Programming Languages and Techniques (CIS120)

Lecture 2

August 28th, 2015

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/spring2015/cis120
- Install OCaml/Eclipse on your laptop; ask if you have questions
  - www.cis.upenn.edu/~cis120/current/ocaml_setup.shtml
- Obtain a clicker from the bookstore
- *No laptops, tablets, smart phones, etc., during lecture*

Registration

- If you are not registered, add your name to the waiting list
  - There are spots available, but we want to make sure that those who need to take the course this semester can register first
- Need a different recitation?
  - If the want you want is open, switch online
  - If you need to attend a closed recitation, add your name to the recitation change request form
  - Go to the recitation you want, even if not registered

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
  - Practice using OCaml to write simple programs
  - Start with first 4 problems (lists next week!)
  - Due: Tuesday, September 8th at 11:59:59pm (midnight)
  - Start early!

Homework Policies

- Projects will be (mostly) automatically graded
  - 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
  - Most assignments will also include style points, added later
  - Your code must compile to get *any* credit
- You will be given your score (on the automatically graded portion of the assignment) immediately
- 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

Important Dates

- Homework:
  - Homework due dates listed on course calendar
  - Tuesdays or Thursdays: see posted schedule (under lectures)

- Exams:
  - 12% First midterm: Friday, October 2nd, in class
  - 12% Second midterm: Friday, November 6th, in class
  - 18% Final exam: TBA
  - Contact instructor *well in advance* if you have a conflict

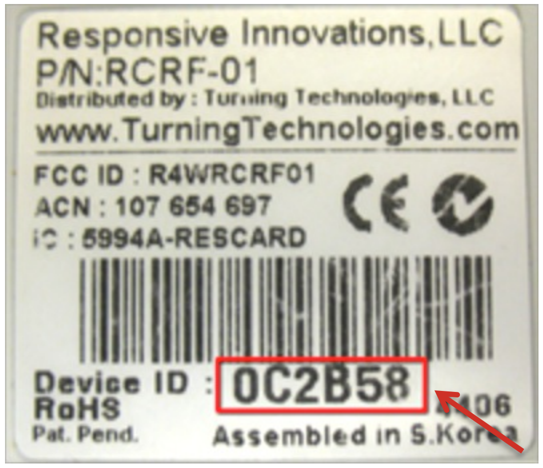
Where to ask questions

- Course material
  - **Piazza Discussion Boards**
  - TA office hours, on webpage calendar
  - Tutoring
  - Prof office hours: Mondays from 3:30 to 5:00 PM, or by appointment (changes will be announced on Piazza)
- HW/Exam Grading: see webpage
- About the CIS majors & Registration
  - Ms. Jackie Caliman, CIS Undergraduate coordinator

Clickers

Clicker Basics

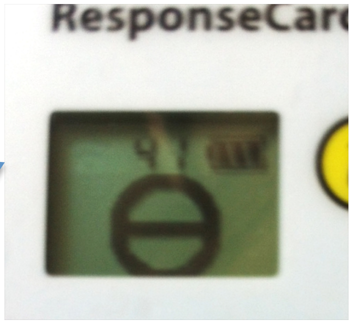
- Beginning today, we'll use clickers in each lecture
  - Grade recording starts next Friday: 9/4/2015
- Any kind of TurningPoint ResponseCard is fine
  - Doesn't have to be the exact model sold in the bookstore
- Use the link on the course website to register your device ID with the course database



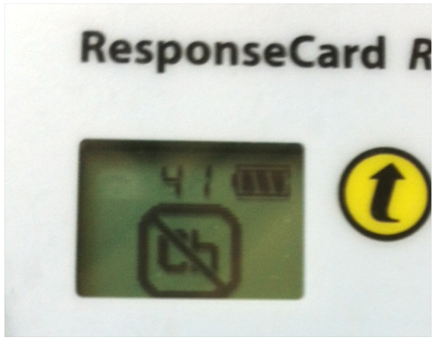
6-character device ID

Test Drive

- Clickers out!
- Press any of the number buttons
  - Make sure the display looks like this:



- If it looks like this…
  - … first check that the *channel is set to 41*
    - If not, try pressing Channel, then 41, then Channel again to reset the channel
  - If this doesn't work come to office hours



