Using and Proving Logical Statements

CS 420

UMass Boston Computer Science

Stephen Chang

Thursday, September 10, 2026

Sentence Type	Symbolic Logic
Simple	p
Negation	$\neg p$
Conjunction	$p \wedge q$
Disjunction	$p \vee q$
Conditional	$p \Rightarrow q$
Biconditional	$p \Leftrightarrow q$

Calcworkshop.com

Announcements

HW

- Weekly
- in/out: **Tues 12:30pm**
- **HW 0** due: **Tues 9/15 12:30pm**

Lectures

- Closely follows listed textbook chapters
  - Differences will be explained
- **Slides** (summary pdf) **posted** (to website)
- **Recordings posted** (to canvas)
- **Nothing replaces attending lecture!**
  - not accepting feedback / questions on supplemental material

All course info available on **web site:**
`cs.umb.edu/~stchang/cs420/f26`

Supplemental questions (in GradeScope) PREVIEW

- Very little affect on final grades
- These are meant to help students who attend class
- Typically “due” with the next HW
- No late submissions (Please do not ask)

Q1 Using a (proof of) an AND statement
1 Point

In this course, if $proof_{A \wedge B}$ is proof of the statement $A \wedge B$, i.e., A AND B, then which of the following is a proof of the statement A

Q2 (Constructing) proof of an AND statement
1 Point

In this course, if $proof_A$ is a proof of statement A and $proof_B$ is a proof of statement B then which of the following is a proof of statement $A \wedge B$

Q4 Using (a proof of) an IF-THEN statement
1 Point

In this course, if $proof_{A \Rightarrow B}$ is a proof of the statement $A \Rightarrow B$, i.e., IF A THEN B, and $proof_A$ is a proof of a statement A, which of the following is a proof of the statement B:

Q3 (Constructing) proof of an IF-THEN statement
1 Point

In this course, which of the following is a proof of the statement $A \Rightarrow B$, i.e., IF A THEN B:

Last Time

This semester, we will …

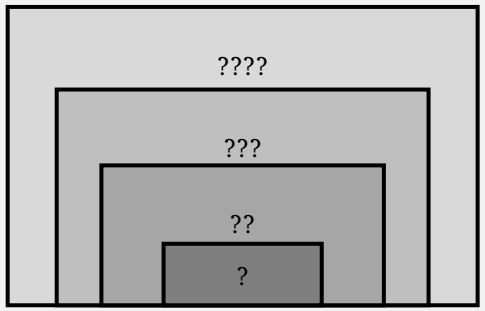
Analogy 1:
Computation Model
(system of definitions and rules)
⇔
Programming Language

1. Define and study **models of computation**
   - **models** will be *as simple as possible* (to make them easier to study)

formal, precisely-defined

Analogy 2:
A **Computation** (in a model)
⇔
A **Program**

2. Compare and contrast models of computation
   - which "programs" are *included / excluded* by a **model**
   - *Equality* or *overlap* between models?

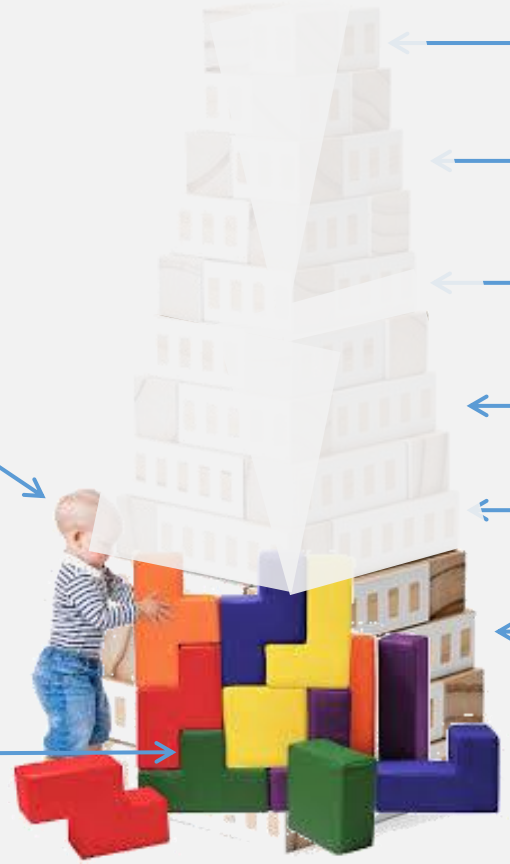


3. Prove things about the models

Analogy 3:
Proving is reasoning about **Program** behavior …
… but the act of **Proving** is (also) **Programming** !
So it's like **writing programs** that **analyze programs!**

