CS 420

(Deterministic) Finite Automata

Tuesday, September 15, 2025

UMass Boston Computer Science

Announcements

HW 0

- ~~Due: 9/15 12:30pm EDT~~

HW 1

- Out: 9/15
- Due: 9/22 12:30pm EDT

Lecture videos

- Posted to Canvas
- Presented as-is, no guarantees
  - Not accepting questions
- For emergency / supplemental use only
- Attendance still taken in lectures

HW Hints and Reminders

- Problems must be: assigned to the correct pages
- Proofs must use constructors and syntax from this semester only!
  - See piazza

How to ask for HW help

(there's **no such thing** as a **stupid question**, but …)

… there **is such thing** as a **less useful question** (gets less useful answers)

- **"Is this correct?"**
- **"I don't get it"**
- **"Give me a hint?"**
- **"Do I need to do the thing DFA thing?"**

Useful question examples (gets useful answers)**:**

- "I don't understand this **notation** $A \otimes B \ggg C$ **…** and I couldn't find it in any lecture"
- "I couldn't find this **word's definition …**"

Most HW questions can be **answered** by **looking up the meaning of a word** or **notation** or **definition!**

Last Time

Mathematical Logic Operators

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

...

(No other symbols allowed! ... unless explicitly introduced)

This semester:

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

We will write proofs using a programming style:

Proving: equivalent to **constructing**, e.g., an **object**

Using: equivalent to **using** the **proof object**, e.g., **GETTING** a field, or **APPLYING** a method

Last Time

Mathematical Statements: AND

Using:

- If we know $A \wedge B$ is **TRUE** ... (i.e., we have a **proof** of it) 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

Programming-style:

if proof_{AB} is a **proof of** $A \wedge B$ ...

then to **get** a **proof of** A ... **FIRST** proof_{AB}

and to **get** a **proof of** B ... **SECOND** proof_{AB}

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

We want

Programming-style:

if we have $\mathrm{pr_A}$ and $\mathrm{pr_B}$ as **proofs of** A and B, respectively …

then to **construct** a **proof of** $A \wedge B$ … **`mk-AND-proof`** $\mathrm{pr_A}\ \mathrm{pr_B}$

Mathematical Statements: IF-THEN

Using:

- If we know $P \Rightarrow Q$ is **TRUE** ...
  what do we know about P and Q individually? **NOTHING!!** (an IF-THEN only says something about them together!)
  - If (we can prove) P is **False**, then Q ... can be TRUE or FALSE (not interesting!)
  - If (we can prove) P is **TRUE**, then (we have proven) Q is TRUE

(modus ponens)

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

Programming-style:

if we have $\mathrm{pr}_{P \Rightarrow Q}$ as **proof of** $P \Rightarrow Q$ ...

... and pr_P as **proof of** P ...

(it's like a function)

then to **get** a **proof of** Q ... **APPLY** $\mathrm{pr}_{P \Rightarrow Q}$ **TO** pr_P

Mathematical Logic Operators: IF-THEN

Using:

- If we know $P \Rightarrow Q$ is **TRUE** …
  what do we know about P and Q individually?
  - If (we can prove) P is **False**, then Q … can be **TRUE** or **FALSE**
  - If (we can prove) P is **TRUE**, then (we have proven) Q is **TRUE**

Proving:

- To prove $P \Rightarrow Q$ is **TRUE**:
  - Either **Prove** P always **FALSE** (usu. impossible), or
  - **Assume** (we have proof that) 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?

Mathematical Logic Operators: IF-THEN

Programming-style:

if we want to **construct** a **proof of** $P \Rightarrow Q$ …

(it's like defining a function!)

Function parameter

`mk-IFTHEN-proof` (with ASSUMPTION = **proof of** P):

… construct **proof of** Q …

… which can use the ASSUMPTION (parameter) **proof of** P
(which is only valid, i.e., "in-scope", in here) …

Body of the function

But how can we **"construct"** a **multi-line proof?**

Proving:

- To prove $P \Rightarrow Q$ is **TRUE**:
  - **Assume** (we have proof that) P is **TRUE**, then **prove** Q is **TRUE**

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 (we have proof that) this AND statement is True

Using AND

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

i.e., $x = a^2 + b^2 + c^2 + d^2$ AND $a \geq 1; b \geq 1; c \geq 1; d \geq 1$

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

Prove this is True, using Assumption

Proving:

To prove $P \Rightarrow Q$ is **True:**

Assume (we have proof that) P is **True**,
then **prove** Q is **True**

Statements / Justifications Table of (sub) **Proofs**

Example: Proving an IF-THEN Statement

Prove: IF IF $x \geq 4$, THEN $2^x \geq x^2$ AND $x = a^2 + b^2 + c^2 + d^2$ AND $a \geq 1;\ b \geq 1;\ c \geq 1;\ d \geq 1$
THEN $2^x \geq x^2$

mk-IFTHEN-Proof (with ASSUMPTION = **proof of** <3clause AND in IF part of **IF-THEN above**>):

	Statement	**Justification**
Proof_1	1. $x = a^2 + b^2 + c^2 + d^2$	1. **SECOND** ASSUMPTION (AND "getter")
Proof_2	2. $a \geq 1;\ b \geq 1;\ c \geq 1;\ d \geq 1$	2. **THIRD** ASSUMPTION
⋮	3. $a^2 \geq 1;\ b^2 \geq 1;\ c^2 \geq 1;\ d^2 \geq 1$	3. **APPLY** ARITH **TO** Proof_2
	6. $2^x \geq x^2$	

Use of "fn parameter"

Some facts from previous courses will available as a "library"

(end with the Statement we want to prove)

But how can we **"construct"** a **multi-line proof?**

Example: Proving an IF-THEN Statement

Prove: IF IF $x \geq 4$, THEN $2^x \geq x^2$ AND $x = a^2 + b^2 + c^2 + d^2$ AND $a \geq 1;\ b \geq 1;\ c \geq 1;\ d \geq 1$
THEN $2^x \geq x^2$

mk-IFTHEN-Proof (with ASSUMPTION = **proof of** <3clause AND in IF part of **IF-THEN above**>):