Have you successfully installed OCaml on your laptop?

```
1) Yes
2) No
```

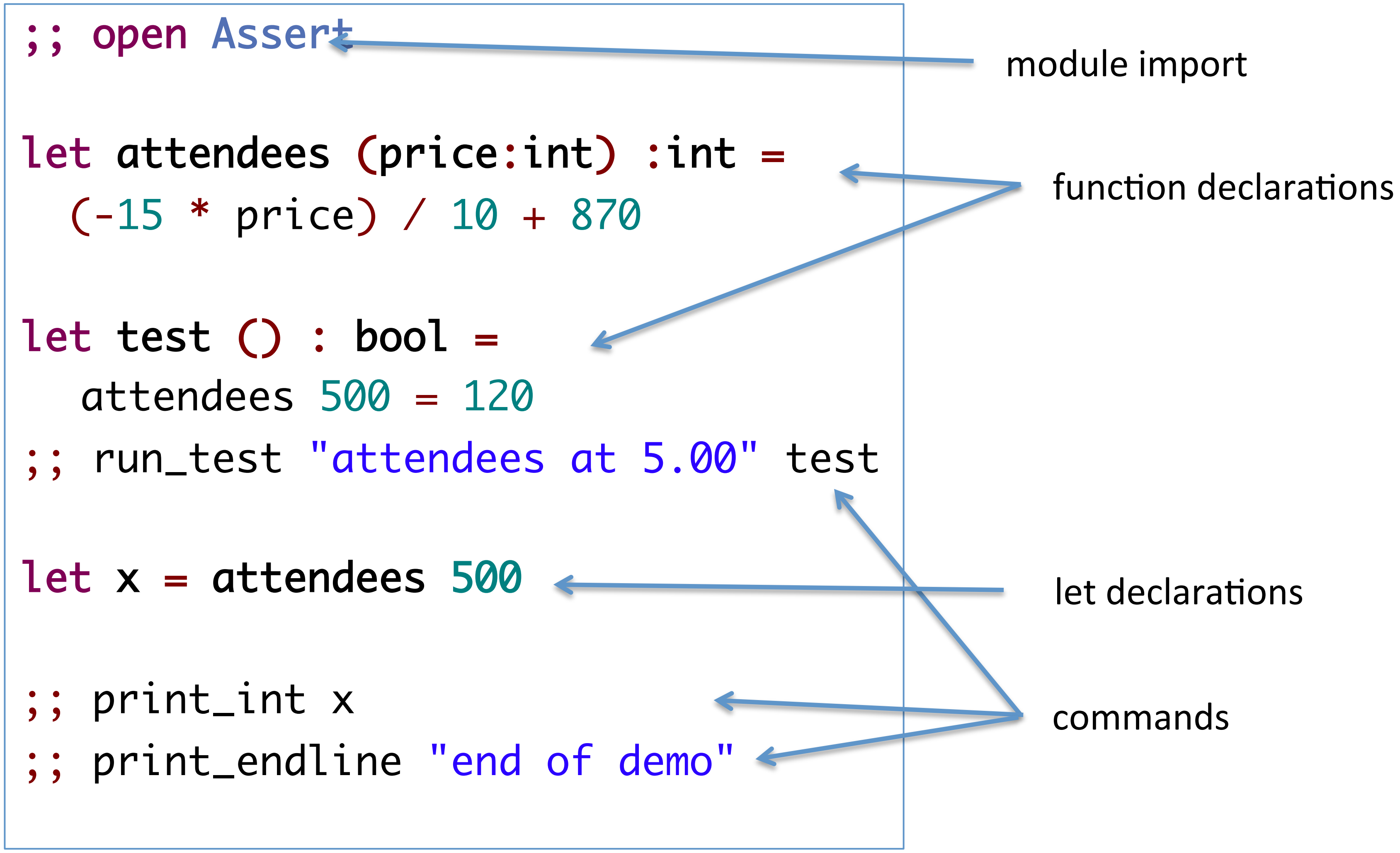
In what language do you have the most significant programming experience?

```
1) Java or C#
2) C, C++, or Objective-C
3) Python, Ruby, Javascript, or MATLAB
4) Clojure, Scheme, or LISP
5) OCaML, Haskell, or Scala
6) Other
```

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 = 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, you 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 likely to be correct
  - is easy to maintain
  - takes skill to develop

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…
    - e.g., `print_endline`, `run_test`

