# ITCS 532:
# 1. Models of Computation

Rob Egrot

# What is computation?

- Given a set $\Gamma$ of first-order sentences, and a first-order sentence $\phi$, does $\Gamma \models \phi$?
- Is there an algorithm that will always answer the above question correctly?
- What exactly is an algorithm anyway?
- What is computation?
- How does this relate to modern computers?

# Finite State Machines

- A *finite state machine* (FSM), AKA *finite automaton*, is an abstract machine.
- At any moment an FSM is in one of a finite number of possible states.
- The state changes in response to input.
- There are only a finite number of possible inputs.
- E.g:
  - ticket machines,
  - vending machines,
  - etc.

# FSM - formal definition

An FSM consists of:

1. A finite set of states $Q$.
2. A distinguished set $H \subseteq Q$. This set contains the *halting* states of $M$. If the machine gets to a state in $H$ then it stops running.
3. A special starting state $q_0 \in Q$.
4. A finite set of possible inputs $\Sigma$ sometimes called the *alphabet* of $M$.
5. A transition function $\delta : (Q \setminus H) \times \Sigma \to Q$. This function controls state change. I.e. $\delta(q, \sigma) = r$ means *if in state $q$ go to state $r$ when receiving input $\sigma$*.

# Warning 1

- There are other definitions for FSMs.
- E.g:
  - $\delta$ may be partial.
  - In this case there must be a rule for dealing with undefined situations.
- Alternative definitions will be essentially equivalent to ours.

# Warning 2

- In the real world, things like ticket machines take user input in real time.
- Users might walk away before completing their purchase.
- The design of the machine must deal with this.
- E.g. after some time of inactivity state of machine must reset.

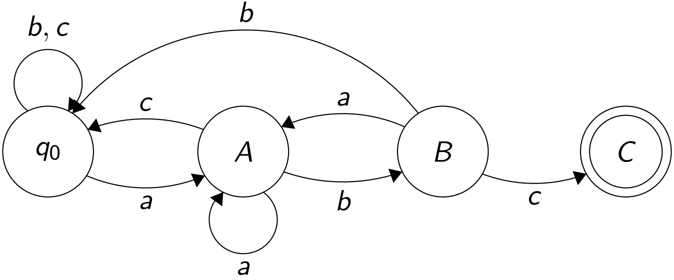
- Waiting time should not be too short or too long.
- Not our problem.
- We abstract away time, and treat input as a pre-determined finite sequence of events.
- We don't worry about what happens after that.

# Running an FSM

- The possible inputs are represented by the symbols in $\Sigma$.
- Sequences of inputs correspond to finite strings from $\Sigma$.
- $\Sigma^*$ is set of all finite strings from $\Sigma$.
- We think of the *input* of an FSM to be a string from $\Sigma^*$.
- Machine acts on each symbol in the input in turn, and changes state according to $\delta$.
- The output is just the state after acting on the final symbol of the input.

# Representing an FSM

This diagram represents an FSM that looks for the sequence *abc* inside strings composed of letters from the alphabet $\Sigma = \{a, b, c\}$.




If it gets to state $C$ it halts, and we have a success. Otherwise once it reaches end of input we consider it a failure. I.e., input does not contain *abc*.

# The formal version

The machine on the previous slide has this formal description:

- $Q = \{q_0, A, B, C\}$.
- $\Sigma = \{a, b, c\}$.
- $\delta : \begin{cases} (q_0, a) \mapsto A \\ (q_0, b) \mapsto q_0 \\ (q_0, c) \mapsto q_0 \\ (A, a) \mapsto A \\ (A, b) \mapsto B \\ (A, c) \mapsto q_0 \\ (B, a) \mapsto A \\ (B, b) \mapsto q_0 \\ (B, c) \mapsto C \end{cases}$
- $q_0$ (the starting state).
- $H = \{C\}$ (the set of halt states).

# Limitations of FSMs

- Finite state machines are useful, but they are quite limited.
- E.g:
  - No FSM to check if a graph is connected.
  - No FSM to check if a given set of polynomial equations has a solution.
  - No FSM to check if a number in binary is prime.
  - Etc.
- A simpler example:
- Why is there no FSM that can add two binary numbers together?

