Programming Languages and Techniques (CIS1200)

Lecture 2

Value-Oriented Programming

CIS 1200

- If you are joining us today…WELCOME!

- Please check Ed for announcements and reminders
  - If you are already registered for the course, you should be signed up automatically
  - If not, you'll get added automatically when you enroll

- Read the course syllabus and Ch. 1 lecture notes and watch Wed's lectures, all available on the website
  - www.cis.upenn.edu/~cis1200/

Announcements (1)

- No class on Monday (MLK Day)
- Recitations start next week
- Dr. Zdancewic will be away next week
  - no office hours
  - Lectures on Weds. and Fri. will be covered by Dr. Weirich (another regular instructor for this course)
  - otherwise, business as usual - I should have access to Ed and email

Announcements (2)

- Please *read*
  - Chapter 2 of the lecture notes
  - OCaml style guide on the course website (https://www.seas.upenn.edu/~cis1200/23sp/ocaml_style)
- Homework 1: OCaml Finger Exercises
  - Instructions are on the Schedule page of course website
  - Code is available on Codio (see Ed)
  - Practice using OCaml to write simple programs
  - Due: *January 24th, 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 with the assignment
  - You'll need to write your own tests
  - Our grading script will apply *additional* tests
  - 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
  - TA manual grading of non-testable properties
  - feedback on style from your TAs

Where to ask questions

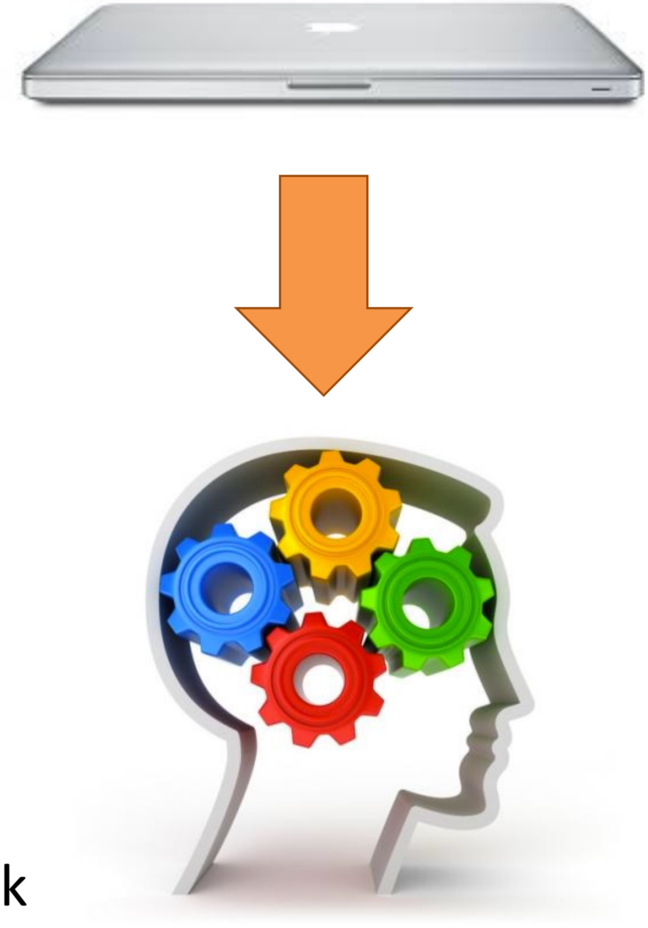
- Course material
  - **Ed Discussion Board**
  - TA office hours (on website calendar, starts Tues 9/6)
  - Prof. office hours:
    Dr. Zdancewic Mon 3.30-5pm Levine 511
    (also by appointment)

- Tutoring available
- HW/Exam Grading: see website FAQ
- About CIS majors & Course Registration
  - CIS Undergraduate coordinators, Levine 308
  - cis-undergrad-advising@seas.upenn.edu

Poll Everywhere

- We will use *Poll Everywhere* for interactive quizzes during lecture
  - Answer with your phone or laptop
  - Completely ungraded
  - Useful for gauging your understanding
- We'll start using it on January 23rd

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 :-)

Programming in OCaml

Codio

- Codio codio.com
  - *see Ed for enrollment info*
  - web-based development environment
  - *remote access for TA help*
- Under the hood:
  - linux virtual machine (Ubuntu)
  - pre-configured per project with everything you need
  - configurable editor

OCaml

- Industrial-strength, statically-typed *functional* programming language
- Lightweight, approachable setting for learning about program design

- See ocaml.org
  - CIS1200 uses only a small part of the language
  - We will cover everything you need to know.

