Previously

How Mathematics (Proofs) Work

Today:

- "Facts" can have many **different "shapes"!**
- How do we **USE known facts?** (We know it's TRUE!)
- How can we **PROVE new facts?** (We don't know it's TRUE!)



More **Theorems**

More **Axioms**

More **Definitions**

Theorem

Theorem

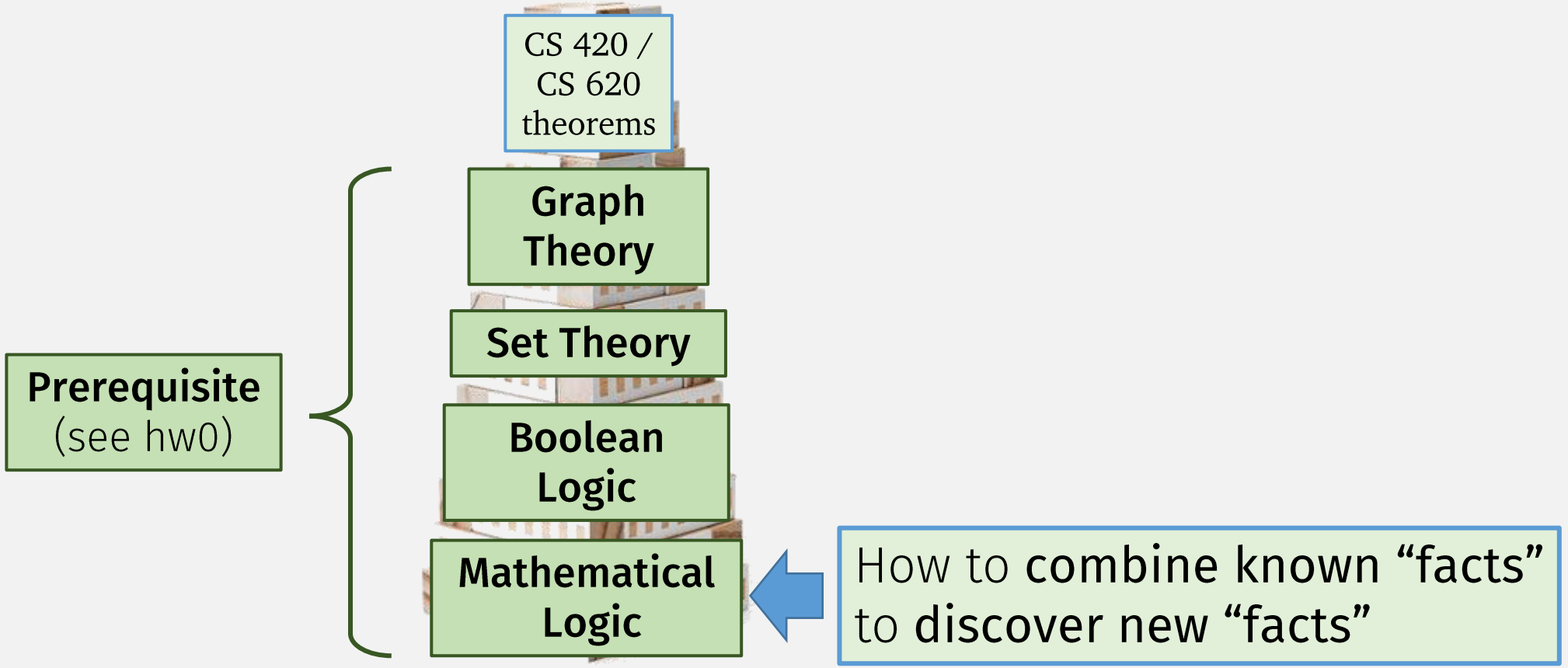
Axioms

Definitions

"facts"

Proofs = Figuring out how to (precisely) stack the pieces together

How This Course Works



Mathematical Logic Operators

- **Conjunction** $(\text{AND}, \wedge)$
- **Disjunction** $(\text{OR}, \vee)$
- **Negation** $(\text{NOT}, \neg, \sim)$
- **Implication** $(\text{IF-THEN}, \Rightarrow, \rightarrow)$

...

This semester:

Must understand difference between **Using** vs **Proving** a mathematical statement!

Mathematical Statements: AND

Using:

- If we know $A \wedge B$ is TRUE ...
  what do we know about A and B individually?
  - A is TRUE, and
  - B is TRUE

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False



We know

Mathematical Statements: AND

Using:

- If we know $A \wedge B$ is TRUE …
  what do we know about A and B individually?
  - A is TRUE, and
  - B is TRUE

Proving:

- To prove $A \wedge B$ is TRUE:
  - Prove $\boxed{A}$ is TRUE, and
  - Prove $\boxed{B}$ is TRUE

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False



We want

Mathematical Statements: IF-THEN

Using:

- If we know $P \to Q$ is TRUE ...
  what do we know about P and Q individually?
  - Either P is FALSE, or (promise unbroken)
  - If (we prove) P is TRUE, then Q is TRUE (promise kept)

(modus ponens)

Proving:

Examples?

