Programming Languages and Techniques (CIS1200)

Lecture 3

Value-Oriented Programming (continued)

Lists and Recursion

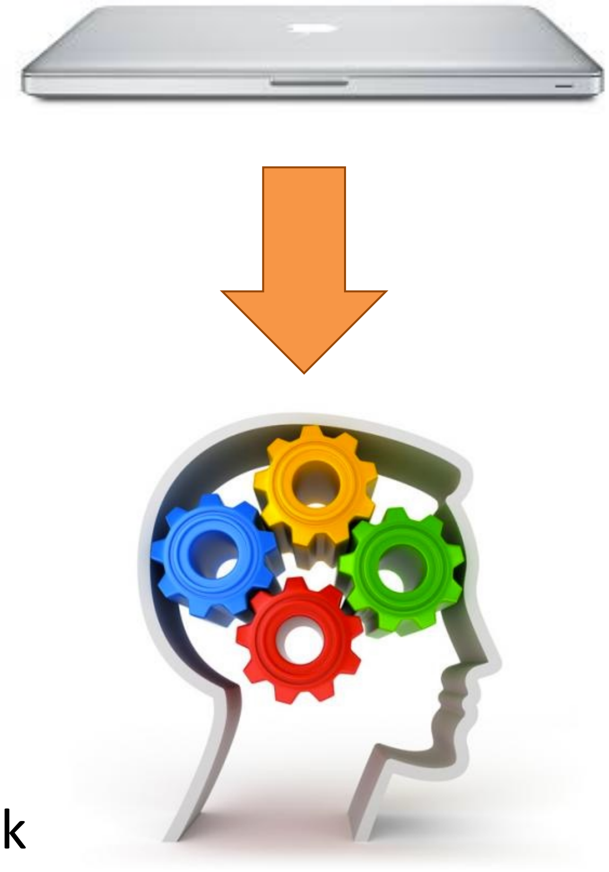
CIS 1200 Announcements

- Homework 1: OCaml Finger Exercises
  - Due: Tuesday, 9/10 at midnight
  - Must submit to Gradescope website
  - Use the ’Zip’ option in the ‘Run Submission’ menu *not* Codio’s “export as zipfile”

- Reading: Please read up through Chapter 3
- Questions?
  - Post to Ed (privately if you need to include code!)
  - Look at HW1 FAQ
- TA and instructor office hours:
  See course Calendar webpage
- Recitations start today!

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 :-)



Poll Everywhere

Poll Everywhere Basics

- Beginning today, we'll use Poll Everywhere in each lecture
  - You can use your phone, laptop, etc., (but only for polls!)

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

Have you created a Codio account?

