

# Using and Proving Logical Statements

CS 420 / CS 620

UMass Boston Computer Science

Instructors: Stephen Chang and Holly DeBlois

Monday, September 8, 2025

Sentence Type	Symbolic Logic
Simple	$p$
Negation	$\sim p$
Conjunction	$p \wedge q$
Disjunction	$p \vee q$
Conditional	$p \rightarrow q$
Biconditional	$p \leftrightarrow q$

# *Announcements*

- **HW**
  - Weekly; in/out Mon noon
    - HW 0 in, HW 1 out
  - ~3-4 questions, Paper-and-pencil proofs (no programming)
  - Discussing with classmates ok
  - Final answers written up and submitted individually
- **Lectures**
  - Slide sketches posted
    - Will not simulate / replace lecture!
  - Closely follow listed textbook chapters
- **Office Hours**
  - On web site
  - Let me know in advance if possible, but drop-ins also fine
  - TAs TBD

# In-class questions (in GradeScope) PREVIEW

- If we *know* statement  $A \wedge B$  is TRUE  
... what do we know about A and B?
- If we *want to prove*  $A \wedge B$  is TRUE  
... what do we need to prove about A and B?
- If we *know* statement  $A \rightarrow B$  is TRUE  
... what do we know about A and B?
- If we *want to prove* statement  $A \rightarrow B$  is TRUE  
... what are *valid* ways to do so?
- If we *want to prove* statement  $A \rightarrow B$  is TRUE  
... what is the *most common* way to do so?

Last Time

This semester, we will ...

Analogy:

**Computation Model**  
(system of definitions and rules)

↔

**Programming Language**

1. Define and study <sup>formal, precisely-defined</sup> **models of computation**

- models will be *as simple as possible* (to make them easier to study)

2. Compare and contrast models of computation

- which “programs” are *included / excluded* by a model
- *Equality* or *overlap* between models?

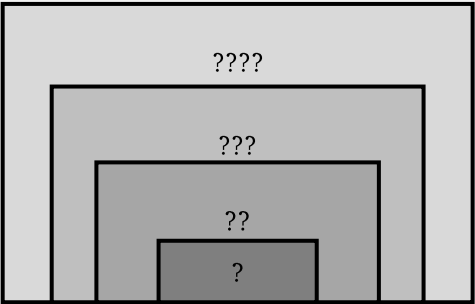
Analogy:

A **Computation** (in a model)

↔

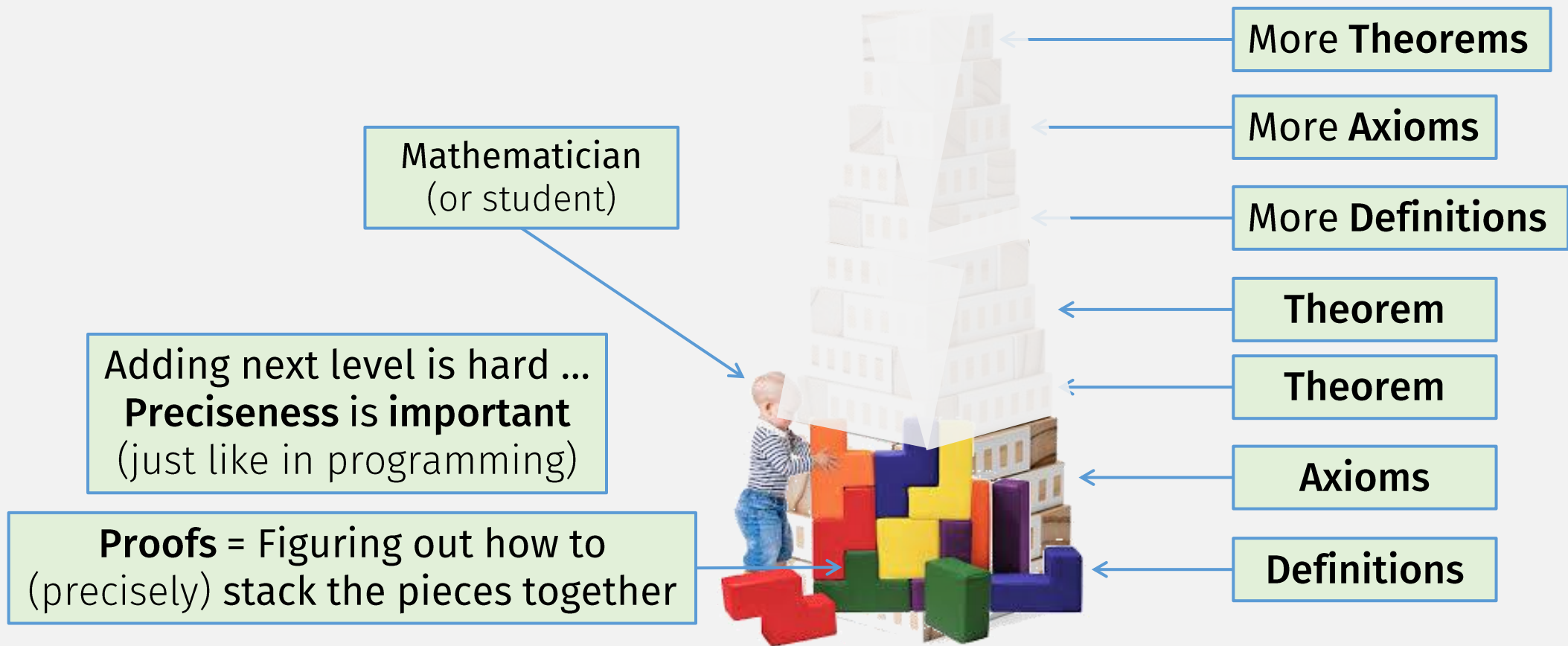
A Program

3. Prove things about the models



*Last Time*

# How Mathematics (Proofs) Work



# The “Modus Ponens” Inference Rule

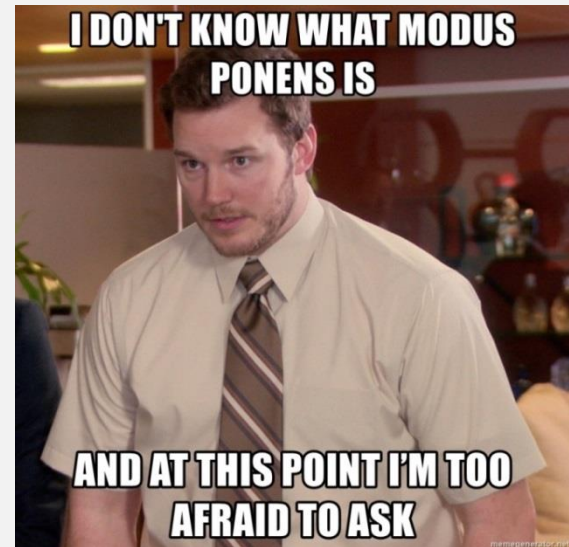
(Precisely Fitting Blocks Together)

