Programming Languages and Techniques (CIS120)

Lecture 2

Value-Oriented Programming

Programming in OCaml

CIS 120 Tools

- OCaml
  - Industrial-strength, statically-typed *functional* programming language
  - Lightweight, approachable setting for learning about program design
  - Web based development: codio.com
- Java
  - Industrial-strength, statically-typed *object-oriented* language
  - Many tools/libraries/resources available
  - Develop using Codio or Eclipse

Why two languages??

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

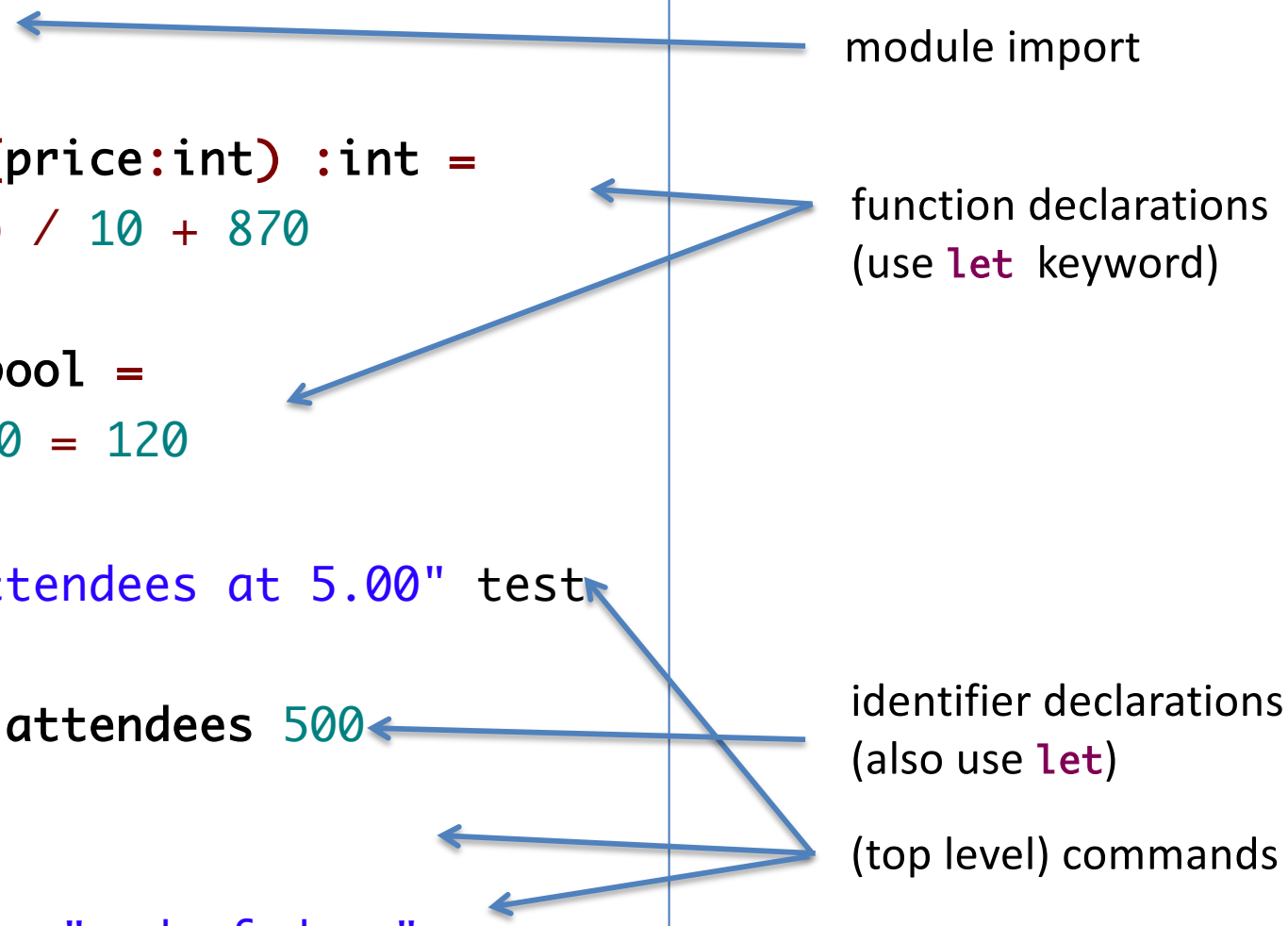
"[The OCaml part of the class] was very essential to getting fundamental ideas of comp sci across. Without the second language it is easy to fall into routine and syntax lock where you don't really understand the bigger picture.''
---Anonymous CIS 120 Student