Who uses OCaml?

intel

facebook

JANE STREET

LexiFi

Google

CiTRIX

Microsoft

MLstate

cea

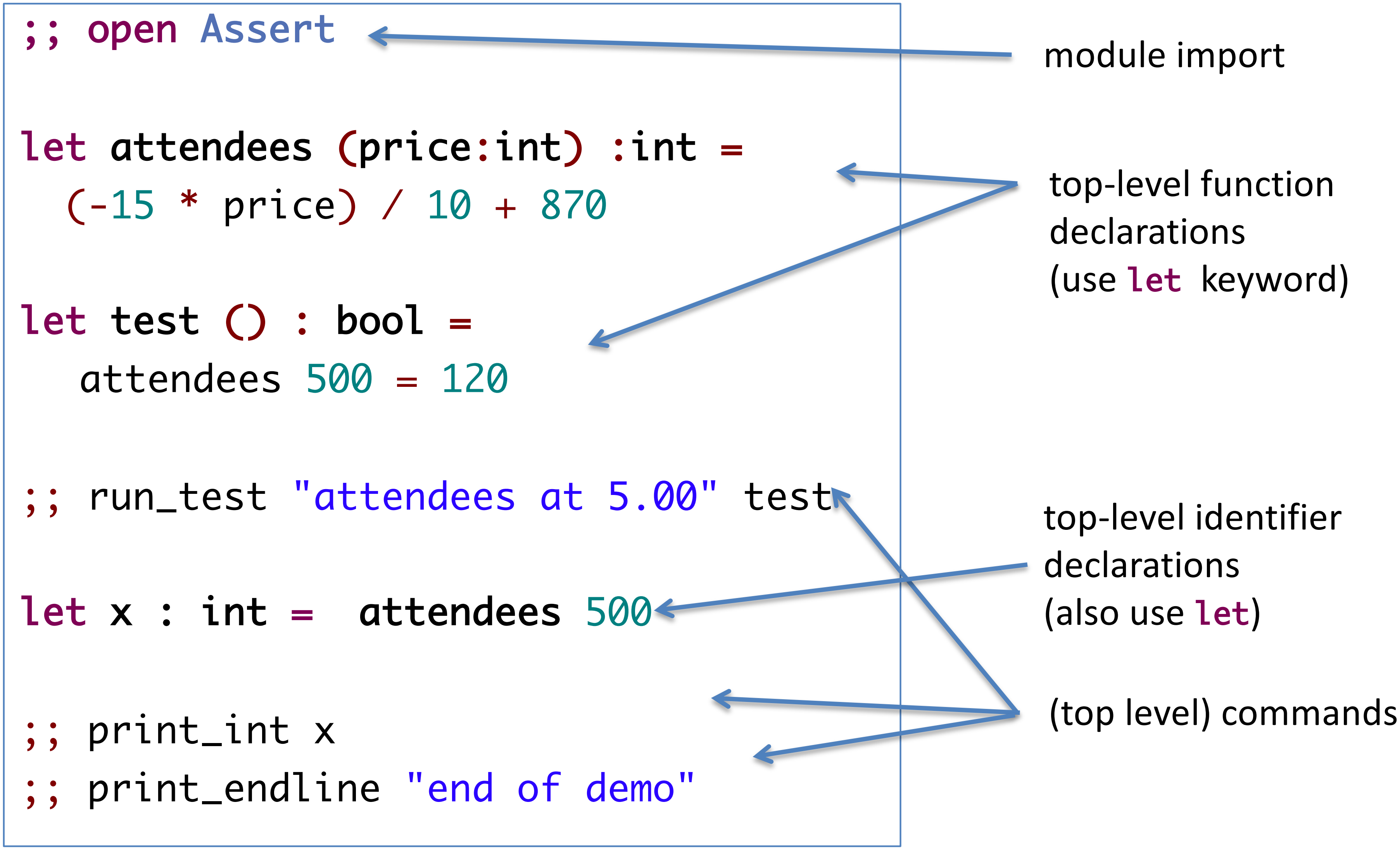
Microsoft Visual F#

mylife

SimCorp

DASSAULT AVIATION

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"
```

module import

top-level function
declarations
(use `let` keyword)

top-level identifier
declarations
(also use `let`)

(top level) commands

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`

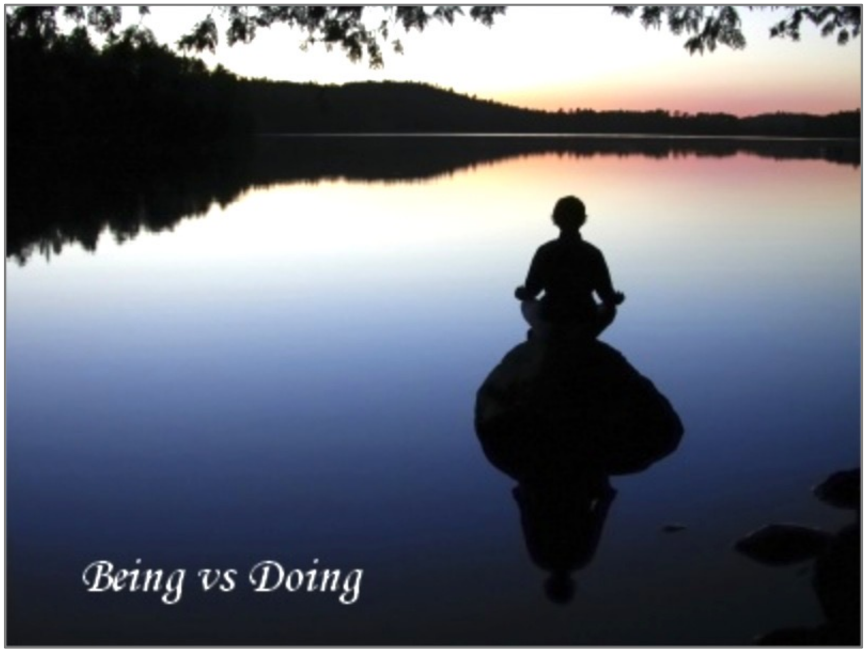
    … but these are used rarely
  - Most of what we write is *expressions* denoting *values*

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

Types, Values, and Expressions