	Statement	**Justification**
$Proof_1$	1. $x = a^2 + b^2 + c^2 + d^2$	1. **SECOND** ASSUMPTION
	2. $a \geq 1;\ b \geq 1;\ c \geq 1;\ d \geq 1$	2. **THIRD** ASSUMPTION
$Proof_3$	3. $a^2 \geq 1;\ b^2 \geq 1;\ c^2 \geq 1;\ d^2 \geq 1$	3. **APPLY** ARITH **TO** $Proof_2$
$Proof_4$	4. $x \geq 4$	4. **SUBST** $Proof_3$ **INTO** $Proof_1$
$Proof_5$	5. IF $x \geq 4$ THEN $2^x \geq x^2$	5. **FIRST** ASSUMPTION
	6. $2^x \geq x^2$	6. **APPLY** $Proof_5$ **TO** $Proof_4$

A proof of an IF-THEN is like a function!

Use an IF-THEN "function" with **APPLY**

New rule: **SUBST** can replace "equals"

(and, **APPLY** ARITH ...)

Substitution Rule – Replace Equivalents

Programming-style:

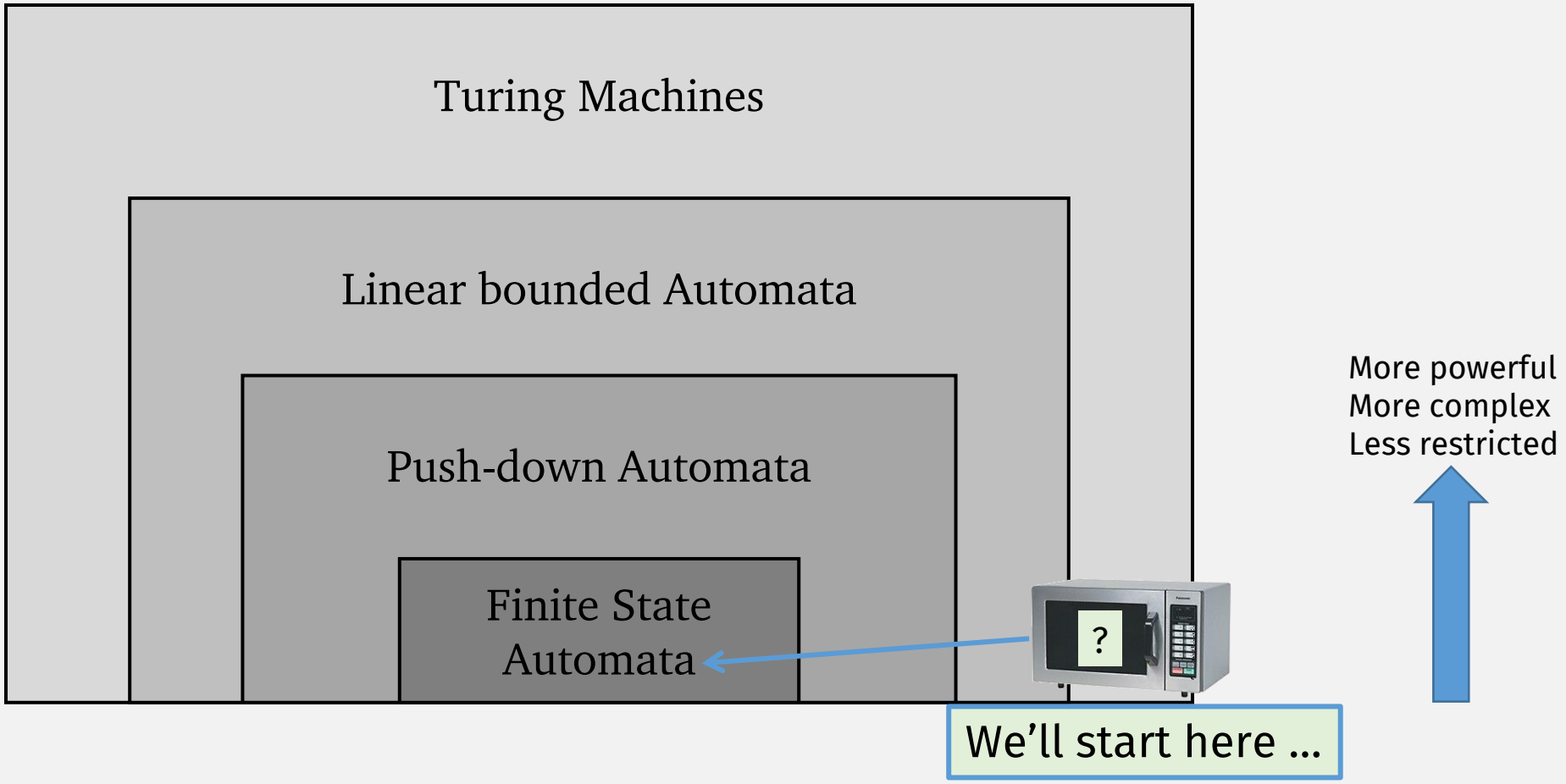
if we have $\mathrm{pr}_{x=y}$ as **proof of** $x=y$ ...

... and pr_{Px} is a **proof of statement** P **that contains** x ...

then **SUBST** $\mathrm{pr}_{x=y}$ **INTO** pr_{Px} ... is a **proof of statement** P **where** y **replaces** x ...

Last Time

Models of Computation Hierarchy



A (Mathematical) Theory …

Mathematical theory

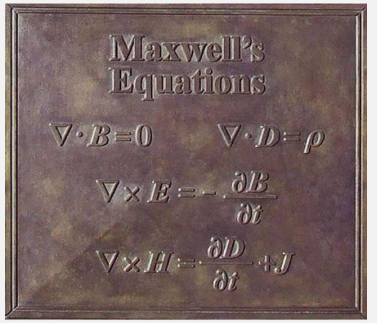
From Wikipedia, the free encyclopedia

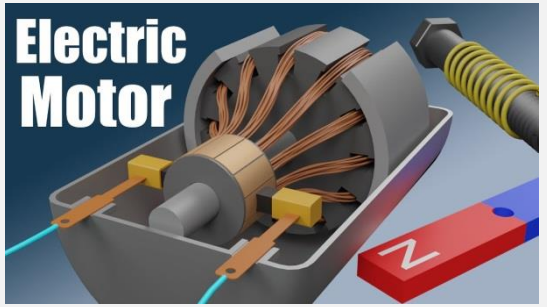
A **mathematical theory** is a mathematical model of a branch of mathematics that is based on a set of axioms. It can also simultaneously be a body of knowledge (e.g., based on known axioms and definitions), and so in this sense can refer to an area of mathematical research within the established framework.[1][2]

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.