**Premises** (if we can show these statements are true)

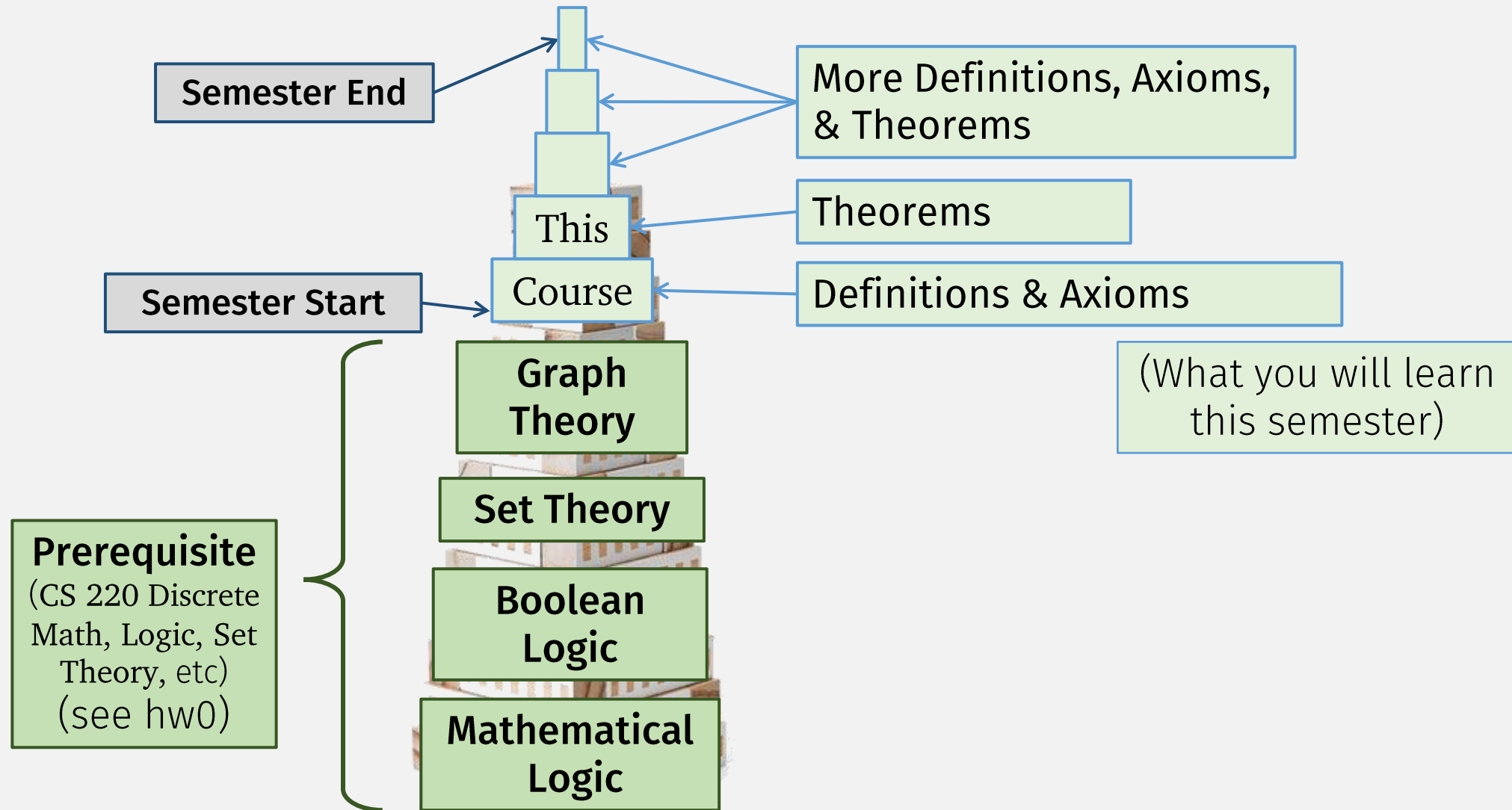
- If  $P$  then  $Q$
- $P$  is TRUE

**Conclusion** (then we can say that this is also true)

- $Q$  must also be TRUE



# How This Course Works



# A Word of Advice

Important:  
**Do not fall behind**  
in this course



To prove a (new) theorem ...


... need to know all axioms,  
definitions, and (previous)  
theorems below it



# Another Word of Advice

HW 1, Problem 1

Prove that  $ABC = XYZ$



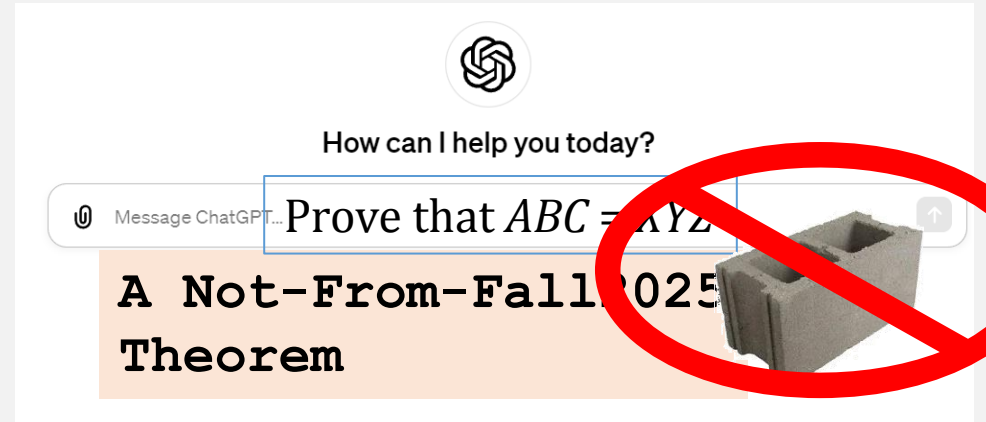
**“Blocks” from outside the course won’t work in the proof**

Remember:  
**Preciseness in proofs**  
(just like in programming) **is critical**  
(Proofs **must** connect  
facts from this course exactly)

HW problems are *graded* on precise steps  
in the **proof**, not on the final theorem itself!

... can be used to **prove** (new)  
**theorems** in this course

Only **axioms, definitions,** and  
**theorems** from this course...



# Grading

- **HW: 80%**
  - Weekly: In / Out Monday
  - Approx. 12 assignments
  - Lowest grade dropped
- **Participation: 20%**
  - Lecture participation, in-class work, office hours, piazza
- **No exams**
- **A range: 90-100**
- **B range: 80-90**
- **C range: 70-80**
- **D range: 60-70**
- **F: < 60**

All course info available on (joint) web sites:

- [cs.umb.edu/~stchang/cs620/f25](http://cs.umb.edu/~stchang/cs620/f25)
- [cs.umb.edu/~stchang/cs420/f25](http://cs.umb.edu/~stchang/cs420/f25)
- [cs.umb.edu/~hdeblois/cs420/f25/](http://cs.umb.edu/~hdeblois/cs420/f25/)

# Textbooks

- Sipser. *Intro to Theory of Computation*, 3<sup>rd</sup> ed.
- Hopcroft, Motwani, Ullman. *Intro to Automata Theory, Languages, and Computation*, 3<sup>rd</sup> ed.
- **Slides** (posted) and **lecture** will try to be **self-contained**,
- **BUT, students who read the book earn higher grades**

All course info available on (joint) web sites:

- [cs.umb.edu/~stchang/cs620/f25](http://cs.umb.edu/~stchang/cs620/f25)
- [cs.umb.edu/~stchang/cs420/f25](http://cs.umb.edu/~stchang/cs420/f25)
- [cs.umb.edu/~hdeblois/cs420/f25/](http://cs.umb.edu/~hdeblois/cs420/f25/)

# Late HW

- Is bad ... try not to do it please
  - Grades get delayed
  - Can't discuss solutions
  - You fall behind!
- Late Policy: **3 late days** to use during the semester

# HW Collaboration Policy

## Allowed

- Discussing HW with classmates (but must cite)
- Using other resources to learn, e.g., youtube, other textbooks, ...
- Writing up answers on your own, from scratch, in your own words

## Not Allowed

- Submitting someone else's answer
- Submitting someone else's answer with:
  - variables changed,
  - thesaurus words,
  - or sentences rearranged ...
- Using sites like Chegg, CourseHero, Bartleby, Study, ChatGPT, etc.
- Using theorems or definitions not from this course

# Honesty Policy

- 1<sup>st</sup> offense: zero on problem
- 2<sup>nd</sup> offense: zero on hw, reported to school
- 3<sup>rd</sup> offense+: F for course

## Regret policy

- If you self-report an honesty violation, you'll only receive a zero on the problem and we move on.