# Memory problems

- Many of the limitations of FSMs come down to memory.
- Specifically, *lack* of memory.
- An FSM records information only in its state.
- Since it has a finite number of states, but inputs can be arbitrarily long, its ability to distinguish between inputs is limited.
- Real computers also have finite memory, but the amount increases all the time.
- Theoretical computer scientists often prefer to study abstract computation by assuming memory is infinite.

# Turing machines

- A Turing machine (TM) is another abstract machine.
- Turing defined this concept to capture the notion of an algorithm.
- It is similar to an FSM but it also has an infinite *tape*.
- The TM can write on the tape, and can access things it has written before.
- The tape provides the TM with an infinite memory.
- A TM has a tape head which moves around on the tape reading symbols.
- During operation, the TM reads a symbol and checks its current state.
- Then it moves into a new state, and acts on the tape in some way, either by writing a new symbol or moving the tape head.

# Turing machines - formal version $(Q, \Sigma, q_0, H, \delta)$

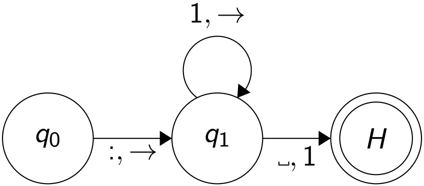
1. A finite set of states $Q$.
2. A finite alphabet $\Sigma$. Also special symbols $\llcorner\!\lrcorner$ (for blank spaces) and $:$ (for start of tape).
3. A one way infinite tape consisting of numbered squares starting at 0. Each square contains one symbol from $\Sigma \cup \{\llcorner\!\lrcorner, :\}$. Symbol $:$ at square 0 (only).
4. A tape head that moves up and down the tape reading symbols.
5. A distinguished starting state $q_0$.
6. A set $H \subseteq Q$ of halting states (maybe partitioned into 'accept' states and 'reject' states).
7. A function $\delta : (Q \setminus H) \times \Sigma \cup \{\llcorner\!\lrcorner, :\} \to Q \times (\Sigma \cup \{\llcorner\!\lrcorner, \to, \leftarrow\})$.

# Running a Turing machine

- The *input* of a TM is the initial state of the tape.
- We assume that the starting tape always contains a finite string from $\Sigma^*$ followed by an infinite sequence of ␣ symbols.
- A *run* of a Turing machine on input $I$ is a sequence of abstract triples representing the state the machine is in, the position of the tape head, and the current state of the tape.
- A run is completely determined by the input $I$ on the tape and the function $\delta$.
- The *output* can either be defined to be the final halting state (e.g. *accept* or *reject*), or the state of the tape up to the first ␣ after halting.
- Runs don't always halt!

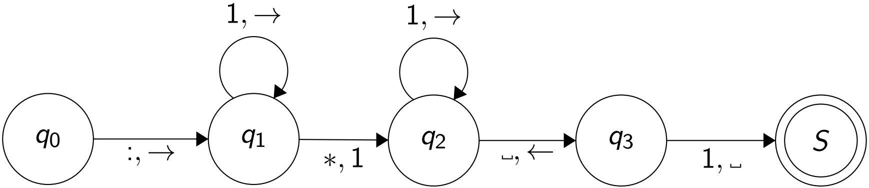
# Example - unary addition of 1

Let $\Sigma = \{1\}$. This machine adds one to a natural number represented in unary notation.



# Example - unary addition of two natural numbers

This machine uses the alphabet $\{1, *\}$ and correct input is of form $*, a*, *b, a*b$ followed by blanks ($a$ and $b$ are strings containing only 1). Here $a$ and $b$ represent the numbers to be added together in unary notation. Output is the unary number that is left on the tape when the machine accepts.



# Decision problems

- A decision problem is a yes or no question. E.g:
  - Given a graph $G$, is $G$ connected?
  - Does a given string of English characters contain the word *biscuits* as a substring?
  - Is a given propositional formula satisfiable?
  - Is a given natural number prime?
- A decision problem has a *general form*, and an *instance*.
- E.g. "Is a given natural number prime?" is a general form, and a specific number $n$ is an instance of this problem.
- Abstractly, a decision problem partitions its set of instances into two parts: one for 'yes' and the other for 'no'.
- Turing machines can try to solve decision problems, but first we must write them in a language they can understand.