Yes

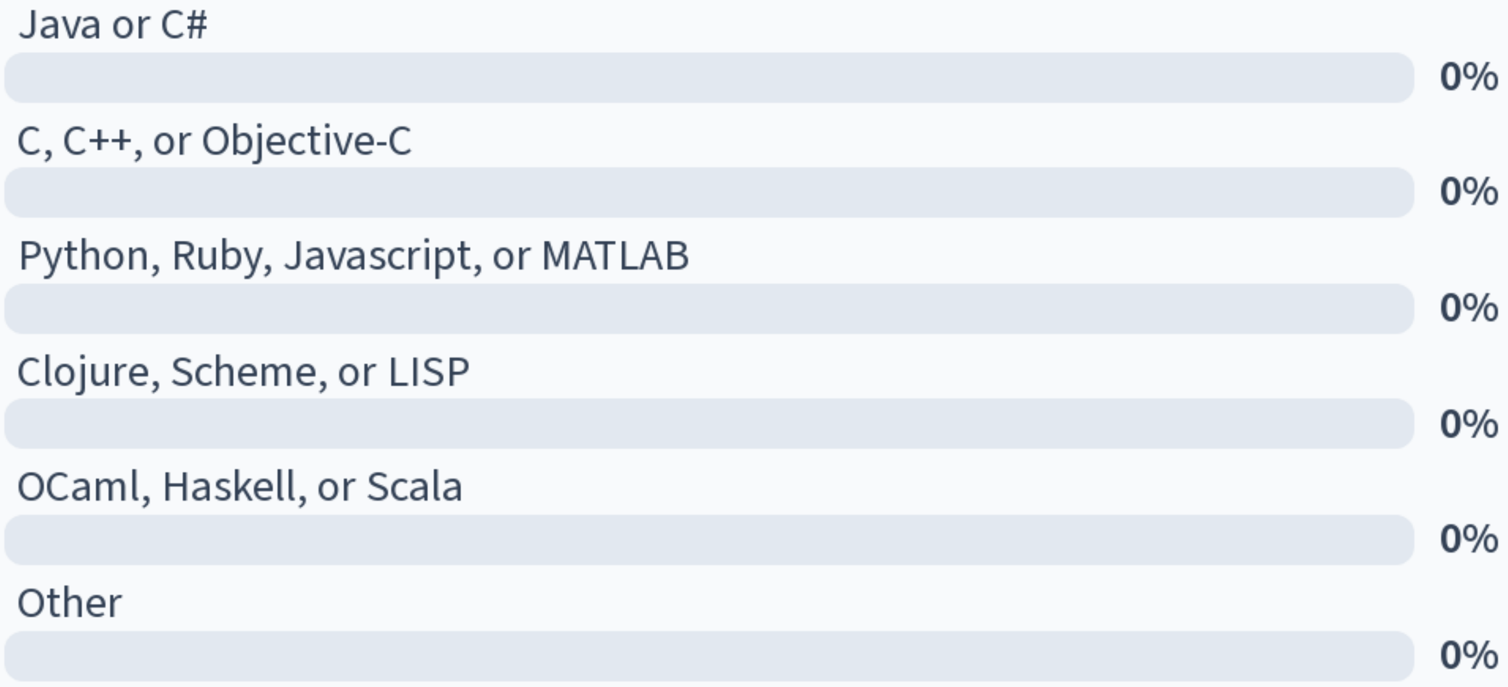
0%

No

0%

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

✓ 0



Java or C#	0%
C, C++, or Objective-C	0%
Python, Ruby, Javascript, or MATLAB	0%
Clojure, Scheme, or LISP	0%
OCaml, Haskell, or Scala	0%
Other	0%

What sort of programming experience do you have?

0

CIS 110 — 0%

High School course (incl. AP CS) — 0%

Camp or other extra-curricular — 0%

Self-taught — 0%

Other — 0%

Have you started working on HW 1?

✓ 0

Yes

0%

No

0%

Review

Value-Oriented Programming

- OCaml promotes a **value-oriented** style:

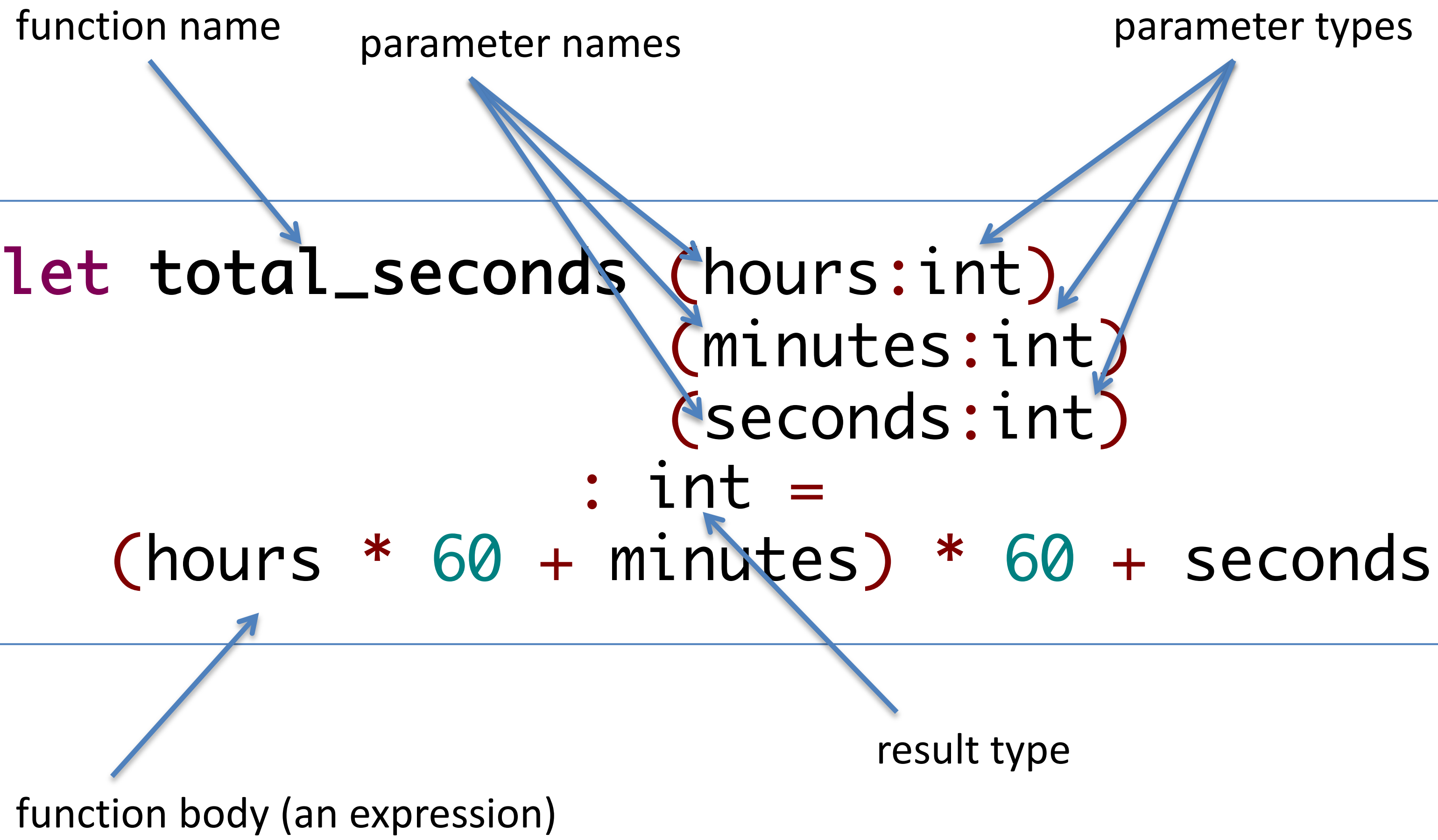
  Most of what we write is *expressions* denoting *values*

- We can visualize running an OCaml program as a sequence of *calculation* or *simplification* steps that eventually lead to a final value

```
   (300 + 12) * 60 + 17
⟼ 312 * 60 + 17
⟼ 18720 + 17
⟼ 18737
```

Functions

(Top-level) Function Declarations



```
let total_seconds (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 sequence of arguments. The whole expression is a *function application*.