Last Time

How Mathematics (Proofs) Work



Mathematician
(or student)

Adding next level is hard …
Preciseness is **important**
(just like in programming)

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

More **Theorems**

More **Axioms**

More **Definitions**

Theorem

Theorem

(user-defined library)

Axioms

Definitions

(standard library)

The "Modus Ponens" Inference Rule

(Precisely Fitting Blocks Together)

Premises (if we can prove these statements are true)

- IF P THEN Q ← More specifically, we say that we **APPLY** (**a proof of**) this
- P is TRUE ← **TO** (**a proof of**) this

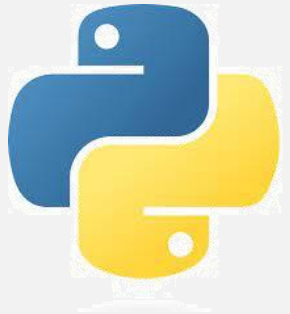
Conclusion (then we have proven that this is also true)

- Q must also be TRUE ← to get a (**a proof of**) this

It's exactly like a **function!**
Functions are **APPLIED TO** an **argument** to get a **result!**

You already do “Proof” when Programming

```
def no_div0(x):
    if (x > 0) | (x < 0) | (x == 0):
        return x + 1
    else:
        return 1 / 0
```



Can this function ever throw `ZeroDivisionError`?

No!

How did you figure out the answer?

You used the **Python model of computation** to **predict the program’s behavior**

You did a proof!

(Let’s write it out more formally)

Deductive Proof Example

```
def no_div0(x): "test expr"
    if (x > 0) | (x < 0) | (x == 0):
        return x + 1   "first branch"
    else:
        return 1 / 0   "second branch"
```

Prove: `no_div0` never throws `ZeroDivisionError`

Proof "construction" :

Prior proven become known facts, can be used to prove later steps!

Statements / Justifications Table
a proof is made of smaller proofs (each line)

	Statements	Justifications
	1. IF running "test expr" is True, THEN "first branch" runs	1. Rules of Python ("built-in" facts)
	2. IF running "test expr" is False, THEN "second branch" runs	2. Rules of Python
$Proof_3$	3. running "test expr" is (always) True	3. Definition of "numbers"
⇒	4. "first branch" (always) runs	4. **APPLY** $Proof_1$ **TO** $Proof_3$

Modus Ponens

If we can prove these:

- If P Then Q
- P

Then we've proved:

- Q

It's exactly like a **function**!
Functions are **APPLIED TO** an **argument** to get a **result**!

7. `no_div0` never throws `ZeroDivisionError`

(ends with the Statement we want to prove)

Deductive Proof Example

```
def no_div0(x):
    if (x > 0) | (x < 0) | (x == 0):
        return x + 1
    else:
        return 1 / 0
```
"second branch"

Prove: `no_div0` never throws `ZeroDivisionError`

Proof "construction" :

	Statements	Justifications
	1. IF running "test expr" is True, THEN "first branch" runs	1. Rules of Python
	2. IF running "test expr" is False, THEN "second branch" runs	2. Rules of Python
	3. running "test expr" is (always) True	3. Definition of "numbers"
$Proof_4$	4. "first branch" (always) runs	4. **APPLY** $Proof_1$ **TO** $Proof_3$
$Proof_5$	5. "second branch" *never* runs	5. Using $Proof_4$ and Rules of Python???
$Proof_6$	6. `no_div0` never runs `1 / 0`	6. Using $Proof_5$ and … ?
➡	7. `no_div0` never throws `ZeroDivisionError`	7. Using $Proof_6$ and Rules of Python???

Is there another inference rule?

(we will learn more rules about writing proofs as semester progresses)

■

Last Time

How Mathematics (Proofs) Work

More **Theorems**

More **Axioms**

More **Definitions**

Theorem

Theorem

(user-defined library)

