

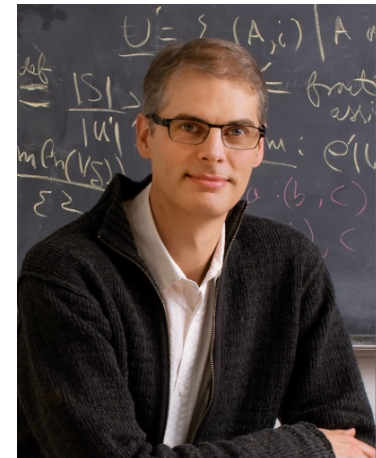
# Programming Languages and Techniques (CIS1200)

## Lecture 1

### Introduction to Program Design

# Introductions

- Swapneel Sheth
  - Levine Hall 264
  - [swapneel@cis.upenn.edu](mailto:swapneel@cis.upenn.edu)
  - <http://www.cis.upenn.edu/~swapneel>
  - Office hours: Tuesdays 10:30am – 12:30pm (& by appointment)
- Benjamin Pierce
  - Levine Hall 562
  - [bcpierce@cis.upenn.edu](mailto:bcpierce@cis.upenn.edu)
  - <http://www.cis.upenn.edu/~bcpierce>
  - Office hours: Mondays 3:30-5:00pm (& by appointment)



# Head Teaching Assistants



Claire Keller



Lauren Velegol



Katrina Liu



Mehak Dhaliwal

# What is CIS 1200?

- CIS 1200 is a course in **program design**
- Practical skills
  - ability to write larger (~1000 lines) programs
  - increased independence ("working without a recipe")
  - test-driven development, principled debugging
- Conceptual foundations
  - common data structures and algorithms
  - several different programming idioms
  - focus on modularity and compositionality
  - derived from first principles throughout
- It will be fun!



# Prerequisites

- We assume you can already write small programs (10 to 100 lines) in some imperative or object-oriented language
  - Java experience is *strongly recommended*
  - CIS 1100 or AP CS is typical
  - You should be familiar with editing code and running programs in some language
- If you're wondering whether you should be in CIS 1100 or 1200, see here:
  - <https://advising.cis.upenn.edu/skip-1100>
  - If you still have doubts, come talk to us

# CIS 1200 Tools

- OCaml
  - Industrial-strength, statically-typed *functional* programming language
  - Lightweight, approachable setting for learning about program design
  - Browser-based development tools: [codio.com](https://codio.com)
- Java
  - Industrial-strength, statically-typed *object-oriented* language
  - Many tools/libraries/resources available
  - Browser-based development or local IDE



# Why two languages??

- Clean pedagogical progression
- Everyone starts at the same place
- Practice in learning new tools
- Different perspectives on programming

“[The OCaml part of the class] was very essential to getting fundamental ideas of comp sci across. Without the second language it is easy to fall into routine and syntax lock where you don't really understand the bigger picture.”

--- CIS 1200 Student

“[OCaml] made me better understand features of Java that seemed innate to programming, which were merely abstractions and assumptions that Java made. It made me a better Java programmer.”

--- CIS 1200 Student

# Course Structure and Logistics

All course material is available on the course website

<http://www.seas.upenn.edu/~cis1200/>



# Course Grade Components

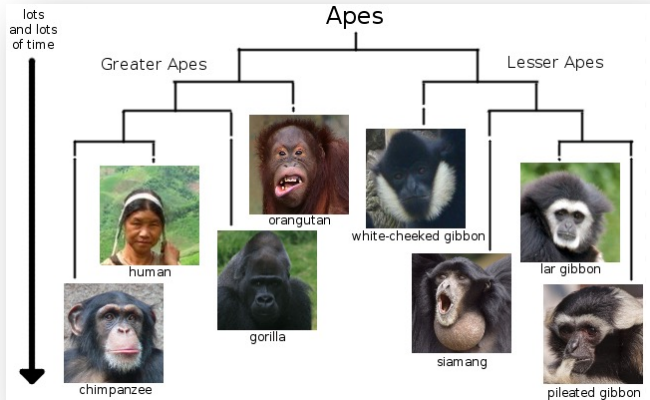
- Lectures (3% of final grade)
  - Presentation of **ideas and concepts**, interactive demos, etc.
  - Lecture notes, slides & video recordings available
  - Participation using “Poll Everywhere”
- Recitations / Labs (8% of final grade)
  - Practice and **discussion in small group setting**
  - Wed/Thurs, grade based on participation
- Homework (40% of final grade)
  - Practice with **individual problem solving**
  - Help available from course staff in office hours
  - Due Tuesdays, grade based on automated tests + style
- Exams (49% of final grade)
  - Test **foundations of program design**
  - Do you understand the terminology? Can you reason about programs? Can you synthesize solutions?
  - 2 midterms (14% each, dates on website) and a final (21%, TBA)

Warning: This will be a *challenging* and *time consuming* ( and rewarding :- ) course!