```
total_seconds 5 30 22
```

(Note: the sequence of arguments is *not* parenthesized.)

Calculating With Functions

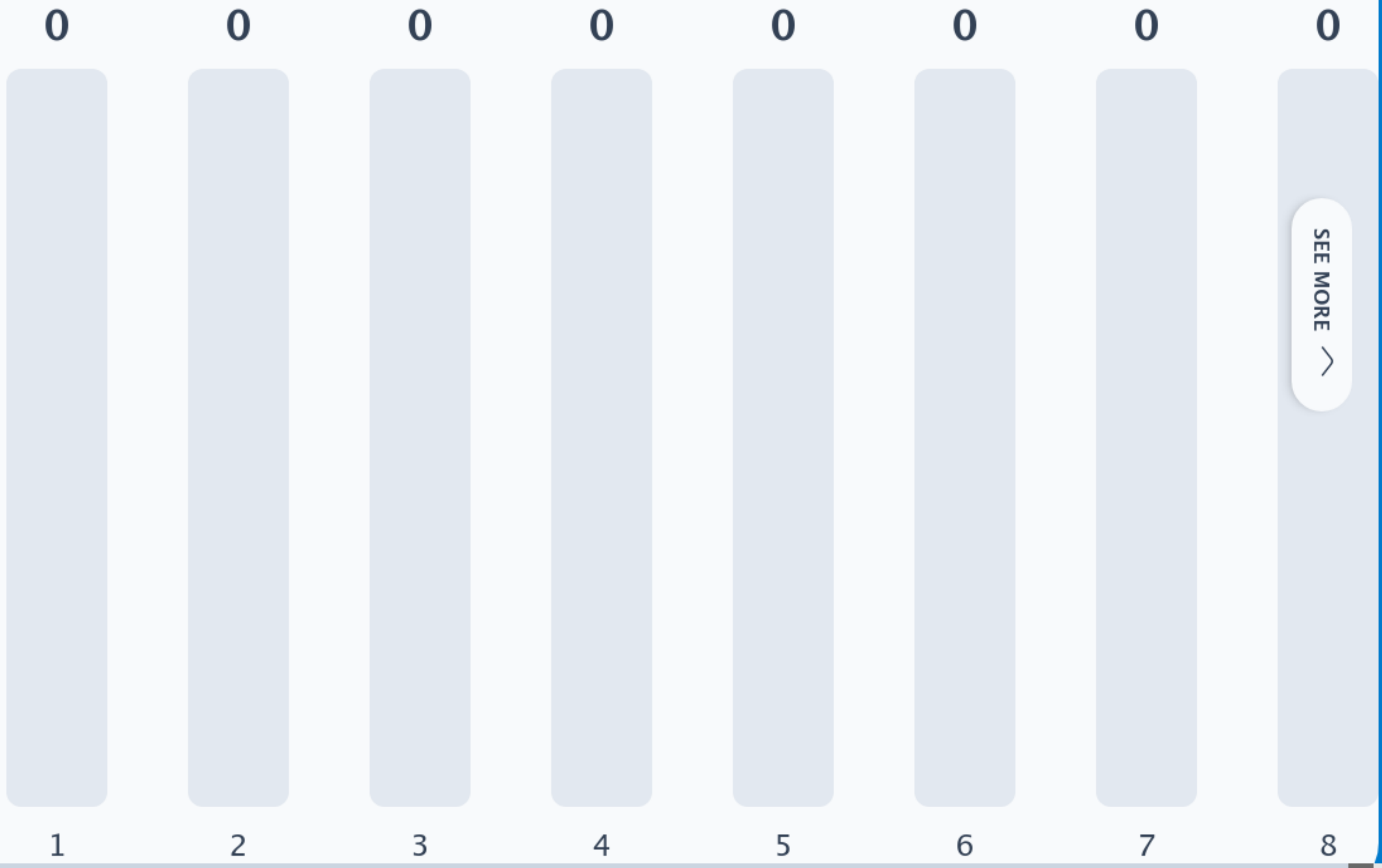
To calculate the value of a function application, first calculate values for its arguments and then *substitute* them for the parameters in the body of the function.

