Welcome to

Theory of Computation

CS 420 / CS 620

UMass Boston Computer Science

Instructors: Stephen Chang and Holly DeBlois

Fall 2025

Today's Theme:
What's this course about?

Welcome to

Theory of Computation

CS 420 / CS 620

UMass Boston Computer Science

Instructors: Stephen Chang and Holly DeBlois

Fall 2025

What's this?

Interlude: Lecture Logistics

- *I expect:* **lecture** to be **interactive**
  - **Participation** is a part of your **grade**
  - Also, **it's the best way to learn!**

- *I may:* **call** on **students randomly**
  - It's **ok to be wrong in class**! – will not affect your grade
  - Also, **it's the best way to learn!**

- *Please:* **tell me your name** before **speaking**
  - Sorry in advance if I get it wrong
  - Also, **it's the best way for me to learn!**

Welcome to

Theory of Computation

CS 420 / CS 620

UMass Boston Computer Science

Instructors: Stephen Chang and Holly DeBlois

Fall 2025

How would you define this?

Computation Is … (via examples)

- 1 + 1 = ??
- = 2

… some **basic** <u>definitions</u> and <u>assumptions</u> (**"axioms"**),
e.g., define "Numbers" to be: 0, 1, 2, 3, … …

- 11 + 11 = ??
- = 22

… and <u>rules that use</u> the **definitions** and **axioms** (**"algorithm"**),
e.g., grade school arithmetic

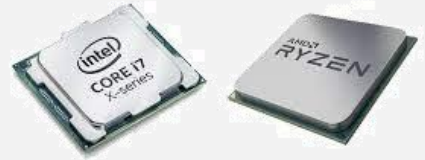
- 9999999999 + 9999999999 = ??
- = 19999999998

Computation rules can be <u>executed</u> by **hand,** or by **machine / automaton**

- 1 +1 = ??
- = 10

(binary)

There are <u>many</u> possible definitions (i.e., **models**) of **computation**



(hint)

Computation Is … Programs!

Every **programming language** <u>is</u> a **model of computation**

```
def bigger(x):
    if x > 0:
        return x + 1
    else:
        return x - 1

print( bigger(10) )
```

→ 11

different???

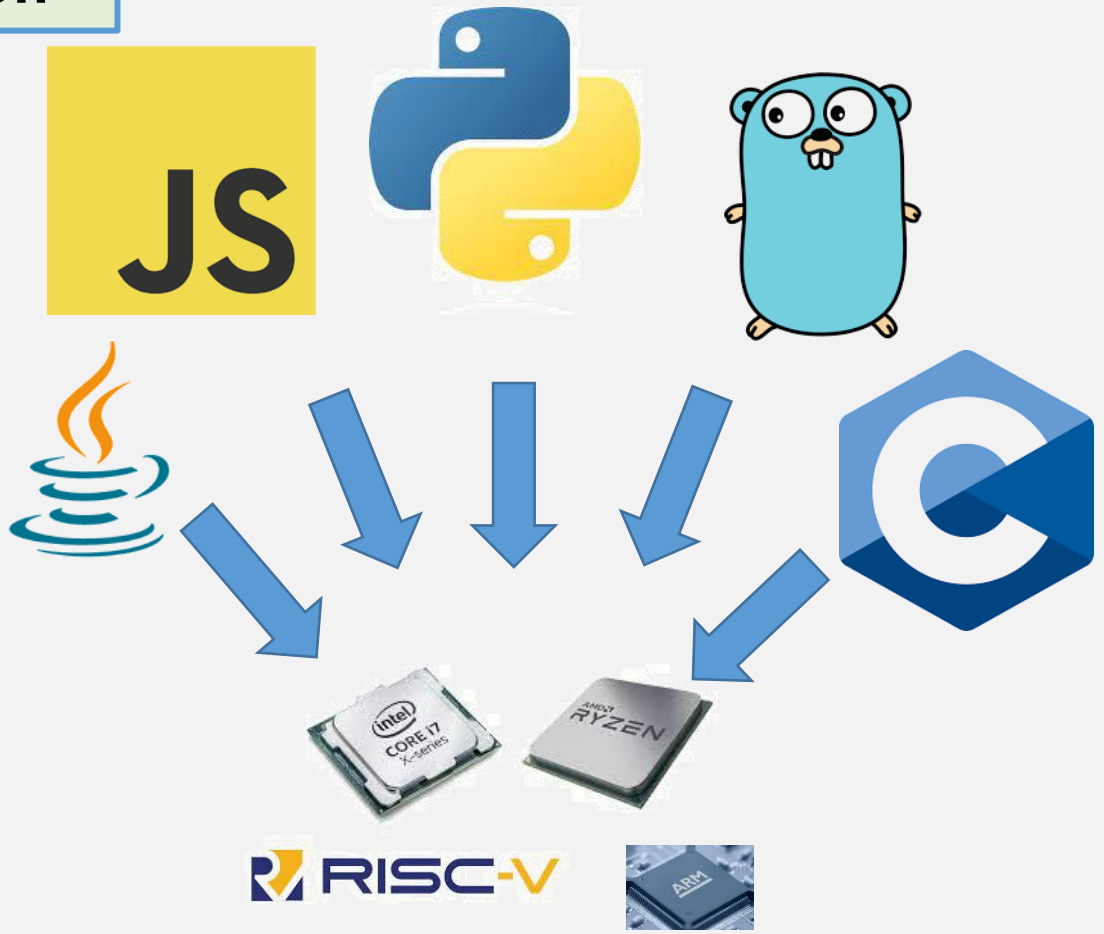
If they are <u>different</u>: how can we know?