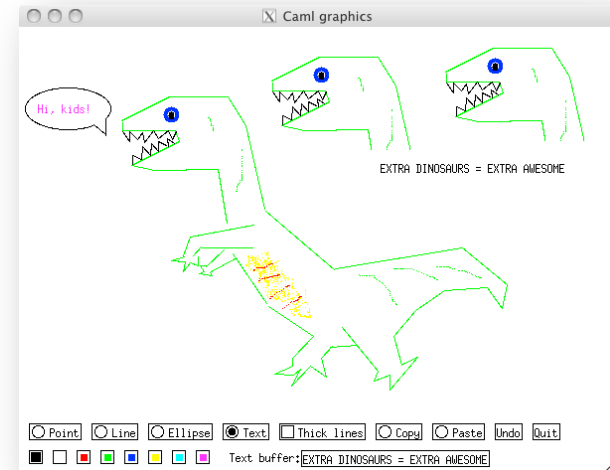
# Homework

- 9 programming assignments
- Submit assignments on the course website
  - You'll get automated grade and style feedback
  - Each assignment will have limits on the number of submission attempts
- Due at midnight (23:59pm ET) on the date announced
- Standard late policy, applies to most situations:
  - 10 points if up to 24 hours late
  - 20 points if 24-48 hours late
  - no submissions accepted after that
- In ***emergencies***, contact course staff at [cis1200@seas.upenn.edu](mailto:cis1200@seas.upenn.edu)

