# Programming Languages and Techniques (CIS120)

Lecture 2

Value-Oriented Programming

# If you are joining us today…

- Read the course syllabus/lecture notes on the website
  - www.cis.upenn.edu/~cis120
- Sign yourself up for Piazza
  - piazza.com/upenn/fall2019/cis120
- Try out Codio
  - www.cis.upenn.edu/~cis120/current/codio.shtml
- Sign yourself up for Poll Everywhere
  - Details are in Piazza

- If you aren't registered for the class yet, please fill out the waitlist form (on the web pages)
- If you are registered, but want to switch the lecture/recitation section, please fill out the "Switch Recitations or Lectures" Form

- *No laptops, tablets, smart phones, etc., during lecture* (except for participating in PollEverywhere quizzes)

# Announcements

- Please *read:*
  - Chapter 2 of the course notes
  - OCaml style guide on the course website (http://www.seas.upenn.edu/~cis120/current/programming_style.shtml)
- Homework 1: OCaml Finger Exercises
  - Available from Schedule page on course website
  - Practice using OCaml to write simple programs
  - Due: *September 10th, at 11:59:59pm* (midnight)
  - Start early!
  - Start with first 4 problems (lists will be introduced next week!)

# Homework Policies

- Projects will be (mostly) automatically graded with immediate feedback
  - We'll give you some tests, as part of the assignment
  - You'll write your own tests to supplement these
  - Our grading script will apply *additional* tests
  - Your score is based on how many of these you pass
  - Your code must compile to get *any* credit
- Multiple submissions *are allowed*
  - First few submissions: no penalty
  - Each submission after the first few will be penalized
  - Your final grade is determined by the *best* raw score
- Late Policy
  - Submission up to 24 hours late costs 10 points
  - Submission 24-48 hours late costs 20 points
  - After 48 hours, no submissions allowed
- Style / Test cases:
  - manual grading of non-testable properties
  - feedback on style from your TAs

# Important Dates

- Homework:
  - Homework due dates will be listed on course calendar
  - Tuesdays at midnight: see schedule on web

- Exams:
  - 12% First midterm: Friday, September 27th in class
  - 12% Second midterm: Friday, November 8th in class
  - 18% Final exam:
    (Tentatively) Tuesday, December 17th at 6:00PM
  - Contact instructor *well in advance* if you have a conflict

  - Make-up Exam Times will be announced beforehand

# Where to ask questions

- Course material
  - **Piazza Discussion Boards**
  - TA office hours, on webpage calendar
  - Tutoring
  - Prof office hours:

    Sheth……Tuesdays 10:30am – 12:30pm
    Zdancewic…..Mondays 3:30 – 5:00pm
    or by appointment (changes will be announced on Piazza)
- HW/Exam Grading: see webpage
- About CIS majors & Registration
  - Desirae Cesar or Laura Fox, Levine 309
    CIS Undergraduate coordinators