"[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.''
--- Anonymous CIS 120 Student

Who uses OCaml?

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`

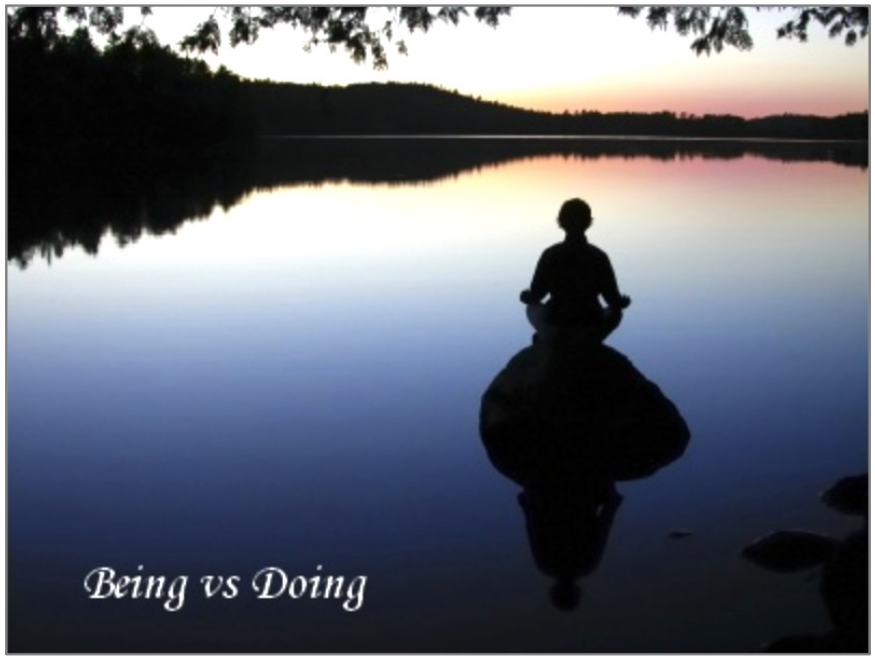
    … but these are used rarely
  - Most of what we write is *expressions* denoting *values*

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 *expression* computes a value (or already is a value)
- Each *type* corresponds to a set of well-typed values