An IF-THEN is a "promise"

p	q	$p \to q$	
True	True	True	← We know
True	False	False	☒
False	True	True	← or we know
False	False	True	← or we know

Using an **IF-THEN** statement: The "Modus Ponens" Inference Rule

Premises (if these statements are true)

- If P then Q
- P is **TRUE**

We know

Conclusion (then this is also true)

- Q must also be **TRUE**

p	q	$p \rightarrow q$
True	True	True
True	False	False
False	True	True
False	False	True

We know

Mathematical Logic Operators: IF-THEN

Using:

- If we know $P \rightarrow Q$ is TRUE,
  what do we know about P and Q individually?
  - Either P is FALSE, or
  - If we **prove** P is TRUE, then Q is TRUE (**modus ponens**)

Proving:

- To prove $P \rightarrow Q$ is TRUE:
  - Either **Prove** P is FALSE (usu. hard or impossible), or
  - **Assume** (not prove!) P is TRUE,
    then **prove** Q is TRUE

p	q	$p \rightarrow q$	
True	True	True	← Want
True	False	False	
False	True	True	← Want?
False	False	True	← Want?

Example: Proving an IF-THEN Statement

Prove the following:

Proving IF-THEN

Using IF-THEN

- IF: If $x \geq 4$, then $2^x \geq x^2$

Assume this (AND stmt) is true

Using AND

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False

AND: x is the sum of the squares of four positive integers

- THEN: $2^x \geq x^2$

Prove this is true

p	q	$p \rightarrow q$
True	True	True
True	False	False
False	True	True
False	False	True

Proving:

To prove $P \rightarrow Q$ is TRUE:

Either Prove P is FALSE (usu. hard or impossible), or

Assume (not prove!) P is TRUE, then prove Q is TRUE

Statements / Justifications Table

Example: Proving an IF-THEN Statement

Prove: IF If $x \geq 4$, then $2^x \geq x^2$ AND x is the sum of the squares of four positive integers
THEN $2^x \geq x^2$

Proof:

Statement	**Justification**
1. $x = a^2 + b^2 + c^2 + d^2$	1. Assumption (IF part of IF-THEN)
2. $a \geq 1;\ b \geq 1;\ c \geq 1;\ d \geq 1$	2. Assumption (IF part of IF-THEN)
5. If $x \geq 4$, then $2^x \geq x^2$	5. Assumption (IF part of IF-THEN)
6. $2^x \geq x^2$	

Statements / Justifications Table

Example: Proving an IF-THEN Statement

Prove: IF If $x \geq 4$, then $2^x \geq x^2$ AND x is the sum of the squares of four positive integers
THEN $2^x \geq x^2$

Proof:

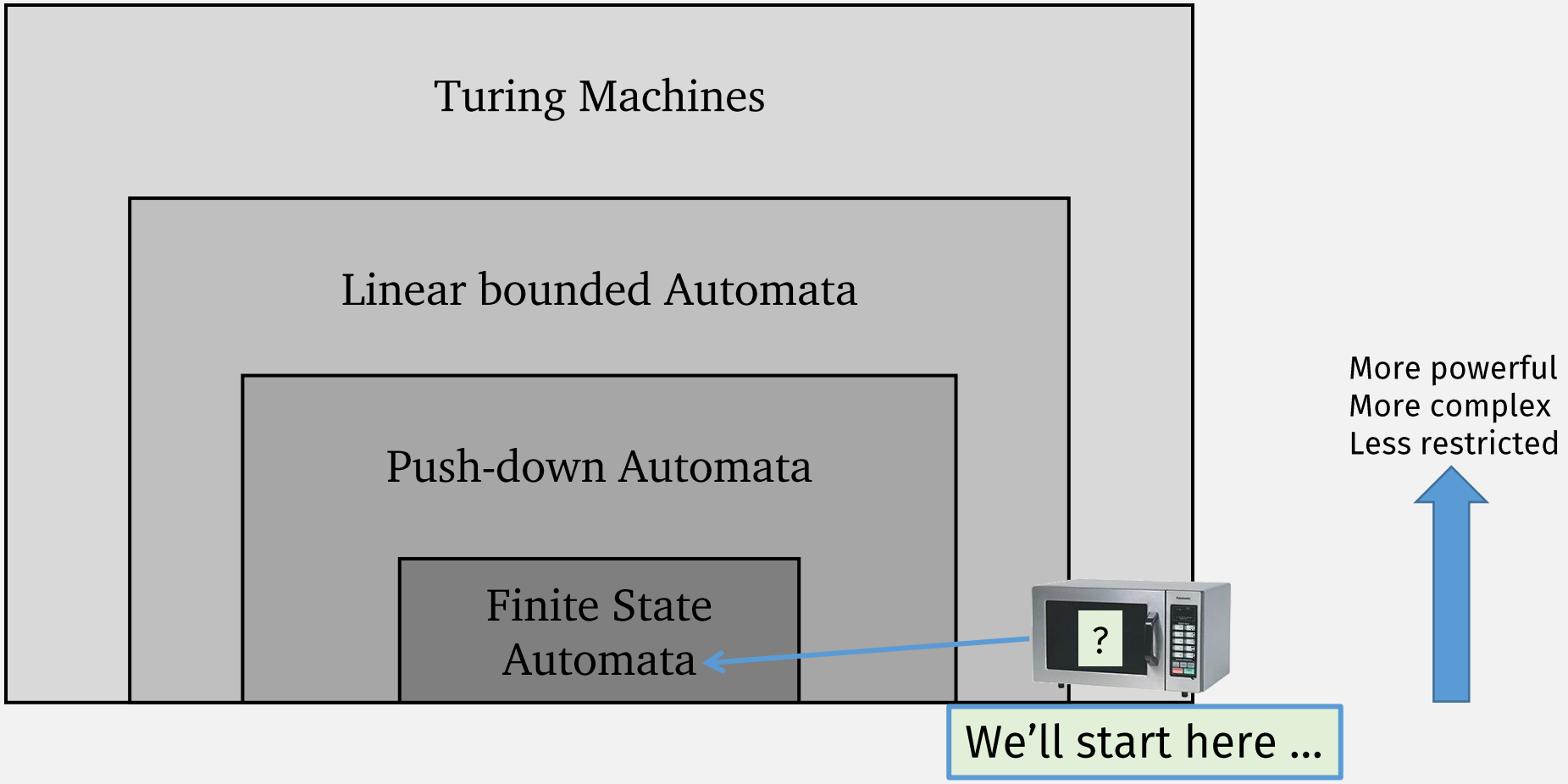
Statement	**Justification**
1. $x = a^2 + b^2 + c^2 + d^2$	1. Assumption (IF part of IF-THEN)
2. $a \geq 1; b \geq 1; c \geq 1; d \geq 1$	2. Assumption (IF part of IF-THEN)
3. $a^2 \geq 1; b^2 \geq 1; c^2 \geq 1; d^2 \geq 1$	3. By Stmt #2 & arithmetic laws
4. $x \geq 4$	4. Stmts #1, #3, and arithmetic
5. If $x \geq 4$, then $2^x \geq x^2$	5. Assumption (IF pa
6. $2^x \geq x^2$	6. Stmts #4 and #5