Poll Everywhere

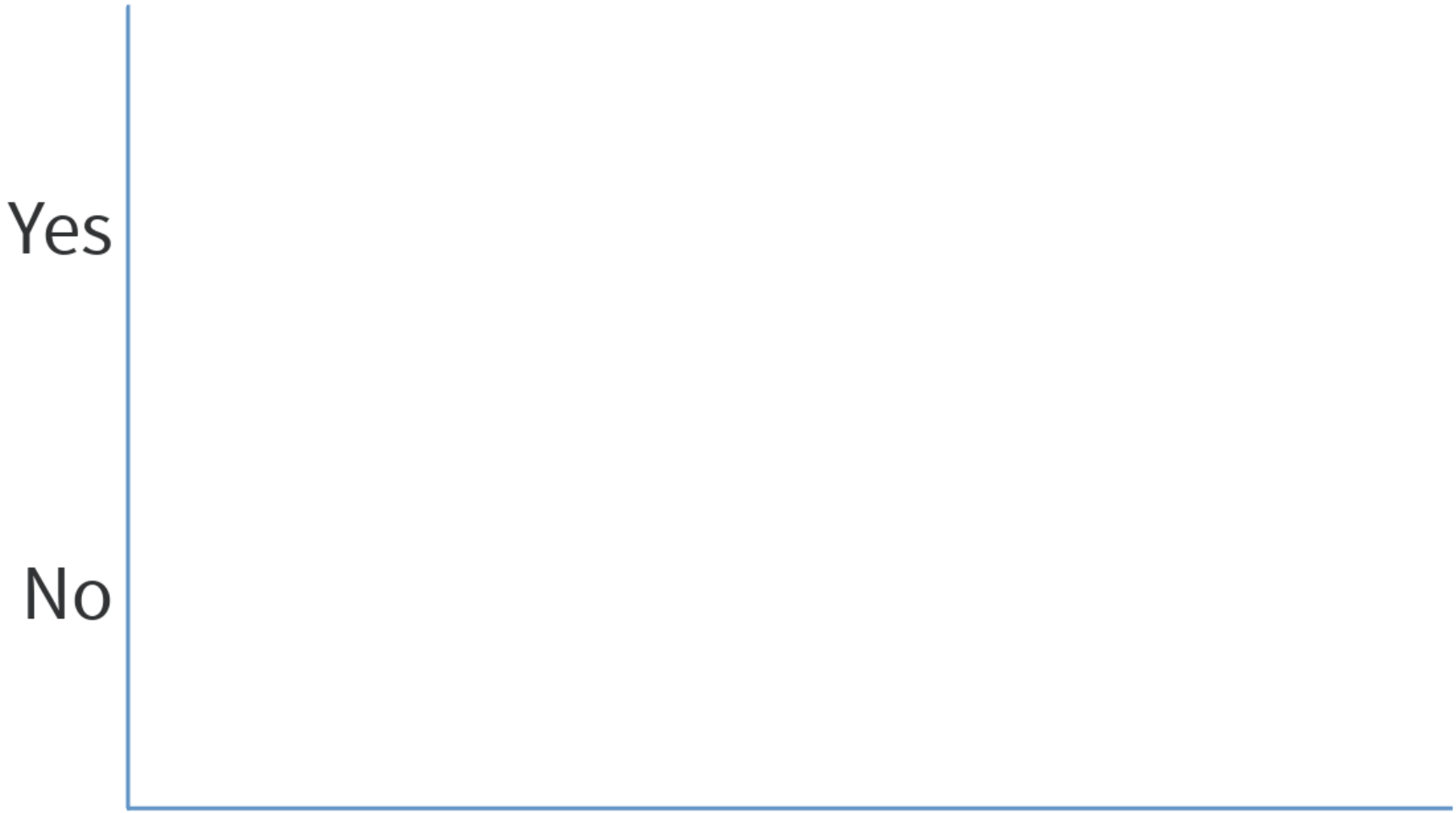
# Poll Everywhere Basics

- Beginning today, we'll use Poll Everywhere in each lecture
  - Grade recording starts *Monday 9/9*
- You can use your phone, laptop, etc. to go the website.
- You can also use your phone to text directly

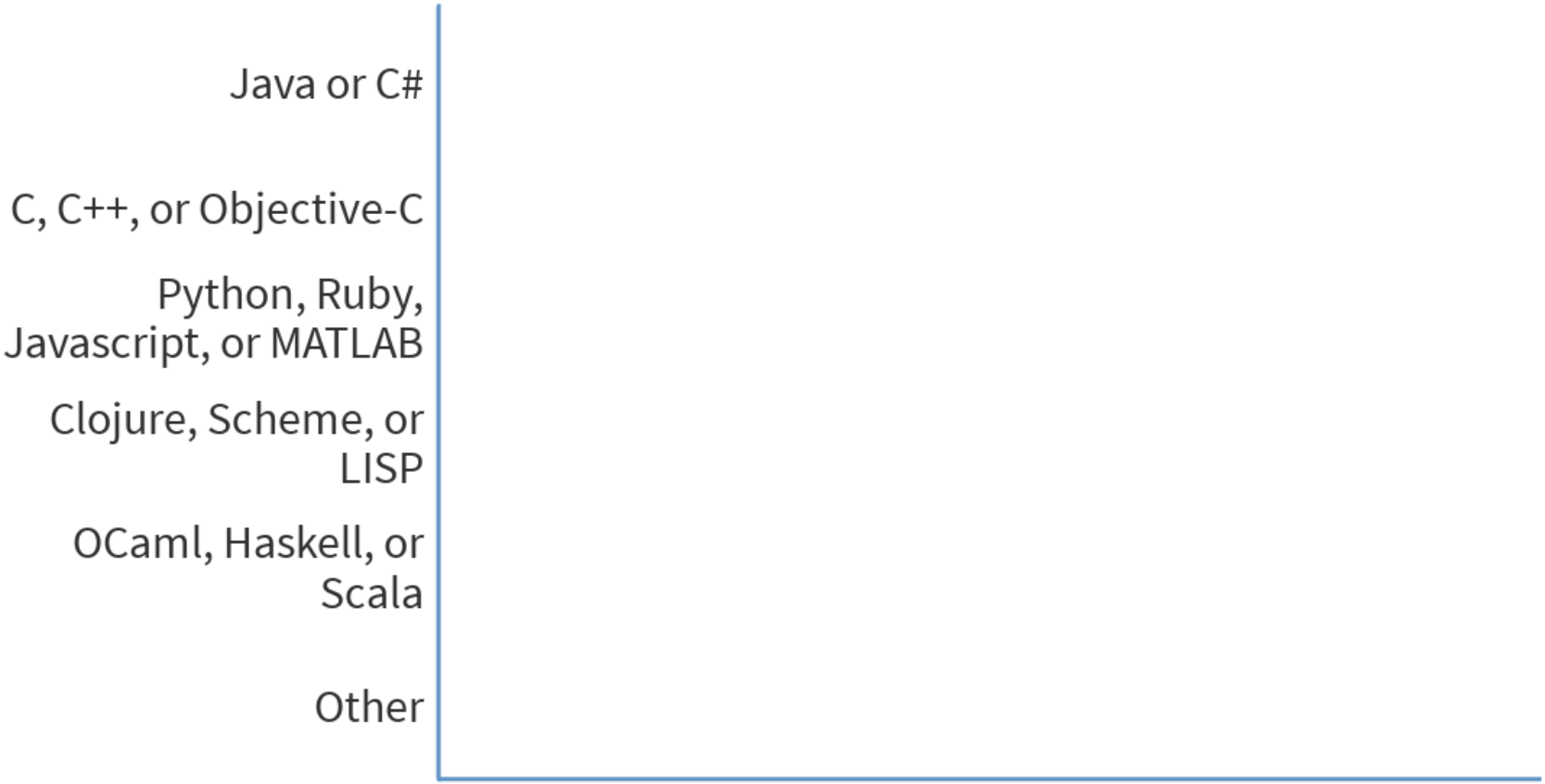
- Polls will be restricted to registered participants
- Register with your Penn Email Address if you haven't already