Axioms

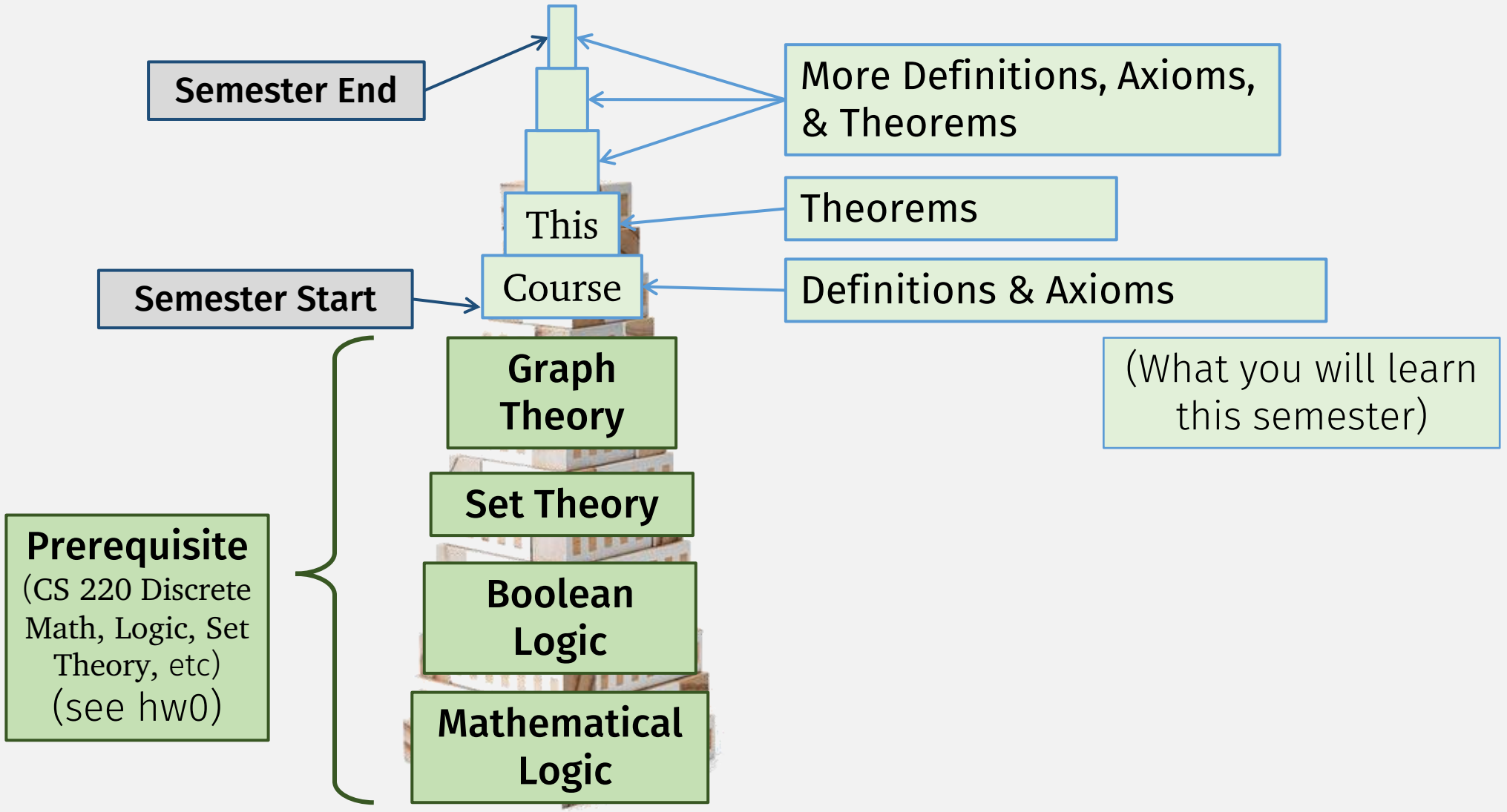
Definitions

(standard library)

Adding next level is hard …
Preciseness is **important**
(just like in programming)