Or same???

???

If they are the <u>same</u>: is there a common model for all?

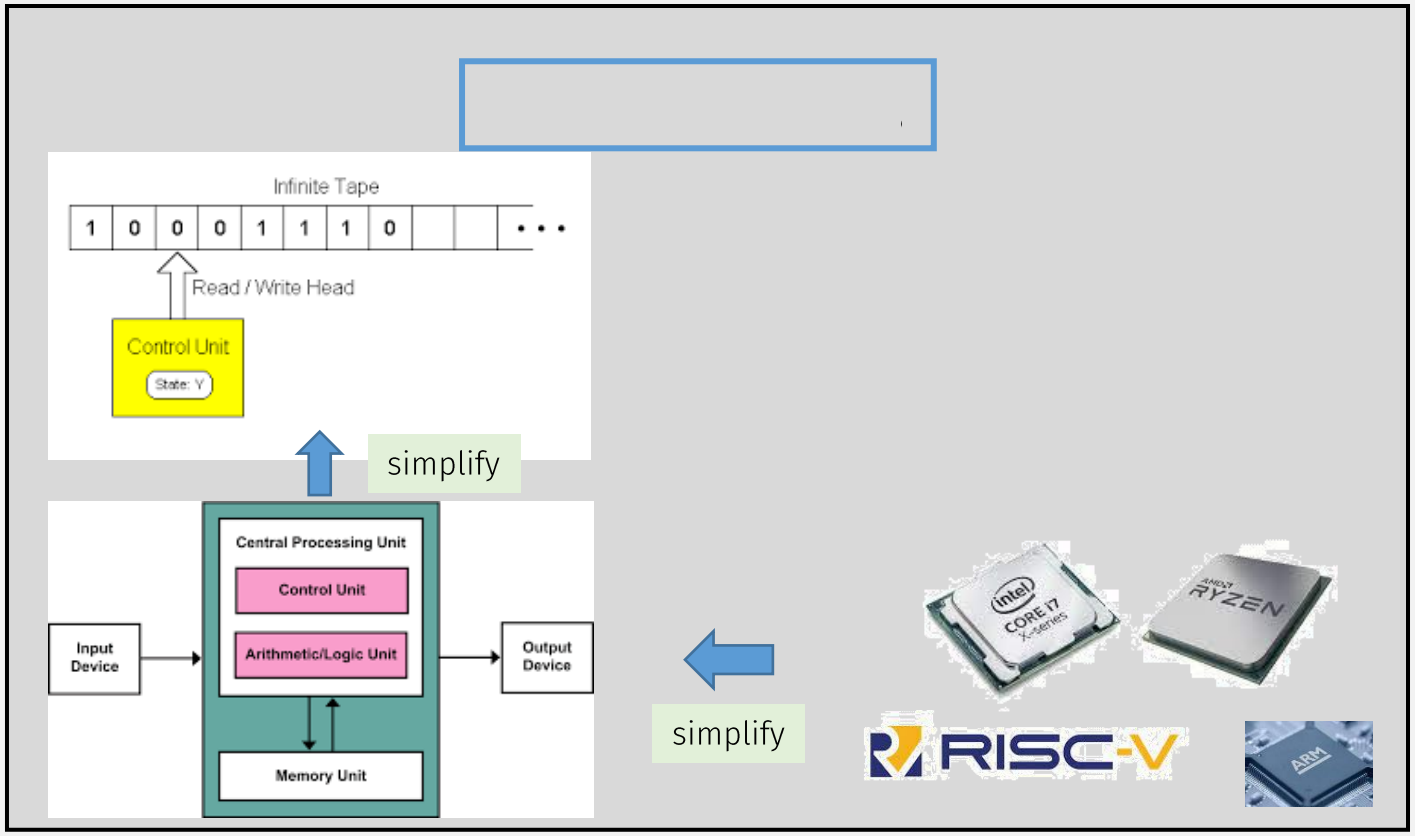
You <u>already</u> use **models of computation!** Every time you reason about code!



This semester, we will …

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

Models of Computation (spoiler alert!)