Types	Values	Operations	Expressions
`int`	`-1 0 1 2`	`+ * - /`	`(3 + y) * x`

- Each *type* corresponds to a set of *values*
- Each *expression* is built from *operations* on values and it simplifies to a value (or already is a value)
- Use parentheses to associate nested expressions

Types, Values, and Expressions

Types	Values	Operations*	Expressions
`int`	`-1 0 1 2`	`+ * - /`	`(3 + y) * 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 *values*
- Each *expression* is built from *operations* on values and it simplifies to a value (or already is a value)
- Use parentheses to associate nested expressions

*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`

Static vs. Dynamic

The term ‘*static*’ indicates something that happens *before* the program is run.

OCaml (like Java) has a static type system: the compiler checks that the program is *well typed* before the program is run.

The term ‘*dynamic*’ refers to something that happens *while* the program is running.

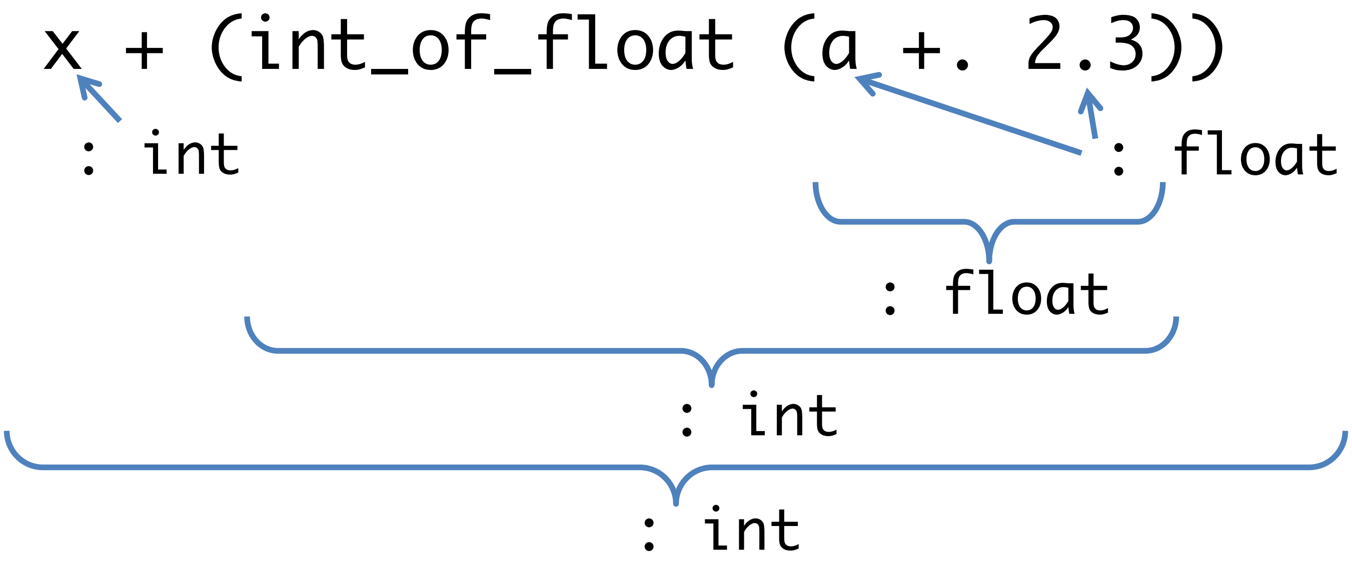
(We will learn about Java’s “dynamic dispatch” later in the course.)

Static Types

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

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

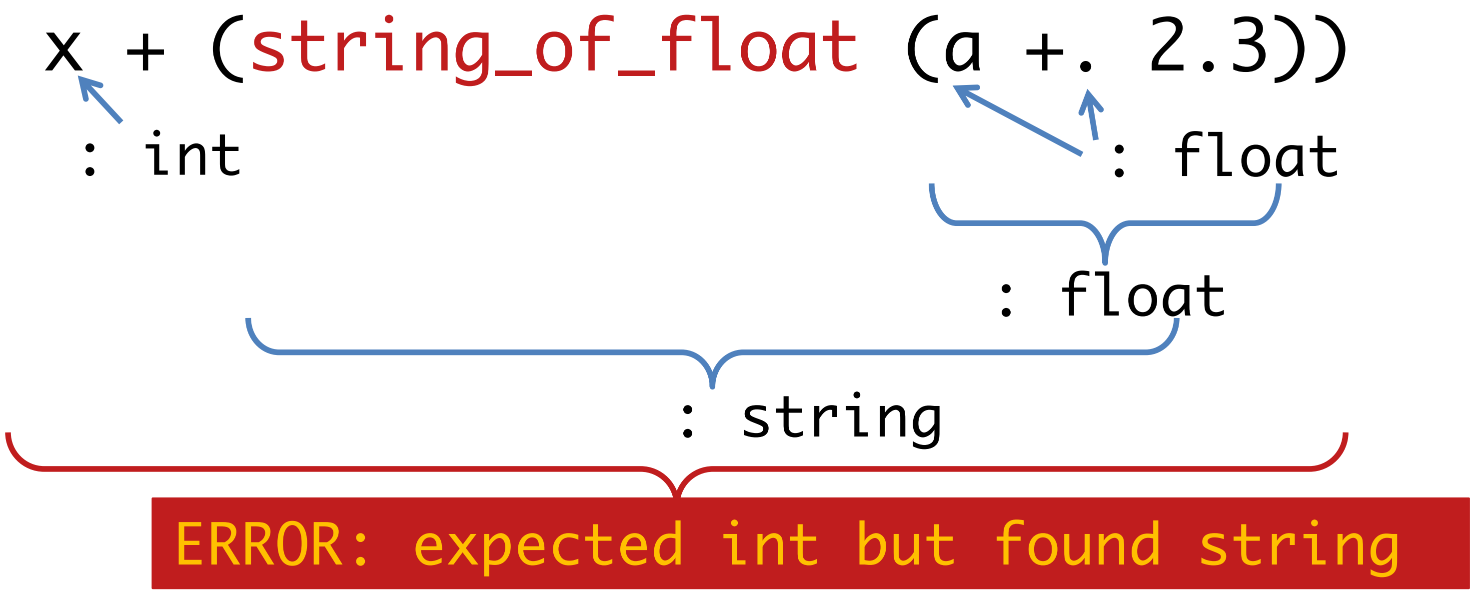
- Every OCaml *expression* has a unique type determined by its constituent *subexpressions*