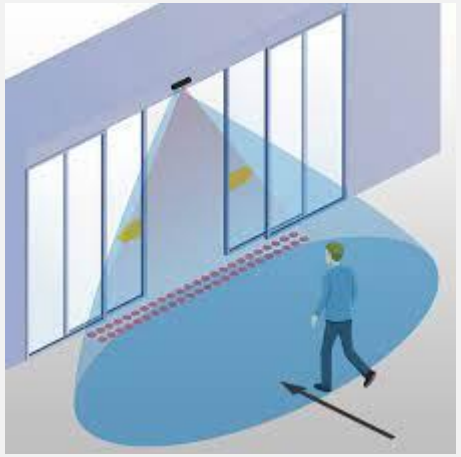
The singular accomplishment of axiomatic set theory is its ability to give a foundation for the derivation of the entirety of classical mathematics from a handful of axioms. The reason set theory is so prized is because of its explanatory depth. So a mathematical theory which just postulates an infinity 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 …

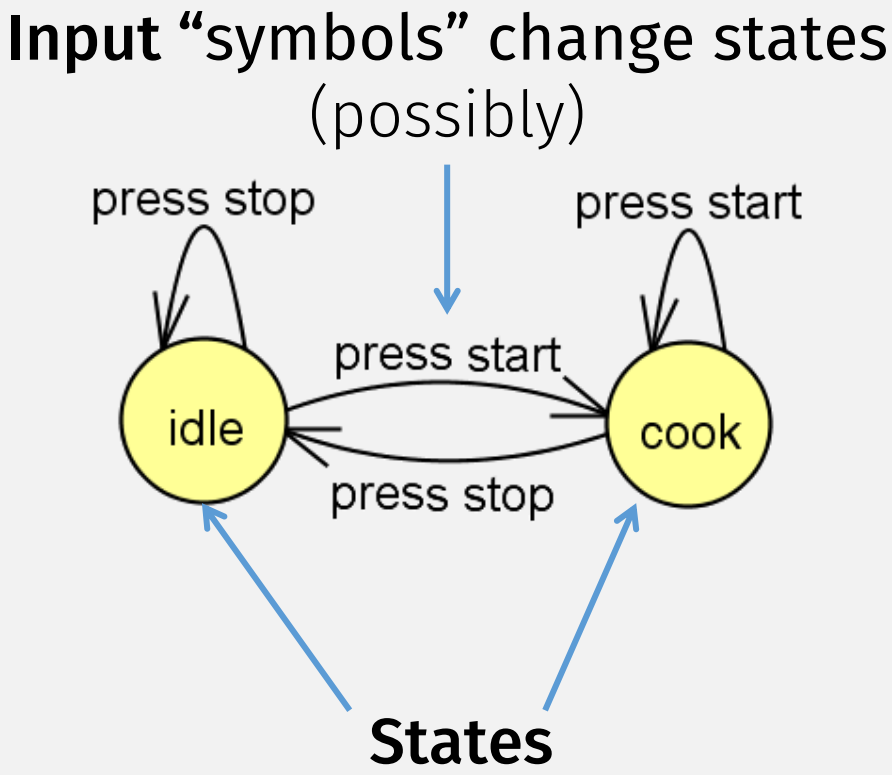




Finite Automata: “Simple” Computation / “Programs”



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)

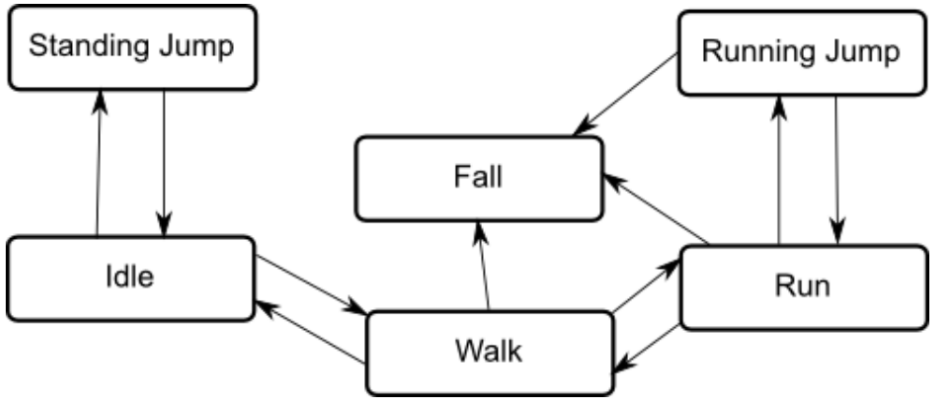
Video Games Love Finite Automata

Unity Documentation Manual

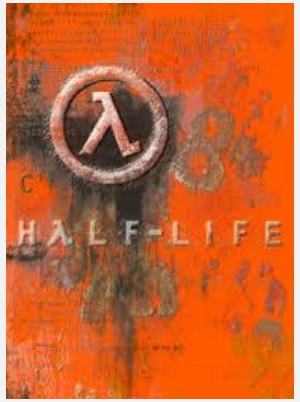
Unity User Manual 2020.3 (LTS) / Animation / Animator Controllers / Animation State Machines / State Machine Basics