This semester, we will …

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

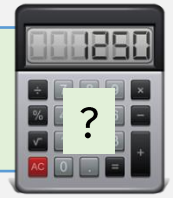
2. Compare and contrast models of computation
   - which “programs” are *included* by a **model**
   - which “programs” are *excluded* by a **model**
   - *overlap* between models?

Models of Computation

Q_1: Are there computational models … other than **Turing Machines?**

Turing Machines

Q_2: Are there computational models … “*weaker*” than **Turing Machines?**



Q_3: Are there computational models … “*more powerful*” than **Turing Machines?**

Q_4: What does “*weaker*” or “*more powerful*” even mean?!

A: Yes, yes, yes, and … stay tuned!

Models of Computation Hierarchy

… and get to here …

… and also look at what's out here???

Turing Machines

Linear bounded Automata

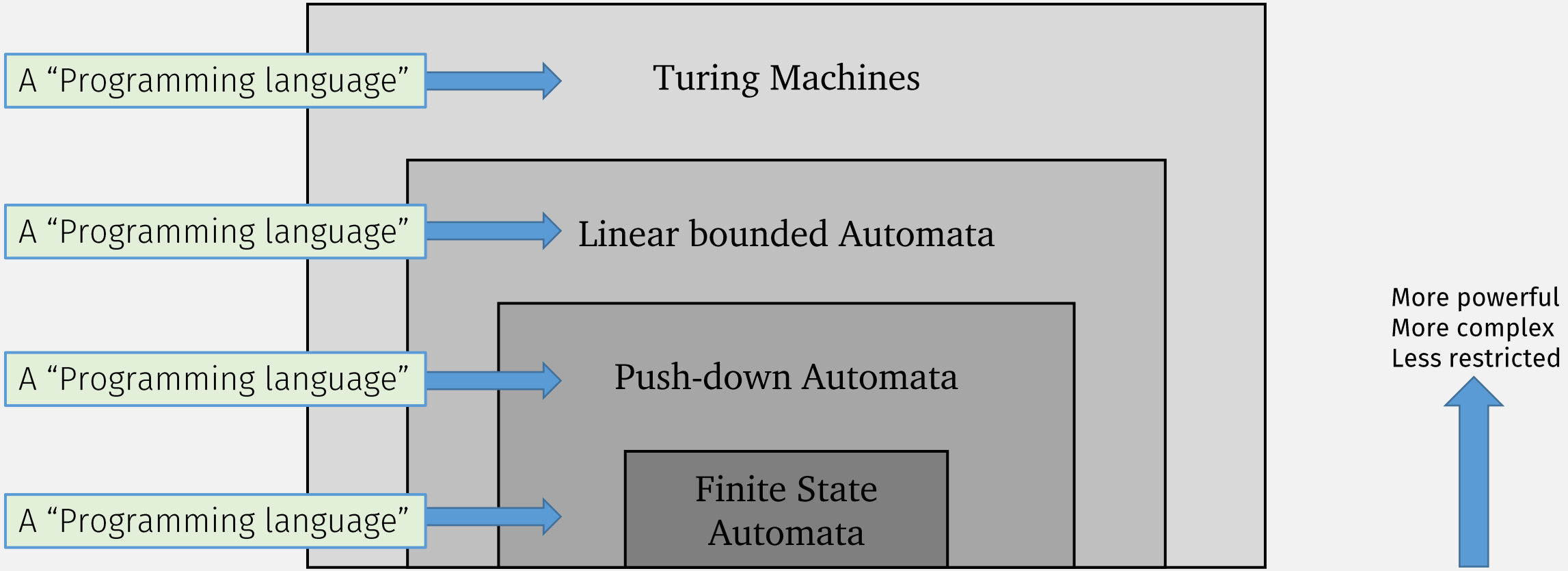
Push-down Automata

Finite State Automata

We'll start here …

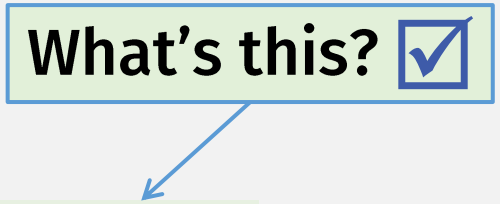
More powerful
More complex
Less restricted

But remember ... Computation = Programs!



Helpful analogy *for this course:*

- a **set** of machines / computational model (a rectangle) ~ a **Programming Language!**
- a **single** machine (one thing in a rectangle) ~ a **Program!**



Welcome to
Theory of Computation

CS 420 / CS 620

UMass Boston Computer Science

Instructors: Stephen Chang and Holly DeBlois

Fall 2025

What's this?

Welcome to Theory of Computation

CS 420 / CS 620

UMass Boston Computer Science

Instructors: Stephen Chang and Holly DeBlois

Fall 2025

"Theory" = math

(This is a math course!)

(But programming is math too!)

Programming Is (What) Math?

Math(ematical) logic!

```
def bigger(x):
    if x > 0:
        return x + 1
    else:
        return x - 1

print( bigger(10) )
```

???