https://pollev.com/penncis120/register

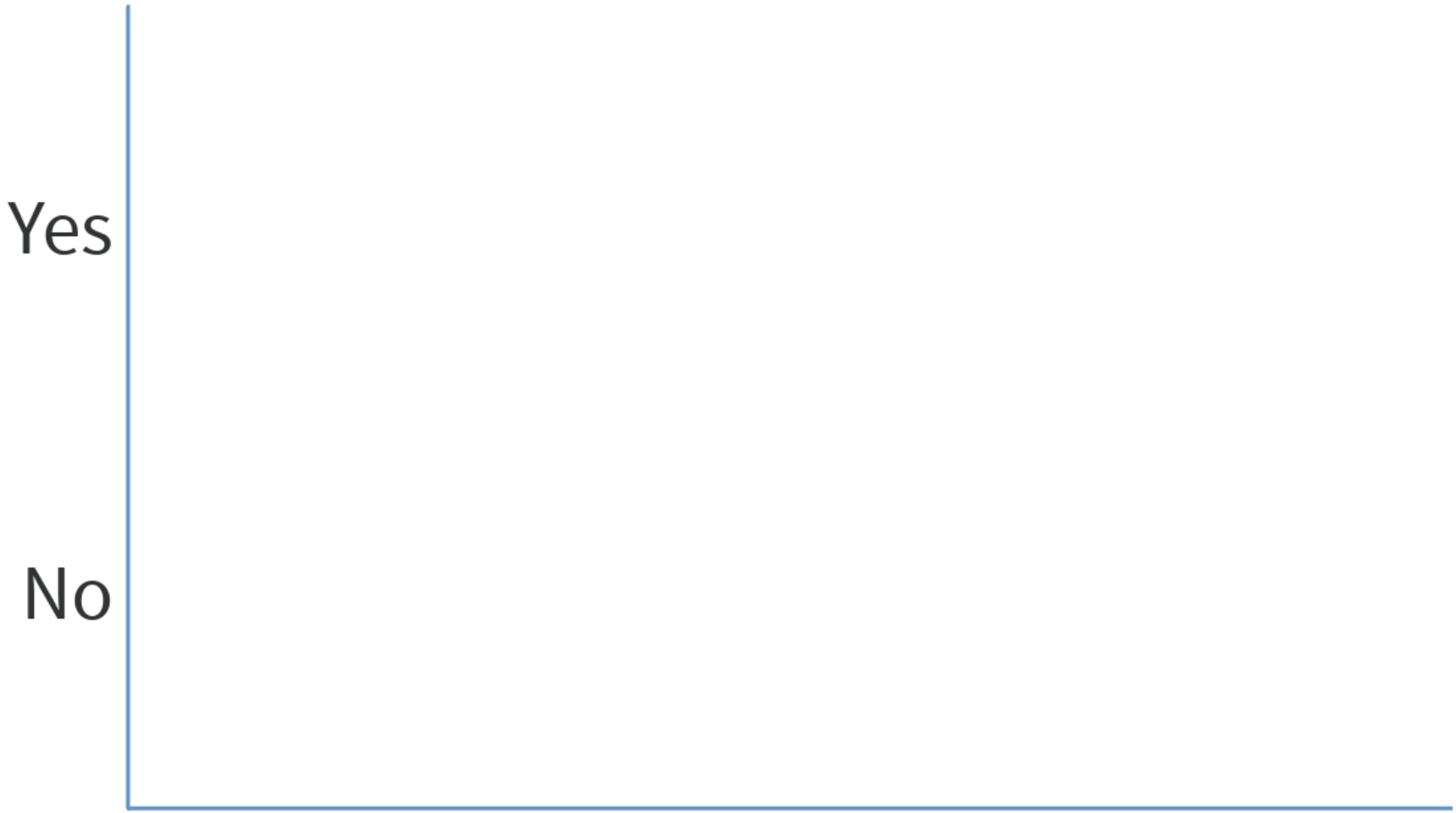
# Have you created a Codio account?



