

# Programming Languages and Techniques (CIS120)

Lecture 2

Jan 13, 2012

Program Design  
OCaml Basics

# Announcements

- If you are joining us today:
  - See Wed's slides/lecture notes on course website  
<http://www.seas.upenn.edu/~cis120/>
  - Sign yourself up for Piazza  
<http://www.piazza.com/>
  - Go through the first lab materials on website
  - If you can't get into the lab you want, send email  
[sweirich@cis.upenn.edu](mailto:sweirich@cis.upenn.edu)
  - No laptops during lecture

# Announcements

- Homework 1: OCaml Finger Exercises
  - practice using OCaml for simple programs
  - Due: Monday, Jan. 23<sup>rd</sup> at 11:59:59pm (midnight)
  - Start early! It takes 2-20 hours to finish the assignment
- Please *read* Chapters 1 & 2 of the course notes, which is available from the course web pages.
  - It covers course introduction, design recipe and introductory OCaml syntax and programming
  - The concepts needed for the first parts of HW 1 are covered
  - We'll add more material as the course goes on...

# Design

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 erroneous 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 the owner of a movie theater who has complete freedom in setting ticket prices. The more he charges, the fewer people can afford tickets. In a recent experiment the owner determined a precise 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. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner \$180. Each attendee costs another four cents (\$0.04). The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.

(Interactive Interlude)

# Step 1: Understand the problem

- In this problem there are five relevant concepts:
    - *(ticket) price*
    - *attendees*
    - *revenue*
    - *cost*
    - *profit*
  - There are relationships among them:
    - $\text{profit} = \text{revenue} - \text{cost}$
    - $\text{revenue} = \text{price} * \text{attendees}$
    - $\text{cost} = \$180 + \text{attendees} * \$0.04$
    - *attendees = some function of the ticket price*
  - Goal is to determine profit, given the ticket price
- So profit, revenue and cost also depend on price.
- 

## Step 2: Formalize the Interface

- Here, there is only one (mildly) interesting choice:  
*How should we represent money?*
  - option 1: integers
  - option 2: floating point numbers
- Either could work\*
  - for simplicity, we choose integers

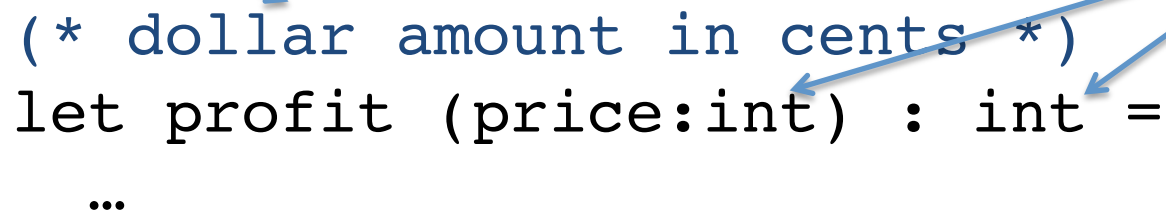
\* Floating point is generally a *bad* choice for representing money: bankers use different rounding conventions than the IEEE floating point standard, and floating point arithmetic isn't as exact as you might like. Try calculating  $0.1 + 0.1 + 0.1$  sometime...



# Formalizing the Interface in OCaml

comment documents  
the design decision

type annotations  
enforce the interface\*



```
(* dollar amount in cents *)  
let profit (price:int) : int =  
  ...
```

The code snippet is enclosed in a blue-bordered box. Two blue arrows point from the text above to the code: one from 'comment documents the design decision' to the opening parenthesis of the comment, and another from 'type annotations enforce the interface\*' to the type annotations '(price:int) : int'.

\*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 informal specification, we can calculate specific test cases

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

## Step 3: Write Test Cases

- By looking at the data from the informal specification, we can calculate\* these tests:
  - profit at \$5.00 is \$415.20
  - profit at \$4.90 is \$476.10
- Working out tests by hand also helps nail-down corner cases and can help you understand the problem better.

\*TIP: The OCaml interactive top level loop can be used as a calculator and to play around with definitions while you're understanding the program and the test cases. You should record the tests you develop as assertions so that they can be run again later when the program changes.