→ 11

How did you figure out the answer?

(But programming is math too!)

Programming = Mathematical logic!

- "logic is the foundation of all computer programming"
  - https://www.technokids.com/blog/programming/its-easy-to-improve-logical-thinking-with-programming/
- "logic is the fundamental key to becoming a good developer"
  - https://www.geeksforgeeks.org/i-cant-use-logic-in-programming-what-should-i-do/
- "Analytical skill and logical reasoning are prerequisites of programming because coding is effectively logical problem solving at its core"
  - https://levelup.gitconnected.com/the-secret-weapon-of-great-software-engineers-22d57f427937

(Studying logic, i.e., this class, will make you a better programmer!)

Programming = Mathematical logic!

Programming Concepts

- Functions
- Variables
- If-then
- Recursion
- Strings
- **Sets** (and other data structures)

Math(ematical Logic) Concepts

- Functions
- Variables
- If-then (implication)
- Recursion
- Strings
- **Sets** (and other groupings of data)

(Studying logic, i.e., this class, will make you a better programmer!)

This semester, we will …

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

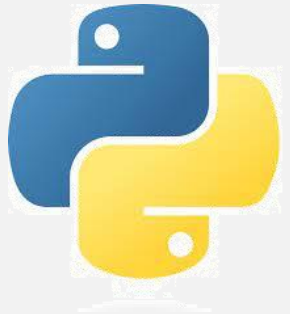
2. Compare and contrast models of computation
   - which "programs" are *included* by a **model**
   - which "programs" are *excluded* by a **model**
   - *overlap* between models?

3. Prove things about the models

Reasoning About Code is ~~Math~~ Proof

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

print( no_div0(10) )???
```



→ 11

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!

A (Mathematical) Theory Is …

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.

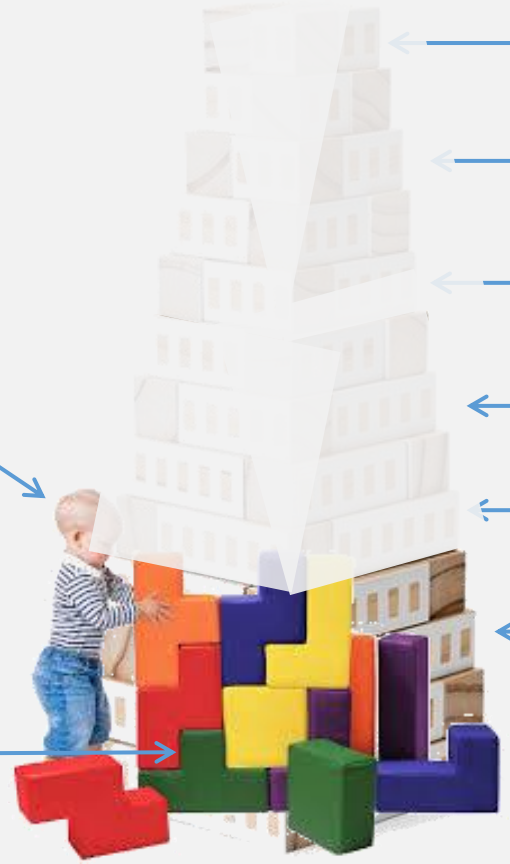
The singular accomplishment of axiomatic set theory is its ability to give a foundation for the derivation of the entirety of classical mathematics from a handful of axioms. The reason set theory is so prized is because of its explanatory depth. So a mathematical theory which just postulates an infinity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

… a mathematical model, i.e., **axioms** and **definitions**, of some domain, e.g. computers …

… that **explains** (**predicts**) some real-world phenomena …

… and can **derive** (prove) additional results (**theorems**) …

How Mathematics (Proofs) Work



More **Theorems**

More **Axioms**

More **Definitions**

Theorem

Theorem

Axioms

Definitions

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

The “Modus Ponens” Inference Rule

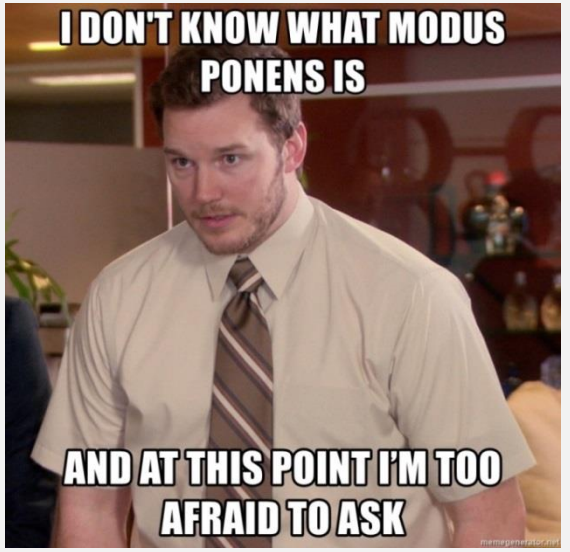
(Precisely Fitting Blocks Together)

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**



Kinds of Mathematical Proof

Deductive Proof

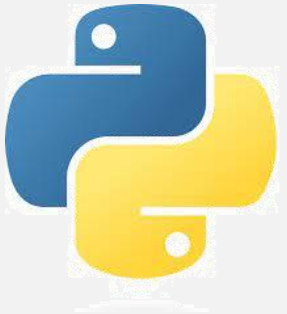
- *Start with*: known facts and statements

- *Use:* logical **inference rules** (like modus ponens) to **prove** new **facts** and **statements**

You already do “Proof” when Programming

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

print( no_div0(10) ) ???
```