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

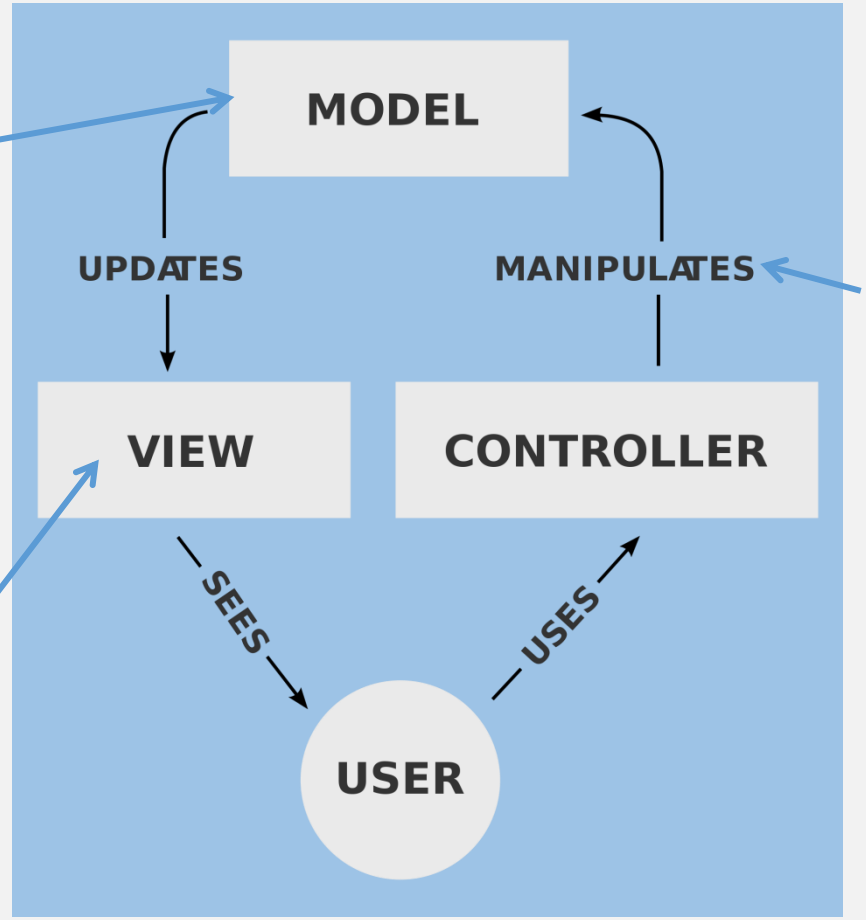
0 contributors

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
...
```

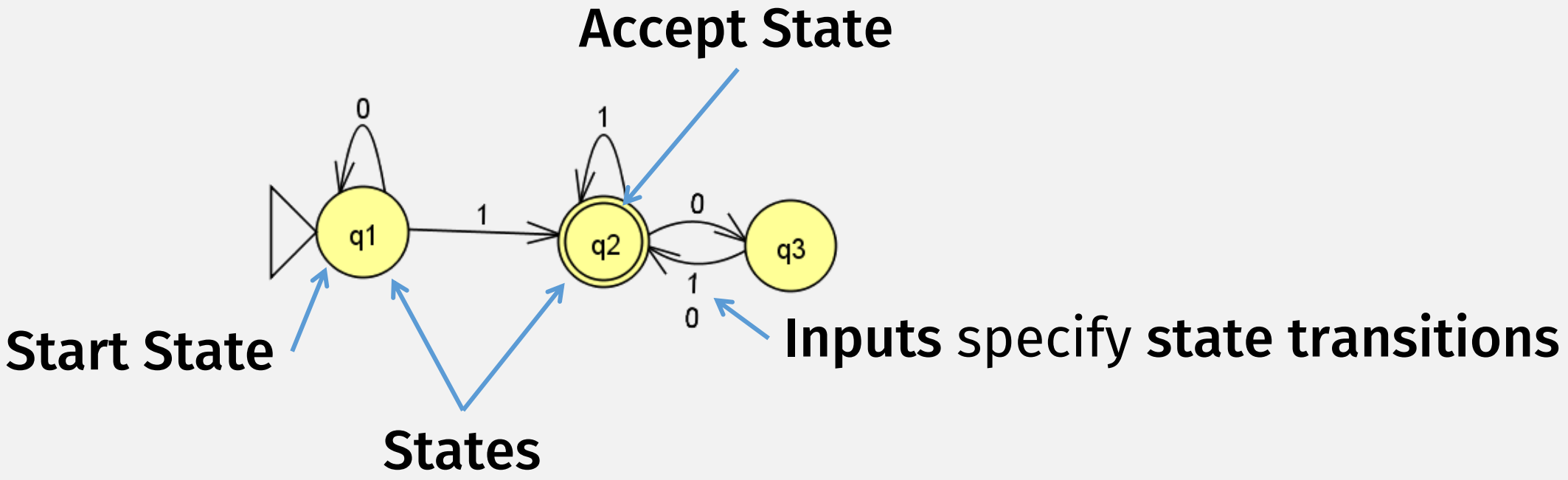
The **View** draws states

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 `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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

A (mathematical) **function** is ...

... a **mapping of elements** in a **domain** set **to elements** in a **range** set

DEFINITION

Constructor function

5 parameters??
(functions can only have one domain set)

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, where

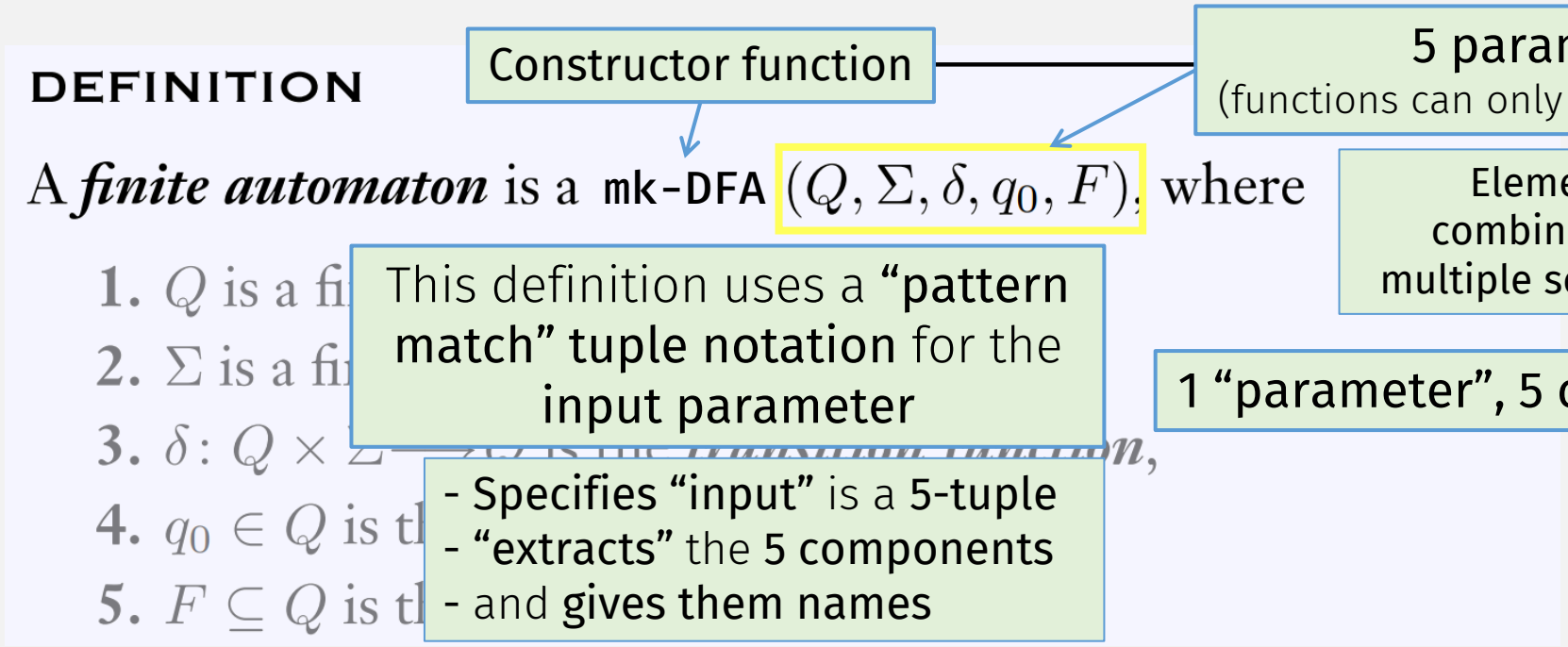
1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

In this course ...

... **function definitions** must have a **signature** (with notation →), which specifies the **domain** set and **range** set

`mk-DFA` : ??? → **DFA**

Finite Automata: The Formal Definition



mk-DFA : ? × ? × ? × ? × ? → **DFA**

mk-DFA : ??? → **DFA**

Finite Automata: The Formal Definition

DEFINITION

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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***.

1 “parameter”, 5 components (tuple)

Let’s play a game:
Set or **sequence**?

`mk-DFA` : ? × ? × ? × ? × ? → `DFA`

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** !

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

DEFINITION

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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 `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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***.