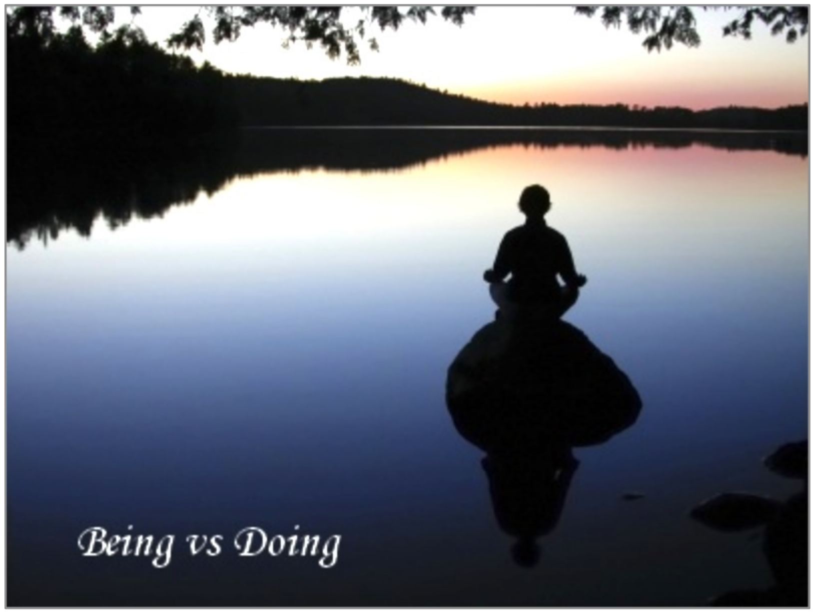
    … but these are used rarely
  - Most of what we write is *expressions* denoting *values*

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, in the end, it leads to code that is simpler to understand…

Values and Expressions

Types	Values	Operations	Expressions
`int`	`-1 0 1 2`	`+ * - /`	`3 + (4 * x)`
`float`	`0.12 3.1415 -2.56`	`+. *. -. /.`	`3.0 *. (4.0 *. x)`
`string`	`“hello” “CIS120”`	`^`	`“Hello, ” ^ x`
`bool`	`true false`	`&& || not`	`(not x) || y`

- Every OCaml *expression* has a type* determined by its constituent subexpressions.
- Each type corresponds to a set of values.
- Later we'll see how to create our own types and values.

*OCaml is a strongly statically-typed language. 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`

Calculating Expression Values

OCaml’s computational model.

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 performs a sequence of *actions* or *events*
    - *… 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*
  - They modify the *implicit, pervasive* state of the machine

Calculating with Expressions

OCaml programs mostly consist of *expressions*.

We understand programs by *simplifying* expressions 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 value <val>.

Note that the symbol '⇒' is *not* OCaml syntax. It's a convenient way to *talk* about the way OCaml programs behave.

Step-wise Calculation

- We can understand $\Rightarrow$ in terms of single step calculations written '$\longmapsto$'

- For example:

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

$\longmapsto$ `5 * (5-2)`	because 2+3 $\longmapsto$ 5
$\longmapsto$ `5 * 3`	because 5-2 $\longmapsto$ 3
$\longmapsto$ `15`	because 5*3 $\longmapsto$ 15