# In what language do you have the most significant programming experience?



# Have you started working on HW 1?

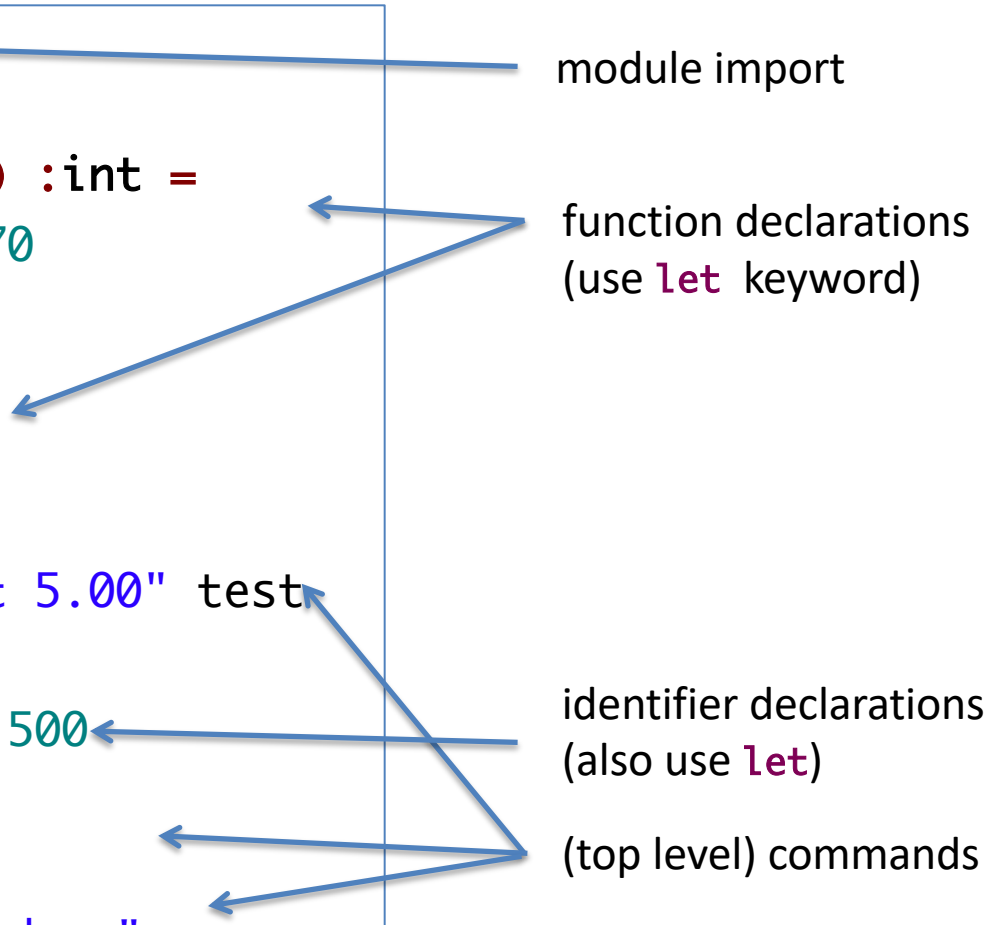


# Programming in OCaml

Read Chapter 2 of the CIS 120 lecture notes, available from the course web page

# What is an OCaml module?

```
;; open Assert

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

let test () : bool =
   attendees 500 = 120

;; run_test "attendees at 5.00" test

let x : int =  attendees 500

;; print_int x
;; print_endline "end of demo"
```



# What does an OCaml program do?

```
;; open Assert

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

let test () : bool =
  attendees 500 = 120

;; run_test "attendees at 5.00" test

let x = attendees 500

;; print_int x
```

To know if the test will pass, we need to know whether this expression is true or false

To know what will be printed we need to know the value of this expression

*To know what an OCaml program will do, we need to know what the value of each expression is*

# Value-Oriented Programming

pure, functional, strongly typed

# Course goal

*Strive for beautiful code.*

- Beautiful code
  - is *simple*
  - is easy to understand
  - is easy(er) to get right
  - is easy to maintain
  - takes skill to write

# Value-Oriented Programming

- Java, C, C#, C++, Python, Perl, etc. are tuned for an **imperative** programming style
  - Programs are full of *commands*
    - *"Change x to 5!"*
    - *"Increment z!"*
    - *"Make this point to that!"*
- OCaml, on the other hand, promotes a **value-oriented** style
  - We've seen that there are a few *commands*…

    `print_endline, run_test`

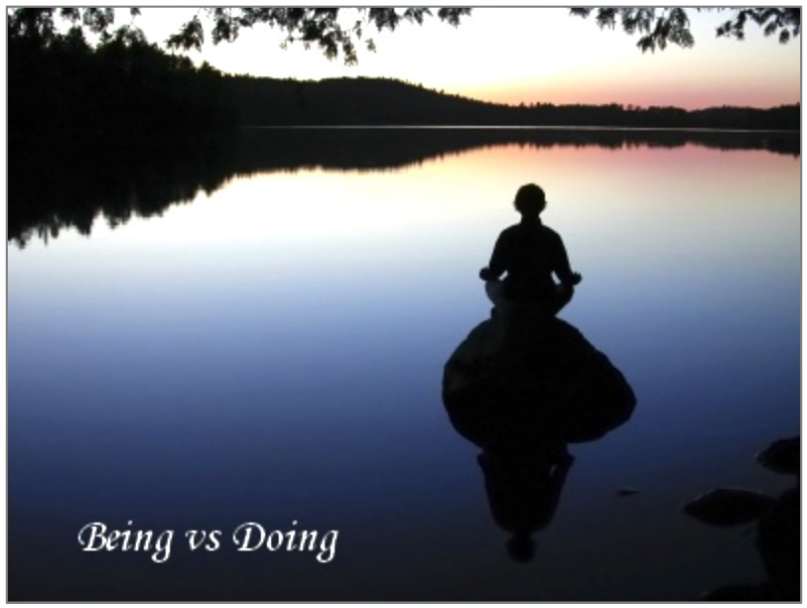
    … but these are used rarely
  - Most of what we write is *expressions* denoting *values*