# Some of the homework assignments...



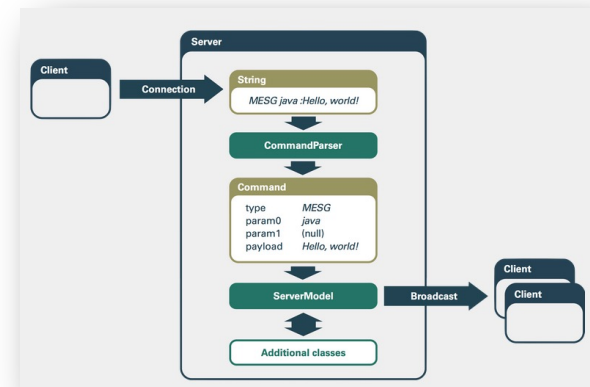
Computing with DNA



Building a GUI Framework

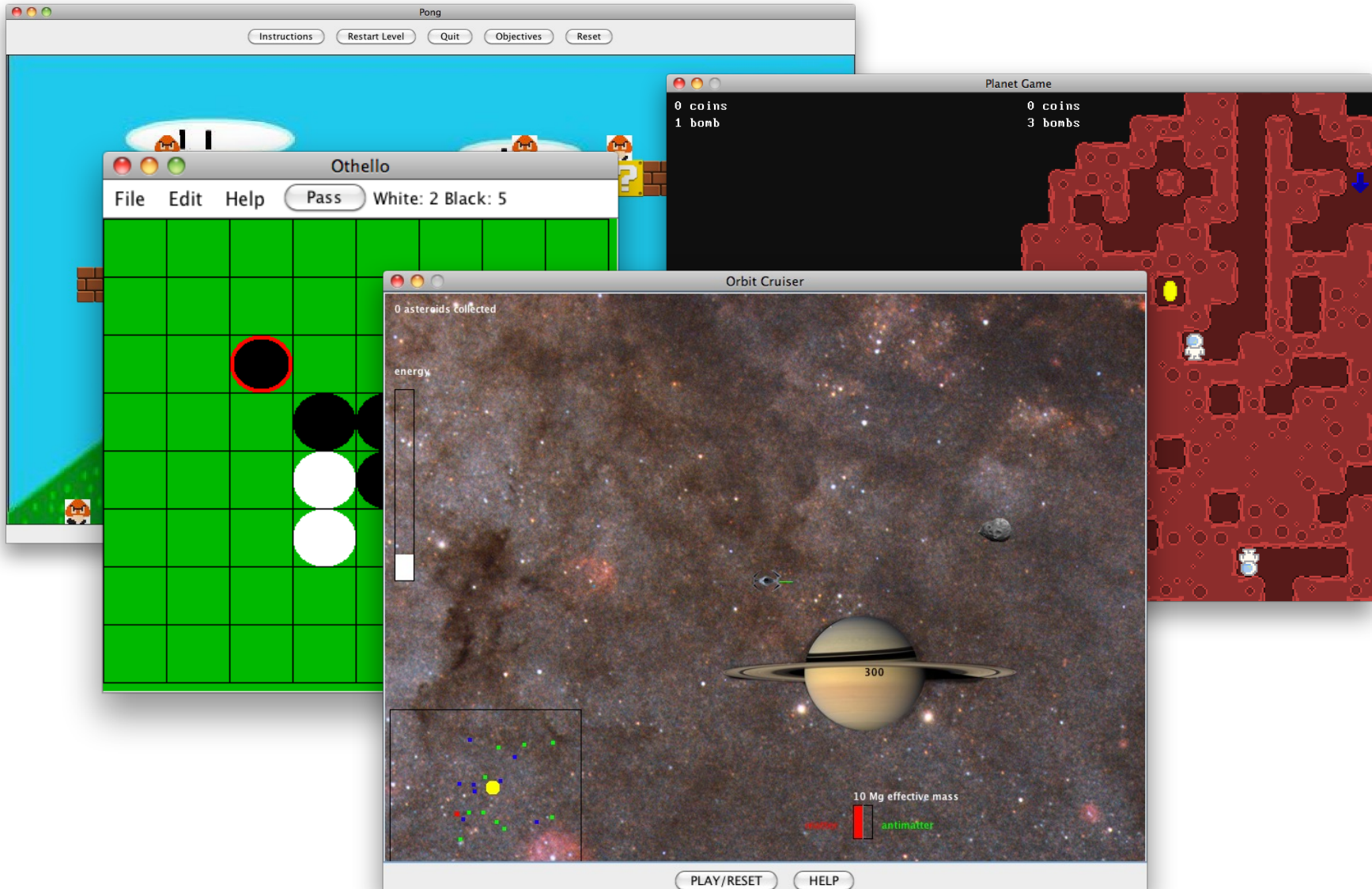


Image Processing



Chat Client/Server

# Final project: Design a Game



# Ed

- We use Ed for most communication in this course
  - from us to you
  - from you to us
  - from you to each other
- If you are already registered for the course, you should be signed up automatically
  - If not, you'll get added when you enroll
- Ed supports *anonymous* questions
  - Please check to see whether your question has already been asked; it helps us deliver higher quality responses.



Look to Ed for course announcements,  
weekly “todo” lists, reminders, etc.

# In-Class Announcements

Each lecture will also start with reminders, announcements, and a short recap

- Make sure you read the syllabus on the course website before the next class

<http://www.seas.upenn.edu/~cis1200/>

# Recitations / Lab Sections

- Recitations start *next week*
  - First meeting September 4/5
  - Room locations on Path@Penn (Moore Labs)
  - Please play a bit with the Codio platform before the first recitation (instructions will be posted)
- Goals of first recitation
  - Meet your TAs and classmates
  - Practice with OCaml before your first homework is due

# Academic Integrity

- Submitted homework must be *your individual work*
- OK (and encouraged!)
  - Discussions of concepts
  - Discussion of debugging strategies
  - Verbally sharing experience
- Not OK
  - Copying or otherwise looking at someone else's code
  - Sharing your code in any way (copy-paste, github, paper and pencil, ...)
  - Using code from a previous semester
  - ChatGPT and similar tools

*Penn's code of academic integrity:*

<https://catalog.upenn.edu/pennbook/code-of-academic-integrity/>



# Enforcement

- Course staff will check for copying
  - We will use plagiarism-detection tools on your code

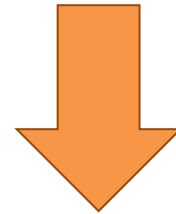
*Violations will be treated seriously!*

- *zero credit*
- *lowered course grade*
- *referral to Center for Community Standards and Accountability*

- *Questions? See the course FAQ. If in doubt, ask.*

# No Devices during Lecture

- Laptops *closed... minds open*
  - Although this is a computer science class, the use of electronic devices – laptops, phones, etc., during lecture (*except for participating in quizzes*) is *prohibited*
- Why?
  - Device users tend to surf/chat/email/game/text/tweet/etc.
  - They also distract those around them
  - Better to take notes *by hand*
  - You will get plenty of time in front of your computer while working on the homework :-)



# COVID

- **Masking is optional (but welcome)** in lectures and recitations
- If you contract COVID (or another communicable illness)
  - follow the university procedures for isolation, quarantine
  - let us know about your status
  - course lectures are recorded; material is online
  - We can make appropriate accommodations for HW, exams, recitation, participation, etc.