`mk-DFA` : `Setof<A>` × `Setof<C>` × (A × C → A) × A × `Setof<A>` → **DFA**

Summary: this is the **only way to construct a DFA!**

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"**)

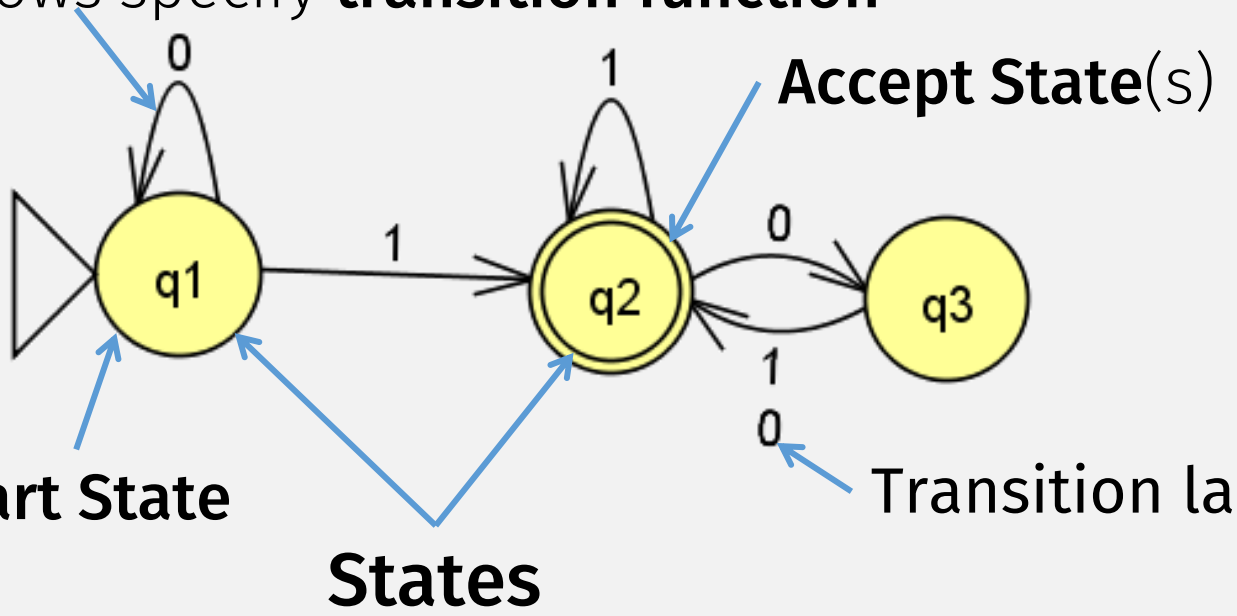
Finite Automata: State Diagram

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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(s)

Start State

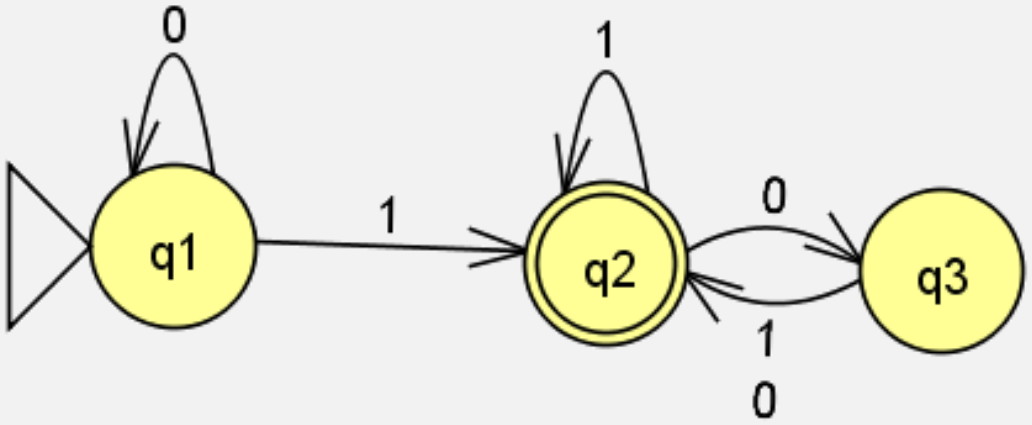
States

Transition labels must be from **alphabet**

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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***.



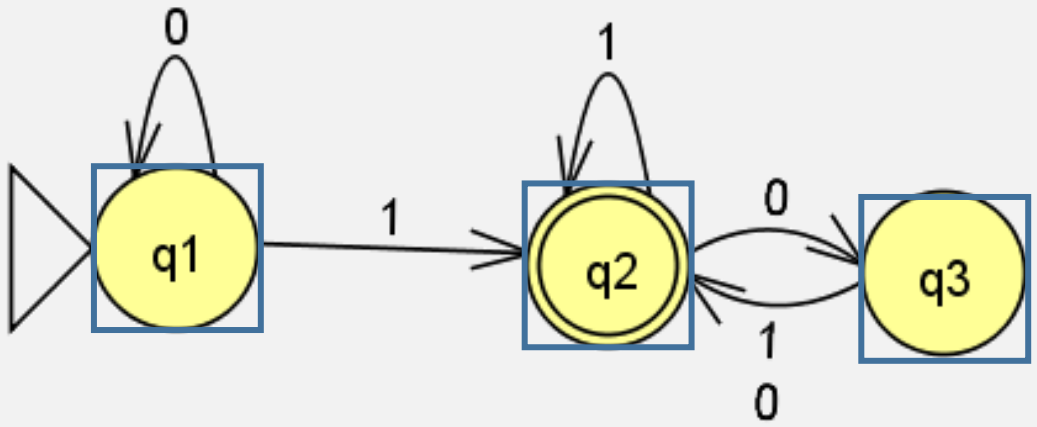
An Example (as **state diagram**)

DEFINITION (constructor function definition) – (function input parameter)

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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**) (constructor function use) (function input argument)

M_1 = `mk-DFA`$(Q, \Sigma, \delta,$ `q1`$, F)$ where

$Q = \{q_1, q_2, q_3\}$,

$\Sigma = \{0,1\}$,

δ is described as

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

braces = set notation (no duplicates)

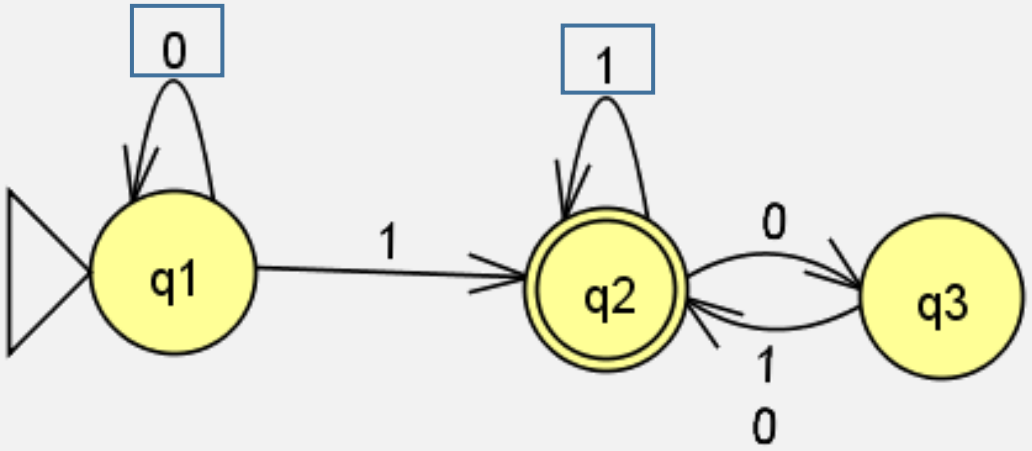
q_1 is the start state

$F = \{q_2\}$.

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ `q1`$,\ F\ )$, where

$Q = \{q_1, q_2, q_3\}$,

$\Sigma = \{0,1\}$; ← Possible chars of input

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,

q_1 is the start state

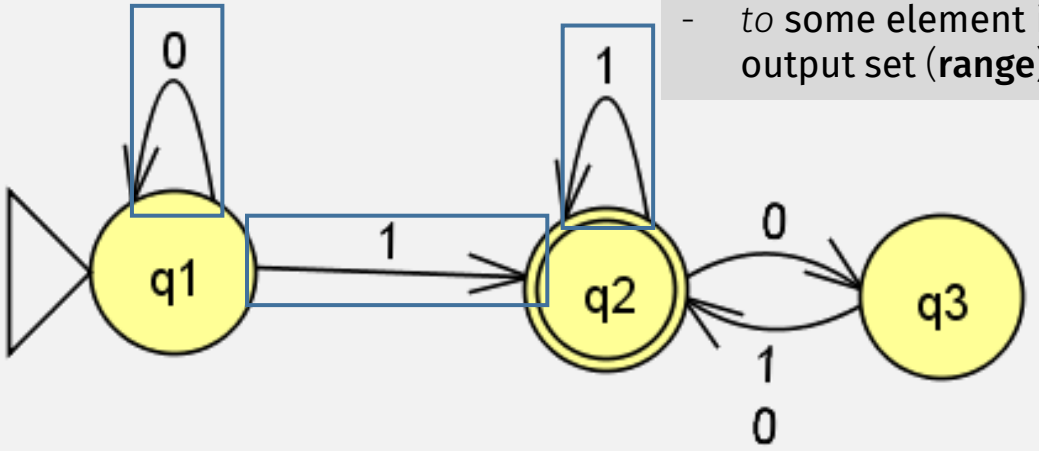