Metaphorically, we might say that

imperative programming is about ***doing***

while

value-oriented programming is about ***being***



# Programming with Values

- Programming in *value-oriented* (a.k.a. *pure* or *functional*) style can be a bit challenging at first

- But it often leads to code that is much more beautiful

# Values and Expressions

| Types | Values | Operations* | Expressions |
|---|---|---|---|
| `int` | `-1 0 1 2` | `+ * - /` | `3 + (4 * x)` |
| `float` | `0.12  3.1415` | `+. *. -. /.` | `3.0 *. (4.0 *. a)` |
| `string` | `“hello” “CIS120”` | `^` *(concatenation)* | `“Hello, ” ^ s` |
| `bool` | `true  false` | `&& \|\| not` | `(not b1) \|\| b2` |

- Each *type* corresponds to a set of well-typed values.

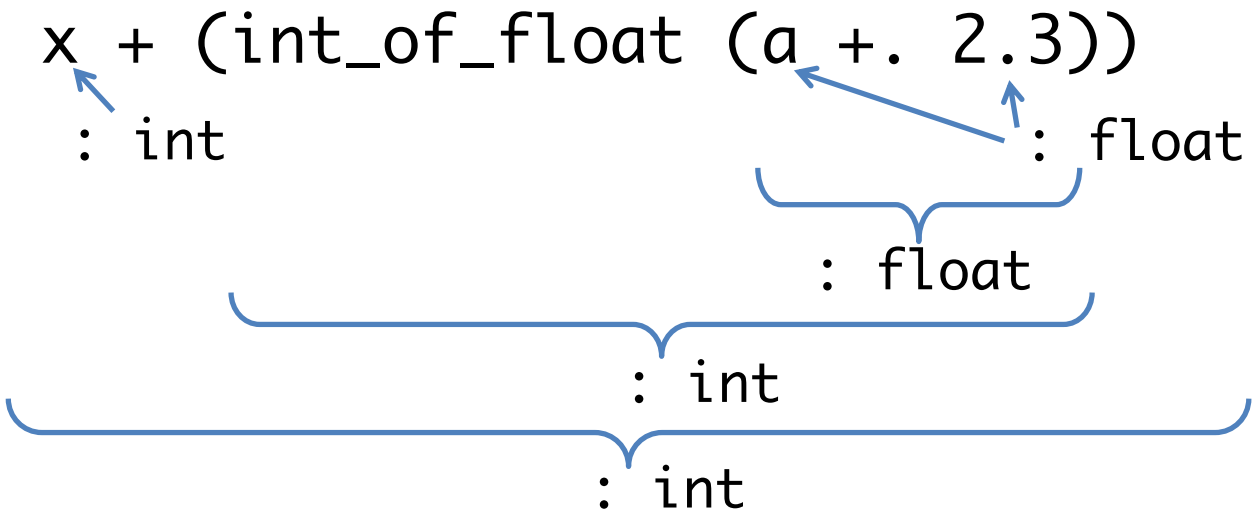
*Note that there is no automatic conversion from float to int, etc., so you must use explicit conversion operations like `string_of_int` or `float_of_int`

# Types

- Every *identifier* has a unique associated type.
- "Colon" notation associates an identifier with its type:

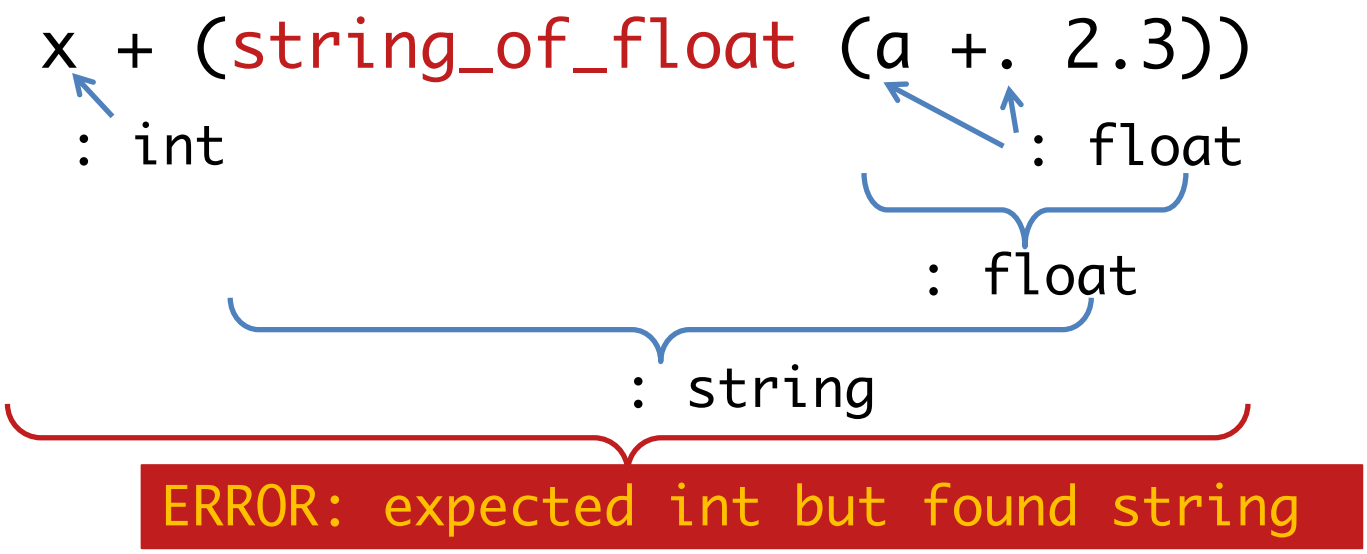
```
x : int          a : float
s : string       b1 : bool
```

- Every OCaml *expression* has a unique type determined by its constituent *subexpressions*



# Type Errors

- OCaml will use *type inference* to check that your program to ensure that it uses types consistently.
  - It will give you an error if not




NOTE: Every time OCaml points out a type error, it is indicating a likely bug! Well-typed Ocaml programs often "just work".