Modus Ponens
If we can prove these:
- If P then Q
- P

Then we've proved:
- Q

Last Time

Models of Computation Hierarchy



Last Time

A (Mathematical) Theory …

Mathematical theory

From Wikipedia, the free encyclopedia

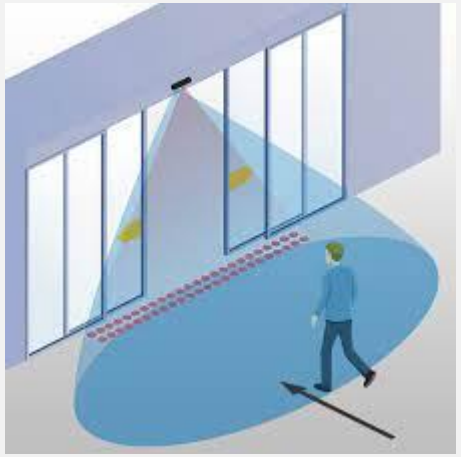
A **mathematical theory** is a mathematical model of a branch of mathematics that is based on a set of axioms. It can also simultaneously be a body of knowledge (e.g., based on known axioms and definitions), and so in this sense can refer to an area of mathematical research within the established framework.[1][2]

Explanatory depth is one of the most significant theoretical virtues in mathematics. For example, set theory has the ability to systematize and explain number theory and geometry/analysis. Despite the widely logical necessity (and self-evidence) of arithmetic truths such as 1<3, 2+2=4, 6-1=5, and so on, a theory that just postulates an infinite blizzard of such truths would be inadequate. Rather an adequate theory is one in which such truths are derived from explanatorily prior axioms, such as the Peano Axioms or set theoretic axioms, which lie at the foundation of ZFC axiomatic set theory.

The singular accomplishment of axiomatic set theory is its ability to give a foundation for the derivation of the entirety of classical mathematics from a handful of axioms. The reason set theory is so prized is because of its explanatory depth. So a mathematical theory which just postulates an infinity of arithmetic truths without explanatory depth would not be a serious competitor to Peano arithmetic or Zermelo-Fraenkel set theory.[3][4]

… must **explain** (**predict**) some real-world phenomena …

Finite Automata: “Simple” Computation / “Programs”



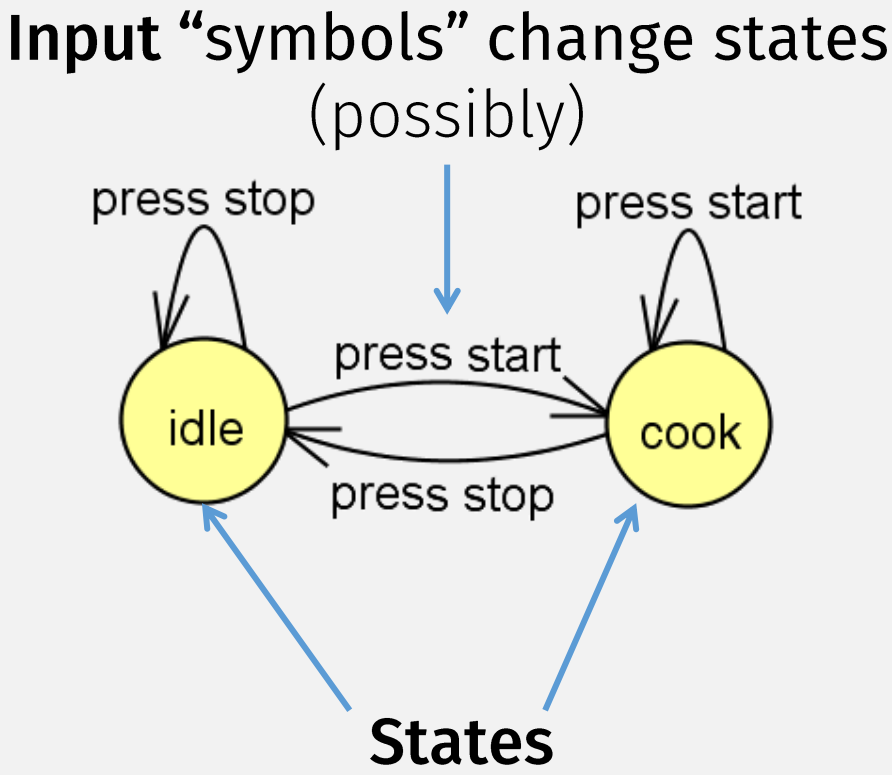




Finite Automata

- A **finite automata** or **finite state machine (FSM) …**
- … computes with a finite number of **states**

A Microwave Finite Automata



Finite Automata: Not Just for Appliances

Finite Automata:
a common
programming pattern



State pattern

From Wikipedia, the free encyclopedia

The **state pattern** is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. This pattern is close to the concept of finite-state machines. The state pattern can be interpreted as a strategy pattern, which is able to switch a strategy through invocations of methods defined in the pattern's interface.