```
  total_seconds (2 + 3) 12 17
⟼ total_seconds 5 12 17
⟼ (5 * 60 + 12) * 60 + 17      substitute args in body
⟼ (300 + 12) * 60 + 17
⟼ 312 * 60 + 17
⟼ 18720 + 17
⟼ 18737
```

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

3. What is the value computed for 'answer'? (1 - 9)

0



What is the value computed for 'answer' in the following program? (0 .. 9)

```
let answer : int =
  let x = 3 in
  let f (y : int) = y + x in
  let x = 1 in
  f x
```

```
let answer : int =
  let f (y : int) = y + 3 in
  let x = 1 in
  f x
```

```
let answer : int =
  let f (y : int) = y + 3 in
  f 1
```

```
let answer : int =
  1 + 3
```

```
let answer : int =
  4
```

Typechecking Functions

Function types are written with “arrow” notation:

```
total_seconds : int -> int -> int -> int
```

the ones to the left of -> are *input types*

the last one is the *result type*

(We’ll have more to say about functions and their types later on…)

Typechecking Function Calls

```
total_seconds 5 20 32
```

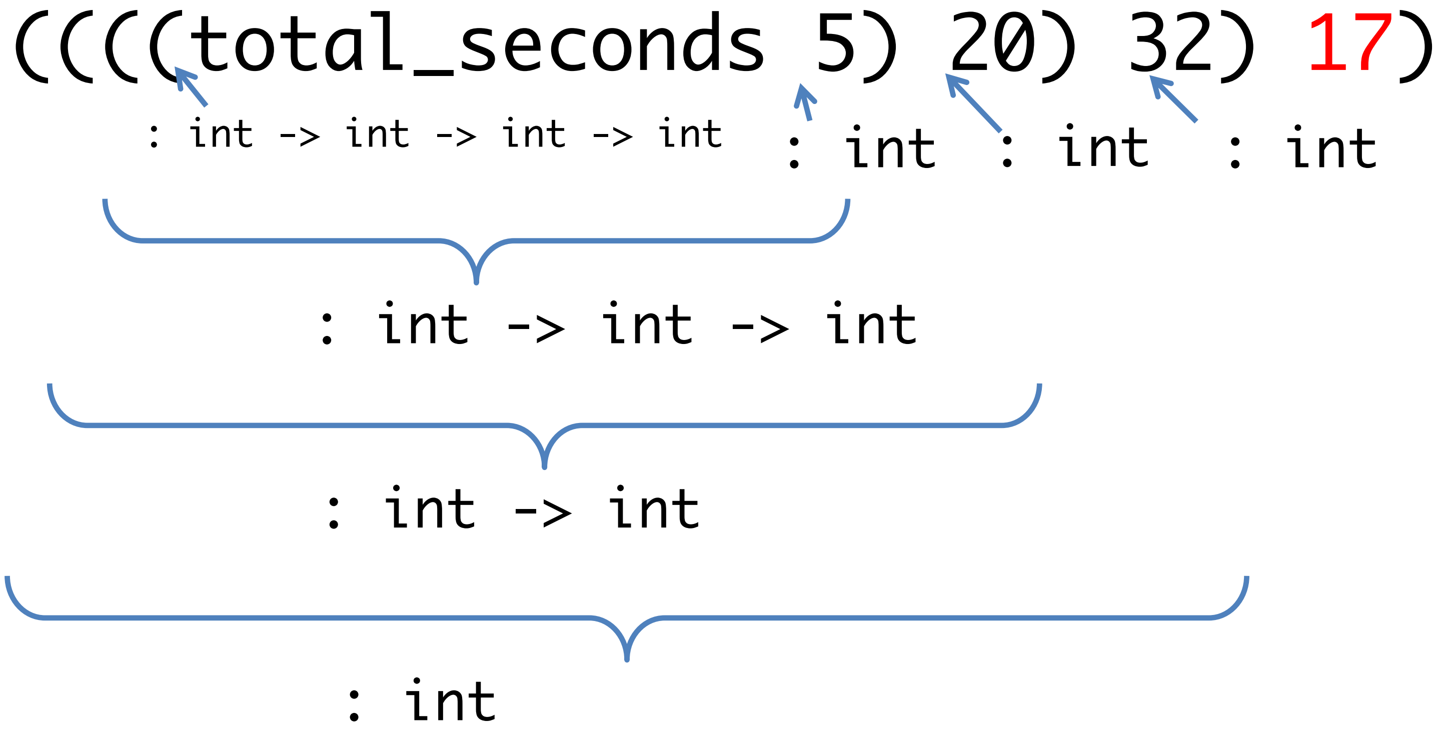
Really means

$$\underbrace{\texttt{(}\underbrace{\texttt{(}\underbrace{\texttt{(}\underset{\texttt{: int -> int -> int -> int}}{\texttt{total\_seconds}}\ \underset{\texttt{: int}}{\texttt{5}}\texttt{)}}_{\texttt{: int -> int -> int}}\ \underset{\texttt{: int}}{\texttt{20}}\texttt{)}}_{\texttt{: int -> int}}\ \underset{\texttt{: int}}{\texttt{32}}\texttt{)}}_{\texttt{: int}}$$

Applying a function matches the input type to the argument type leaving the type on the right-hand side of the ->.

Too Many Arguments = Type Error