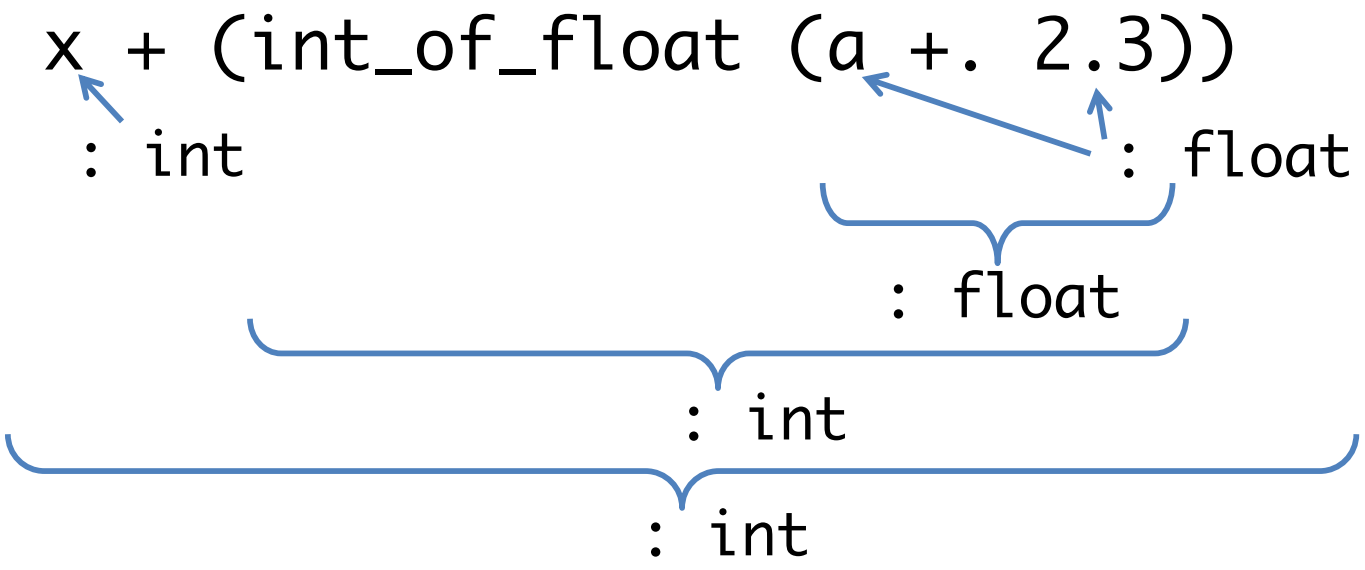
*Note that there is no automatic conversion from float to int, etc., so you must use explicit conversion operations like `string_of_int` or `float_of_int`

Types

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

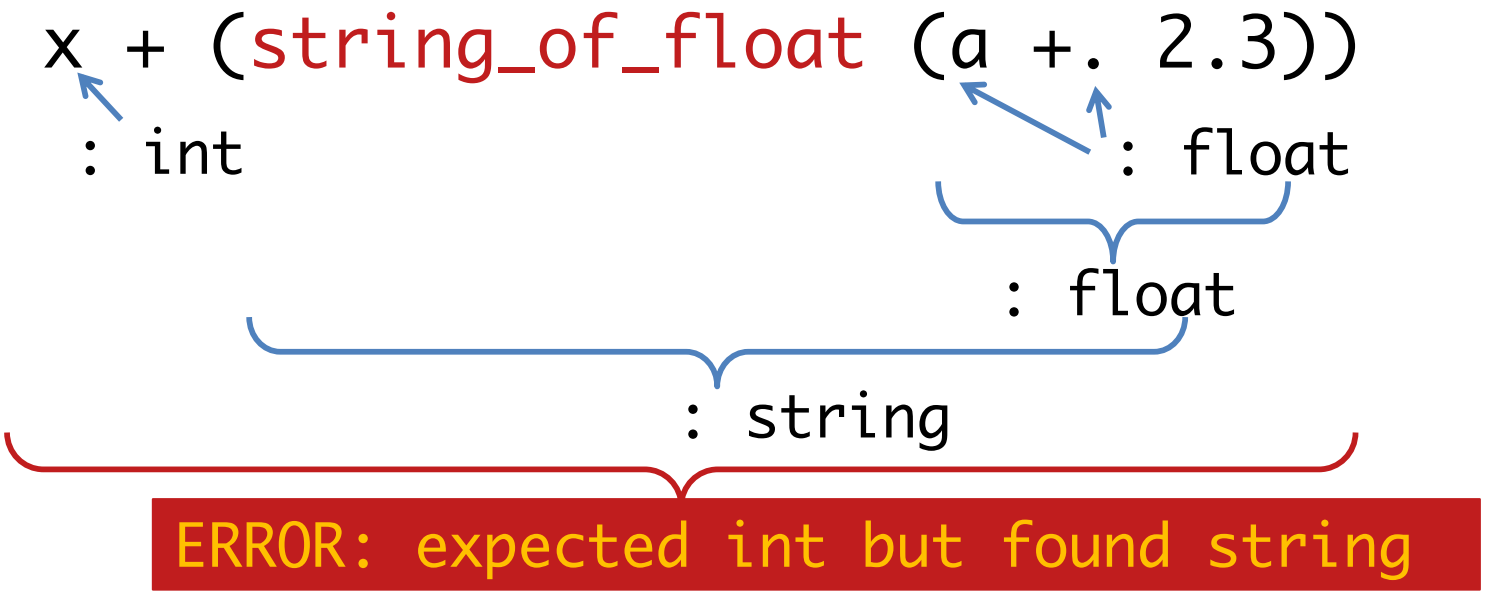
```
x : int          a : float
s : string       b1 : bool
```