# Working with strings

- A (finite) alphabet $\Sigma$ is a finite set of symbols.
- A string over $\Sigma$ is a sequence of characters, e.g. 01001, or the digits of $\pi$.
- $|s|$ denotes the length of $s$. E.g $|01001| = 5$, and the length of the string defined by the digits of $\pi$ is $\omega$ (countable infinity).
- The *empty string* is denoted $\epsilon$, and $|\epsilon| = 0$.
- We can *concatenate* finite strings. Given $a$ and $b$ we write $ab$.
- For all finite strings $a$ we have $\epsilon a = a$ and $a\epsilon = a$.
- $\Sigma^*$ is the set of all finite strings from $\Sigma$.

# Formal languages

- A *formal language* over $\Sigma$ is a subset of $\Sigma^*$.
- If $L$ is a formal language over $\Sigma$ then we can define the *characteristic function* of $L$ by $\chi_L : \Sigma^* \to \{0, 1\}$ and
$$\chi_L(s) = \begin{cases} 1 \text{ if } s \in L \\ 0 \text{ if } s \notin L \end{cases}$$

# Encoding schemes

- We want to take a decision problem and turn it into a language problem.
- I.e. given a decision problem $D$ we want to associate it with a formal language $L_D$ for some alphabet $\Sigma$.
- We have to choose $\Sigma$, and choose a system for writing the instances of $D$ as strings from $\Sigma^*$.
- I.e. each instance $x$ should correspond to some $s_x \in \Sigma^*$.
- This system should be sensible. I.e:
  - Different instances should be written as different strings.
  - We should be able to tell if a string from $\Sigma^*$ corresponds to an instance of $D$ or not.
  - We should be able to work out the string from the instance, and also the instance from the string.

# Encoding schemes - formal version

Definition 1 (encoding scheme)

If $\Sigma$ is a finite alphabet, $D$ is a decision problem, and $I_D$ is the set of all instances of $D$, an encoding scheme for $D$ using $\Sigma$ is a function $\textbf{code} : I_D \to \Sigma^*$ such that:

1. if $x, y \in I_D$ and $x \neq y$ we must have $\textbf{code}(x) \neq \textbf{code}(y)$ (i.e. **code** is $1 - 1$),
2. it is possible to work out if a string in $\Sigma^*$ is $\textbf{code}(x)$ for some $x \in I_D$, and
3. the process for converting $x$ to $\textbf{code}(x)$ and $\textbf{code}(x)$ to $x$ is well defined.

# Decision problems and formal languages

- Any formal language $L$ defines a decision problem $D_L$.
  - "is this word from $\Sigma^*$ in $L$?".
- Using encoding we can turn decision problems into formal languages as follows.

### Definition 2 ($L_D$)

Given a decision problem $D$, a finite alphabet $\Sigma$, and an encoding for $D$ using $\Sigma$, the language defined by $D$ is the set $L_D = \{\mathbf{code}(x) : x \text{ is a yes instance of } D\}$.

# Decidability

### Definition 3 (Decidable)

An (encodable) decision problem $D$ is decidable if there is a Turing machine $T$ that accepts when its input is the code of a yes instance of $D$, and rejects when its input is the code of a no instance (or not the code of an instance at all). We say $T$ decides $D$.

- Is every decision problem that can be encoded using a finite alphabet decidable?

# Semidecidability

### Definition 4 (Semidecidable)

An (encodable) decision problem $D$ is semidecidable if there is a Turing machine $T$ that accepts when its input is the code of a yes instance of $D$, and does not halt (i.e. it runs forever) when its input is the code of a no instance (or not the code of an instance at all). We say $T$ semidecides $D$.

- Is every decision problem that can be encoded using a finite alphabet semidecidable?
- Why is every decidable problem also semidecidable?

# Decidability and formal languages

### Definition 5 (recursive)

A formal language $L$ over a finite alphabet $\Sigma$ is recursive if there is a Turing machine $T$ that takes input $x$ from $\Sigma^*$ and accepts when $x \in L$ and rejects when $x \notin L$.