So much for logistics... how about some computer science?

# Program Design

# Fundamental Design Process

*Design* is the process of translating informal specifications (“word problems”) into running code

1. *Understand the problem*

What are we trying to achieve?

What are the relevant concepts and how do they relate?

2. *Formalize the *interface**

How should the program interact with its environment?

3. *Write *test cases**

How does the program behave on typical inputs?

On unusual ones? On invalid ones?

4. *Implement the required behavior*

Often by decomposing the problem into simpler ones  
and applying the same recipe to each

# A design problem

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased cost: each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180. What profit do you make at any given price?

# Step 1: Understand the problem

## What are we trying to achieve?

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased cost; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180.

What profit do you make at any given price?



# Step 1: Understand the problem

## What are we trying to achieve?

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased cost; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180.

**What profit do you make at any given price?**

# Step 1: Understand the problem

## What are we trying to achieve?

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased cost; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180.

What profit do you make at any given price?

Calculate profit as a function of ticket price

# Step 1: Understand the problem

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased cost; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180. What profit do you make at any given price?

# Step 1: Understand the problem

## What are the relevant concepts?

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased cost; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180.

What profit do you make at any given price?

# Step 1: Understand the problem

## What are the relevant concepts?

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the **price** of a ticket and average **attendance**. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased **cost**; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180. What **profit** do you make at any given price?

# Step 1: Understand the problem

What are the relationships among them?

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the **price** of a ticket and average **attendance**. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. However, increased attendance also comes at increased **cost**; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180.

What **profit** do you make at any given price?

$\text{profit} = \text{revenue} - \text{cost}$

$\text{revenue} = \text{price} * \text{attendees}$

$\text{cost} = \$180 + \text{attendees} * \$0.04$

$\text{attendees} = \text{some function of the ticket price}$

## Step 2: Formalize the Interface

*Goal: write a function that returns the profit when given the price*

*Idea: we'll represent money in cents, using integers*

comment documents  
the design decision

type annotations  
declare the input  
and output types



```
(* Money is represented in cents. *)  
let profit (price : int) : int = ...
```

The diagram shows two blue arrows. One arrow points from the text 'comment documents the design decision' to the green comment line in the code. The other arrow points from the text 'type annotations declare the input and output types' to the red type annotations in the code.

N.b. Floating point is generally a *bad* choice for representing money: bankers use different rounding conventions than the IEEE floating point standard, and floating point arithmetic isn't as exact as you might like. Try calculating  $0.1 + 0.1 + 0.1$  sometime in your favorite programming language...

N.b. OCaml will let you omit these type annotations, but including them is *mandatory* for CIS1200. Using type annotations is good documentation; they also improve the error messages you get from the compiler. When you get a type error message from the compiler, the first thing you should do is check that your type annotations are correct.

## Step 3: Write test cases

The design problem gives us an easy way to calculate the expected result for one specific test case:

```
let profit_500 : int =  
  let price      = 500 in  
  let attendees  = 120 in  
  let revenue    = price * attendees in  
  let cost       = 18000 + 4 * attendees in  
  revenue - cost
```



# Write test cases

With a little thinking, we pretty easily write down another test case:

```
let profit_490 : int =  
  let price      = 500 - 10 in  
  let attendees  = 120 + 15 in  
  let revenue    = price * attendees in  
  let cost       = 18000 + 4 * attendees in  
  revenue - cost
```

Recall: “Decreasing the price by a dime (\$.10) increases attendance by 15”

# Add the Test Cases to the Program

Record the test cases as *assertions*:

- the command `run_test` executes a test

a *test* is just a function that takes no input and returns true if the test succeeds

```
let test500p () : bool =  
    (profit 500) = profit_500  
  
;; run_test "profit at $5.00" test500p
```

double semicolons  
mark top-level commands

the string in quotes identifies  
the test in printed output  
(in case it fails)