```
x + (int_of_float (a +. 2.3))
```

Static Type Errors

- OCaml uses *type inference* to check that your program uses types consistently



`x + (string_of_float (a +. 2.3))`

`: int`

`: float`

`: float`

`: string`

`ERROR: expected int but found string`

Because + expects both of its inputs to be of type `int`.

NOTE: Every time OCaml points out a type error, it is indicating a likely bug. Well-typed OCaml programs often "just work"!
TIP: Adding type annotations can help track down type checking errors.

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 the Values of Expressions

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
- In 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")
```

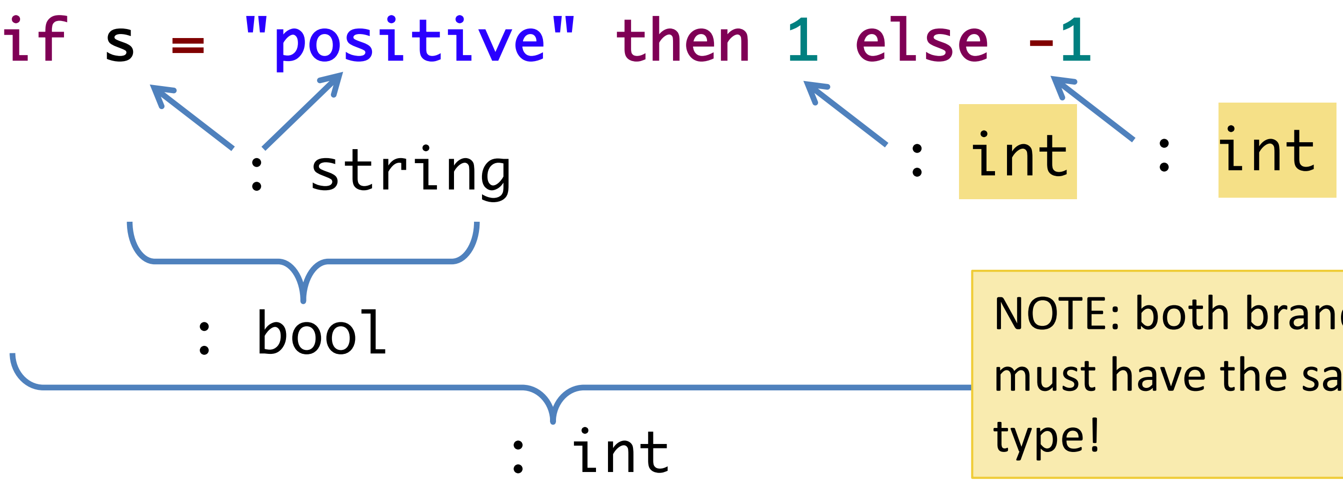
Simplifying Conditional Expressions

- 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
```

- It doesn’t make sense to leave out the ‘else’ branch in an ‘if’. (What would the value be if the test was ‘false’?)

Typing Conditional Expressions



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

: string

: int

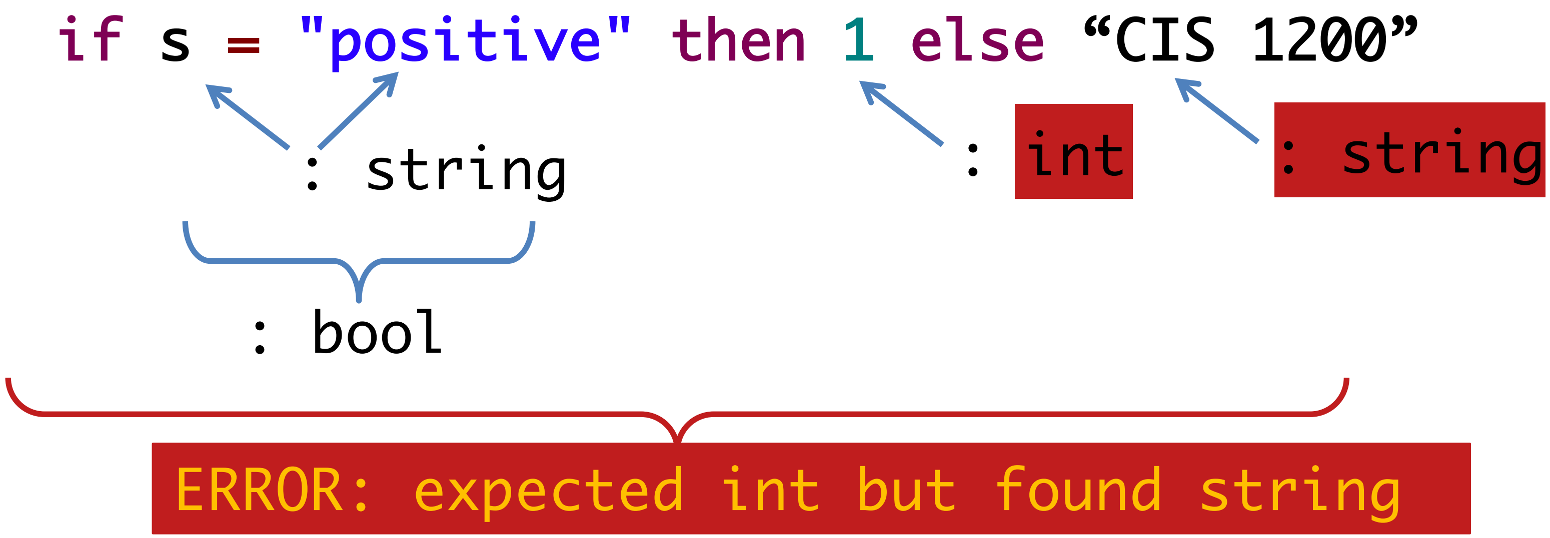
: int

: bool

: int

NOTE: both branches must have the same type!

Type Errors