### Definition 6 (recursively enumerable)

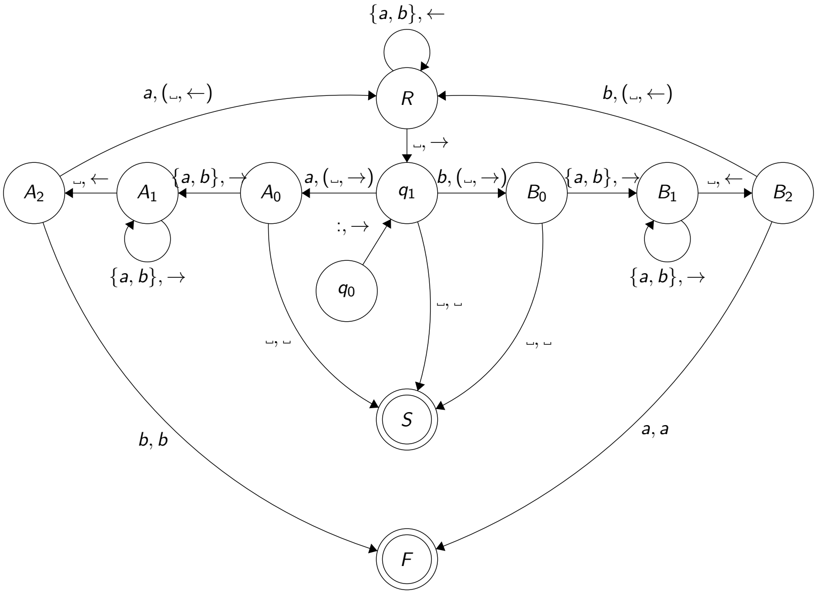
A formal language $L$ over a finite alphabet $\Sigma$ is recursively enumerable if there is a Turing machine $T$ that takes input $x$ from $\Sigma^*$ and accepts when $x \in L$ and runs forever when $x \notin L$. We often shorten *recursively enumerable* to *r.e.*).

# Some equivalences

By definition of $L_D$ and $D_L$ we have the following correspondences for encodable decision problems:

$$\begin{aligned}
D \text{ is decidable} &\iff L_D \text{ is recursive}\\
D_L \text{ is decidable} &\iff L \text{ is recursive}\\
D \text{ is semidecidable} &\iff L_D \text{ is recursively enumerable}\\
D_L \text{ is semidecidable} &\iff L \text{ is recursively enumerable}
\end{aligned}$$

# Example - palindromes



# Pushdown automata

- Turing machines are strictly more powerful than FSMs due to them effectively having infinite memory.
- There are things we can do with a TM that can not be done by an FSM.
- Is there a model of computation between FSMs and TMs?
- It turns out the answer is yes.
- A *pushdown automaton* is another abstract computation system based on FSMs.
- A pushdown automaton also has access to infinite memory in the form of a stack.
- It turns out that this makes them strictly more powerful than FSMs, but strictly less powerful than Turing machines.
- Pushdown automata with two stacks turn out to be equivalent to Turing machines.

# Computation and language

## Definition 7 (Recognize)

Let $M$ be an abstract machine capable of acting on input from $\Sigma^*$ for a finite alphabet $\Sigma$, and let $L \subseteq \Sigma^*$ be a language. We say $M$ recognizes $L$ if $M$ accepts on input $x$ if and only if $x \in L$.

- Every recursively enumerable language has a Turing machine that recognizes it (this is just the definition of r.e.).
- Moreover, every Turing machine $T$ defines a recursively enumerable language (just take the set of all $x$ such that $T(x)$ halts). So...

## Theorem 8

*The class of recursively enumerable languages is precisely the class of languages that can be recognized by a Turing machine.*

# The Chomsky heirarchy

| language | model of computation |
|---|---|
| recursively enumerable | Turing machines |
| context-sensitive | linear bounded automata |
| context-free | pushdown automata |
| regular | finite state machines |

- The Chomsky hierarchy of formal languages and their associated models of computation.
- Models of computation are arranged in order of power.
- On this course we are mainly interested in Turing machines.
- Note that it is possible to refine this hierarchy by adding extra classes, and this may result in something that is not linearly ordered.