→ `11`

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**

(Let’s write it out formally)

You did a proof!

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: Prior steps are already-proved, can be used to prove later steps!

Statements / Justifications Table

	Statements	Justifications
1.	If running "test expr" is `True`, then "first branch" runs	Rules of Python
2.	If running "test expr" is `False`, then "second branch" runs	Rules of Python
3.	running "test expr" is (always) `True`	Definition of "numbers"
4.	"first branch" (always) runs	By steps **1, 3,** and **modus ponens**
7.	`no_div0` never throws `ZeroDivisionError`	

Modus Ponens

If we can prove these:

- If P then Q
- P

Then we've proved:

- Q

Deductive Proof Example

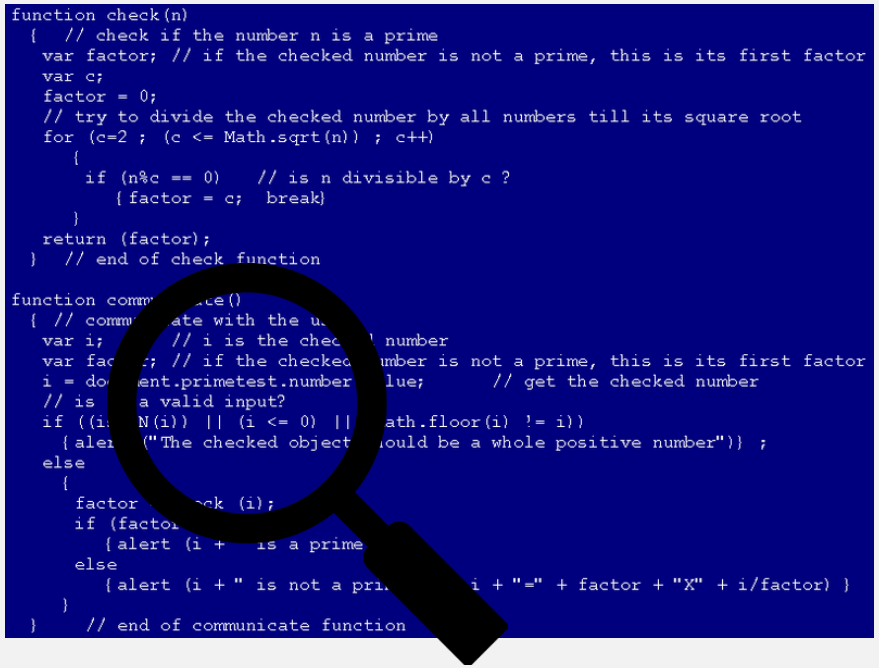
```
def no_div0(x):
    if (x > 0) | (x < 0) | (x == 0):
        return x + 1
    else:
        return 1 / 0
```
"second branch"

Prove: `no_div0` never throws `ZeroDivisionError`

Proof:

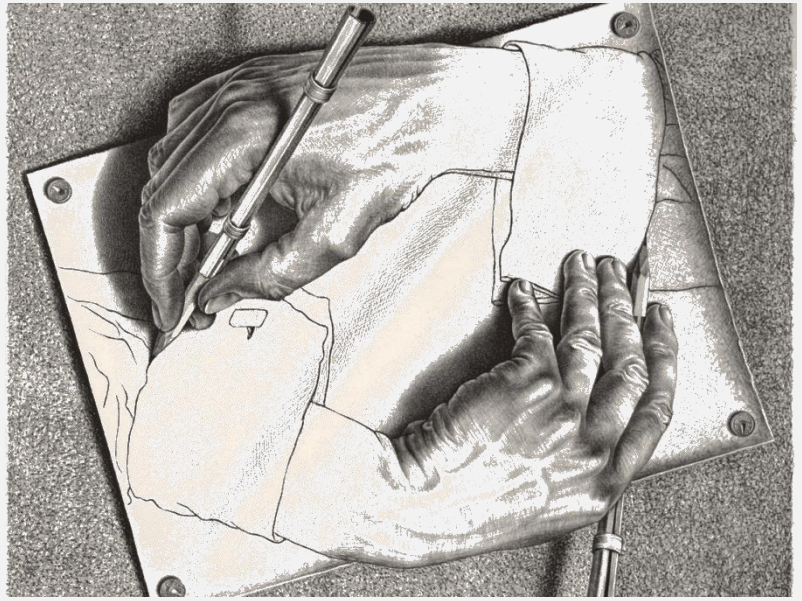
	Statements	Justifications
1.	If running "test expr" is `True`, then "first branch" runs	1. Rules of Python
2.	If running "test expr" is `False`, then "second branch" runs	2. Rules of Python
3.	running "test expr" is (always) `True`	3. Definition of "numbers"
4.	"first branch" (always) runs	4. By steps **1**, **3**, and modus ponens
5.	"second branch" *never* runs	5. By step **4**, and Rules of Python???
6.	`no_div0` never runs `1 / 0`	6. By step **5**
7.	`no_div0` never throws `ZeroDivisionError`	7. By step **6** and Rules of Python???