(More powerful?) **Computation**
"Simulating" other (weaker?) **Computation**
(a common theme this semester)



Video Games Love Finite Automata

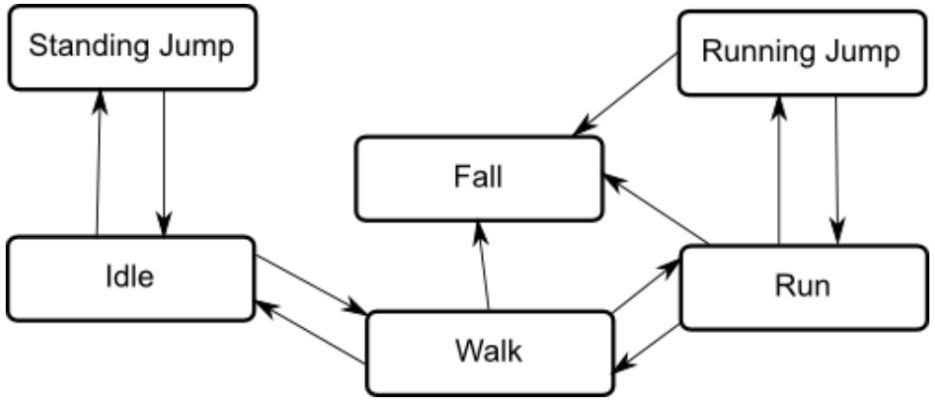
Unity Documentation

Manual

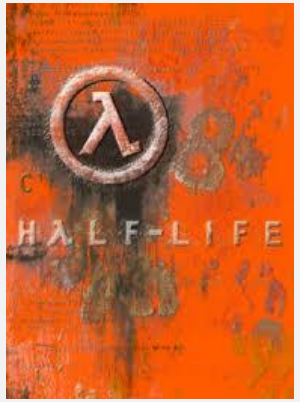
Unity User Manual 2020.3 (LTS) / Animation / Animator Controllers / Animation State Machines / State Machine Basics

The basic idea is that a character is engaged in some particular kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like idling, walking, running, jumping, etc. These actions are referred to as **states**, in the sense that the character is in a "state" where it is walking, idling or whatever. In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other. For example, a running jump can only be taken when the character is already running and not when it is at a standstill, so it should never switch straight from the idle state to the running jump state. The options for the next state that a character can enter from its current state are referred to as state transitions. Taken together, the set of states, the set of transitions and the variable to remember the current state form a **state machine**.

The states and transitions of a state machine can be represented using a graph diagram, where the nodes represent the states and the arcs (arrows between nodes) represent the transitions. You can think of the current state as being a marker or highlight that is placed on one of the nodes and can then only jump to another node along one of the arrows.