- *Every* form of expression can be simplified with $\longmapsto$

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'-expression or its 'else'-expression, 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 (one!) 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
```

- There is no way of *assigning* a new value to an identifier after it is declared – it is *immutable*.
- The *scope* of a top-level identifier is the rest of the file after the declaration.

*We sometimes call these identifiers *variables*, but the terminology is a bit confusing because in languages like Java and C a variable is something that can be modified over the course of a program. In OCaml, like in mathematics, once a variable's value is determined, it can never be modified… As a reminder of this difference, for the purposes of OCaml we'll try to use the word "identifier" when talking about the name bound by a let.

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

Typing Let Expressions

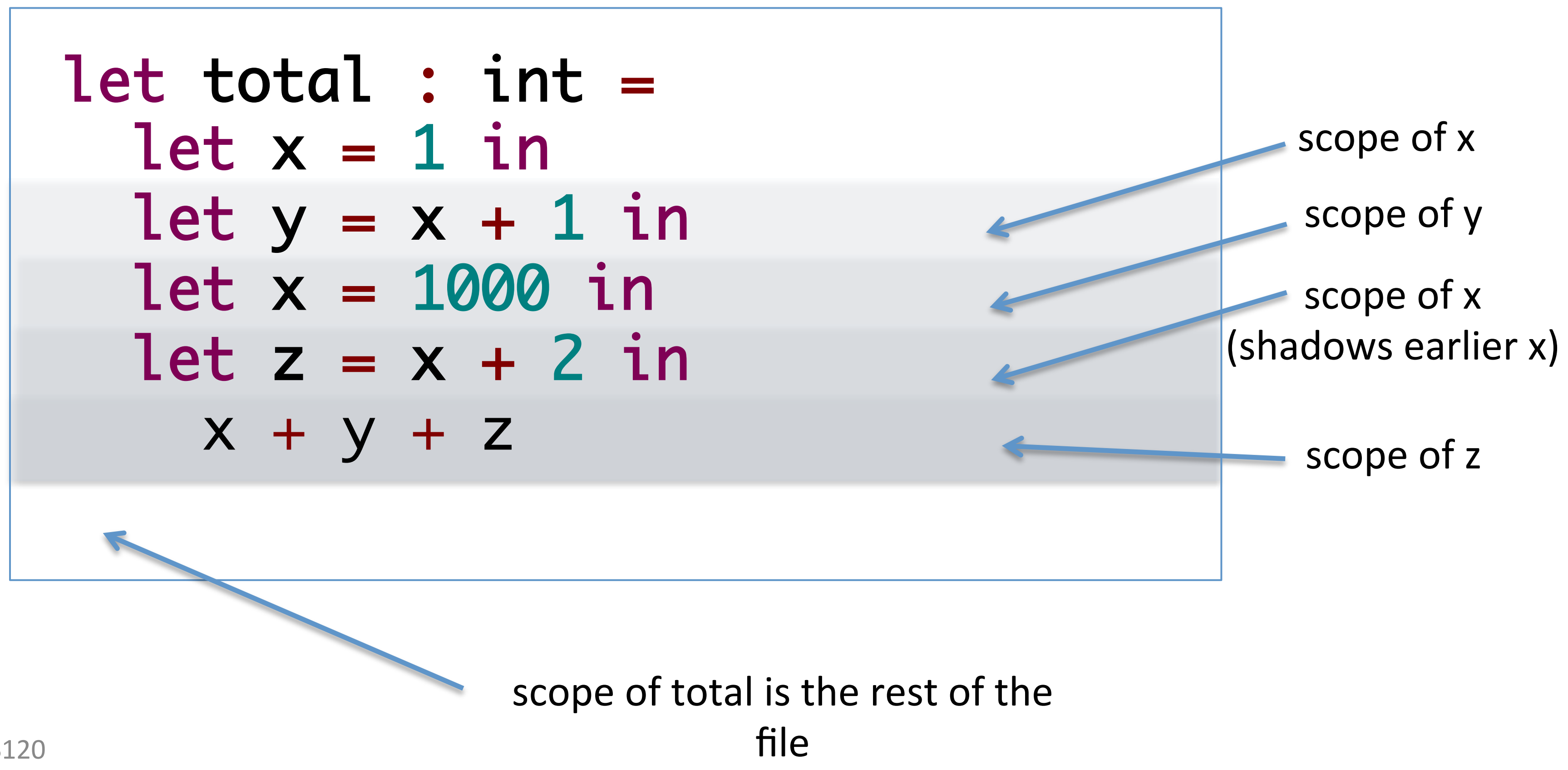
$$\underbrace{\texttt{let x =}\ \underbrace{\texttt{3 + 5}}_{:\ \texttt{int}}\ \texttt{in}\ \underbrace{\texttt{string\_of\_int (}\underset{:\ \texttt{int}}{\texttt{x}}\ \texttt{*}\ \underset{:\ \texttt{int}}{\texttt{x}}\texttt{)}}_{:\ \texttt{string}}}_{:\ \texttt{string}}$$

- Inside its scope, 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'
- Type annotations in OCaml 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
```

