Programming Languages and Techniques (CIS120)

Lecture 1

Introduction to Program Design

Program Design

Fundamental Design Process

Design is the process of translating informal specifications ("word problems") into running code

1. *Understand* the problem
   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

5. Revise / Refactor / Edit

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.

At what price do you make the highest profit?

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.

At what price do you make the highest profit?

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.

At what price do you make the highest profit?

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.

At what price do you make the highest profit?

profit = revenue – cost
revenue = price * attendees
cost = $180 + attendees * $0.04
attendees = *some function of the ticket price*

Step 1: Understand the problem

What are you 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.

At what price do you make the highest profit?

Determine how profit depends on the ticket price, which will allow you to maximize profit by changing the price

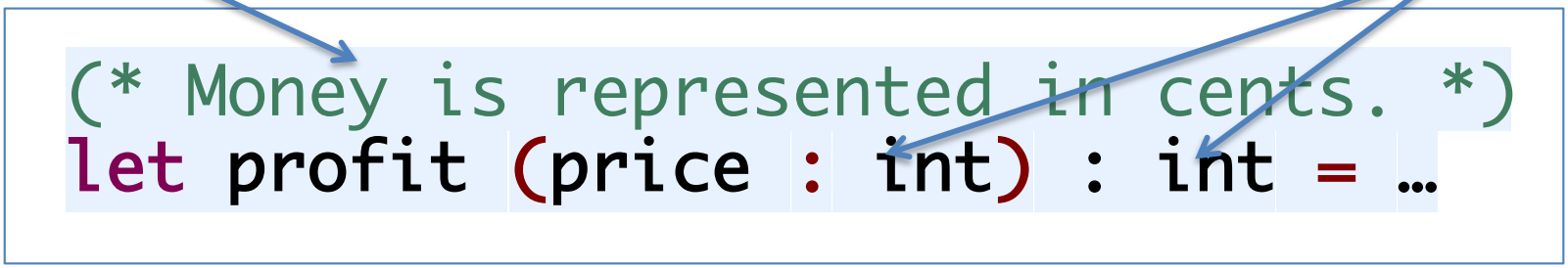
Step 2: Formalize the Interface

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 = …
```

* 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…

**OCaml will let you omit these type annotations, but including them is *mandatory* for CIS120. 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 there and that they are what you expect.

Step 3: Write test cases

- By looking at the design problem, we can calculate specific test cases

```
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
```

Writing the Test Cases in OCaml

- Record the test cases as assertions in the program:
  - 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 test () : bool =
   (profit 500) = profit_500

;; run_test "profit at $5.00" test
```