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

How This Course Works



A Word of Advice

Important:
Do not fall behind
in this course

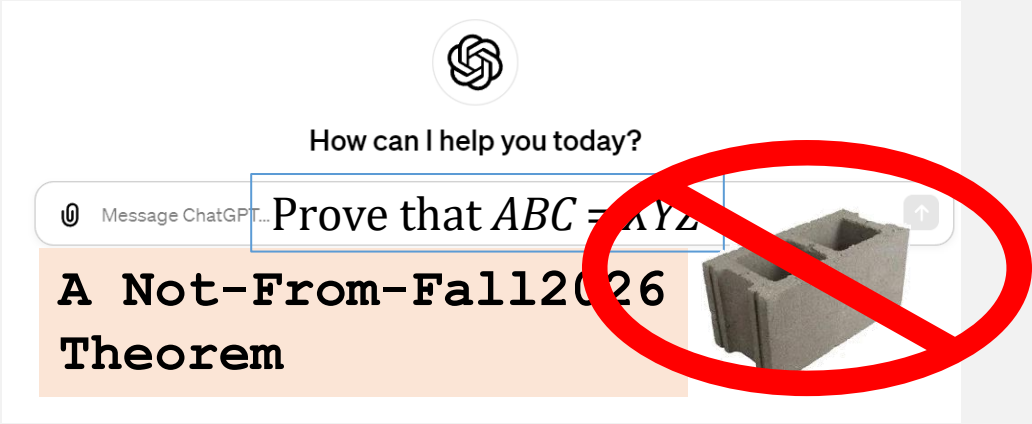
To **prove** a (new) **theorem** …

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

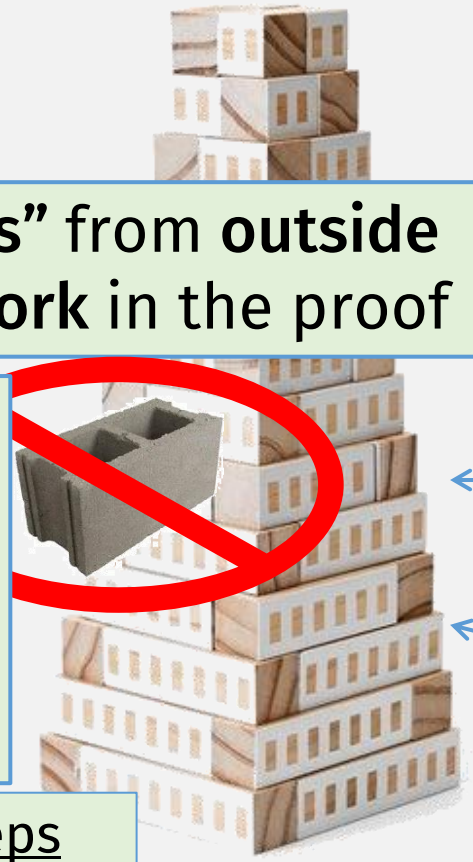
Another Word of Advice

HW 1, Problem 1

Prove that $ABC = XYZ$



"Facts" and **"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)

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

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

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

Grading

HW: 80%

- Weekly: In / Out Tuesday
- Approx. 12 assignments
- Lowest grade dropped (i.e., helps the most)

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 **web site:**
`cs.umb.edu/~stchang/cs420/f26`

Textbooks

- Sipser. *Intro to Theory of Computation*, 3rd ed.

- Hopcroft, Motwani, Ullman. *Intro to Automata Theory, Languages, and Computation*, 3rd 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 **web site:**
`cs.umb.edu/~stchang/cs420/f26`

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)
- Discussing HW in office hours
- Using other resources to learn, e.g., youtube, other textbooks, …
  - But: only theorems, syntax, formatting from this course can be used
- Students must always write up answers on their own, from scratch, in their own words

Not Allowed

- Submitting someone else's answer
- Submitting someone else's answer with:
  - variables changed,
  - thesaurus words,
  - or sentences rearranged …
- Theorems or definitions not from this course
  - You must use required "programming language"
- No AI (ChatGPT, Claude, DeepSeek, etc)

This is an AI-free UMB course

Required Message from the Provost:

"In this class, all submitted work must be original and created by the student(s) alone or in groups. Students should not have another person or entity write *any* portion of an assignment, including hiring others or using AI tools like ChatGPT. Always cite sources for quoted or referenced material. If unsure about a source's appropriateness, consult the instructor."

https://www.umb.edu/academics/provost/academic-integrity/

Honesty Policy

- 1st offense: zero on problem
- 2nd offense: zero on hw, reported to school
- 3rd 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):