# All Up to Date Course Info

Survey, Schedule, Office Hours, HWs, ...

See course website(s):

- [cs.umb.edu/~stchang/cs620/f25](http://cs.umb.edu/~stchang/cs620/f25)
- [cs.umb.edu/~stchang/cs420/f25](http://cs.umb.edu/~stchang/cs420/f25)
- [cs.umb.edu/~hdeblois/cs420/f25/](http://cs.umb.edu/~hdeblois/cs420/f25/)

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

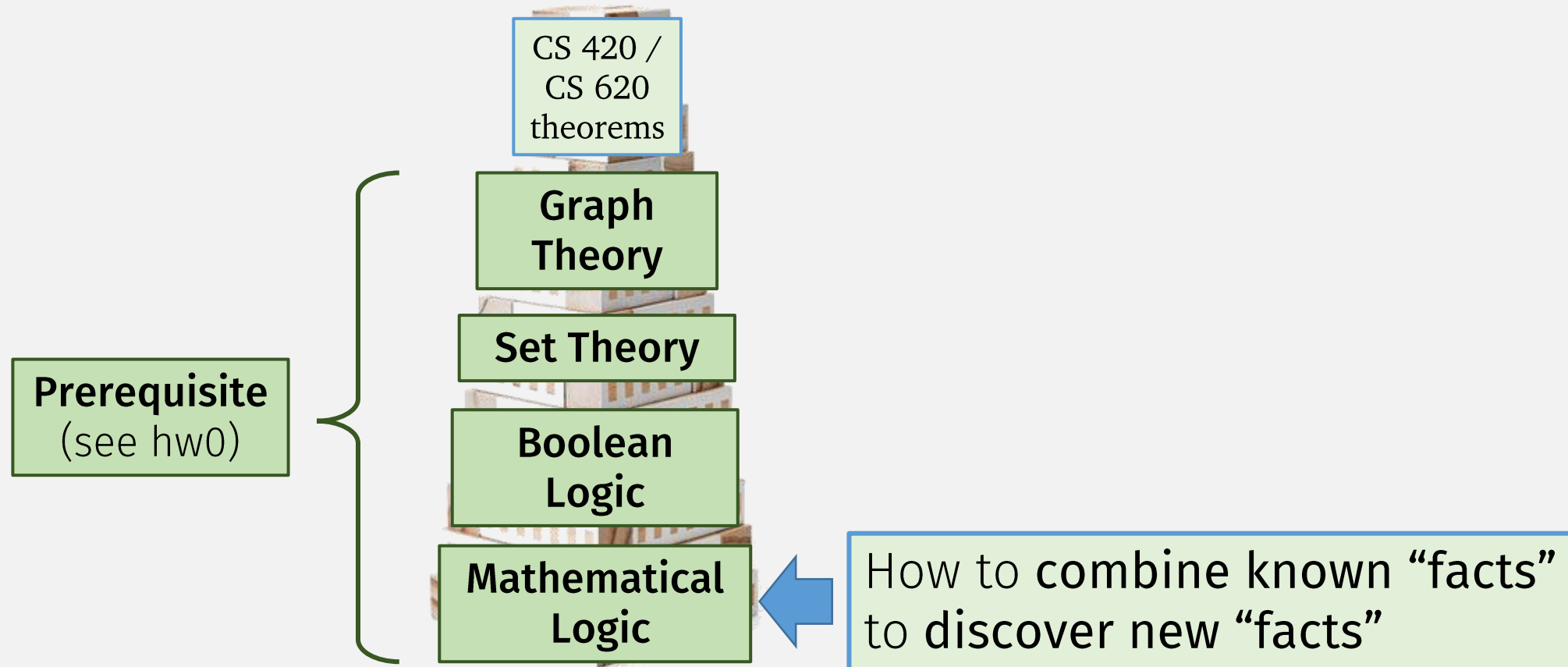
“facts”

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





# How This Course Works



# Mathematical Logic Operators

- **Conjunction** (AND,  $\wedge$ )
- **Disjunction** (OR,  $\vee$ )
- **Negation** (NOT,  $\neg$ ,  $\sim$ )
- **Implication** (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  $A$  is TRUE, and
  - Prove  $B$  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  $P \rightarrow Q$  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)

Examples?


An IF-THEN is a “promise”

## Proving:

$p$	$q$	$p \rightarrow q$	
True	True	True	← We <u>know</u>
True	False	False	⊗
False	True	True	← or we <u>know</u>
False	False	True	← or we <u>know</u>

# 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**)

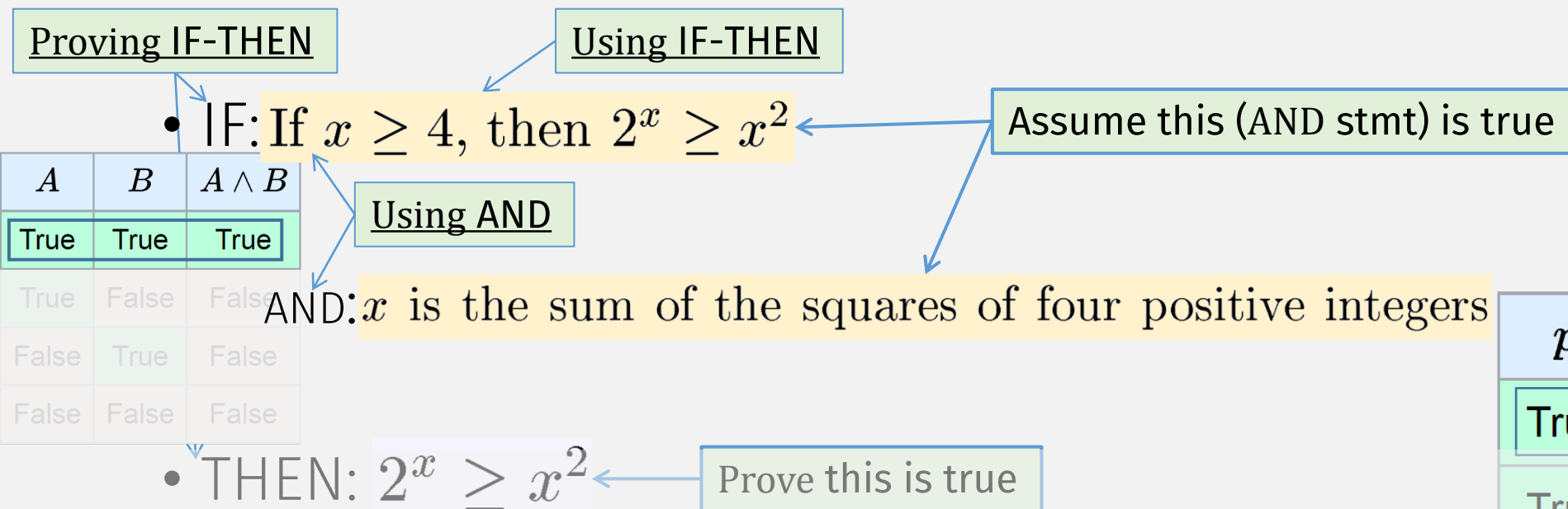
## Proving:

- 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	← <u>Want</u>
True	False	False	
False	True	True	← <u>Want?</u>
False	False	True	← <u>Want?</u>

# Example: Proving an IF-THEN Statement

Prove the following:



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
True	False	False
False	True	True
False	False	True





## 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

1.  $x = a^2 + b^2 + c^2 + d^2$
2.  $a \geq 1; b \geq 1; c \geq 1; d \geq 1$

5. If  $x \geq 4$ , then  $2^x \geq x^2$
6.  $2^x \geq x^2$

### Justification

1. Assumption (IF part of IF-THEN)
  2. Assumption (IF part of IF-THEN)
- 
5. Assumption (IF part of IF-THEN)

# 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

- 1.  $x = a^2 + b^2 + c^2 + d^2$
- 2.  $a \geq 1; b \geq 1; c \geq 1; d \geq 1$
- 3.  $a^2 \geq 1; b^2 \geq 1; c^2 \geq 1; d^2 \geq 1$

4.  $x \geq 4$

5. If  $x \geq 4$ , then  $2^x \geq x^2$

→ 6.  $2^x \geq x^2$

## Justification

- 1. Assumption (IF part of IF-THEN)
- 2. Assumption (IF part of IF-THEN)
- 3. By Stmt #2 & arithmetic laws
- 4. Stmts #1, #3, and arithmetic
- 5. Assumption (IF part of IF-THEN)
- 6. Stmts #4 and #5

### Modus Ponens

If we can prove these:

- If  $P$  then  $Q$

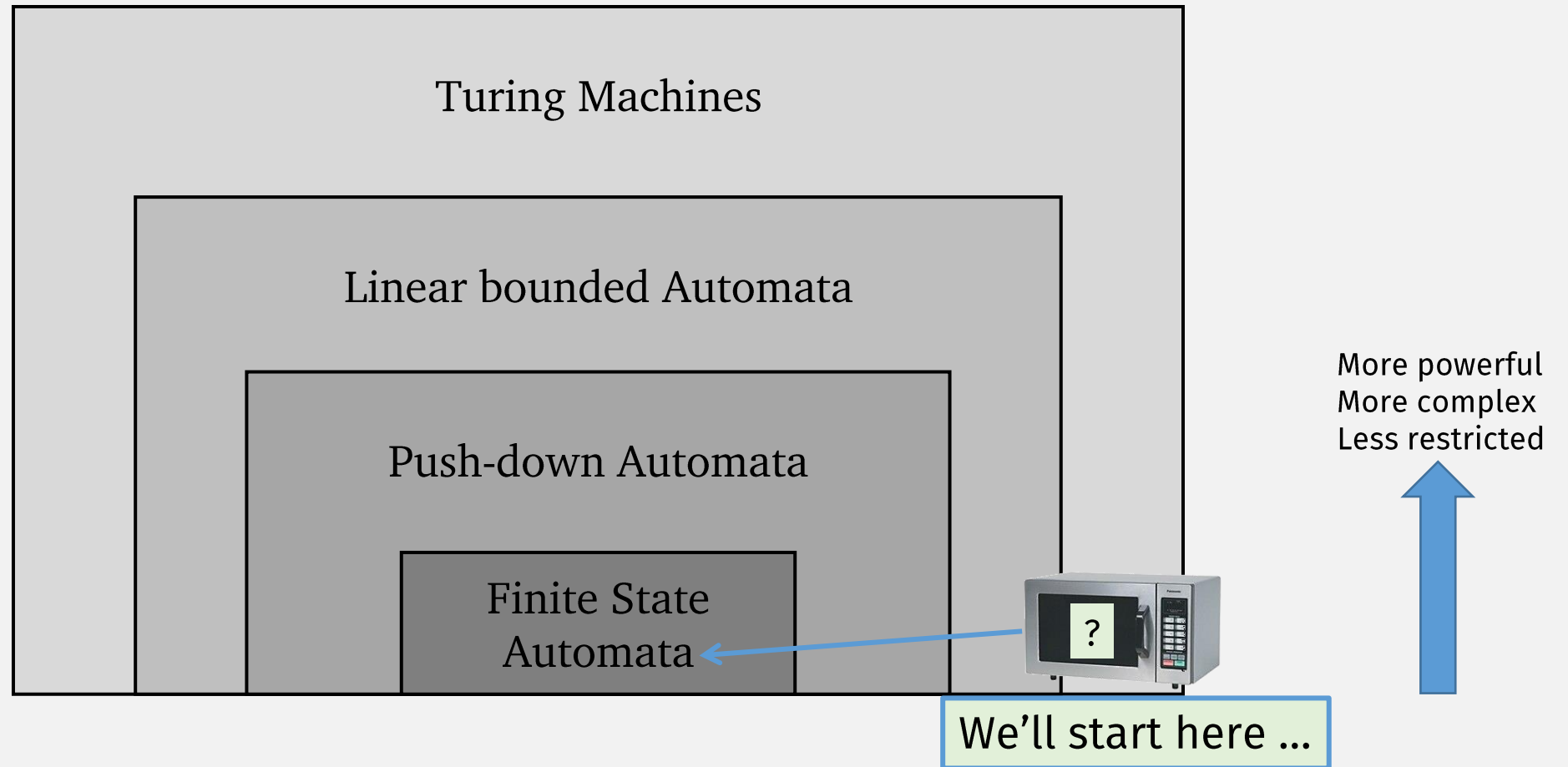
-  $P$

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.<sup>[1][2]</sup>

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.<sup>[3][4]</sup>

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

# Finite Automata: “Simple” Computation / “Programs”

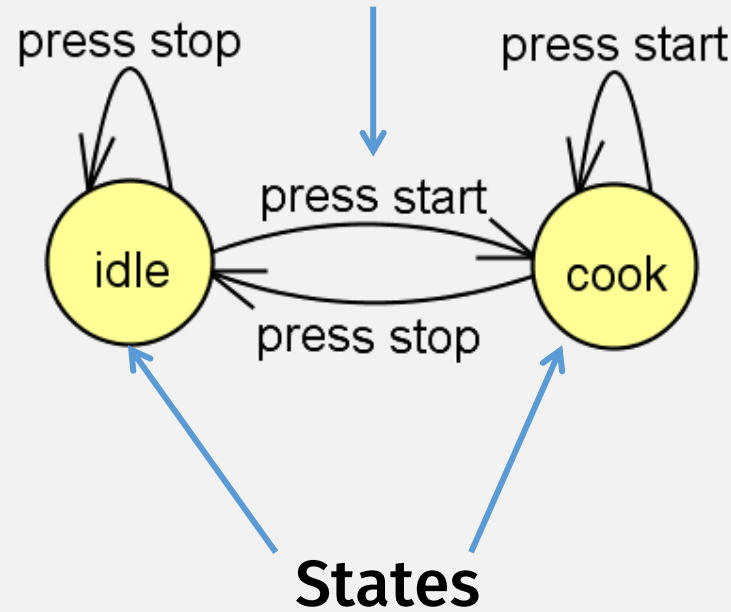


# Finite Automata

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

# A Microwave Finite Automata

**Input** “symbols” change states  
(possibly)



# 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 or methods defined in the pattern's interface.

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

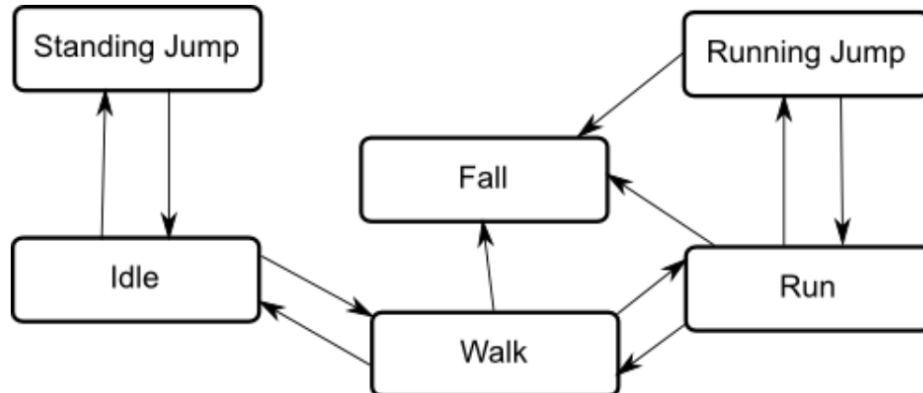




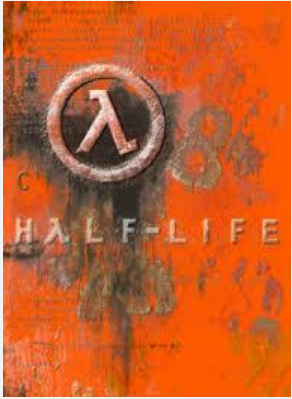
# Video Games Love Finite Automata

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 1.6k Issues Pull requests 23 Actions Projects Wiki

5d761709a3 halflife / game\_shared / bot / simple\_state\_machine.h

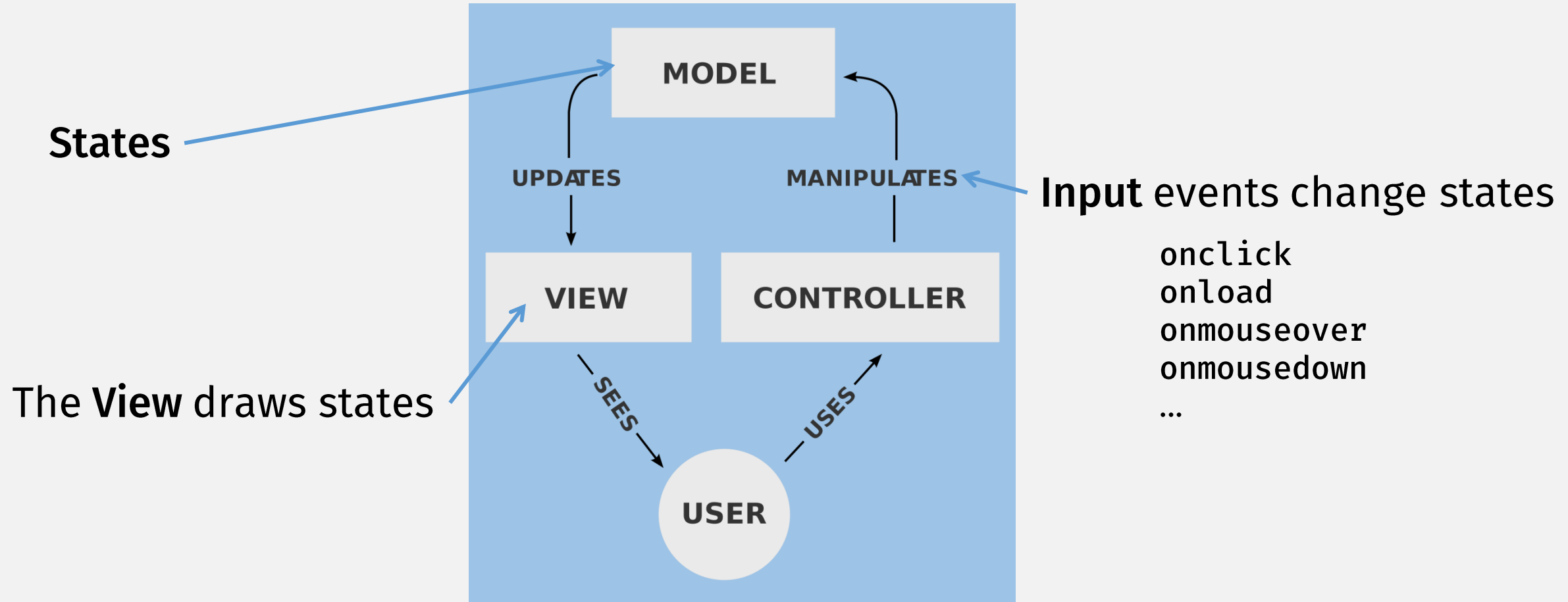
Alfred Reynolds initial seed of Half-Life 1 SDK