$F = \{q_2\}$.

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ `q1`$,\ F\ )$, where

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

Functions must be defined with signatures specifying domain and range sets!

$$\delta : Q \times \Sigma \rightarrow Q$$

	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"

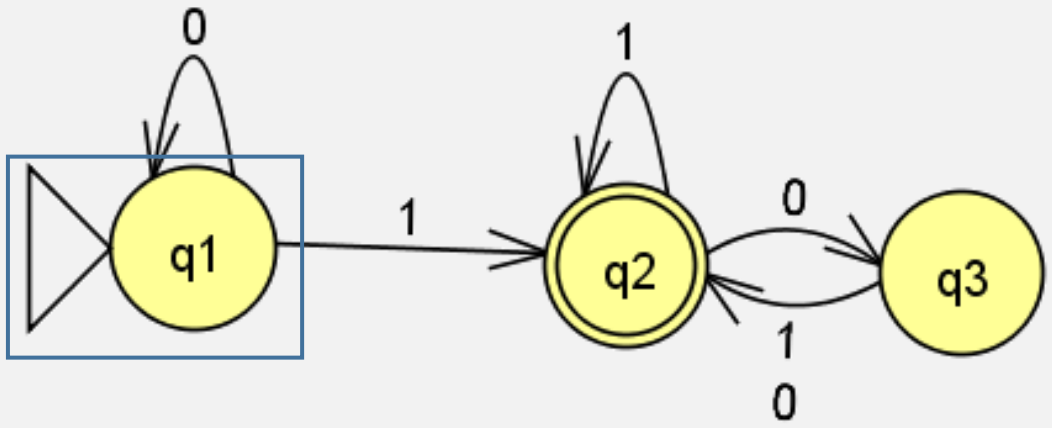
q_1 is the start state

$F = \{q_2\}.$

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ q1, $F\ )$, where

We already wrote it

$$Q = \{q_1, q_2, q_3\},$$
$$\Sigma = \{0,1\},$$
$$\delta : Q \times \Sigma \to Q$$

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

q1 (is the start state)

(do we need to write this?)

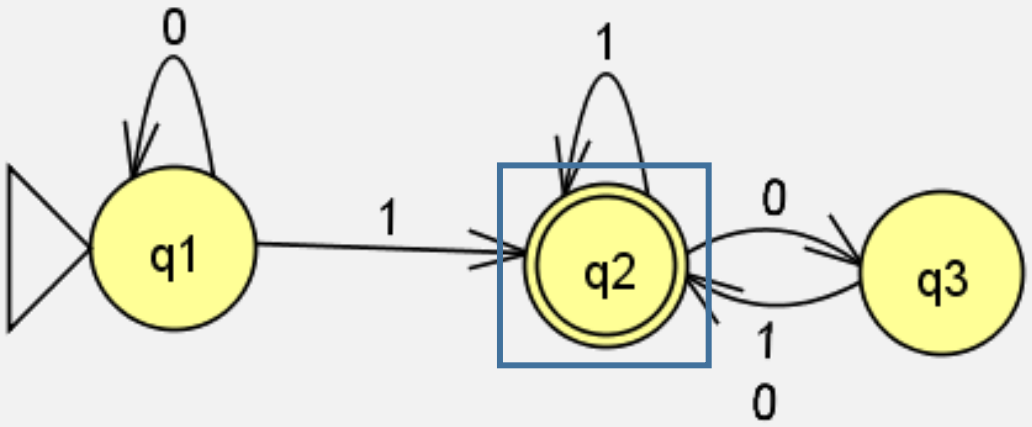
$$F = \{q_2\}.$$

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ `q1`$,\ F\ )$, where

$$Q = \{q_1, q_2, q_3\},$$
$$\Sigma = \{0,1\},$$
$$\delta : Q \times \Sigma \to Q$$

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

~~q1 (is the start state)~~

$$F = \{q_2\}.$$

WARNING: This is a **set**!

Writing a non-set here makes the whole thing **an invalid DFA**

DEFINITION

A “Programming Language”

A *finite automaton* is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, where

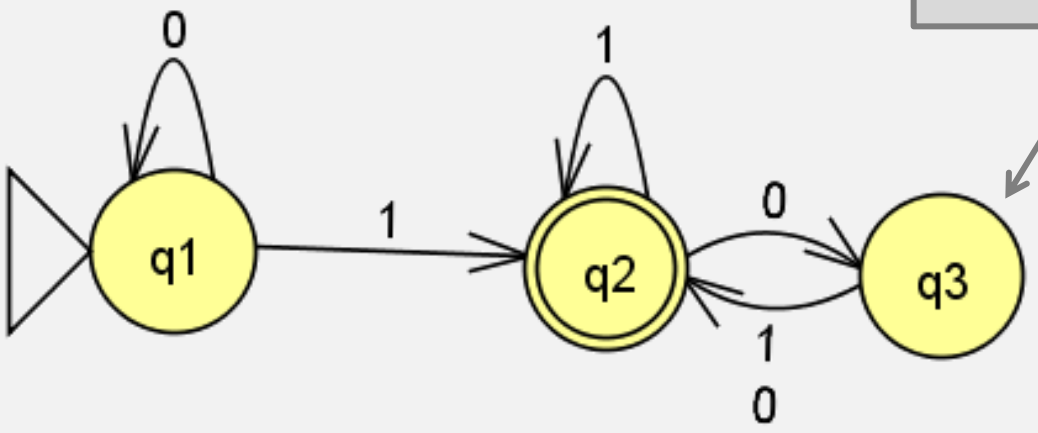
1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

An Example (as formal description)

M_1 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ q1, $F\ )$, where

$$Q = \{q_1, q_2, q_3\},$$
$$\Sigma = \{0,1\},$$
$$\delta : Q \times \Sigma \to Q$$

A “Program”



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