Finite Automata in Video Games



ValveSoftware / **halflife**

<> Code | Issues 1.6k | Pull requests 23 | Actions | Projects | Wiki

5d761709a3 — halflife / game_shared / bot / simple_state_machine.h

Alfred Reynolds initial seed of Half-Life 1 SDK

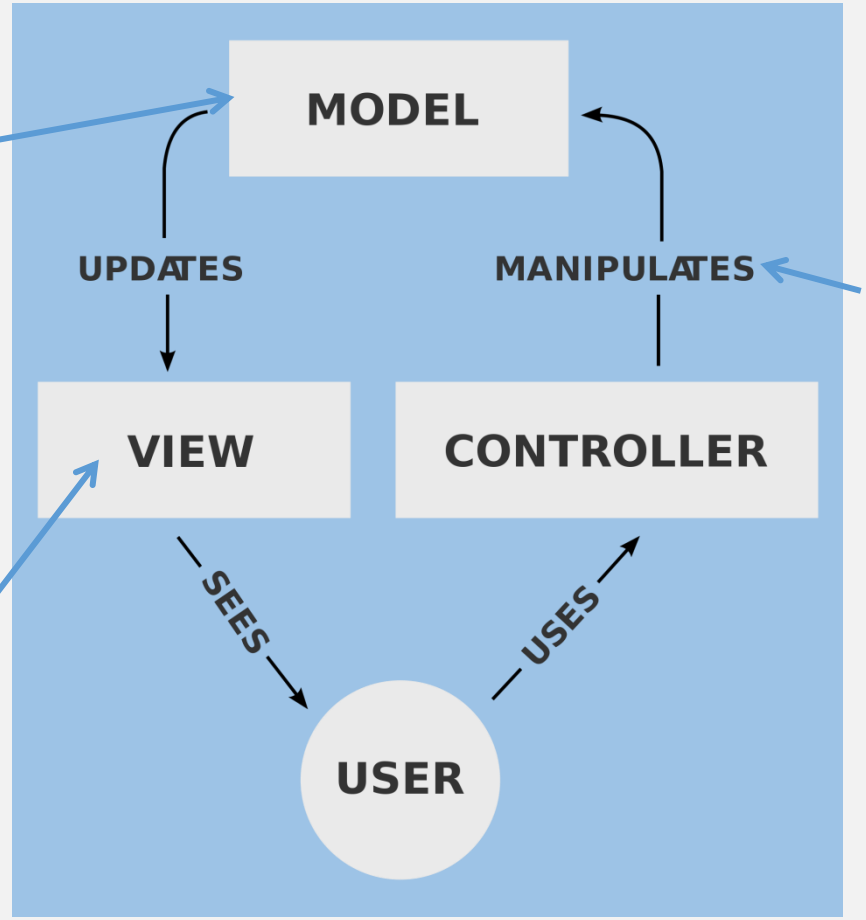
0 contributors

85 lines (67 sloc) | 2.15 KB

```
1  // simple_state_machine.h
2  // Simple finite state machine encapsulation
3  // Author: Michael S. Booth (mike@turtlerockstudios.com), November 2003
4
5  #ifndef _SIMPLE_STATE_MACHINE_H_
6  #define _SIMPLE_STATE_MACHINE_H_
7
8  //--------------------------------------------------------------------------------------------------------
9  /**
10  * Encapsulation of a finite-state-machine state
11  */
12 template < typename T >
13 class SimpleState
```

Model-view-controller (MVC) is an FSM

States



Input events change states

```
onclick
onload
onmouseover
onmousedown
...
```

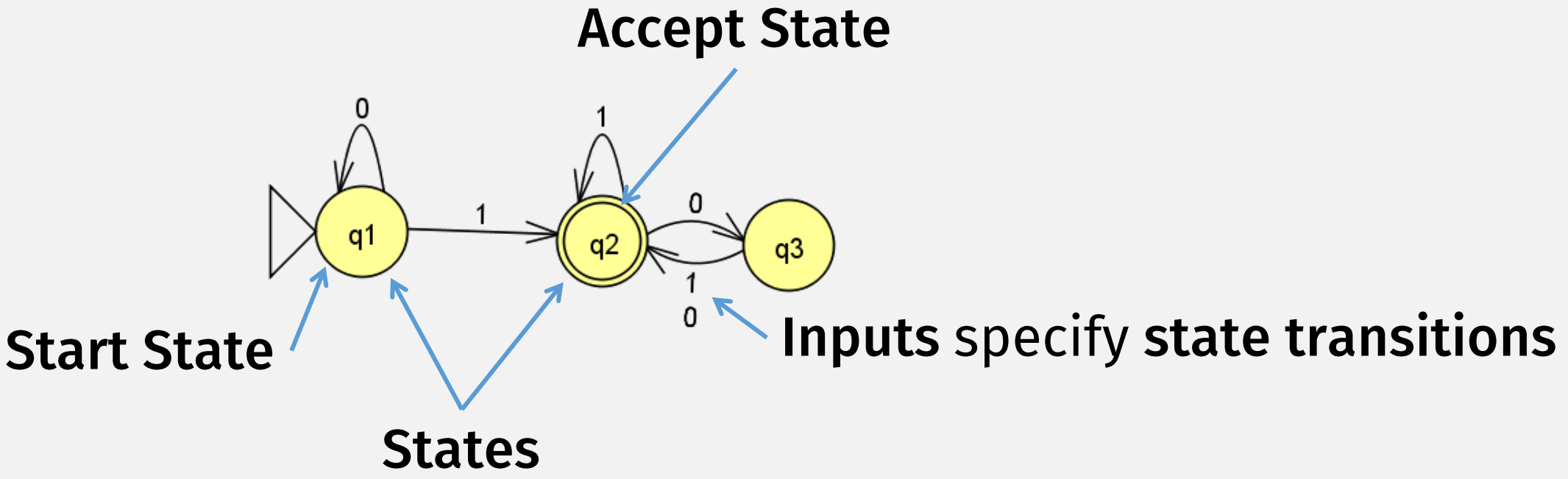
The **View** draws states

Analogy: Finite Automata is a "Program"

- A **restricted "program"** with access to **finite memory**
  - Actually, only **1 "cell"** of **memory!**
  - Possible **contents** of **memory = # of "states"**

- **Finite Automata has different representations:**
  - **Code** (won't use in this class)
  - ➢ **State diagrams**

Finite Automata state diagram



Analogy: Finite Automata is a "Program"

- A **restricted "program"** with access to **finite memory**
  - Only **1 "cell" of memory!**
  - Possible **contents** of **memory = # of states**

- **Finite Automata has different representations:**
  - **Code** (won't use in this class)
  - ➢ **State diagrams**

Analogy: Finite Automata is a "Program"

- A **restricted "program"** with access to **finite memory**
  - Only **1 "cell" of memory**!
  - Possible **contents** of **memory = # of states**

- **Finite Automata has different representations:**
  - **Code** (won't use in this class)
  - **State diagrams**
  - ➢ **Formal math description**
    (essentially same as **code** but **in a very different "programming language"**)

Finite Automata: The Formal Definition

DEFINITION

deterministic

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(*DFA*)

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

This semester

Things in **bold** have **precise formal definitions.**
(be sure to look up and review the definition whenever you are unsure)

Analogy

This is the **"programming language"** for **(deterministic) finite automata** "programs"

Finite Automata: The Formal Definition

Set or **sequence**?