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
```

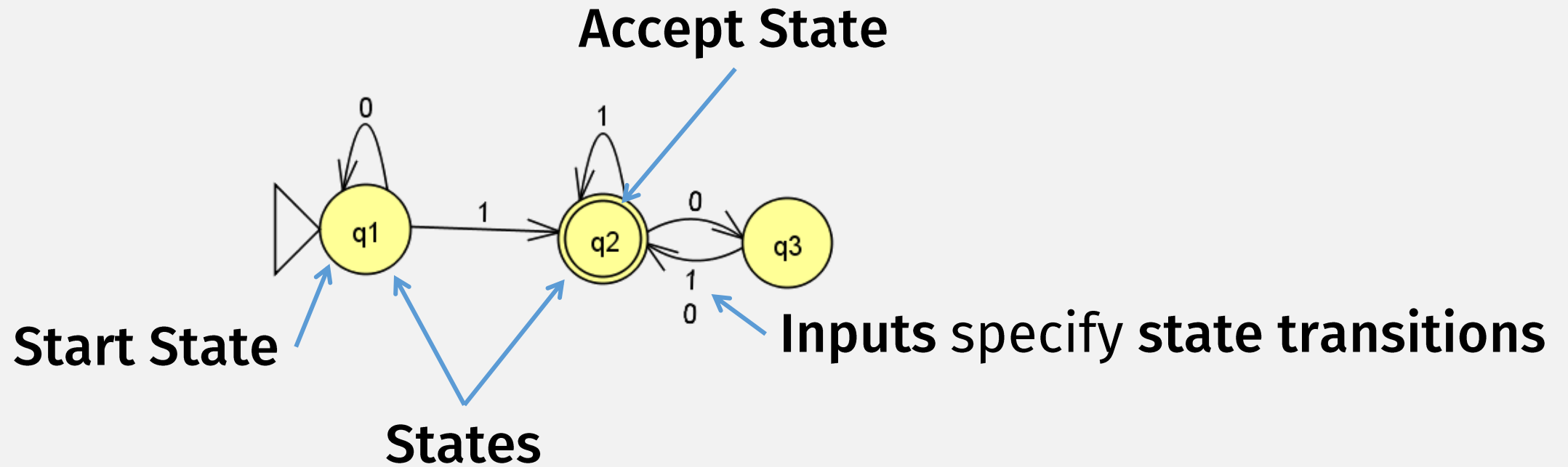
# Model-view-controller (MVC) is an FSM



# *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.  $\Sigma$  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

## DEFINITION

Set or sequence?

5 components

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

1.  $Q$  is a finite set called the *states*,
2.  $\Sigma$  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

- Unordered
- Duplicates not allowed
- Notation: { }
- **Empty set** written:  $\emptyset$  or { }
- A **language** is a (possibly infinite) set of strings

A set used a lot in this course

## Sequences

- Ordered
- Duplicates ok
- Notation: varies: ( ), comma, or concat
- **Empty sequence:** ( )
- A **tuple** is a finite sequence
- A **string** is a finite sequence of characters

sequences used a lot in this course

# Set or Sequence ?

A **function** is ...

... a **set** of **pairs**  
(1<sup>st</sup> of each pair from **domain**, 2<sup>nd</sup> from **range**)

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

## DEFINITION

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

**set**

1.  $Q$  is a finite set called the *states*,

**Set of pairs  
(domain)**

2.  $\Sigma$  is a finite set called the *alphabet*,

**set**

3.  $\delta: Q \times \Sigma \rightarrow Q$  is the *transition function*,

4.  $q_0 \in Q$  is the *start state*, and

**Set (range)**

5.  $F \subseteq Q$  is the *set of accept states*.

Don't know!  
(states can be  
anything)

**set**

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

# Finite Automata: The Formal Definition

## DEFINITION

5 components

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

1.  $Q$  is a finite set called the *states*,
2.  $\Sigma$  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”)

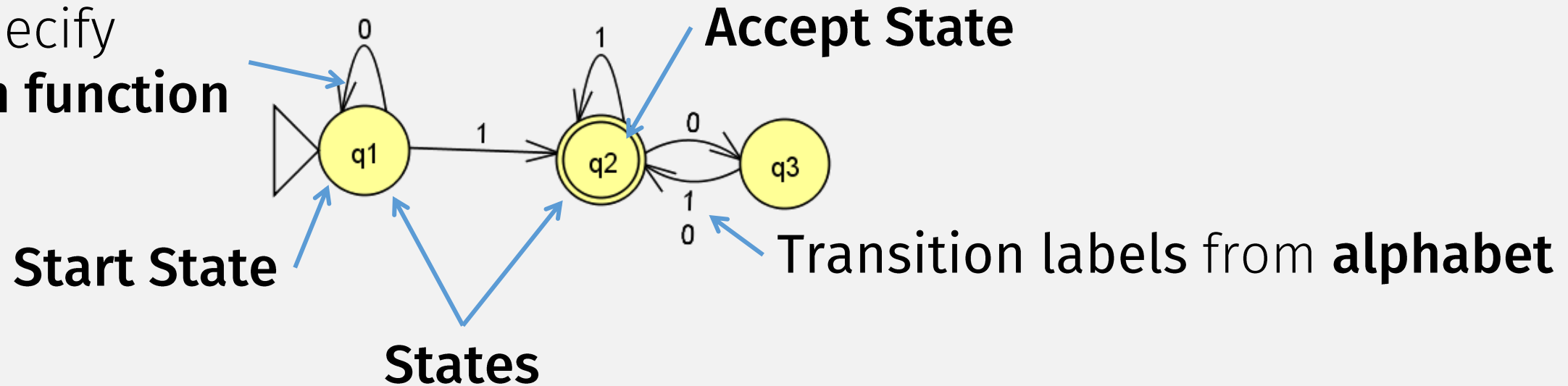
## DEFINITION

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

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

## Finite Automata: State Diagram

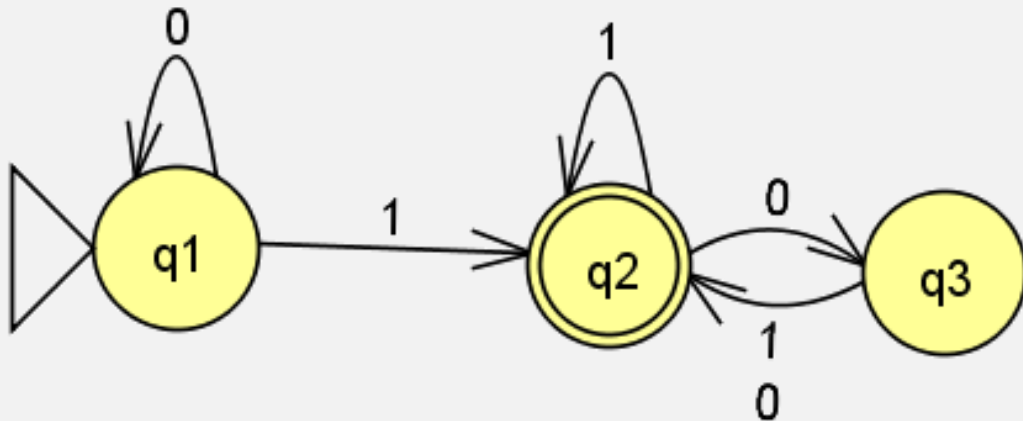
Arrows specify  
**transition function**



## 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.  $\Sigma$  is a finite set called the *alphabet*,
3.  $\delta: Q \times \Sigma \rightarrow Q$  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.  $\Sigma$  is a finite set called the *alphabet*,
3.  $\delta: Q \times \Sigma \rightarrow Q$  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 formal description)

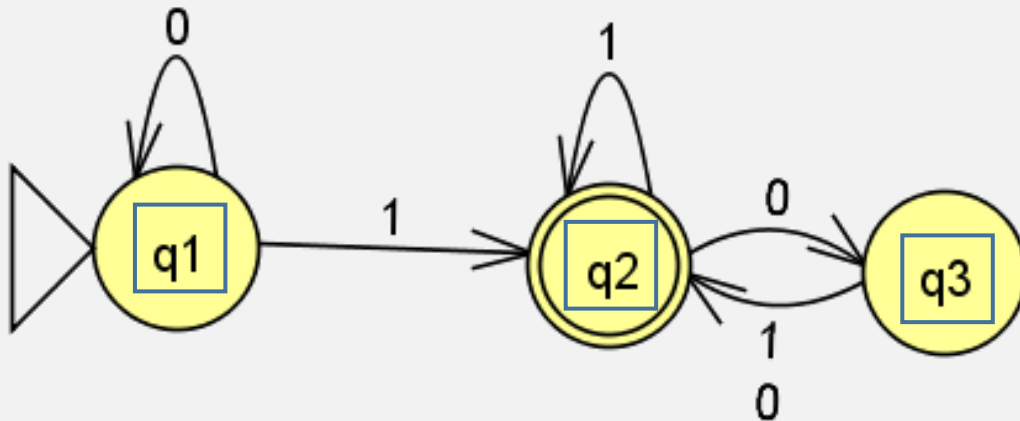
$M_1 = (Q, \Sigma, \delta, q_1, F)$ , where

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

braces =  
set notation  
(no duplicates)