## Step 4: Implement the Behavior

**profit**, **revenue**, and **cost** are easy to define:

```
let attendees (price : int) : int = ...write it later...
```

```
let revenue (price : int) : int =  
  price * (attendees price)
```

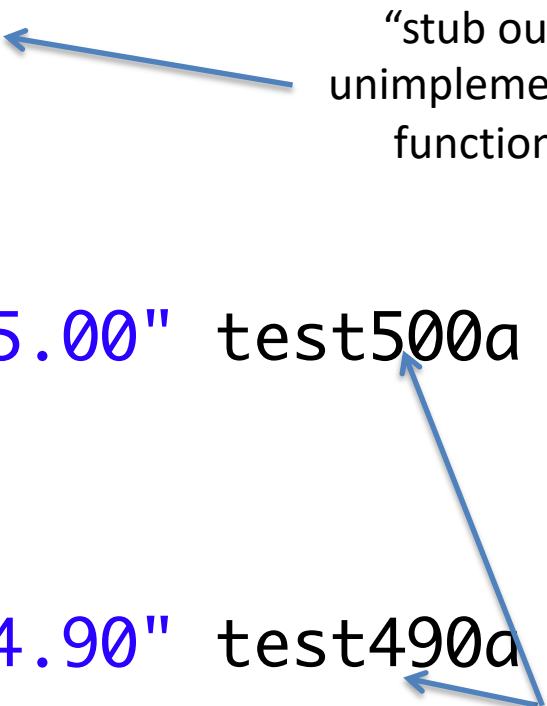
```
let cost (price : int) : int =  
  18000 + (attendees price) * 4
```

```
let profit (price : int) : int =  
  (revenue price) - (cost price)
```

# Apply the Design Pattern Recursively

attendees requires a bit of thought. Start with tests...

```
let attendees (price : int) : int =  
  failwith "unimplemented"  
  
let test500a () : bool =  
  (attendees 500) = 120  
;; run_test "attendees at $5.00" test500a  
  
let test490a () : bool =  
  (attendees 490) = 135  
;; run_test "attendees at $4.90" test490a
```