All course info available on **web site:**
`cs.umb.edu/~stchang/cs420/f26`

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

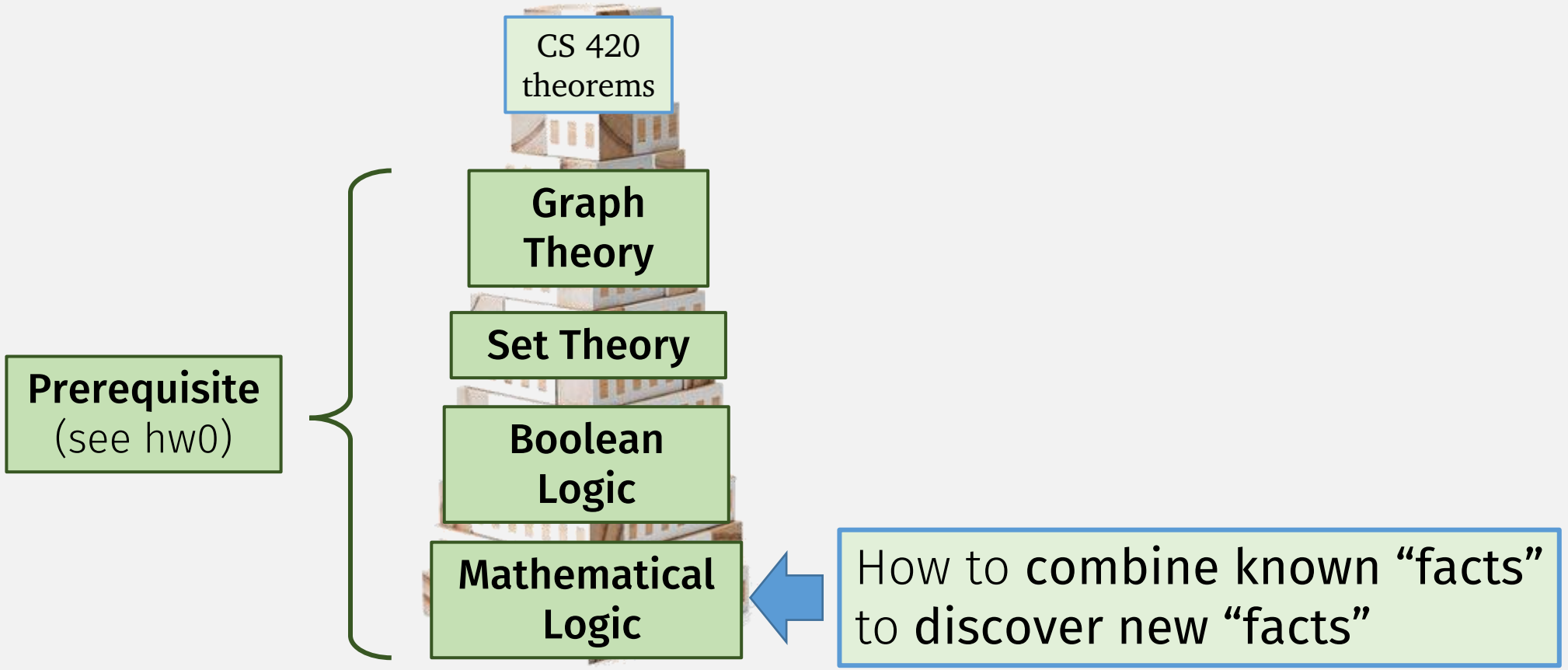
Axioms

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, $\neg$)
- **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

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**

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}

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

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

Examples?

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

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

Programming-style:

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

then to **get** a **proof of** Q ... **APPLY** $\mathrm{pr}_{P \Rightarrow Q}$ **TO** pr_P (it's like a function)

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** (AND "getter")
Proof_1	1. $x = a^2 + b^2 + c^2 + d^2$	1. **SECOND** ASSUMPTION — Use fn parameter
⋮	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
	4. $x \geq 4$	4. **SUBST** Proof_3 **INTO** Proof_1
	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!

(end with the Statement we want to prove)

