSM takes an input symbol and produces a **set of possible next states**.

FSM and NFSM Definitions

Definition: An input sequence is **accepted** by an FSM iff *at least one trace* for the sequence ends in an accepting state.

Definition: If for any input sequence of an FSM M there is *at most one trace*, then M is said to be **deterministic** (also called DFSM for deterministic finite-state machine).

Definition: Any FSM M that is not deterministic, is said to be **nondeterministic** (NFSM).

The power of FSMs

Essentially an FSM is a model of a computer with an extremely limited amount of memory (Qualitative, not Quantitative). The only mechanism that it has to "remember" something is to move into a different state.

Despite this, we can produce FSMs that do some surprising things, such as adding two numbers.

Question: What are the differences are between deterministic and nondeterministic FSMs? Do they recognise the same or different languages ? Why ?

NFSM example

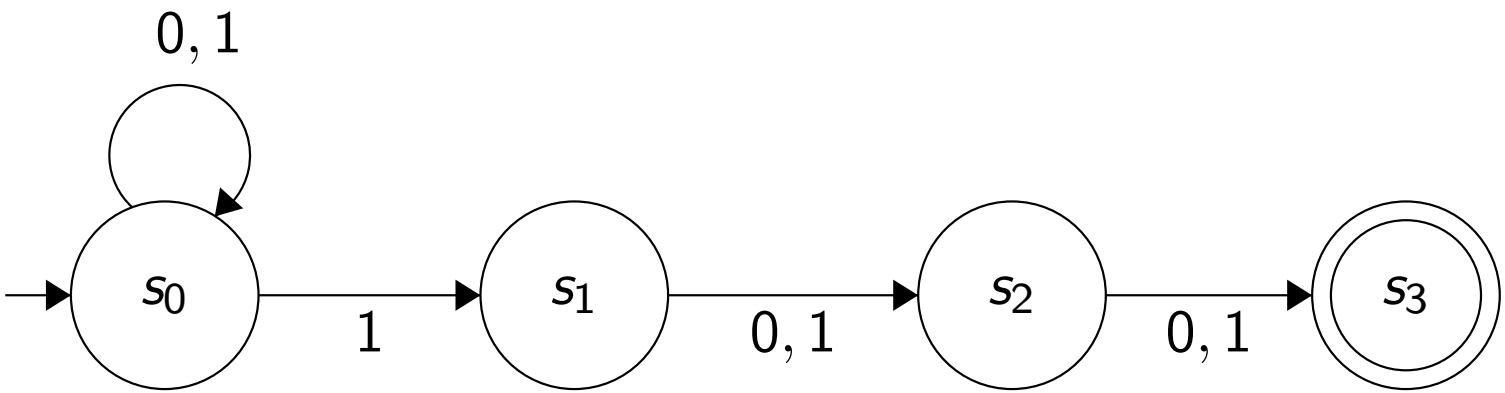
Nondeterminism seems to give the builder great opportunities to be efficient. We can use the power of guessing.

Example: Construct a nondeterministic FSM that accepts binary strings with a 1 in the third last position.

NFSM example

Problem: Construct a NFSM that accepts binary strings with a 1 in the third last position

Solution: Guess where the 1 occurs, then after 2 more symbols the NFSM accepts.