What else can we prove about programs?



Proof = prediction about program result
… without running the program

Can we make predictions about computation?



It's tricky: **Trying to predict computation requires computation!**

Can we make predictions about computation?

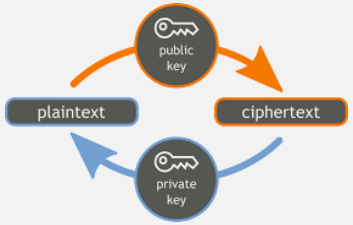
- The **Halting Lemma** says:

- And **Rice's Theorem** says:
  - "all non-trivial, semantic properties of programs are undecidable"

Knowing What Computers Can't Do is Still Useful!

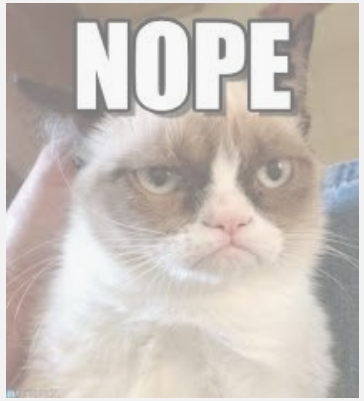
In Cryptography:

- Perfect secrecy is impossible in practice
- But with slightly imperfect secrecy (i.e., a computationally bounded adversary) we get:



Can we make predictions about computation?

- The **Halting Lemma** says:



- And **Rice's Theorem** says:
  - "all non-trivial, semantic properties of programs are undecidable"

Actually:

- it depends on the computation model!

Predicting What Some Programs Will Do …

microsoft.com/en-us/research/project/slam/

SLAM is a project for checking that software satisfies critical behavioral properties of the interfaces it uses and to aid software engineers in designing interfaces and software that ensure reliable and correct functioning. Static Driver Verifier is a tool in the Windows Driver Development Kit that uses the SLAM verification engine.

"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability." **Bill Gates, April 18, 2002. Keynote address at WinHec 2002**

Predicting things about programs … is the **Holy grail of CS**!

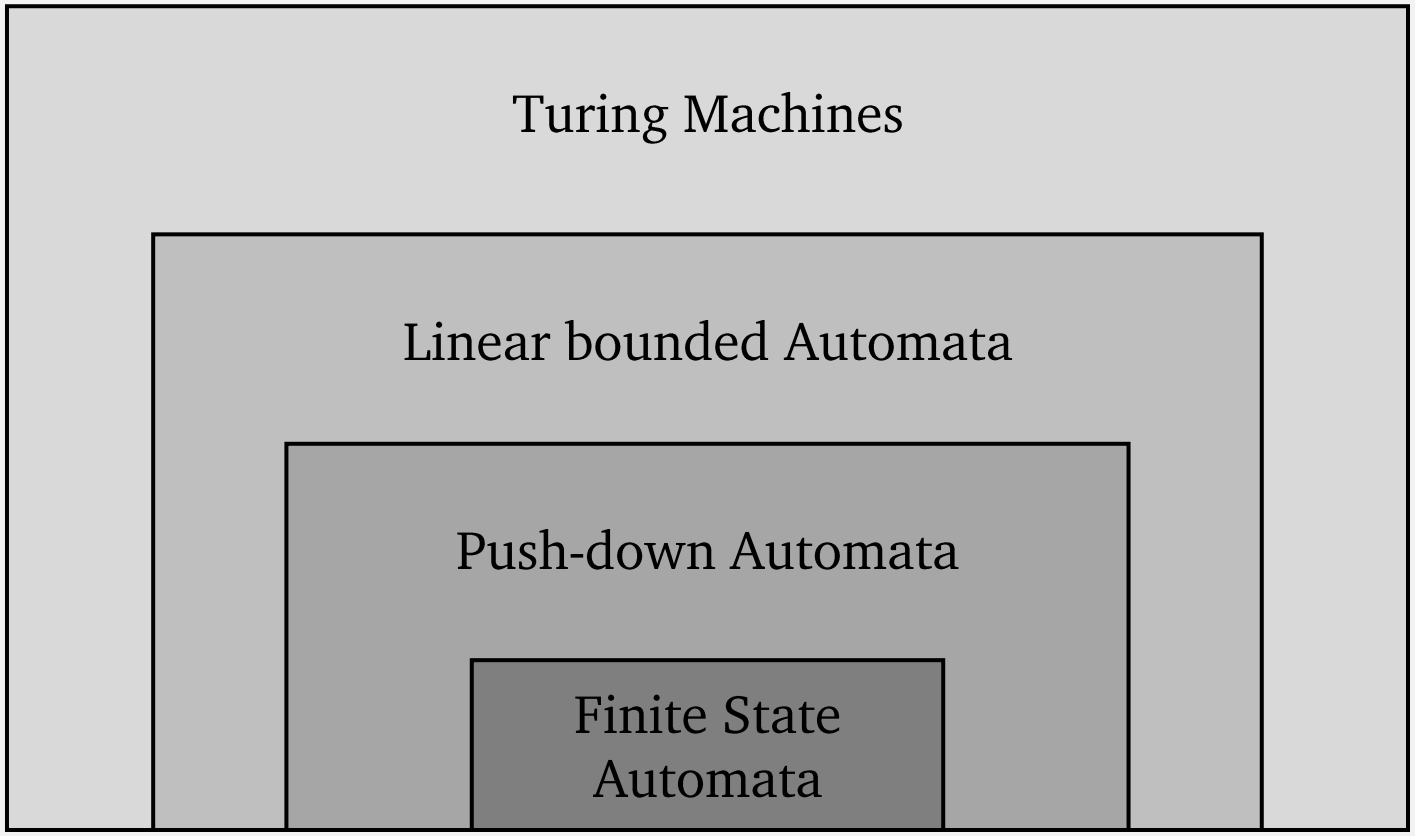
Static Driver Verifier Research Platform README