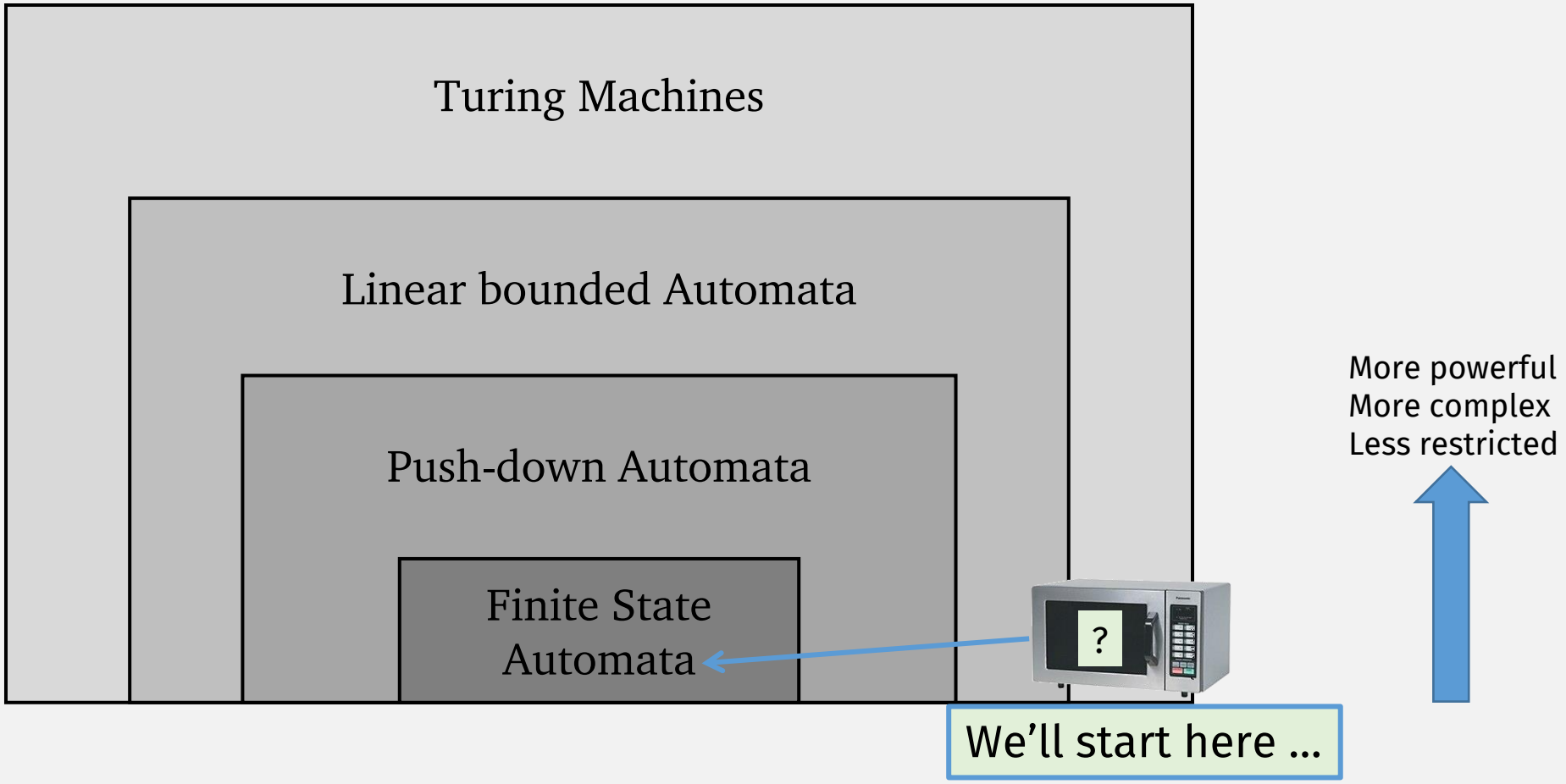
Some facts from previous courses will available as a "library"

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

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

Last Time

Models of Computation Hierarchy



A (Mathematical) Theory …

Mathematical theory

From Wikipedia, the free encyclopedia

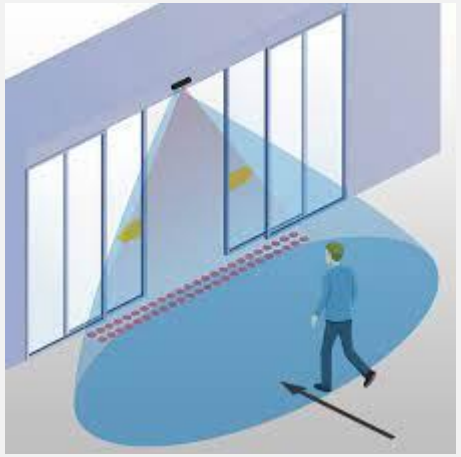
A **mathematical theory** is a mathematical model of a branch of mathematics that is based on a set of axioms. It can also simultaneously be a body of knowledge (e.g., based on known axioms and definitions), and so in this sense can refer to an area of mathematical research within the established framework.[1][2]