5 components

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Interlude: Sets and Sequences

- Both are: **mathematical objects that group other objects**
- **Members** of the group are called **elements**
- Can be: **empty, finite,** or **infinite**
- Can contain: **other sets** or **sequences**

Sets	**Sequences**
• Unordered	• Ordered
• Duplicates not allowed	• Duplicates ok
• Notation: { }	• Notation: **varies:** ( ), **comma,** or **concat**
• **Empty set** written: ∅ or { }	• **Empty sequence:** ( )
• A **language** is a (**possibly infinite**) **set** of **strings**	• A **tuple** is a **finite sequence**
	• A **string** is a **finite sequence** of **characters**

A **set** used a lot in this course

sequences used a lot in this course

Set or Sequence ?

A **function** is …

… a **set** of **pairs**
(1st of each pair **from domain**, 2nd **from range**)

… has many representations:
a mapping, a table, …

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

sequence

set

set

Set of pairs (**domain**)

Set (**range**)

Don't know!
(states can be anything)

set

A **pair** is … a **sequence** of 2 elements

Finite Automata: The Formal Definition

5 components

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Analogy: Finite Automata is a "Program"

- A **restricted "program"** with access to **finite memory**
  - Only **1 "cell" of memory**!
  - Possible **contents** of **memory = # of states**

- Finite Automata has different equivalent representations:
  - **Code** (won't use in this class)
  - ➢State diagrams
  - ➢Formal math description
    (think of it as **code in a very different "programming language"**)

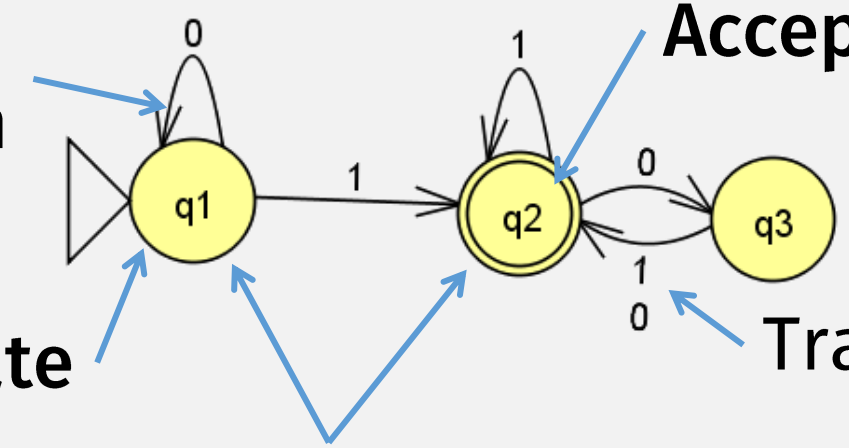
Finite Automata: State Diagram

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Arrows specify **transition function**



Accept State

Start State

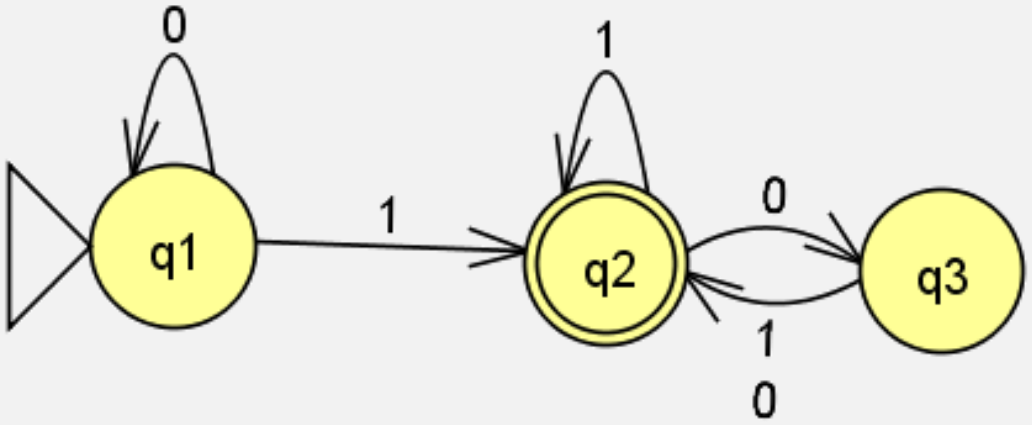
States

Transition labels from **alphabet**

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.



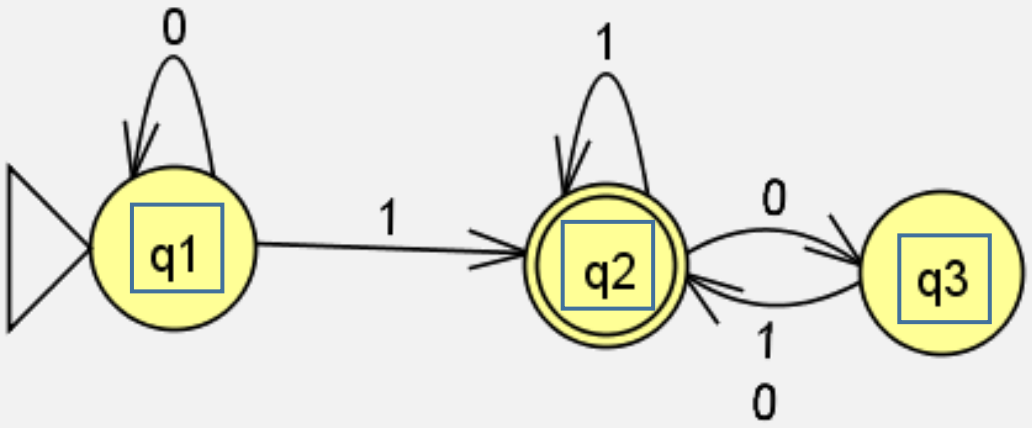
An Example (as **state diagram**)