```
total_seconds 5 20 32 17
```



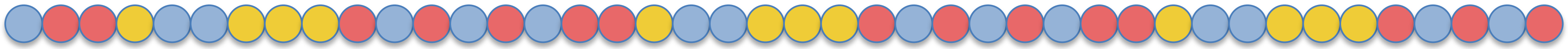
ERROR: Expected int -> int but found int!

Lists

A Value-Oriented Approach
to Sequential Data

Lists: Sequences of Data

- Often, we collect information that …
  - is ordered in some way
  - allows repeated values
  - may be of unknown size
- Examples:
  - words in a sentence: `[“the”; “quick”; “brown”; “fox”; …]`
  - DNA sequences of amino acids: `[G;A;T;T;A;C;A]`
  - phone numbers in a contacts list, voicemail list, etc.
  - options in a menu: `[Open; Save; Close; Export;…]`
  - and many others…



What is a list?

A *list value* is either:

`[]`	the *empty* list, sometimes called *nil*

or

`v :: tail`	a head value v, followed by a list of the remaining elements (the *tail*)

- Here, the infix operator ‘`::`’ *constructs* a new list from a head element and a shorter list
  - This operator is pronounced “cons” (short for “construct”)
- Importantly, *there are no other kinds of lists*

Example Lists

To build a list, we “cons together” its elements, ending with the empty list:

`1::2::3::4::[]`	a list of (four) ints
`"abc"::"xyz"::[]`	a list of (two) strings
`(false::[])::(true::[])::[]`	a list of lists that each contain booleans
`[]`	the empty list

Explicitly parenthesized

‘: :’ is a binary operator like + or ^; it takes an element and a *list* of further elements as its two inputs:

`1::(2::(3::(4::[])))`	a list of four numbers
`"abc"::("xyz"::[])`	a list of two strings
`true::[]`	a list of one boolean
`[]`	the empty list

Unlike + and ^, cons is *right associative*: a :: b :: c means a :: (b :: c) and not (a :: b) :: c

Convenient Syntax

A lighter notation: enclose a list of elements in [and] separated by ;

`[1;2;3;4]`	a list of (four) ints
`["abc";"xyz"]`	a list of (two) strings
`[[false];[true]]`	a list of two lists that each contain a single boolean
`[]`	the empty list

Convenient Syntax

The two ways of writing lists can be freely mixed.

```
1 :: [2;3;4]
```

a list of (four) ints

Some Non-Lists

These are *not* lists:

`[1;true;3;4]`	different element types*
`1::2`	2 is not a list
`3::[]::[]`	different element types

*Lists in OCaml are *homogeneous* – all of the list elements must be of the same type.

List Types

The type of lists of integers is written

```
int list
```

The type of lists of strings is written

```
string list
```

The type of lists of booleans is written

```
bool list
```

The type of lists of lists of strings is written

```
(string list) list
```

or

```
string list list
```

etc.

In OCaml, all types are "first class," so *any* type of values can be stored in a list. (We'll see more about about that in a few lectures.)

3: Which of the following expressions has type int list?

0

[3; true] 0%

[1;2;3]::[1;2] 0%

[]::[1;2]::[] 0%

(1::2)::(3::4)::[] 0%

[1;2;3;4] 0%

Which of the following expressions has the type
`int list` ?