# 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_five_dollars  
;; run_test "profit at $5.00" test  
  
let test () : bool =  
    (profit 490) = 47610  
;; run_test "profit at $4.90" test
```

note the very stylized use of double semicolons for commands

the string identifies the test in printed output

## Step 4: Implement the Behavior

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

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

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

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

# Apply the Design Pattern Recursively

attendees\* requires a bit of thought:

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

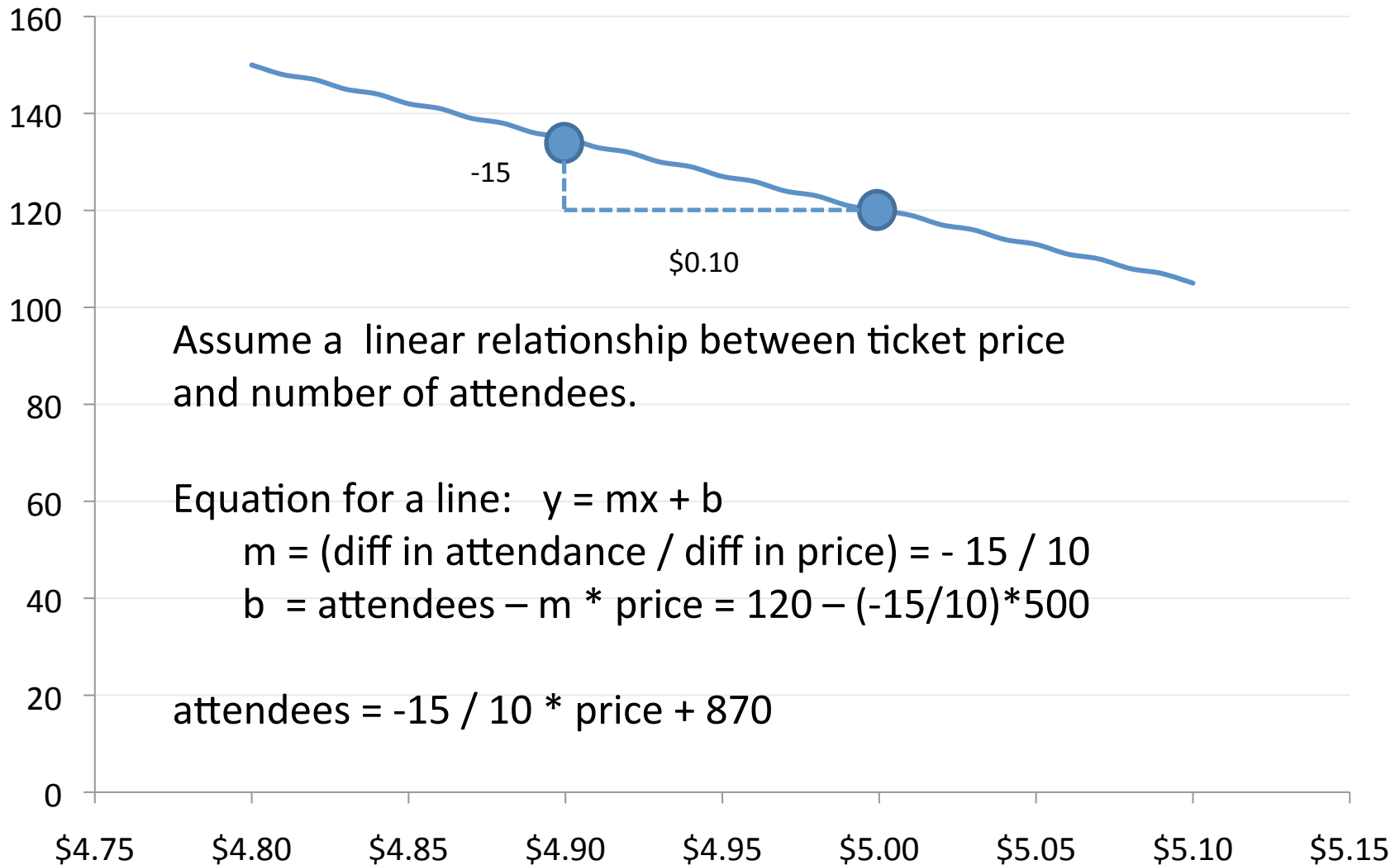
```
let test () : bool =  
    (attendees 500) = 120  
;; run_test "atts. at $5.00" test
```

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

\*Note that the definition of attendees must go *before* the definitions of cost and revenue because the latter make use of the attendees function. Similarly, cost and revenue must be defined before profit.

generate the tests  
from the problem  
statement *first*.

# Attendees vs. Ticket Price



# Run the program!

- One of 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  $-15 * \text{price}$  before dividing by  $10$  increases the precision because rounding errors don't creep in.



# 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 this program still passes all the tests!*

# Evolving/Refactoring Code

- For this simple problem, this design methodology may seem like overkill.
  - The real benefits are to be had in bigger programs
  - But, even *small* programs evolve over time
- Suppose that, based on the problem description, you decided to define cost in terms of the number of attendees directly, rather than calling the attendees from within cost.
  - How do our tools and this design methodology help?

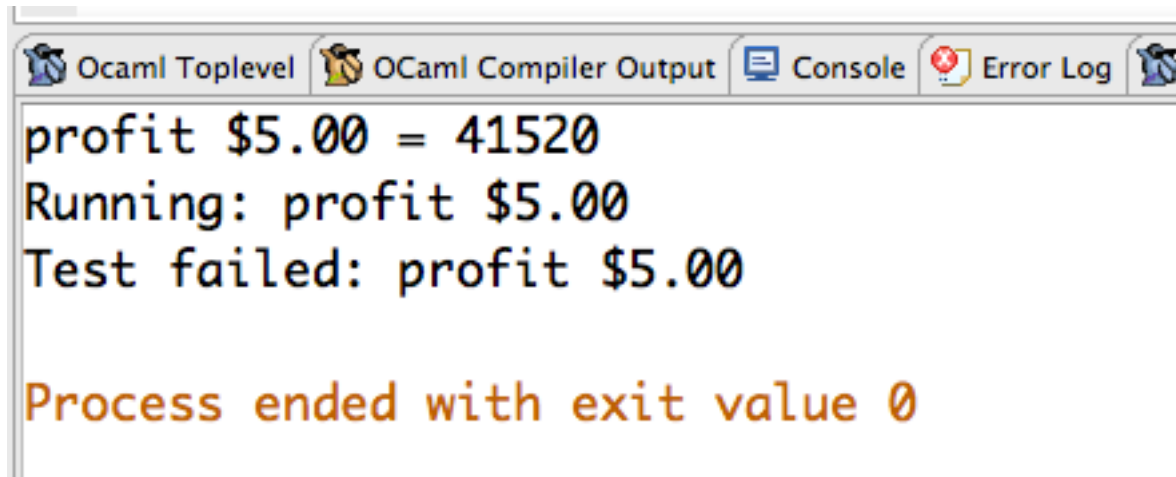
## Example Refactoring: Change 'cost'

cost is simplified:

```
(* atts is the number of attendees *)  
let cost (atts:int) : int =  
    18000 + 4 * atts
```

... but suppose we forget to change profit, which calls cost. (As might easily happen in a big program.)

# Test Case for Profit Fails



The screenshot shows a window titled "Ocaml Toplevel" with tabs for "Ocaml Compiler Output", "Console", and "Error Log". The console output displays the following text:

```
profit $5.00 = 41520  
Running: profit $5.00  
Test failed: profit $5.00  
  
Process ended with exit value 0
```

We need to fix profit like this:

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

# 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've provided one or more tests for each of the problems. You should *start* each problem by making up some more tests.

# Essential OCaml Cheatsheet

See also Chapter 2 of the CIS 120 lecture notes available from the web pages.

# Caveat

Many people find programming in OCaml a little disorienting at first. The syntax is unfamiliar, but more importantly OCaml embodies a *value-oriented* programming style that takes a little while to get used to.

For the moment, we ask you to trust that this is all going to feel much more natural in a couple of weeks and enjoy the challenge of learning to think about programming a little differently.

# Primitive Types and Constants

OCaml's built-in primitive types include...

- `int`

0, 1, 42, -1, 999

- `string`

"hello world"

- `float`

3.14159, 0.123

- `bool`

true, false



# Operators

## Comparisons:

=	equality
<>	inequality
<	less than
>=	greater than or equal

(these can be used with any type of data – numbers, strings, characters, etc.)

## Boolean (logical) operators:

not	logical negation
&&	and
	or

## String operators:

^	string concatenation
---	----------------------

# Expressions

Numeric expressions (ints):

$1 + 2$	addition
$1 - 2$	subtraction
$2 * 3$	multiplication
$10 / 3$	integer division
$10 \text{ mod } 3$	modulus (remainder)

From constants and operations, we can build bigger expressions:

$$(1 + 2 * (10 \text{ mod } 4)) / 4$$

# Value-Oriented Programming

- We run OCaml programs by *calculating* expressions to values:

`3 ⇒ 3`                    values compute to themselves

`3 + 4 ⇒ 7`

`2 * (4 + 5) ⇒ 18`

`true && (false || true) ⇒ true`

The notation `<exp> ⇒ <val>` means that the expression `<exp>` computes to the value `<val>`.

Note: the symbol '`⇒`' is *not* OCaml syntax. It's a convenient way to *talk* about OCaml syntax.

# Step-wise Calculation

- We can understand  $\Rightarrow$  in terms of single step calculations written ' $\mapsto$ '
  - Single step calculations do “the expected thing” for primitive operations
- For example:

$$(2+3) * (5-2)$$

$$\mapsto 5 * (5-2)$$

because  $2+3 \mapsto 5$

$$\mapsto 5 * 3$$

because  $5-2 \mapsto 3$

$$\mapsto 15$$

because  $5*3 \mapsto 15$

# Conditional Expressions

```
if s = "positive" then 1 else -1
```

```
if day >= 6 && day <= 7  
then "weekend" else "weekday"
```

OCaml conditionals are *expressions*: they can be used inside of other expressions:

```
(if 3 > 0 then 2 else -1) * 100
```

```
if x > y then "x is bigger"  
else if x < y then "y is bigger"  
else "same"
```

# Running Conditional Expressions

- A conditional expression yields the value of either its ‘then’-branch expression or its ‘else’-branch expression, depending on whether the test is ‘true’ or ‘false’.

- For example:

`(if 3 > 0 then 2 else -1) * 100`

$\mapsto$  `(if true then 2 else -1) * 100`

$\mapsto$  `2 * 100`

$\mapsto$  `200`

- Note: this means that it’s not sensible to leave out the ‘else’ branch. (What would be the result if the test was ‘false’?)