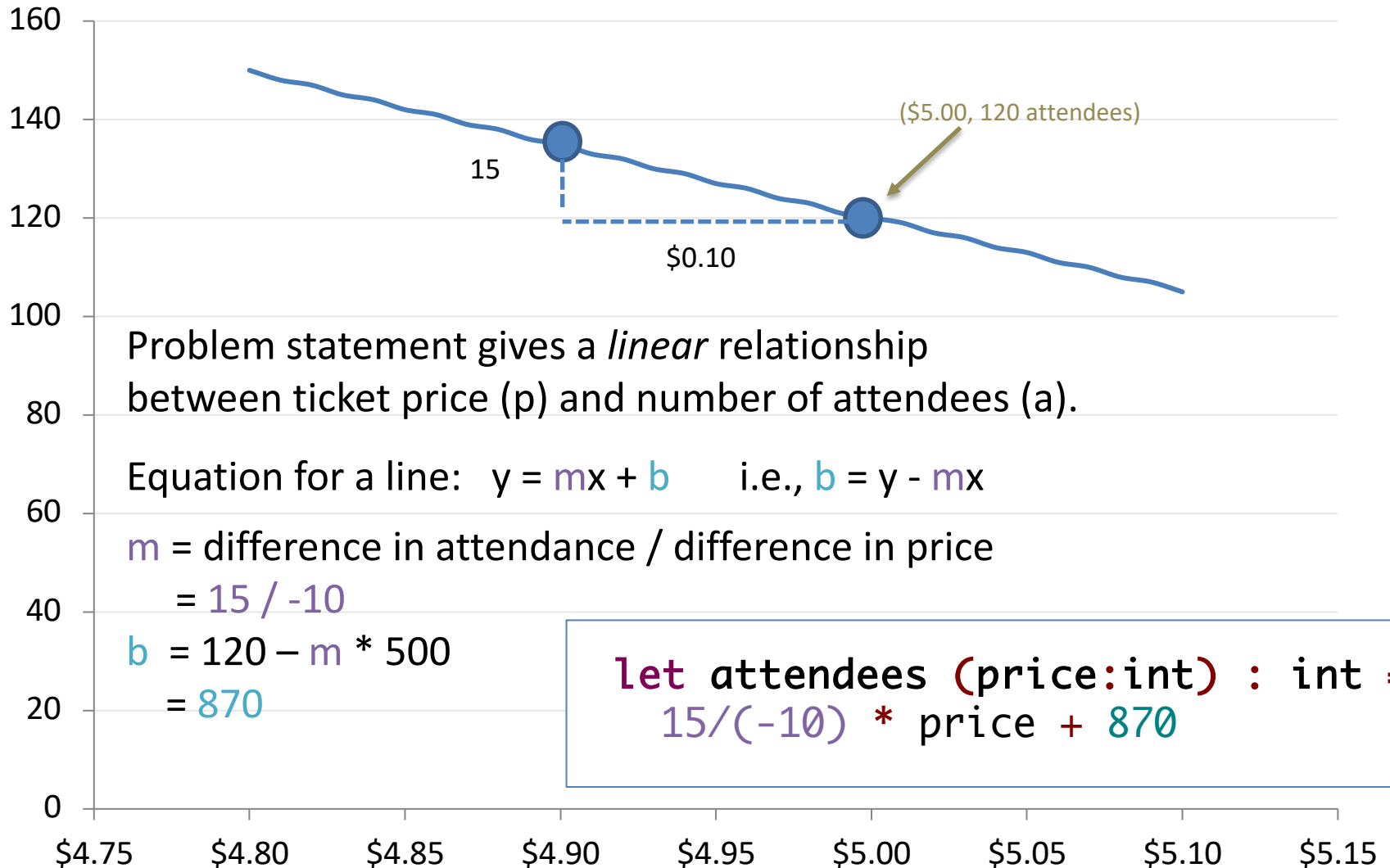


"stub out"  
unimplemented  
functions

create the tests  
from the problem  
statement *first*.

\*Note that the definition of attendees must go *before* the definition of profit because profit uses attendees.

# Attendees vs. Ticket Price







Run it!

CodioProjectFileEditFindViewToolsEducationHelpRun Project

Filetree x




SSHETH  
LectureDemos-22fa



LectureDemos-22fa (master)

- build
- full
- stub
- .codio
- .gitignore
- .merlin
- .ocamlinit
- .settings
- assert.ml
- assert.mli
- Makefile
- README.md
- tickets.ml
- tickets.native

README.mdtickets.mlRun Project x

OCAMLRUNPARAM=b ./tickets.native  
Running: profit\_500 ... Test passed!  
Running: profit at \$5.00 ...  
Test failed: profit at \$5.00   
  
Running: attendees @ 500 ...  
Test failed: attendees @ 500   
  
Running: attendees at \$4.90 ...  
Test failed: attendees at \$4.90   
  
codio@comradematrix-frogeducate:~/workspace\$

# Uh Oh...

The test cases for attendees failed!

Why?

```
let attendees (price:int) : int =  
    15/(-10) * price + 870
```



# Uh Oh...

The test cases for attendees failed!