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



Type Errors

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



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 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"
```

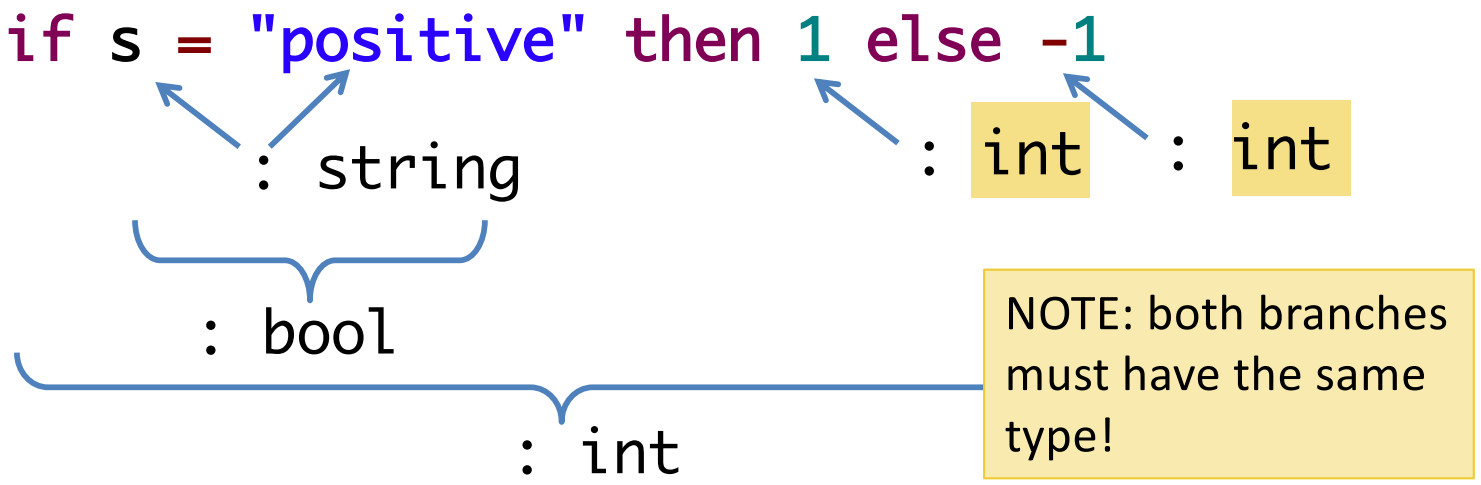
Simplifying Conditional Expressions

- 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:

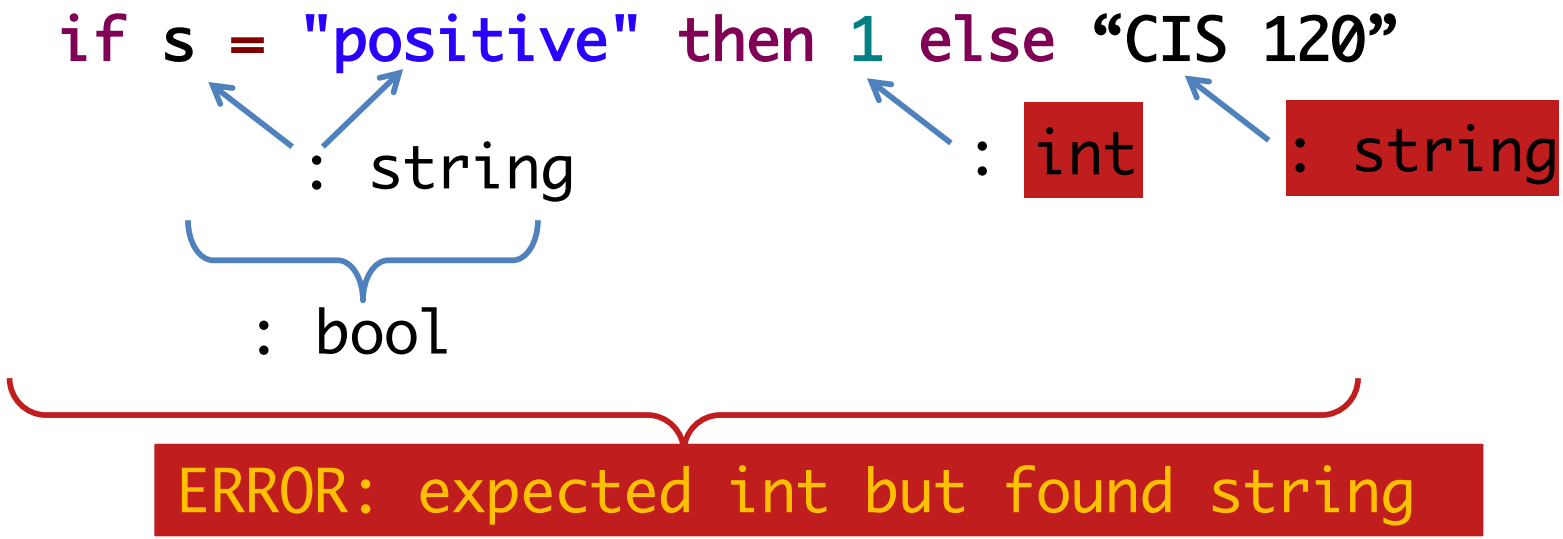
```
  (if 3 > 0 then 2 else -1) * 100
⟼ (if true then 2 else -1) * 100
⟼ 2 * 100
⟼ 200
```

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