	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.  $F = \{q_2\}$ .



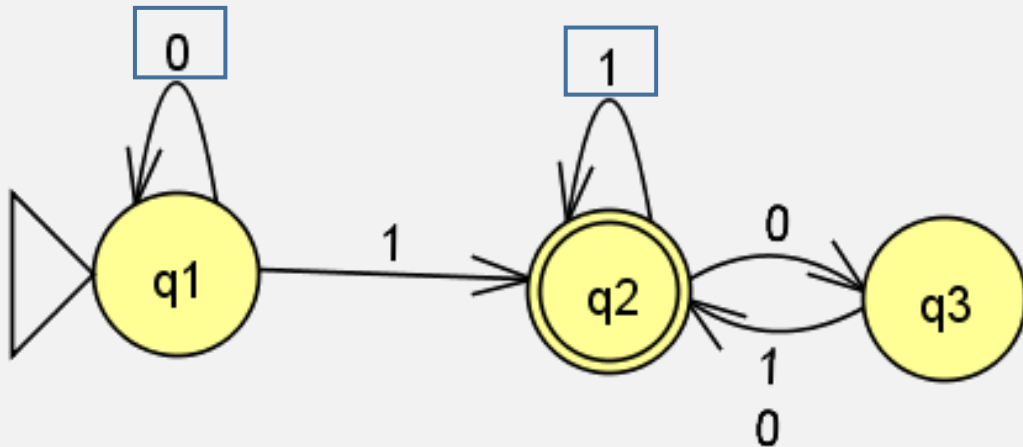
An Example (as **state diagram**)



## 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.  $\Sigma$  is a finite set called the *alphabet*,
3.  $\delta: Q \times \Sigma \rightarrow Q$  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.  $\delta$  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.  $F = \{q_2\}$ .

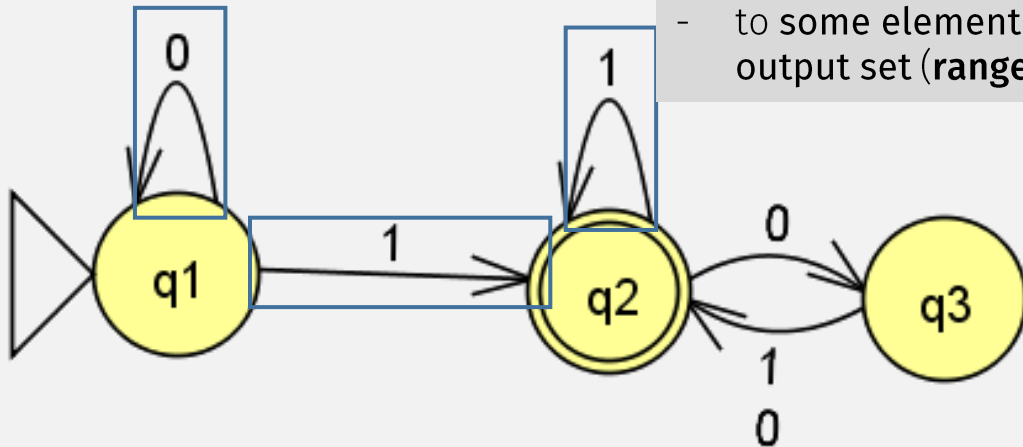
## DEFINITION

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

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

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.  $\delta$  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”

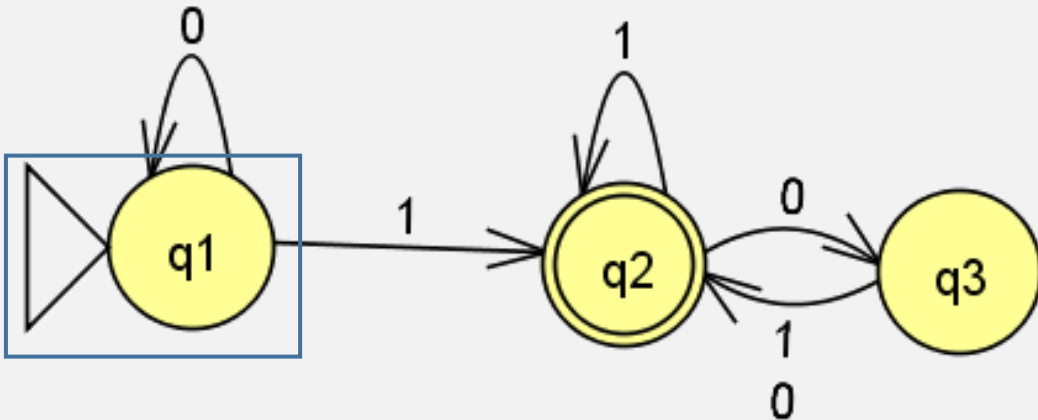
“Then go to this state”

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.  $\Sigma$  is a finite set called the *alphabet*,
3.  $\delta: Q \times \Sigma \rightarrow Q$  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.  $\delta$  is described as

	0	1
$q_1$	$q_1$	$q_2$
$q_2$	$q_3$	$q_2$
$q_3$	$q_2$	$q_2$

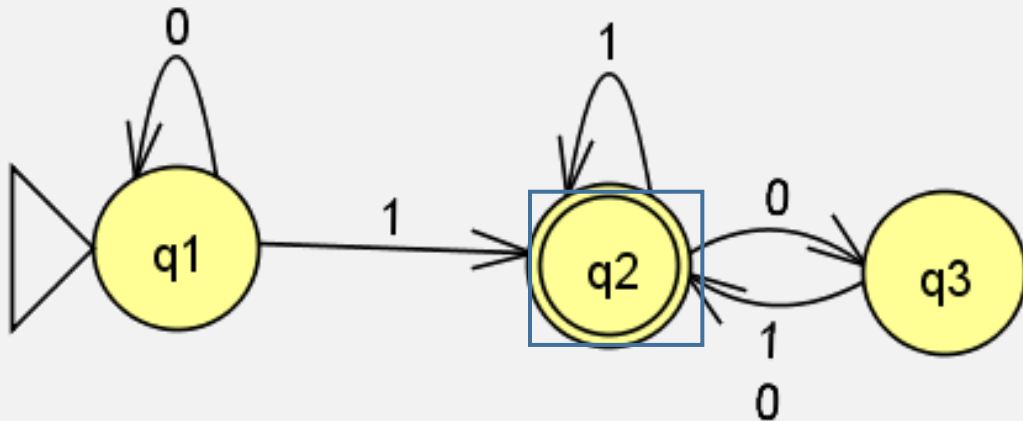
4.  $q_1$  is the start state, and
5.  $F = \{q_2\}$ .

## 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.  $\Sigma$  is a finite set called the *alphabet*,
3.  $\delta: Q \times \Sigma \rightarrow Q$  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.  $\delta$  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

## DEFINITION

### A “Programming Language”

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

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

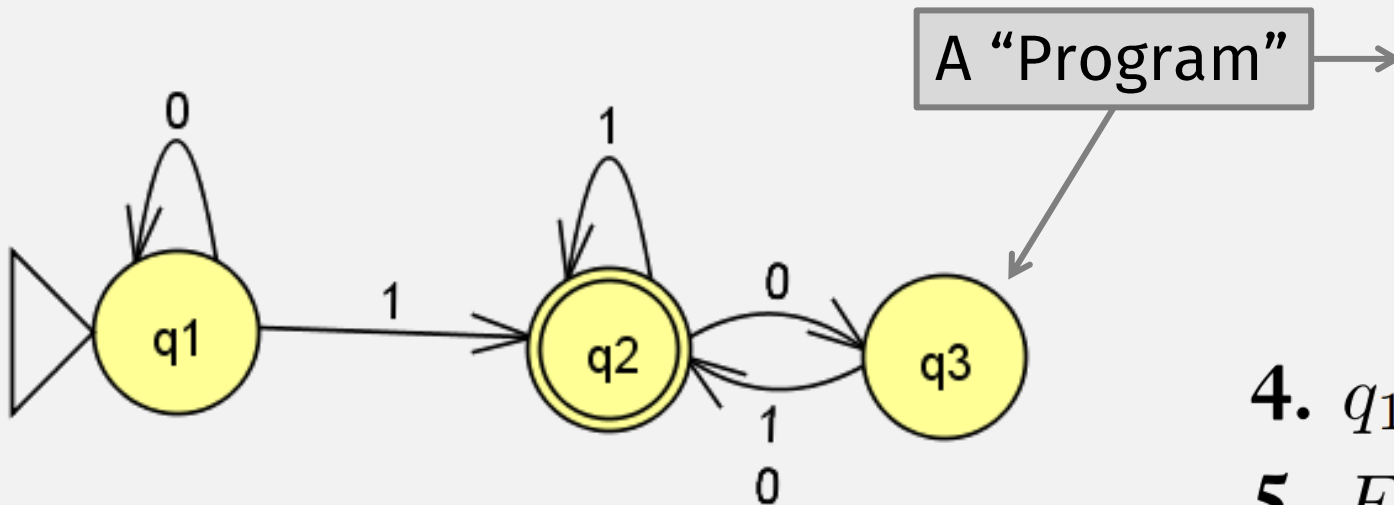
An Example (as formal description)

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

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



*“Programming” Analogy*

This “analogy” is meant to help your intuition

But it’s important not to confuse with **formal definitions**.

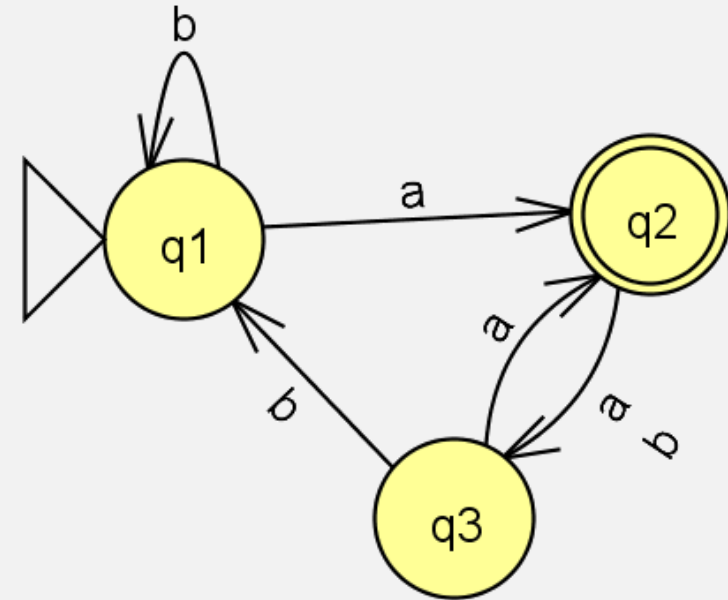
# In-class Exercise

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.  $\Sigma$  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

- $Q = \{q1, q2, q3\}$

- $\Sigma = \{ \mathbf{a}, \mathbf{b} \}$

- $\delta$

- $\delta(q1, \mathbf{a}) = q2$

- $\delta(q1, \mathbf{b}) = q1$

- $\delta(q2, \mathbf{a}) = q3$

- $\delta(q2, \mathbf{b}) = q3$

- $\delta(q3, \mathbf{a}) = q2$

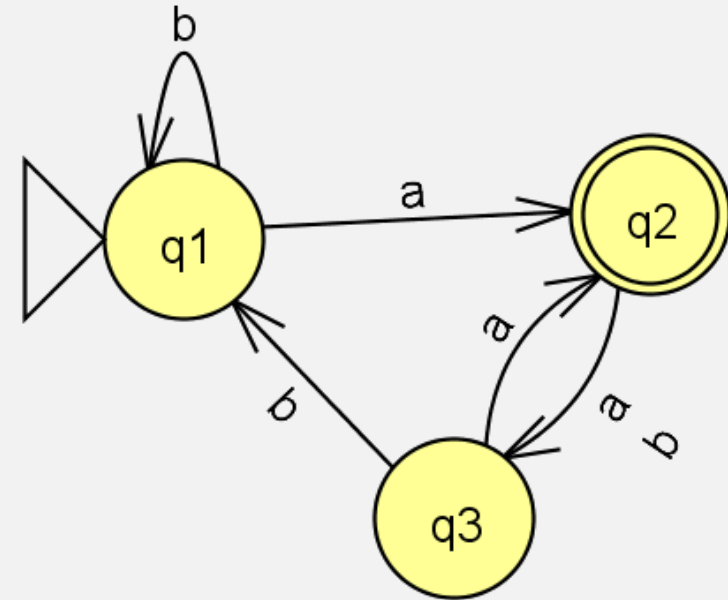
- $\delta(q3, \mathbf{b}) = q1$

- $q_0 = q1$

- $F = \{q2\}$

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

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



# 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 parser ([Grammar/python.gram](#)). The version here omits details related to code generation and error recovery.