~~q1 (is the start state)~~

$$F = \{q_2\}.$$

“Programming” Analogy

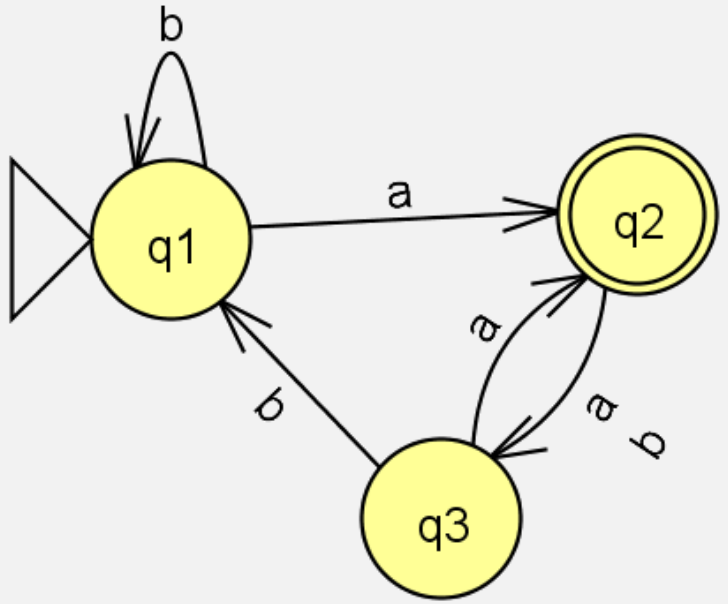
This “analogy” is meant to help your intuition

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

Submit in-class work in gradescope!

In-class Exercise (5min)

Come up with a formal description of the following machine:



DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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***.

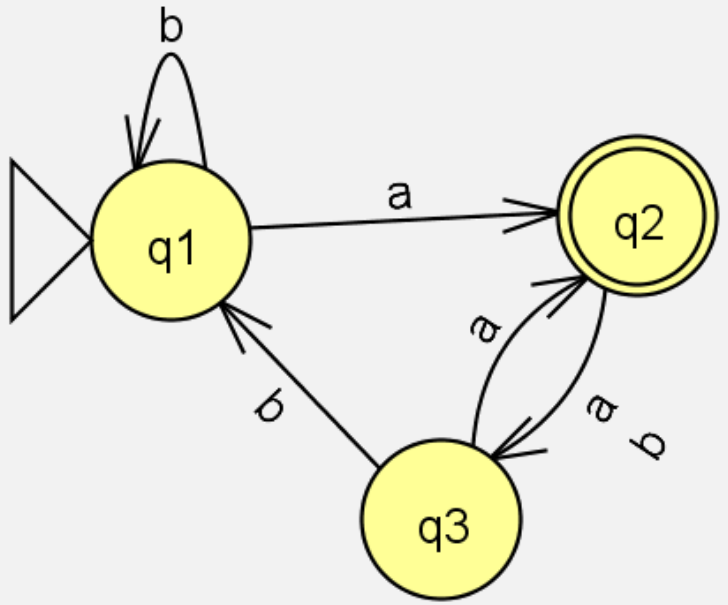
In-class Exercise: solution

- Q = {q1, q2, q3}
- Σ = { **a**, **b** }
- $\delta : Q \times \Sigma \rightarrow Q$
  - 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 (do we need to write this?)
- F = ~~q2~~ {q2}
  ???

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

Yes! Otherwise q_0 is an unbound variable!

M = **mk-DFA**(Q, Σ, δ, q_0, F), where



A Computation Model is … (from lecture 1)

- Some **definitions …**

e.g., A **Natural Number** is either
- Zero
- a Natural Number + 1

- And **rules** that describe how to **compute** with the **definitions …**

To **add** two **Natural Numbers**:
- Add the ones place of each num
- Carry anything over 10
- Repeat for each of remaining digits …

A Computation Model is … (from lecture 1)

- Some **definitions …**

docs.python.org/3/reference/grammar.html

10. Full Grammar specification

This is the full Python grammar, derived directly from the grammar used to generate the CPython pa
Grammar/python.gram). The version here omits details related to code generation and error recover