Typing Conditional Expressions



Type Errors


if s = "positive" then 1 else “CIS 120”
: string
: int
: string
: bool
ERROR: expected int but found string

Let Declarations

naming, not assigning

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'

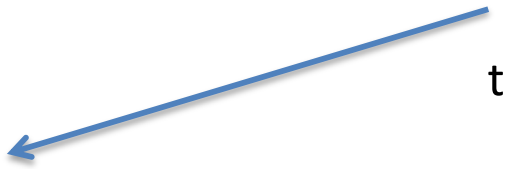
'x = 4' asks does 'x equal 4'? (a boolean value, false)

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

Local Let Expressions

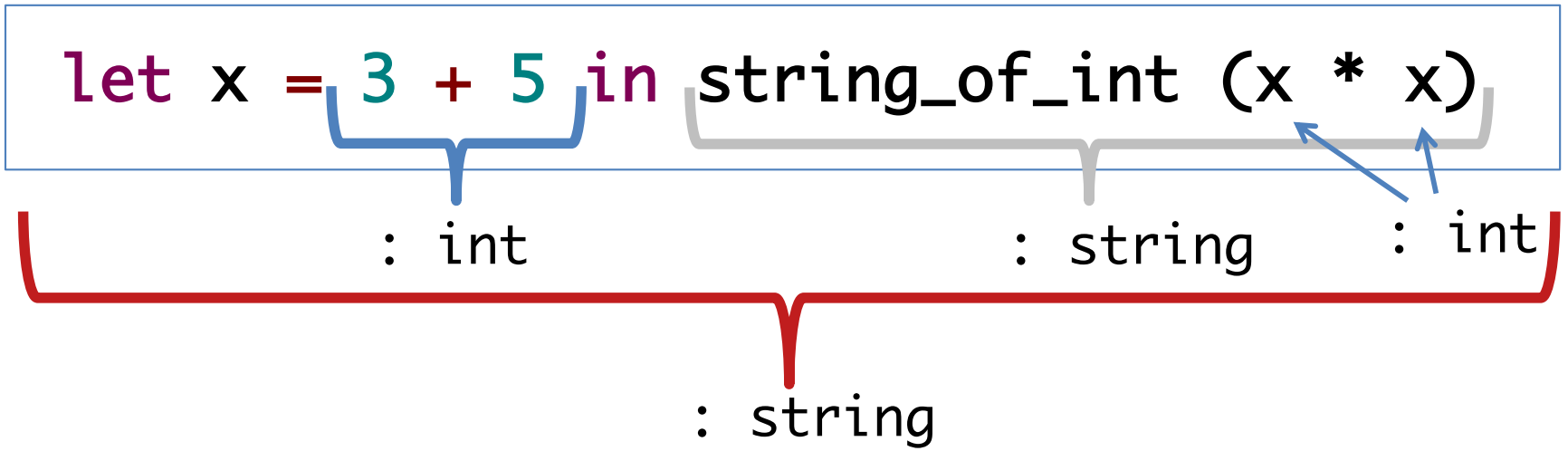
- Let declarations can appear both at top-level and *nested* within other expressions.

