Save the Environment!

A closure is a pair of an environment and an association of a pattern (e.g. \((v_1,\ldots,v_n)\) giving the input variables) with an expression (the function body), written:

\[< (v_1,\ldots,v_n) \rightarrow \text{exp}, \rho >\]

Where \(\rho\) is the environment in effect when the function is defined (for a simple function)

Evaluating declarations

- Evaluation uses an environment \(\rho\)
- To evaluate a (simple) declaration \(\text{let x = e}\)
  - Evaluate expression \(e\) in \(\rho\) to value \(v\)
  - Update \(\rho\) with \(x\) \(v\): \(\{x \rightarrow v\} + \rho\)
- Update: \(\rho_1 + \rho_2\) has all the bindings in \(\rho_1\) and all those in \(\rho_2\) that are not rebound in \(\rho_1\)
  \(\{x \rightarrow 2, y \rightarrow 3, a \rightarrow "hi"\} + \{y \rightarrow 100, b \rightarrow 6\}\)
  \(= \{x \rightarrow 2, y \rightarrow 3, a \rightarrow "hi", b \rightarrow 6\}\)

Evaluating expressions in OCaml

- Evaluation uses an environment \(\rho\)
- A constant evaluates to itself, including primitive operators like + and =
- To evaluate a variable, look it up in \(\rho\): \(\rho(v)\)
- To evaluate a tuple \((e_1,\ldots,e_n)\), evaluate each \(e_i\) to \(v_i\), right to left for OCaml
  - Then make value \((v_1,\ldots,v_n)\)

Evaluation of Application with Closures

- Given application expression \(f\ e\)
- In OCaml, evaluate \(e\) to value \(v\)
- In environment \(\rho\), evaluate left term to closure, \(c = <(x_1,\ldots,x_n) \rightarrow b, \rho'>\)
  - \((x_1,\ldots,x_n)\) variables in (first) argument
  - \(v\) must have form \((v_1,\ldots,v_n)\)
- Update the environment \(\rho'\) to \(\rho'' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'\)
- Evaluate body \(b\) in environment \(\rho''\)
Recursive Functions

```ocaml
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive and Induction

```ocaml
# let rec nthsq n =
    match n with
    | 0 -> 0
    | n -> (2 * n - 1) + nthsq (n - 1);
val nthsq : int -> int = <fun>
# nthsq 3;;
- : int = 9
```

Lists

```ocaml
# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```

Recursion Example

```ocaml
Compute n^2 recursively using:
\[ n^2 = (2 * n - 1) + (n - 1)^2 \]

# let rec nthsq n =
    match n with
    | 0 -> 0
    | n -> (2 * n - 1) + nthsq (n - 1);
val nthsq : int -> int = <fun>
# nthsq 3;;
- : int = 9
```

Lists are Homogeneous

```ocaml
# let bad_list = [1; 3.2; 7];;
Characters 19-22:
let bad_list = [1; 3.2; 7];;;
^^^^
This expression has type float but is here used with type int
```
Question

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [“hi”; “there”; [“wahcha”]; [ ]; [“doin”]]

Answer

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [“hi”; “there”; [“wahcha”]; [ ]; [“doin”]]

3 is invalid because of last pair

Functions Over Lists

```ocaml
# let rec double_up list =
  match list
  with [ ] -> [ ]  (* pattern before ->, expression after *)
  | (x :: xs) -> (x :: x :: double_up xs);
val double_up : 'a list -> 'a list = <fun>

# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1; 1]

# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]

# let rec poor_rev list =
  match list
  with [] -> []
  | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

# poor_rev silly;;
-: string list = ["there"; "there"; "hi"; "hi"]
```

Structural Recursion

Functions on recursive datatypes (eg lists) tend to be recursive

Recursion over recursive datatypes generally by structural recursion

- Recursive calls made to components of structure of the same recursive type
- Base cases of recursive types stop the recursion of the function

Question: Length of list

Problem: write code for the length of the list

How to start?

let rec length list =
**Question: Length of list**

- Problem: write code for the length of the list
  - How to start?

```ocaml
let rec length list =
  match list with
  [] ->
  | a :: bs ->
```

- What patterns should we match against?

```ocaml
let rec length list =
  match list with
  [] -> 0
  | a :: bs ->
```

- What result do we give when list is empty?

```ocaml
let rec length list =
  match list with
  [] -> 0
  | a :: bs -> 1 + length bs
```
**Structural Recursion: List Example**

```ocaml
# let rec length list = match list with [ ] -> 0 (* Nil case *) | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case `[]` is base case
- Cons case recurses on component list `bs`

**Same Length**

- How can we efficiently answer if two lists have the same length?

```ocaml
let rec same_length list1 list2 = match list1 with [ ] -> true | (y::ys) -> false | (x::xs) -> (match list2 with [ ] -> false | (y::ys) -> same_length xs ys)
```

**Your turn: doubleList : int list -> int list**

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```ocaml
let rec doubleList list = match list with [ ] -> [] | x :: xs -> (2 * x) :: doubleList xs
```

**Your turn: doubleList : int list -> int list**

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2
Higher-Order Functions Over Lists

```ocaml
# let rec map f list =
  match list with [ ] -> [] | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```ocaml
# map plus_two fib5;;
: int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x -1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list =
  List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
```

```ocaml
# doubleList [2;3;4];;
:- int list = [4; 6; 8]
```

Folding Recursion

- Another common form “folds” an operation over the elements of the structure

```ocaml
# let rec length list =
  match list with [ ] -> 0 | a :: bs -> 1 + length bs;;
val length : 'a list -> int = <fun>
```

```ocaml
# length [5; 4; 3; 2];;
:- int = 4
```

What do `multList` and `length` have in common?