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 …

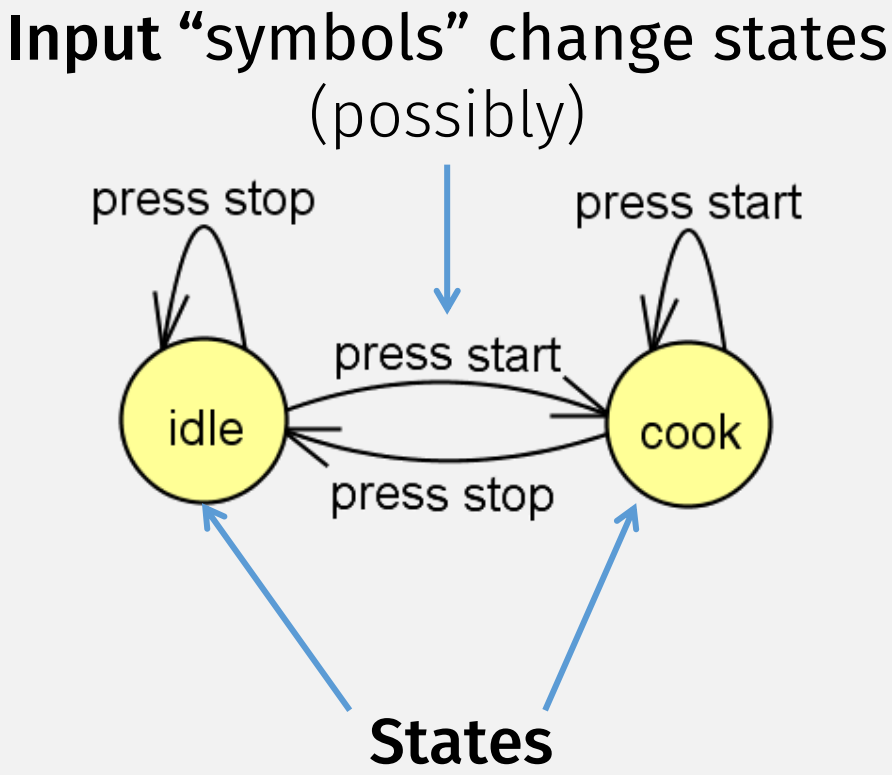
Finite Automata: “Simple” Computation / “Programs”



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)

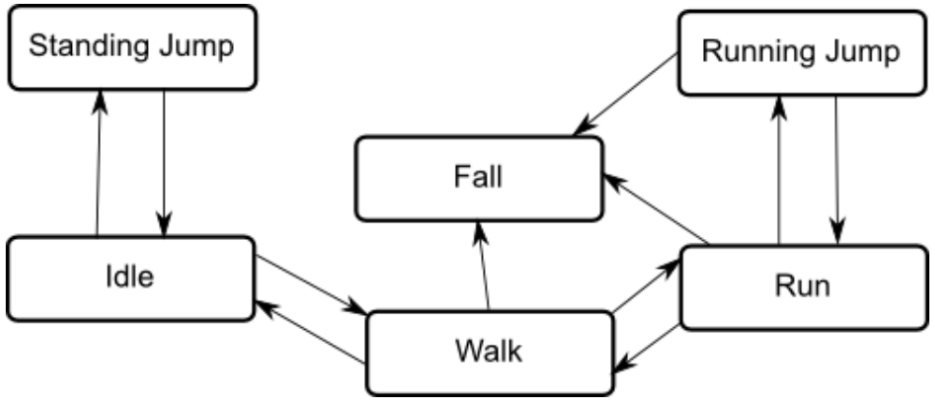
Video Games Love Finite Automata

Unity Documentation Manual

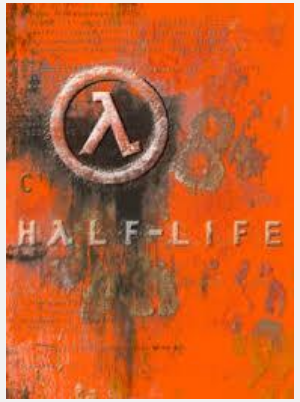
Unity User Manual 2020.3 (LTS) / Animation / Animator Controllers / Animation State Machines / State Machine Basics