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Note:
Not the same Q



An Example (as **state diagram**)

An Example (as **formal description**)

$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

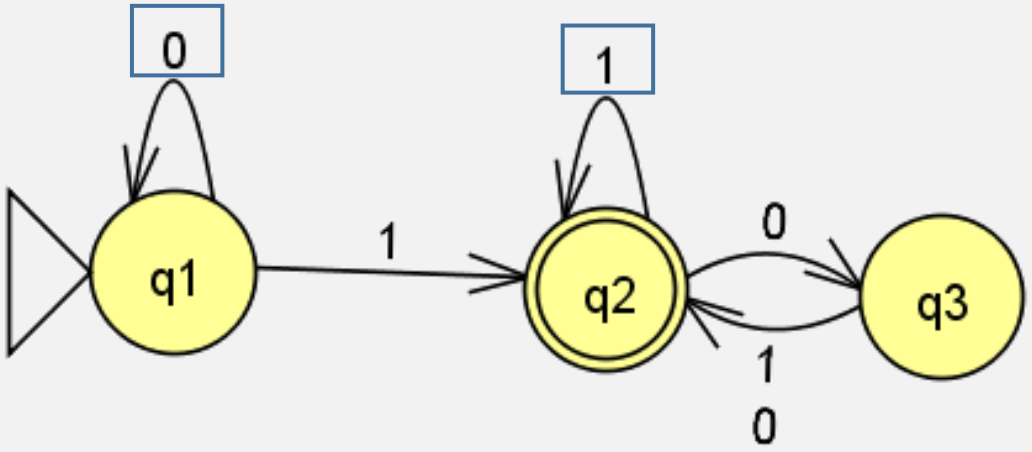
4. q_1 is the start state, and
5. $F = \{q_2\}$.

braces =
set notation
(no duplicates)

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.



$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,

Possible chars of input

3. δ is described

Alphabet defines all possible input strings for the machine

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

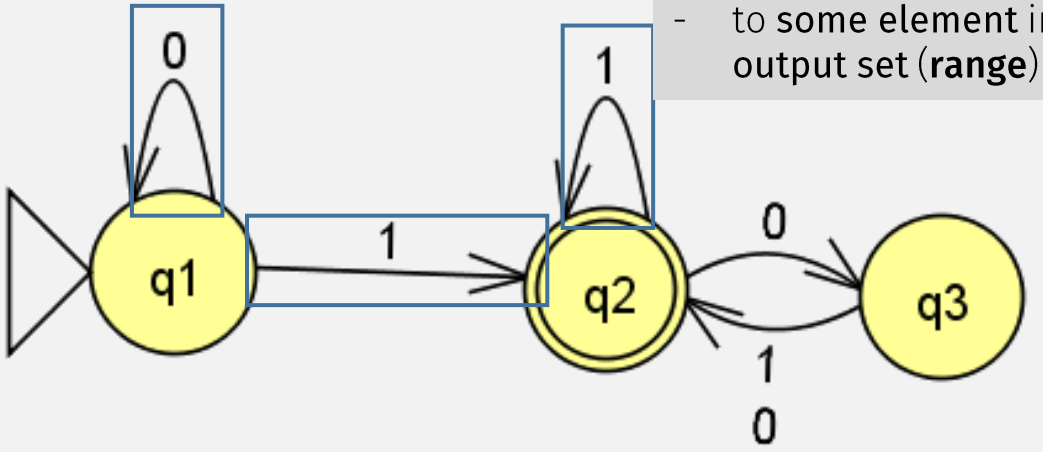
4. q_1 is the start state, and
5. $F = \{q_2\}$.

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept st***

There are many different ways to write a **function**, i.e., a **mapping** ...
- from every element in the input set(s) (**domain**)
- to some element in the output set (**range**)



$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

"And this is next input symbol"

"If in this state"

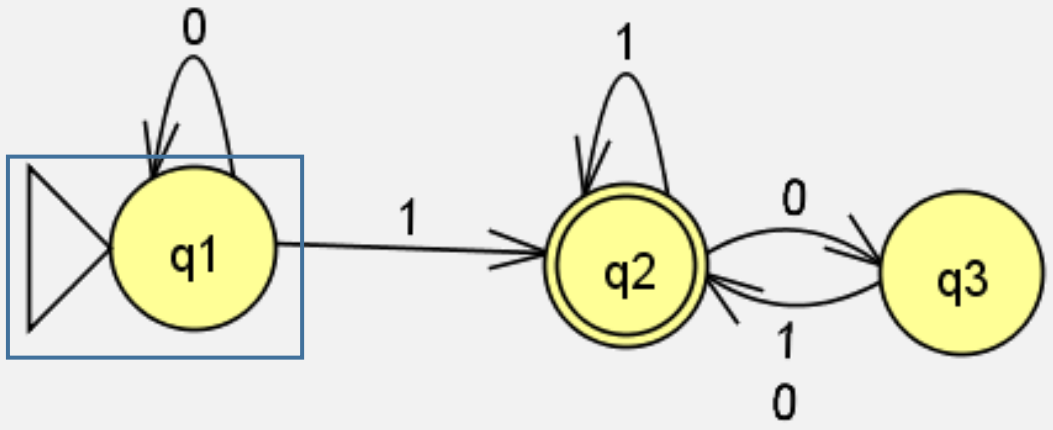
"Then go to this state"

4. q_1 is the start state, and
5. $F = \{q_2\}$.

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.