```
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

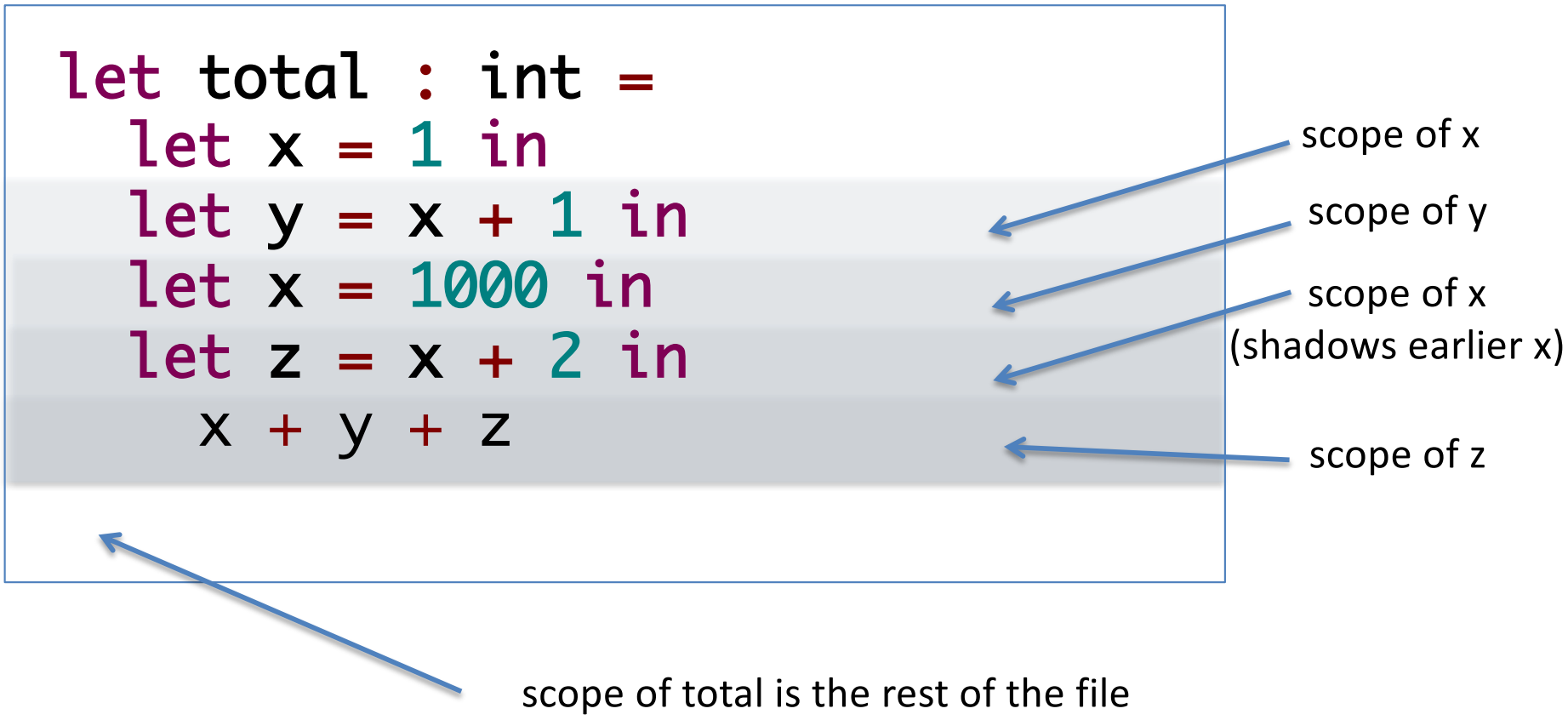


- A let-bound identifier has the type of the expression it is bound to.
- The type of the whole local let expression is the type of the expression after the 'in'