```
if s = "positive" then 1 else “CIS 1200”
```

: string

: bool

: int

: string

ERROR: expected int but found string

Let Declarations

naming, not "assigning"

Top-level Let Declarations

- A let declaration gives a *name* (a.k.a. *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

The “*scope*” of a name is “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++ / python / …

imperative update

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

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

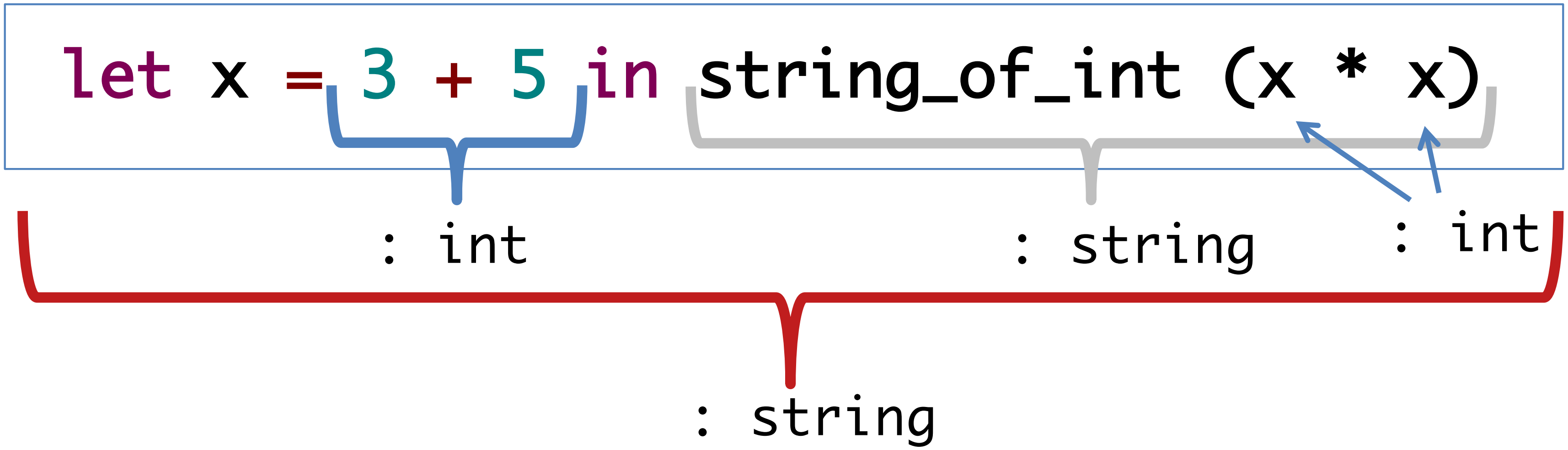
- Let declarations can appear both at top level and *nested* within other expressions.

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

The scope of attendees is the expression after the 'in'