$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

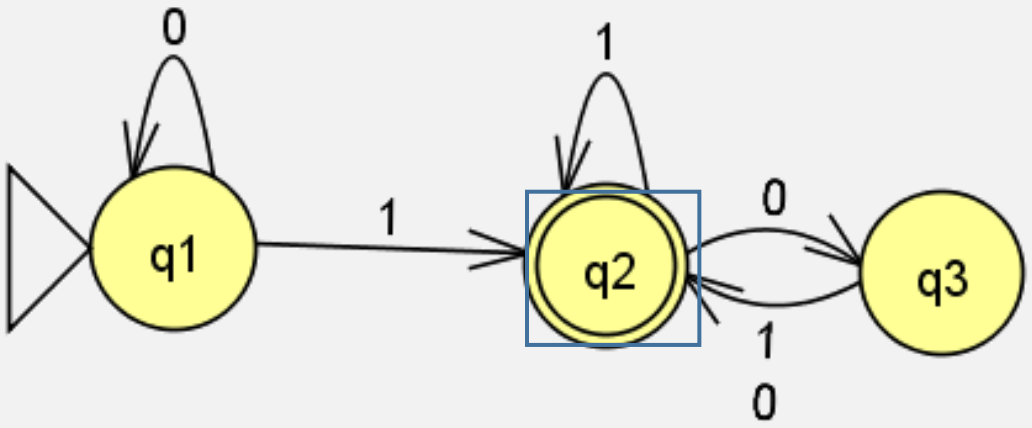
4. q_1 is the start state, and
5. $F = \{q_2\}$.

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

WARNING: This is a **set**!



$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

4. q_1 is the start state, and
5. $F = \{q_2\}$.

WARNING: This is a **set**!

Writing a non-set here makes this **not a DFA**

A "Programming Language"

DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

An Example (as formal description)

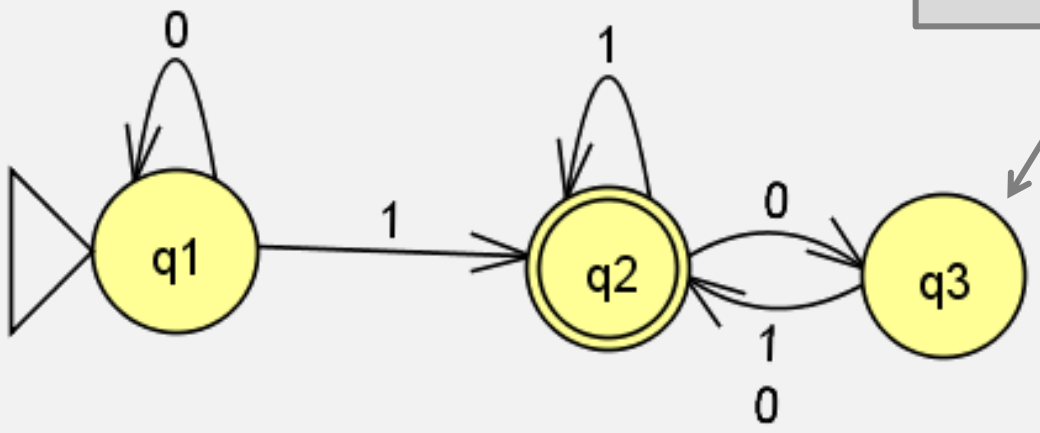
$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

4. q_1 is the start state, and
5. $F = \{q_2\}$.

A "Program"



"Programming" Analogy

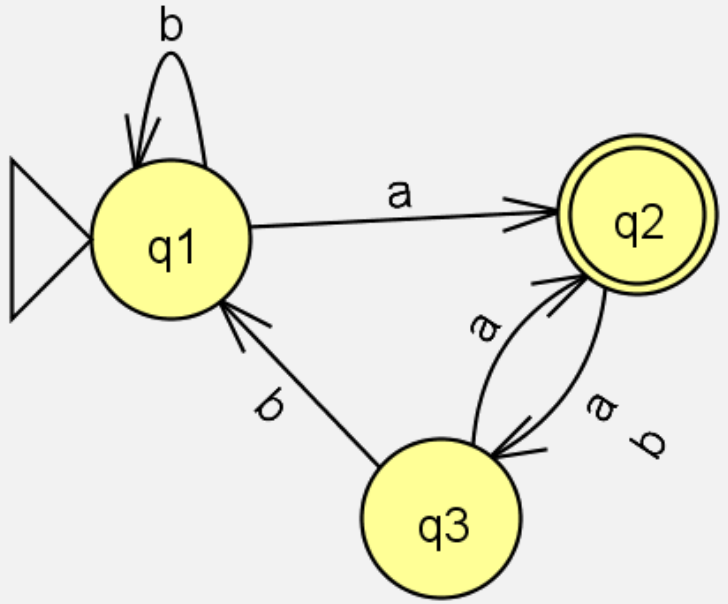
This "analogy" is meant to help your intuition

But it's <u>important</u> not to confuse with **formal definitions.**

Submit in-class work in gradescope!

In-class Exercise

Come up with a formal description of the following machine:



DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

In-class Exercise: solution

- Q = {q1, q2, q3}
- Σ= { **a**, **b** }
- δ
  - δ(q1, **a**) = q2
  - δ(q1, **b**) = q1
  - δ(q2, **a**) = q3
  - δ(q2, **b**) = q3
  - δ(q3, **a**) = q2
  - δ(q3, **b**) = q1
- q_0 = q1
- F = {q2}

(there are **many ways to define a function**, i.e., a **mapping from domain elements to range elements**)

$$M = (Q, \Sigma, \delta, q_0, F)$$