- Recall: type annotations are written using colon:

```
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

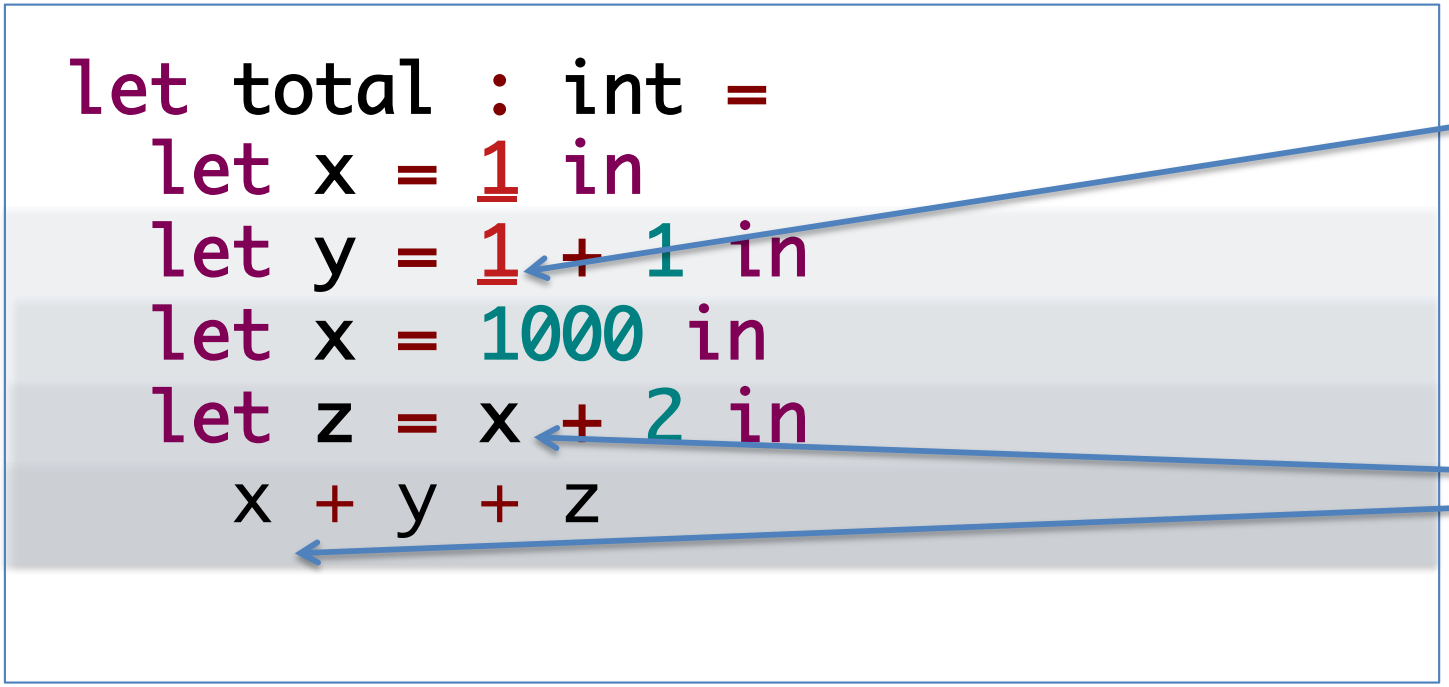
- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let...in' part
  - simplify what's left

```
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
```