- Local 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 Local Let Expressions



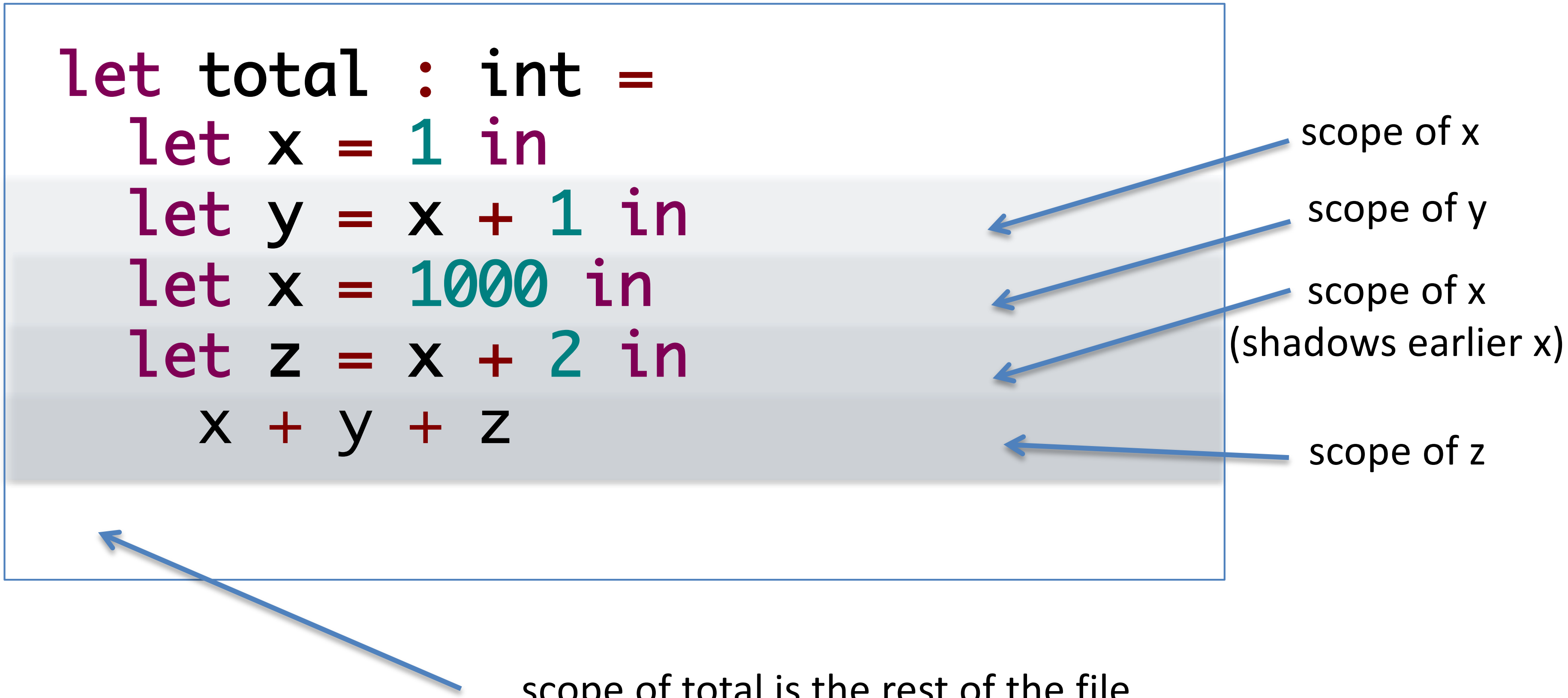
```
let x = 3 + 5 in string_of_int (x * x)
```

- 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.



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

scope of x

scope of y

scope of x
(shadows earlier x)

scope of z

scope of total is the rest of the file

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

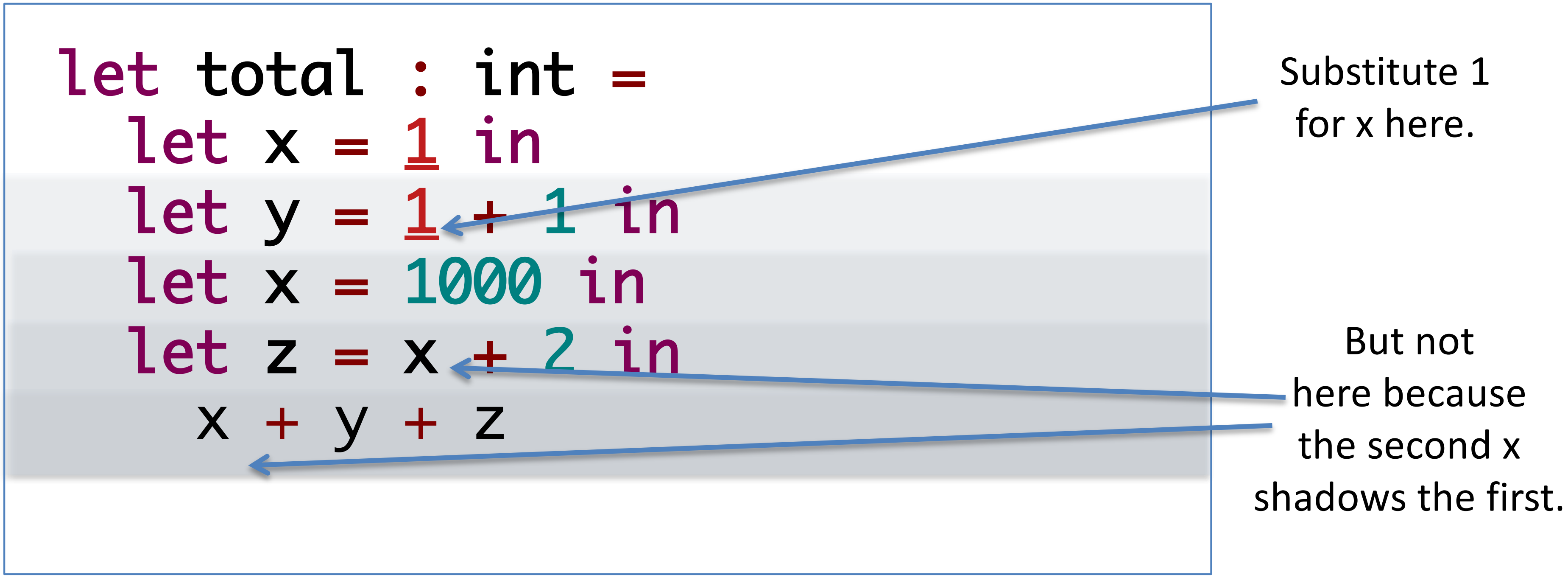
- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let…in’ part
  - simplify what's left

```
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



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

Substitute 1 for x here.

But not here because the second x shadows the first.

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
```

Lexical Scopes

When reading code: a variable refers to the nearest enclosing let-binding.

- Be sure to account for nested expressions

```
let answer : int =
  let x = 1 in
  let y = let x = 2 in x + x in
  x + y
```

For example:
answer = 5

With explicit parentheses:

```
let answer : int =
  let x = 1 in
  let y = (let x = 2 in x + x) in
  x + y
```

These occurrences of ‘x’ refer to ‘x = 2’

This ‘x’ refers to ‘x = 1’. (The other let binding doesn’t enclose this x!)

Things (for you) to do…

- Sign up for Codio
- Check Ed for announcements
- 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!