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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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’ for local lets (it’s no longer needed)
  - simplify the expression remaining in the scope

```
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

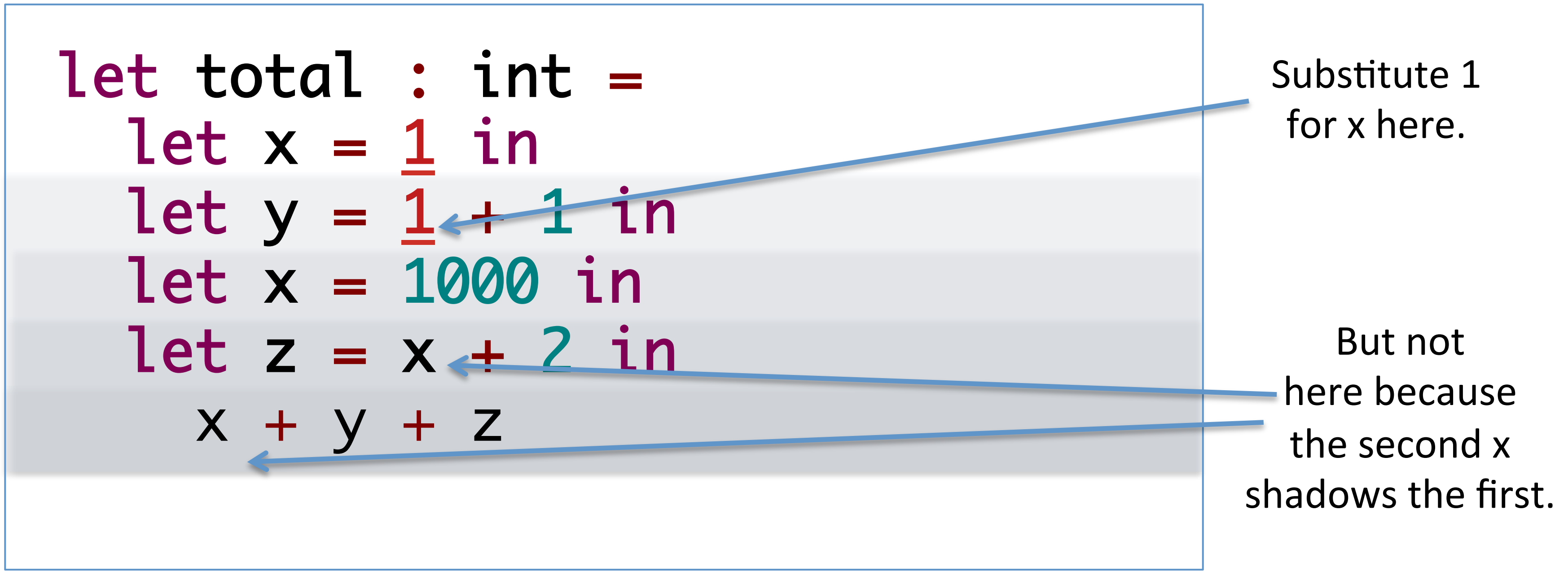
- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the 'let – in' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope



```
let total : int =
  let x = 1 in
  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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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’ for local lets (it’s no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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’ for local lets (it’s no longer needed)
  - simplify the expression remaining in the scope

```
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’ for local lets (it’s no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