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

note the use of double semicolons before top level commands

Step 4: Implement the Behavior

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

```
let attendees (price : int) : int = ...


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

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

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

Apply the Design Pattern Recursively

`attendees` requires a bit of thought:

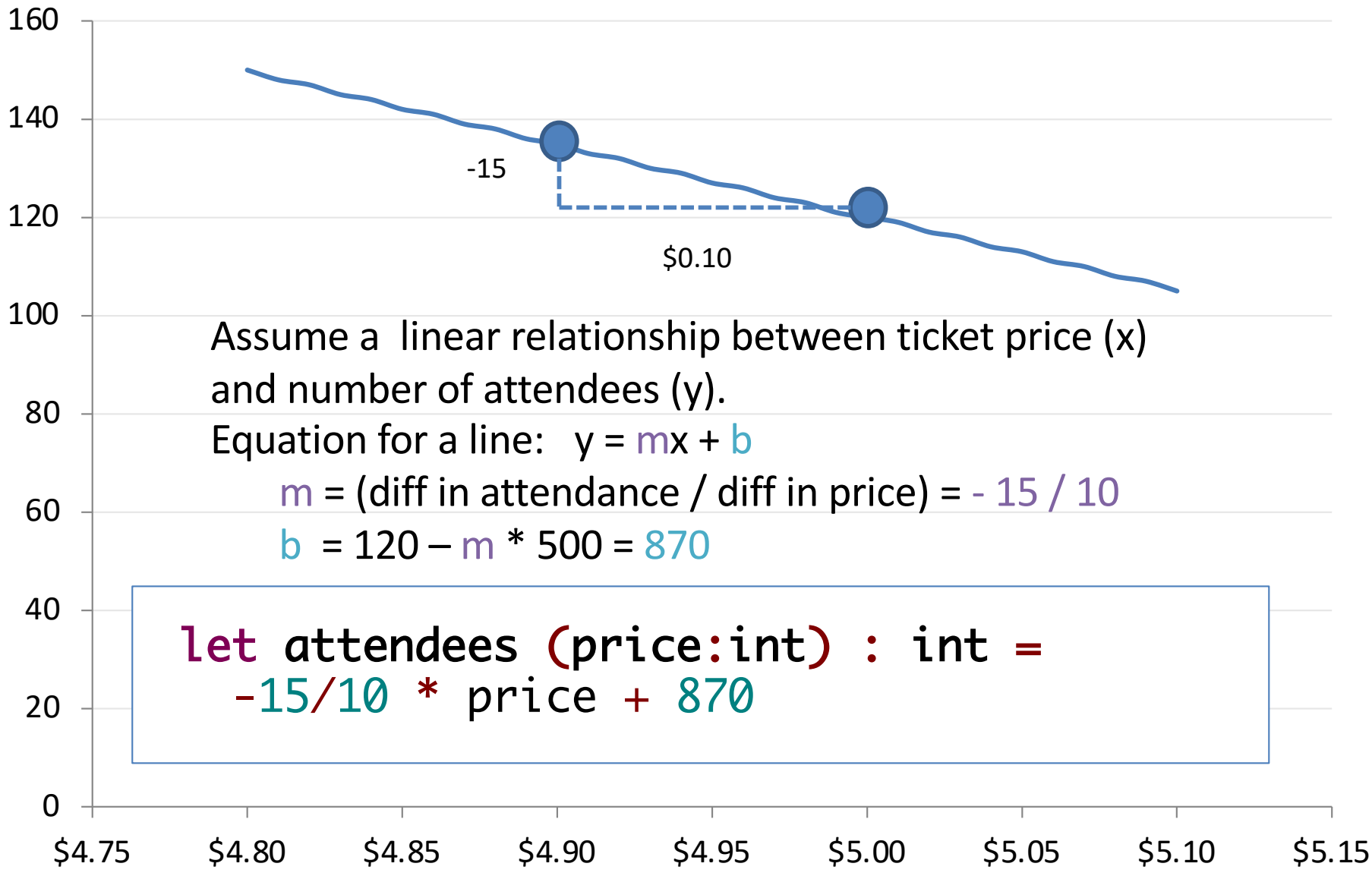
```
let attendees (price : int) : int =
   failwith "unimplemented"

let test () : bool =
   (attendees 500) = 120
;; run_test "attendees at $5.00" test

let test () : bool =
   (attendees 490) = 135
;; run_test "attendees at $4.90" test
```

"stub out" unimplemented functions

generate the tests from the problem statement *first*.

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

Attendees vs. Ticket Price



Assume a linear relationship between ticket price (x) and number of attendees (y).

Equation for a line: y = mx + b

m = (diff in attendance / diff in price) = - 15 / 10

b = 120 – m * 500 = 870

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

Run!

Run the program!

- Our test cases for attendees failed…
- Debugging reveals that integer division is tricky*

Here is the fixed version:

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

*Using integer arithmetic, -15 / 10 evaluates to -1, since -1.5 rounds to -1. Multiplying `-15*price` before dividing by 10 increases the precision because rounding errors don't creep in.

Using Tests

Modern approaches to software engineering advocate *test-driven development*, where tests are written very early in the programming process and used to drive the rest of the process.

We are big believers in this philosophy, and we'll be using it throughout the course.

In the homework template, we may provide one or more tests for each of the problems. They will generally not be complete. You should *start* each problem by making up *more* tests.

How *not* to Solve this Problem

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

This program is bad because it

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

Note that it still *passes all the tests!*

Summary

- CIS120 promotes an iterative *design process*
  1. Understand the problem
  2. Formalize the interface
  3. Write test cases
  4. Implement the desired behavior

- Test early!
  - Helps clarify interfaces and understanding
  - Helps when debugging
  - Useful in the long term too