# Sneak Preview

- OCaml has a rich *type structure*:

```
(+) : int -> int -> int
string_of_int : int -> string
```

*function types*

```
() : unit
(1, 3.0) : int * float
```

*tuple types*

```
[1;2;3]  : int list
```

*list types*

- We will see all of these (and how to define our own brand new types) in upcoming lectures…

# Calculating Expression Values

OCaml's model of computation

# Simplification vs. Execution

- We can think of an OCaml expression as just a way of writing down a *value*
- We can visualize running an OCaml program as a sequence of *calculation* or *simplification* steps that eventually lead to this value
- (By contrast, a running Java program is best thought of as performing a sequence of *actions* or *commands*
  - *… a variable named x gets created*
  - *… then we put the value 3 in x*
  - *… then we test whether y is greater than z*
  - *… the answer is true, so we put the value 4 in x*

  – Each command modifies the *implicit, pervasive* state of the machine)

# Calculating with Expressions

OCaml programs mostly consist of *expressions*.

Expressions *simplify* to values:

```
3 ⇒ 3
3 + 4 ⇒ 7
2 * (4 + 5) ⇒ 18
attendees 500 ⇒ 120
```

(values compute to themselves)

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

Note that the symbol '⇒' is *not* OCaml syntax. We're using it to *talk* about the way OCaml programs behave.

# Step-wise Calculation

- We can break down ⇒ in terms of *single step* calculations, written ⟼

- For example:

```
    (2+3) * (5-2)
⟼ 5 * (5-2)          because 2+3 ⟼ 5
⟼ 5 * 3              because 5-2 ⟼ 3
⟼ 15                 because 5*3 ⟼15
```

# Conditional Expressions

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

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

- OCaml conditionals are also *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"
```

# Simplifying Conditional Expressions

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

```
  (if 3 > 0 then 2 else -1) * 100
⟼ (if true then 2 else -1) * 100
⟼ 2 * 100
⟼ 200
```

- The type of a conditional expression is the (single!) type shared by *both* of its branches.
- It doesn’t make sense to leave out the ‘else’ branch in an ‘if’. (What would be the result if the test was ‘false’?)

# Top-level Let Declarations

- A let declaration gives a *name* (a.k.a. an *identifier*) to the value denoted by some expression

```
let pi : float = 3.14159
let seconds_per_day : int = 60 * 60 * 24
```

- The *scope* of a top-level identifier is the rest of the file after the declaration.

"scope" of a name = "the region of the program in which it can be used"

# Immutability

- Once defined by `let`, the binding between an identifier and a value cannot be changed!

```
int x = 3;
x = 4;
```

**Java / C / C++ / …**

*imperative update*

'x = 4' is a *command* that means 'update the contents of location x to be 3'

The state associated with 'x' *changes* as the program runs.