```
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’ for local lets (it’s no longer needed)
  - simplify the expression remaining in the scope

```
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' for local lets (it's no longer needed)
  - simplify the expression remaining in the scope

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

Purity

- The meaning ("denotation") of a pure expression doesn't change over time:

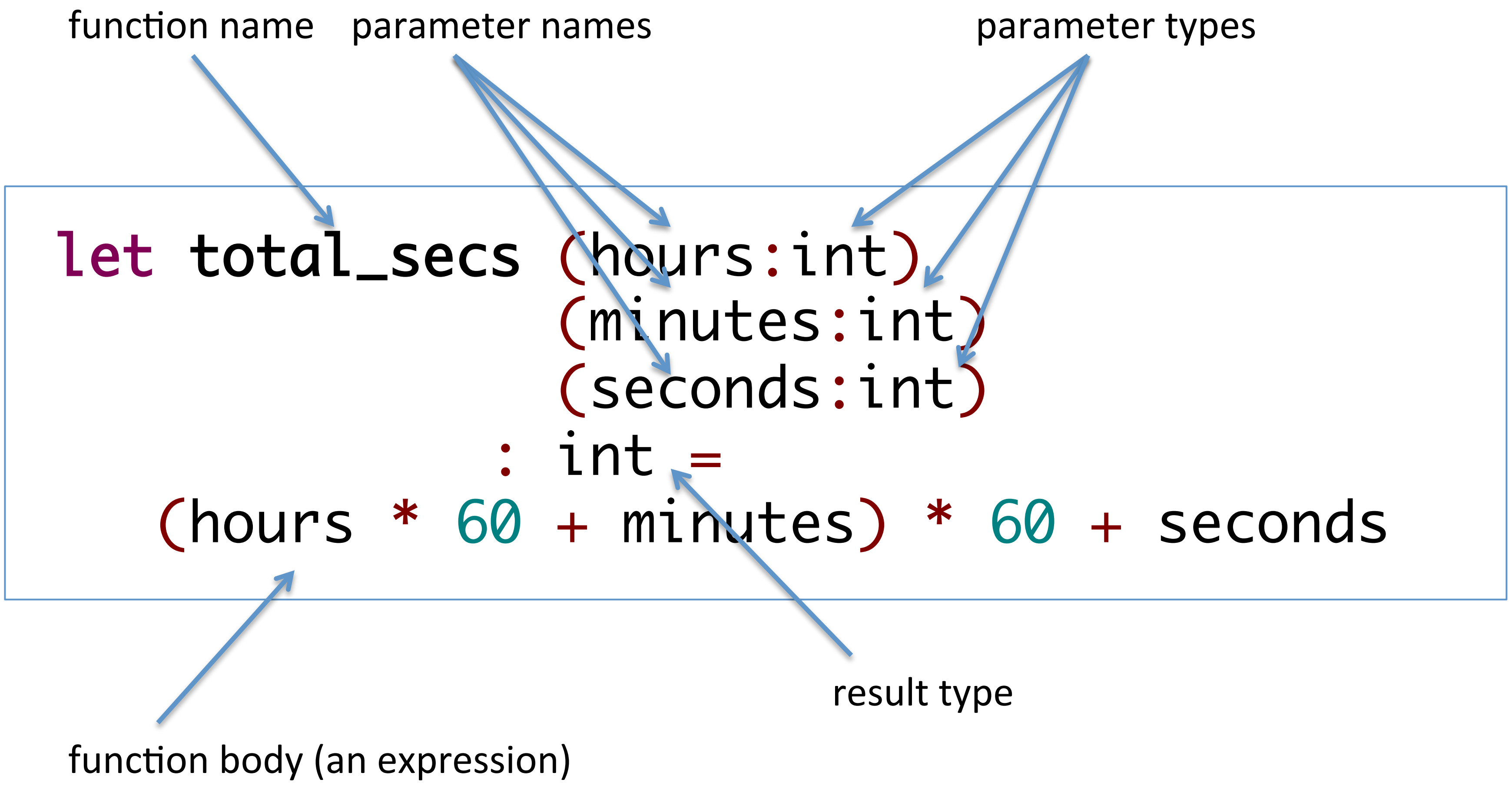
```
let atts : int =
  (attendees 500)

let atts2 : int = atts + atts
```

```
let atts2 : int =
    (attendees 500)
  + (attendees 500)
```

Both ways of computing atts2 are equivalent (in contrast with an imperative language, which runs attendees' *effects* twice.)

(Top-level) Function Declarations



```
let total_secs (hours:int)
               (minutes:int)
               (seconds:int)
             : int =
   (hours * 60 + minutes) * 60 + seconds
```

Function Calls

Once a function has been declared, it can be invoked by writing the function name followed by a list of arguments. This is *function application* expression.

```
total_secs 5 30 22
```

(Note that the list of arguments is *not* parenthesized.)

Calculating With Functions

- To calculate the value of a function application, first calculate values for its arguments and then *substitute* them for the parameters in the body of the functions.

```
   total_secs (2 + 3) 12 17
⟼ total_secs 5 12 17
⟼ (5 * 60 + 12) * 60 + 17      subst. the args in the body
⟼ (300 + 12) * 60 + 17
⟼ 312 * 60 + 17
⟼ 18720 + 17
⟼ 18737
```

```
let total_secs (hours:int)
               (minutes:int)
               (seconds:int)
             : int =
(hours * 60 + minutes) * 60 + seconds
```

TODO

- Sign up for Piazza
  - via link on course web page
- Obtain a clicker
- Homework 1: OCaml Finger Exercises
  - Practice using OCaml to write simple programs
  - Start with first 4 problems (lists next week!)
  - Start early!