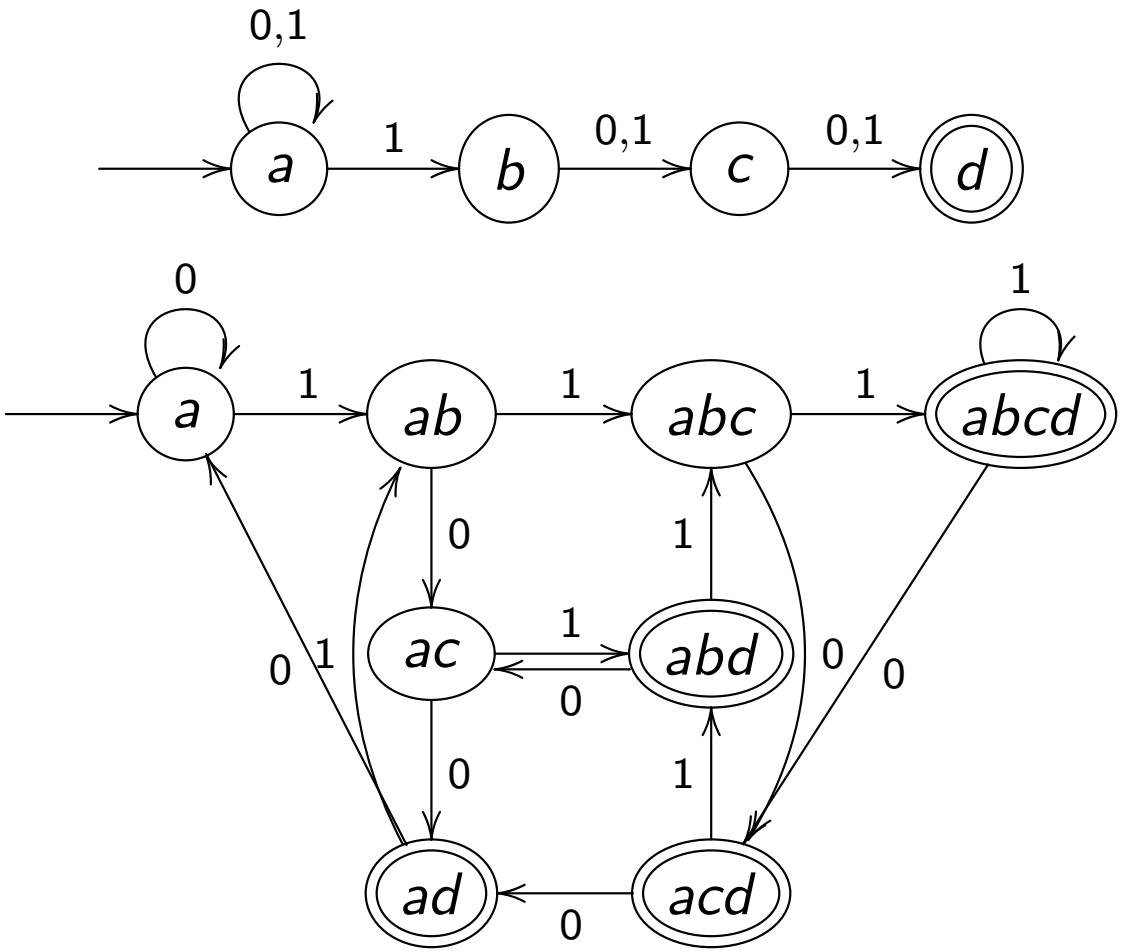


NFSM example, continued

Consider how complicated it is to do the same thing with a deterministic FSM.

Example: Construct a deterministic FSM for the same language.

Example



An NFSM with an equivalent DFSM (constructed via the power set construction which we see in a moment but ignoring unreachable states)

The surprising and useful equivalence of NFSM and DFSM

NFSM and DFSM are equivalent

Theorem: Every NFSM can be converted into an equivalent DFSM

The powerset construction for this conversion was published by M.O. Rabin and Dana Scott in 1959.

Given an NFSM we construct an equivalent DFSM as follows.

- states are $\mathcal{P}(Q)$ (the power set of states of the NFSM)
- initial state is the state corresponding to the union of initial states in the NFSM
- accepting states are those containing any accepting states in the NFSM
- transition function maps each compound state to its matching one

Applications

That NFSM and DFSM are equivalent is surprising:

"guessing" and "calculating" have the same power.

This equivalence is very useful in practice:

- NFSMs are often easier to construct than DFSMs, so solve a problem with an NFSM and then convert it to a DFSM for execution
- However, note for an NFSM with n states, the DFSM may have up to 2^n states
- NFSMs are used for easier proofs of properties, such as closure, in the theory of computation:

Languages as sets

Languages as sets

- We have seen that languages recognized by FSMs are *sets* of strings, and the set of strings recognized by any FSM is a subset of Σ^*
- Since a language is a set, this means we can apply the usual set operations to it – we can take the union or intersection or difference of two languages, for instance
- And we can even take complements, with reference to Σ^* as a universal set

Complements of language

- It turns out the complement of a language recognized by an FSM is *also* a language that can be recognized by an FSM.
- We can construct an FSM to recognize it by swapping the accepting and non-accepting states.

An FSM that recognizes the complement

Given a diagram that represents an FSM M. M recognizes a language $L(M) \subseteq \Sigma^*$. how can we turn it into a FSM diagram that recognises the complement of L (i.e. $\Sigma^* - L(M)$)?

1. If the FSM is non-deterministic, find the equivalent deterministic FSM.
2. Add all 'error' states and missing transitions that might not be shown in the diagram.
3. Turn all accepting into non-accepting states and vice versa.

Summary

- Finite State Machines (FSMs): a simple model of computation with limited memory (Qualitative, not quantitative)
- Acceptors: An FSM accepts a string if that string of input symbols takes the FSM to an accepting state
- Recognisers: An FSM recognises the language comprising the set of all the strings it accepts.
- Nondeterministic FSMs: nondeterministic (guessing) FSMs have the same power as deterministic (calculating) ones.