```
1) [3; true]

2) [1;2;3]::[1;2]

3) []::[1;2]::[]

4) (1::2)::(3::4)::[]

5) [1;2;3;4]
```

Answer: 5

3: Which of the following expressions has type (int list) list?

[3; true] 0%

[1;2;3]::[1;2] 0%

[]::[1;2]::[] 0%

(1::2)::(3::4)::[] 0%

[1;2;3;4] 0%

Which of the following expressions has the type

`(int list) list ?`

```
1) [3; true]

2) [1;2;3]::[1;2]

3) []::[1;2]::[]

4) (1::2)::(3::4)::[]

5) [1;2;3;4]
```

Answer: 3

Calculating With Lists

Calculating with lists is like calculating with arithmetic expressions: just simplify each subexpression in the list expression.

(2+3)::(12 / 5)::[]	
$\longmapsto$ 5::(12 / 5)::[]	because 2+3 $\Rightarrow$ 5
$\longmapsto$ 5::2::[]	because 12/5 $\Rightarrow$ 2

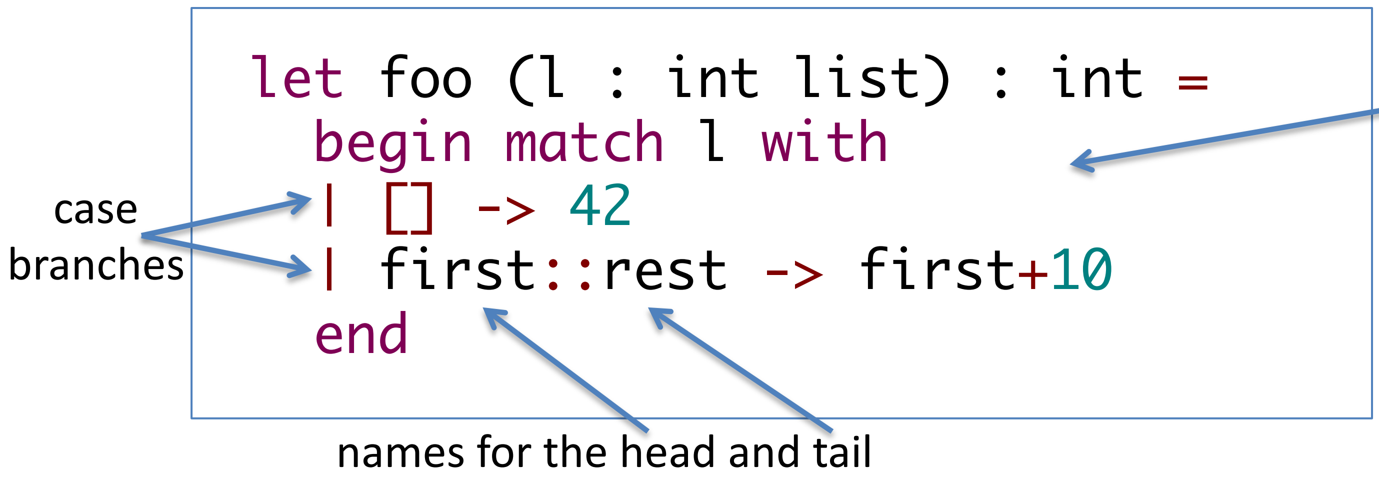
A list is a *value* whenever all of its elements are values.

Inspecting lists

- So far, we’ve seen how to *build* lists in OCaml
- To write list-processing programs, we also need to be able to *inspect* existing lists (so that we can do things with the data in them)…

Pattern Matching

OCaml provides a *pattern matching* construct for inspecting a list and giving names to its subcomponents.