```
# ========================= START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
```

- And **rules** that describe how to **compute** with the **definitions …**

docs.python.org/3/reference/executionmodel.html

4. Execution model

4.1. Structure of a program

A Python program is constructed from code blocks. A *block* is a piece of Python program text that is executed
a unit. The following are blocks: a module, a function body, and a class definition. Each command typed inter
tively is a block. A script file (a file given as standard input to the interpreter or specified as a command line a
ment to the interpreter) is a code block. A script command (a command specified on the interpreter command
with the `-c` option) is a code block. A module run as a top level script (as module `__main__`) from the comma
line using a `-m` argument is also a code block. The string argument passed to the built-in functions `eval()` a
`exec()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for
bugging) and determines where and how execution continues after the code block's execution has complete

4.2. Naming and binding

A Computation Model is … (from lecture 1)

- Some **definitions …**

DEFINITION

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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***.

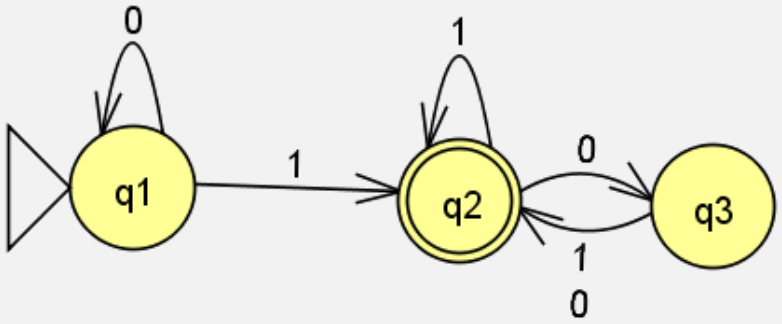
- And **rules** that describe how to **compute** with the **definitions …**

???

Sipser Fig 1.4

Computation with DFAs (JFLAP demo)

- DFA:



- Input: “1101”

HINT: always work out concrete examples to understand how a machine works