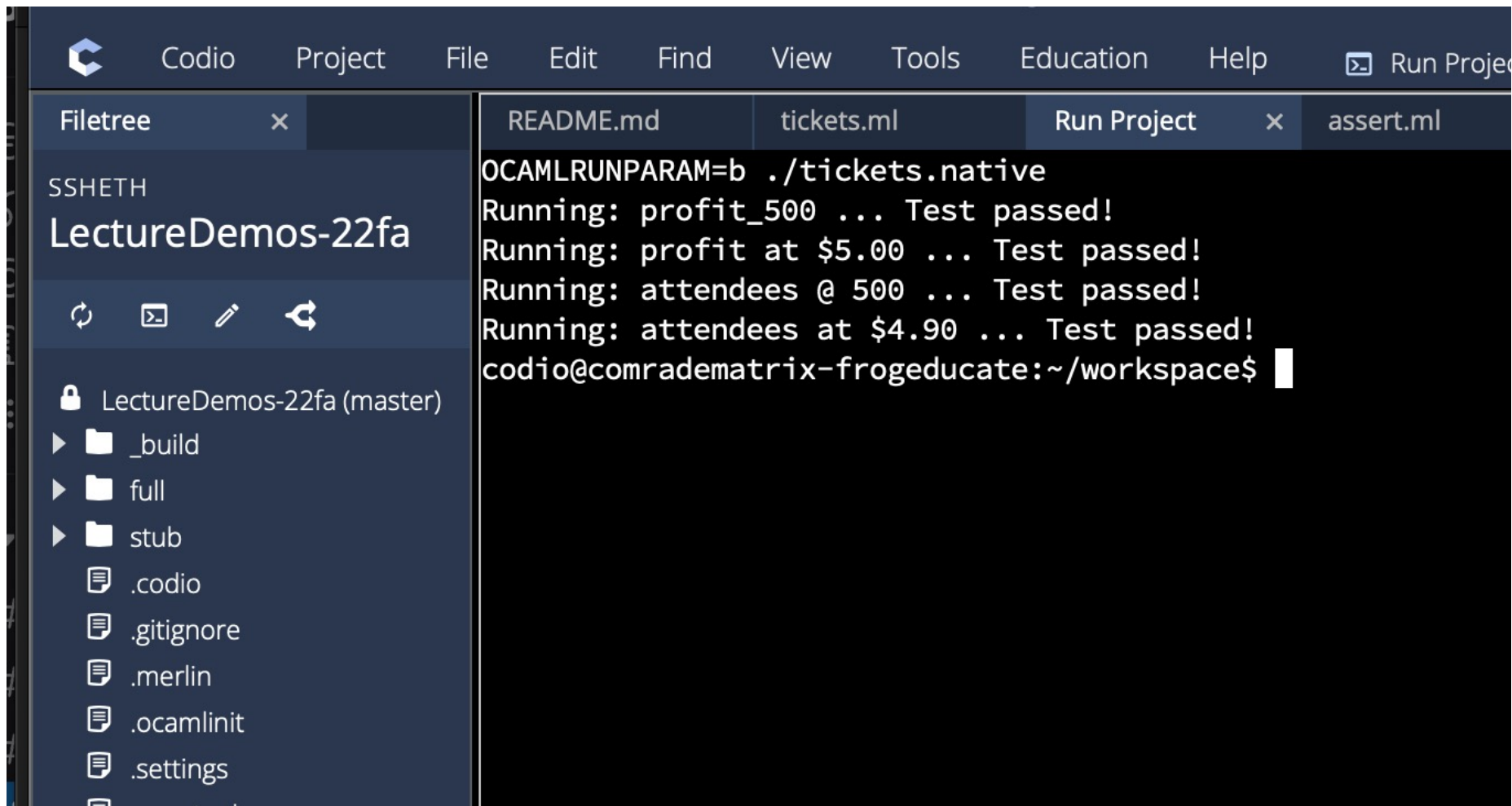
The issue here is that integer division is imprecise

- Using integer arithmetic,  $15 / -10$  evaluates to  $-1$ , since  $-1.5$  rounds to  $-1$

Here is a better\* version:

```
let attendees (price:int) : int =  
    (15 * price) / (-10) + 870
```

\* Multiplying `15*price` before dividing by `-10` increases the precision because rounding errors don't creep in.



# How *Not* to Solve This Problem

*This program also passes all our tests...*

```
let profit price =  
  price * (15 * price / (-10) + 870) -  
  (18000 + 4 * (15 * price / (-10) + 870))
```

Nevertheless, it is bad because it...

- hides the structure and abstractions of the problem
- duplicates code that could be shared
- doesn't document its interface via types and comments

# Summary (I)

CIS1200 promotes a *structured design process*:

1. Understand the problem
2. Formalize the interface
3. Write test cases
4. Implement the desired behavior

# Summary (II)

Modern software development relies heavily on *test-driven development* in *strongly typed languages*

- Write tests early in the programming process and use them to drive the rest of the process

*Types help structure the code.*  
*Tests help get the details right.*

For CIS 1200 homework projects:

- We will provide tests for each part of the project
- They will generally not be complete
- You should *start* each part by making up *more* tests

# What's Next?

Date	Topic	Slides	Code	Reading
<b>Week 1 Videos</b>				
Wed 8/28	Introductions, Program Design	<a href="#">lec01.pdf</a>	<a href="#">assert.ml</a> <a href="#">tickets.ml</a>	<a href="#">Chapter 1</a>
Fri 8/30	Value-Oriented Programming			
<b>Week 2 Videos</b>				
Mon 9/2	<i>No Class: Labor Day</i>			
Wed 9/4	Functions, Lists and Recursion			
Fri 9/6	Lists, Tuples, Nested Patterns			
<b>Week 3 Videos</b>				
Mon 9/9	Datatypes and Trees			
Tue 9/10	HW01 (Finger Exercises) due			