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

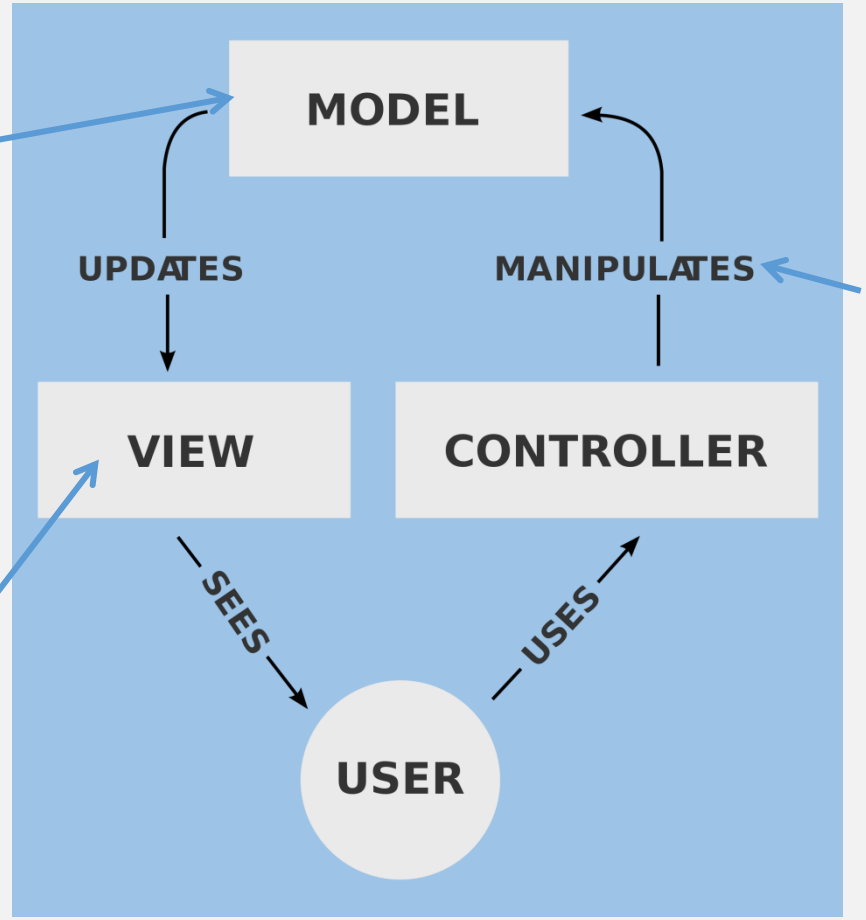
0 contributors

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
...
```

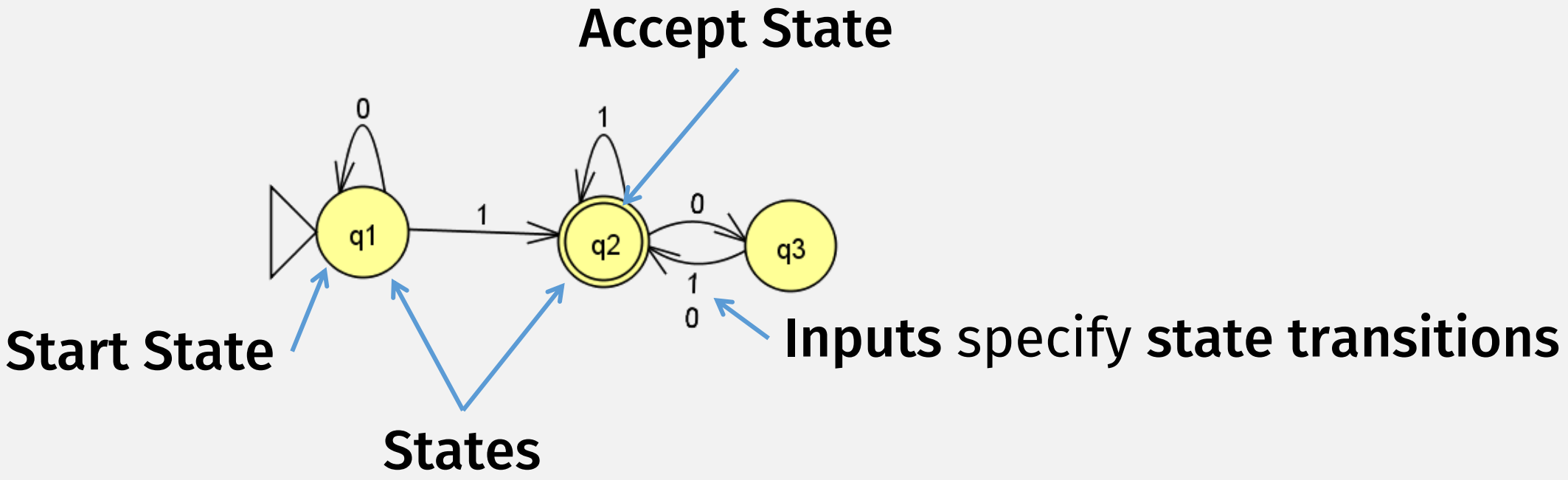
The **View** draws states

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 `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"