```
let x : int = 3 in
  x = 4
```

**Ocaml**

*named expressions*

'let x : int = 3 ' simply gives the value 3 the *name* 'x'

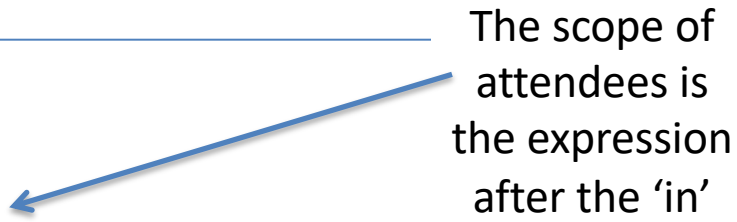
'x = 4' asks does 'x equal 4'? (a boolean value, false)

Once defined, the value bound to 'x' never changes

# Local Let Expressions

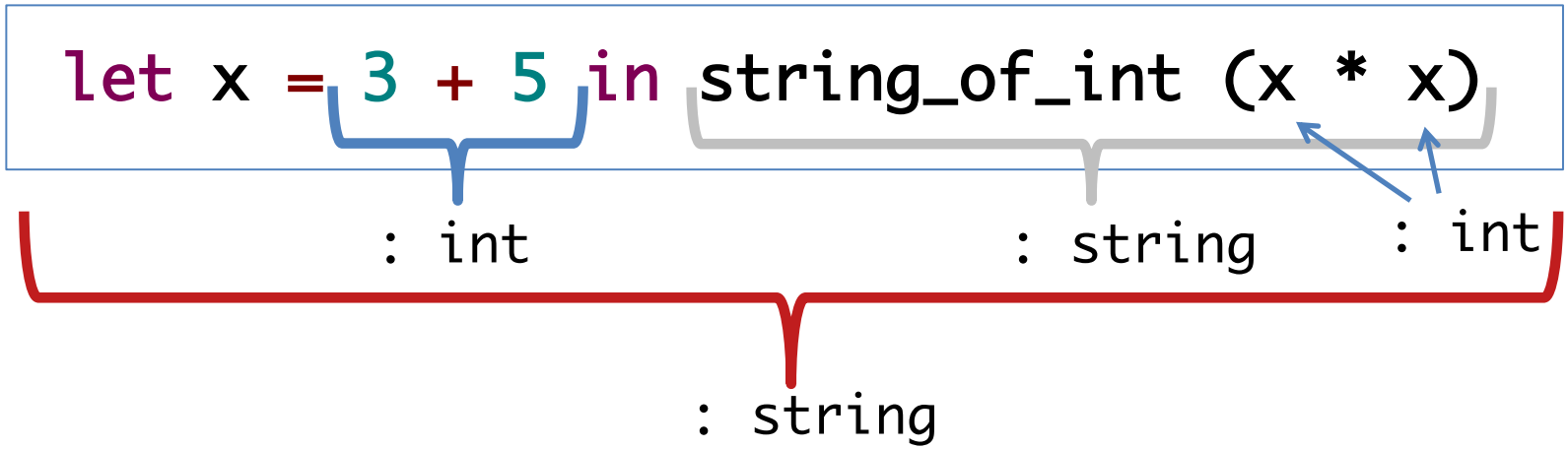
- Let declarations can appear both at top-level and *nested* within other expressions.

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




- Local (nested) let declarations are followed by 'in'
  - e.g. attendees, revenue, and cost
- Top-level let declarations do not use 'in'
  - e.g. profit_500
- The scope of a local identifier is just the expression after the 'in'

# Typing Let Expressions

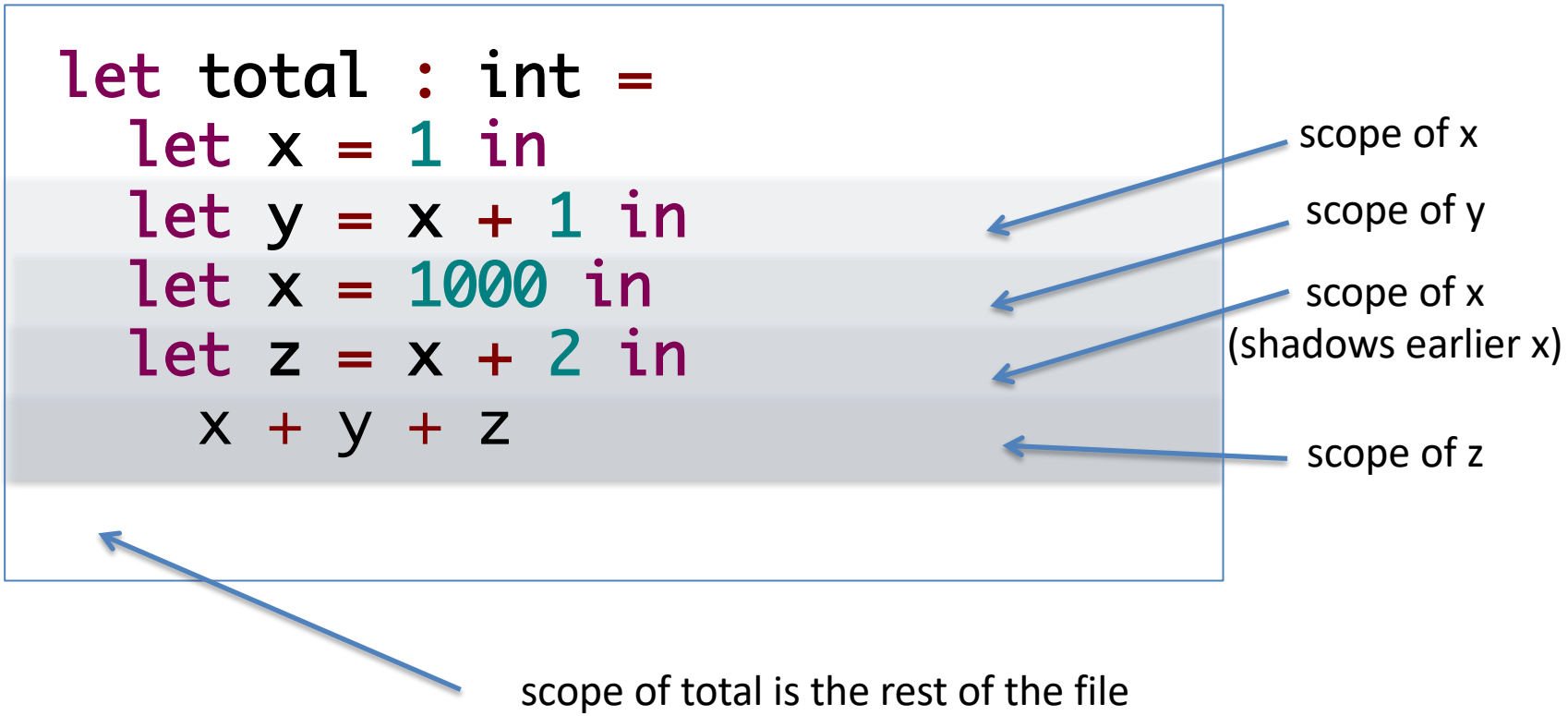



- A let-bound identifier has the type of the expression it is bound to.
- The type of the whole local let expression is the type of the expression after the 'in'
- Recall: type annotations are written using colon:

```
let x : int = … ((x + 3) : int) …
```

# Scope

Multiple declarations of the same variable or function name are allowed. The later declaration *shadows* the earlier one for the rest of the program.



# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =
  let x = 1 in
  let y = x + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =
  let x = 1 in
  let y = x + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

First, we simplify the right-hand side of the declaration for identifier total.

# Simplifying Let Expressions

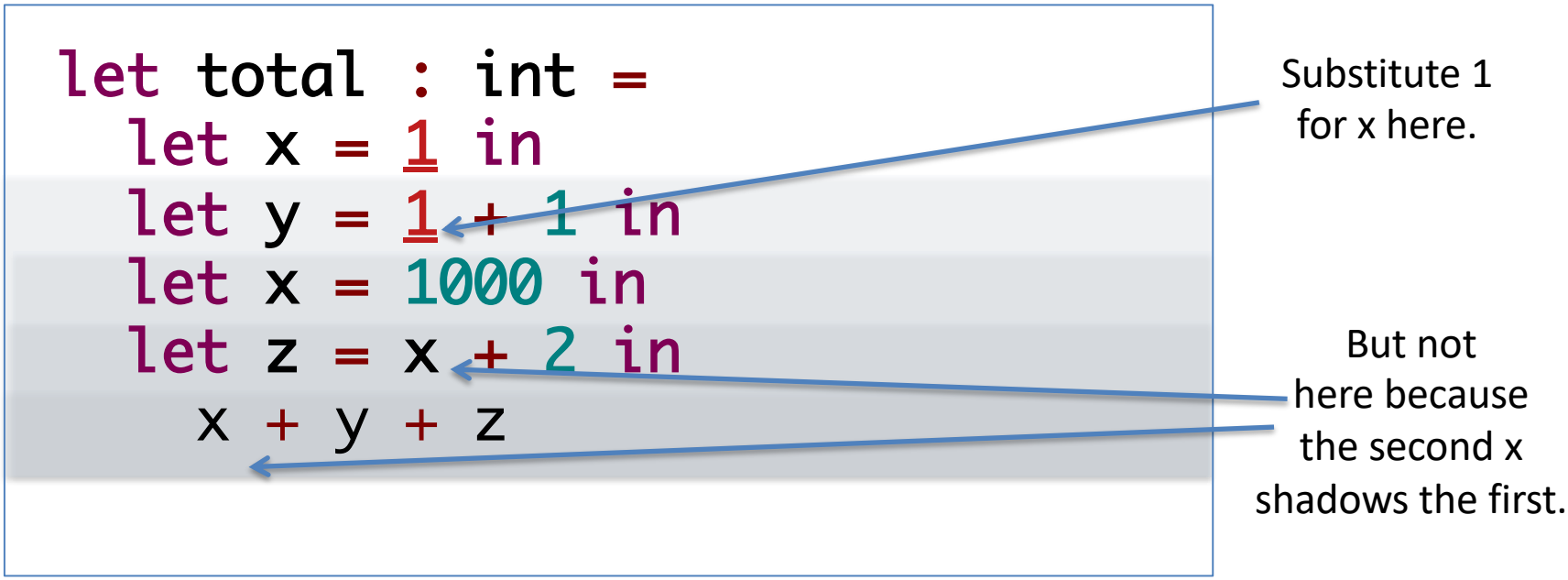
- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =
  let x = 1 in
  let y = x + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

This r.h.s. is already a value.

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left



# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =
  let x = 1 in
  let y = 1 + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

Discard the local let since it's been substituted away.

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =

  let y = 1 + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =

   let y = 1 + 1 in
   let x = 1000 in
   let z = x + 2 in
     x + y + z
```

Simplify the expression remaining in scope.

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =

   let y = 1 + 1 in
   let x = 1000 in
   let z = x + 2 in
     x + y + z
```

Repeat!

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =

   let y = 2 in
   let x = 1000 in
   let z = x + 2 in
     x + y + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =

   let y = 2 in
   let x = 1000 in
   let z = x + 2 in
     x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =

   let y = 2 in
   let x = 1000 in
   let z = x + 2 in
     x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let x = 1000 in
   let z = x + 2 in
     x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let x = 1000 in
   let z = x + 2 in
     x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let x = 1000 in
   let z = x + 2 in
     x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let x = 1000 in
   let z = 1000 + 2 in
     1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let x = 1000 in
   let z = 1000 + 2 in
     1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let z = 1000 + 2 in
     1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let z = 1000 + 2 in
     1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let z = 1000 + 2 in
     1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let z = 1002 in
     1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let z = 1002 in
     1000 + 2 + 1002
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


   let z = 1002 in
     1000 + 2 + 1002
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =


    1000 + 2 + 1002
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int =
```

```
1000 + 2 + 1002      ⇒    2004
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
let total : int = 2004
```

# Things (for you) to do…

- Sign up for Piazza if necessary
- Sign up for Codio
- Homework 1: OCaml Finger Exercises
  - Practice using OCaml to write simple programs
  - Start with first 4 problems
    - (needed background on lists coming next week!)
  - Start early!