Overview of Static Driver Verifier Research Platform

Static Driver Verifier (SDV) is a compile-time static verification tool, included in the Windows Driver Kit (WDK). The SDV Research Platform (SDVRP) is an extension to SDV that allows you to adapt SDV to:

- Support additional frameworks (or APIs) and write custom SLIC rules for this framework.
- Experiment with the model checking step.

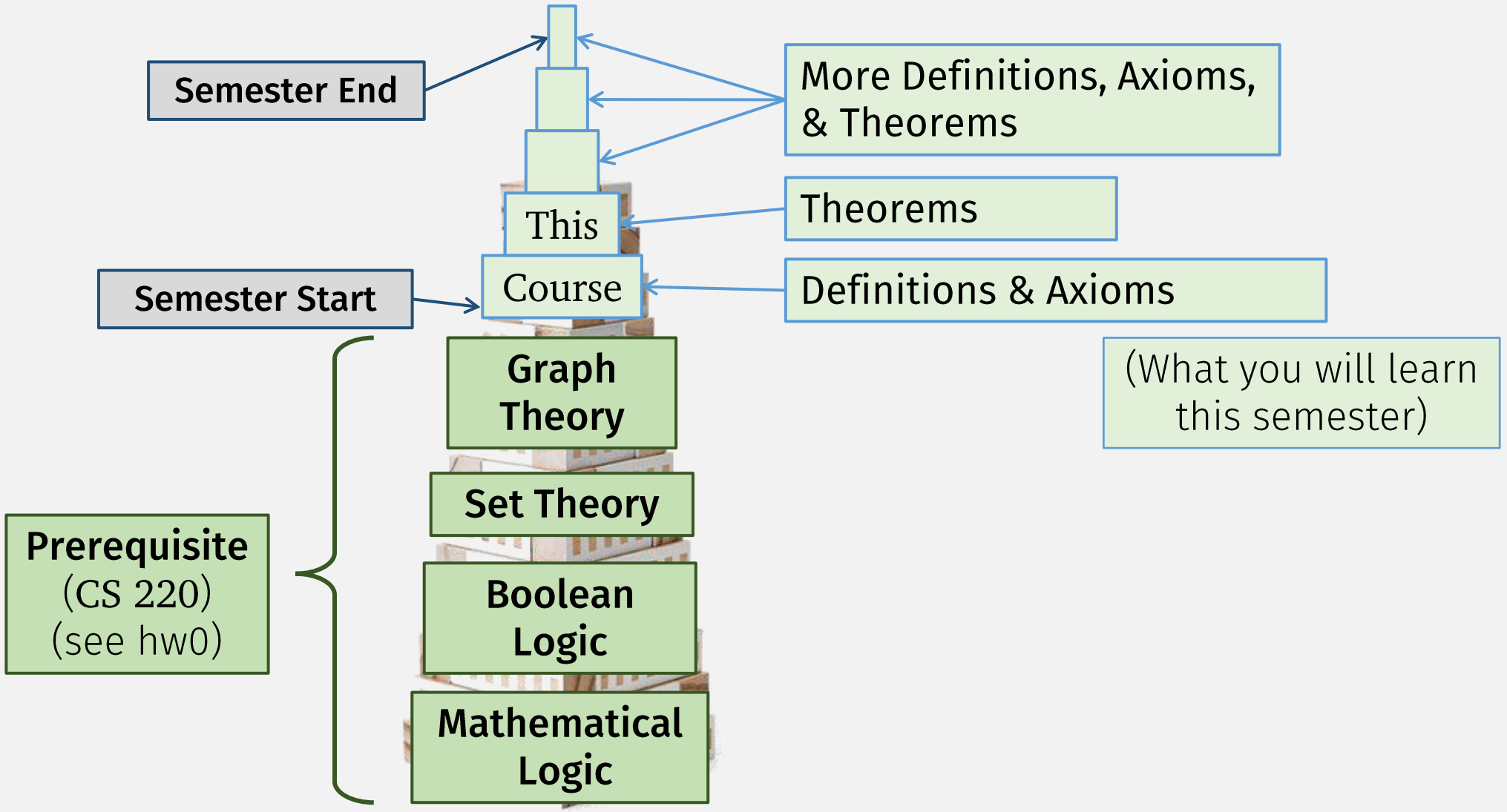
Proofs About Computational Models … in this class