```
let foo (l : int list) : int =
  begin match l with
  | [] -> 42
  | first::rest -> first+10
  end
```

match expression syntax is:

```
begin match … with
  | … -> …
  | … -> …
end
```

Case analysis is justified because there are only *two* shapes a list can have.

Note that `first` and `rest` are identifiers that are bound in the body of the branch

- `first` names the head of the list; its type is the *element* type.
- `rest` names the tail of the list; its type is the *list* type

The type of the match expression is the (one) type shared by its branches.

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match 1::(2::(3::[])) with
  | [] -> 42
  | first::rest -> first + 10
end
```

Recall: `[1;2;3]` means `1::(2::(3::[]))`

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match 1::(2::(3::[])) with
  | [] -> 42
  | first::rest -> first + 10
end
```

match checks each branch in sequence:

(1). pattern [] *does not* match 1::(2::(3::[]))

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match 1::(2::(3::[])) with
   | [] -> 42
   | first::rest -> first + 10
end
```

match checks each branch in sequence:

~~(1). pattern [] *does not* match 1::(2::(3::[]))~~

(2). pattern `first::rest` *does* match 1::(2::(3::[]))

`first` = 1

`rest` = (2::(3::[]))

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

$\longmapsto$

1 + 10

$\longmapsto$

11

match checks each branch in sequence:

~~(1). pattern [] *does not* match 1::(2::(3::[]))~~

(2). pattern `first::rest` *does* match 1::(2::(3::[]))

`first` = 1

`rest` = (2::(3::[]))

...so: substitute in that branch.

Recursion

The Inductive Nature of Lists

A list value is either:

`[]` the *empty* list, sometimes called *nil*

or

`v :: tail` a *head* value v, followed by a list value containing the remaining elements, the *tail*

- Why is this well-defined? The definition of list mentions 'list'!
- Solution: 'list' is *inductive*:
  - The empty list [] is the (only) list of 0 elements
  - To construct a list of n+1 elements, add a head element to an *existing* list of n elements
  - The set of list values contains *all and only* values constructed this way
- Corresponding computation principle: *recursion*

Recursion

Recursion principle:

Compute a function value for a given input by combining the results for strictly smaller parts of the input.

– The structure of the computation follows the inductive structure of the input.

- Example:

```
length (1::2::3::[])  =  1 + length (2::3::[])
length (2::3::[])     =  1 + length (3::[])
length (3::[])        =  1 + length []
length []             =  0
```

Recursion Over Lists

The function calls itself *recursively,* so the function declaration must be marked with rec.

Lists are either empty or nonempty; *pattern matching* determines which.

```
let rec length (l : string list) : int =
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```

If the list is non-empty, then “x” is the first string in the list and “rest” is the remainder of the list.

Structural Recursion Over Lists

Structural recursion builds up an answer from answers for smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … f rest …
  end
```

The branch for `[]` calculates the value (`f []`) directly

- this is the *base case* of the recursion

The branch for `hd::rest` calculates `f (hd::rest)` given hd and (`f rest`).

- this is the *inductive case* of the recursion

Calculating with pattern matching and recursion

Calculating with Recursion

```
   length ["a"; "b"]
↦     (substitute the list for l in the function body)
      begin match "a"::"b"::[] with
      | [] -> 0
      | ( x :: rest ) -> 1 + length rest
      end
↦     (second case matches with rest = "b"::[])
   1 + length ("b"::[])
↦     (substitute the list for l in the function body)
   1 + begin match "b"::[] with
         | [] -> 0
         | ( x :: rest ) -> 1 + length rest
         end
↦     (second case matches again, with rest = [])
   1 + (1 + length [])
↦     (substitute [] for l in the function body)
   …
↦  1 + 1 + 0 ⇒ 2
```

```
let rec length (l:string list) :
int=
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```