```
# ===== 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 PFG).  
#
```



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

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

### 4.2. Naming and binding

# A Computation Model is ... (from lecture 1)

- Some definitions ...

## DEFINITION

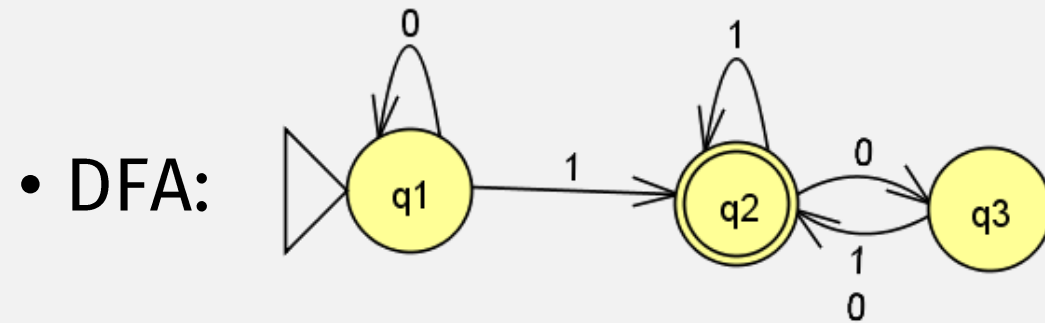
A *finite automaton* is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ , where

1.  $Q$  is a finite set called the *states*,
2.  $\Sigma$  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** ...

???

# Computation with DFAs (JFLAP demo)



• Input: “1101”

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