More powerful
More complex
Less restricted

In this class, we will **prove** things about **simple computational models** (not Python …)

How This Course Works



A Word of Advice

Important:
Do not fall behind
in this course

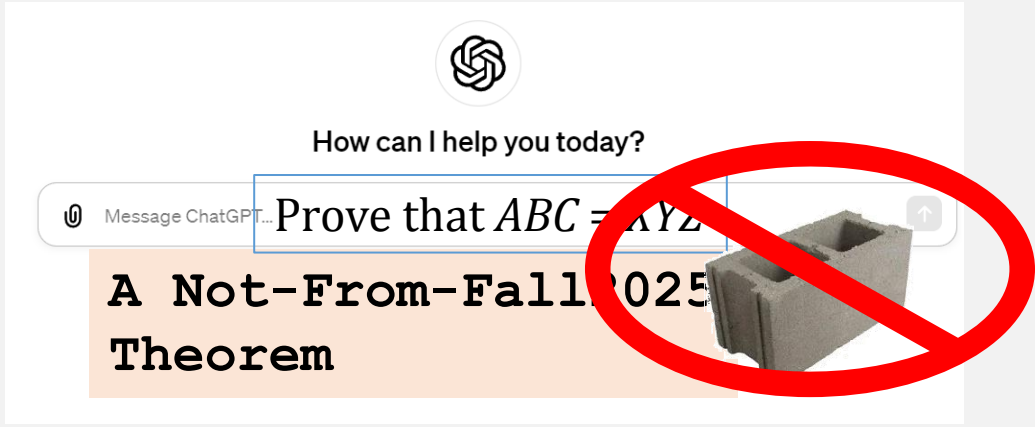
To **prove** a (new) **theorem** …

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

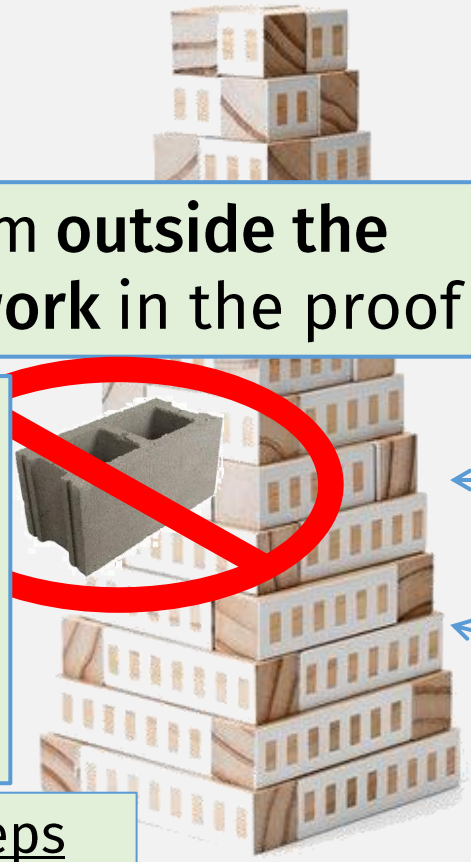
Another Word of Advice

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)

… 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!

How to Do Well in this Course

- Learn the " building blocks"
  - I.e., axioms, definitions, and theorems
- To solve a problem (prove a new theorem) …
  … think about how to (precisely) combine existing "blocks"
- HW problems graded on steps to the answer (not final theorem)
- Don't Fall Behind!
  - Start HW Early (HW 0 due Monday 9/8 12pm EST noon)
- Participate and Engage
  - Lecture
  - Office Hours
  - Message Boards (piazza)

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

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.

hw0 (pre-req quiz)

(see gradescope)