Evaluating declarations

- Evaluation uses an environment $\rho$
- To evaluate a (simple) declaration $\text{let } x = e$
- Evaluate expression $e$ in $\rho$ to value $v$
- Then update $\rho$ with $x v$: $\{x \to v\} + \rho$

Evaluating expressions

- Evaluation uses an environment $\rho$
- A constant evaluates to itself
- To evaluate a variable, look it up in $\rho$ ($\rho(v)$)
- To evaluate uses of $+, -, \text{ etc}$, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: $\text{let } x = e_1 \text{ in } e_2$
  - Eval $e_1$ to $v$, then eval $e_2$ using $\{x \to v\} + \rho$

Eval of App with Closures in OCaml

1. Evaluate the right term to values, $(v_1, \ldots, v_n)$
2. In environment $\rho$, evaluate left term to closure, $c = \langle x_1, \ldots, x_n \to b, \rho \rangle$
3. Match $(x_1, \ldots, x_n)$ variables in (first) argument with values $(v_1, \ldots, v_n)$
4. Update the environment $\rho$ to $\rho' = \{x_1 \to v_1, \ldots, x_n \to v_n\} + \rho$
5. Evaluate body $b$ in environment $\rho'$

OCaml Example 1

```ocaml
# (print_string "a";
     (fun x -> (print_string "b";
                    (fun y -> (print_string "c";
                                  x + y))))));

(print_string "d"; 3)
(print_string "e"; 5));;
```

edabc- : int = 8

#
Your turn now

Try Problem 1 on HW3

# let f = (print_string "a"; (fun x -> (print_string "b"; (fun y -> (print_string "c"; x + y)))))) in
let u = (print_string "d"; 3) in
let g = f u in
let v = (print_string "e"; 5) in g v;;
adbec- : int = 8

Higher Order Functions

- A function is higher-order if it takes a function as an argument or returns one as a result
- Example:
  # let compose f g = fun x -> f (g x);;
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
- The type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b is a higher order type because of ('a -> 'b) and ('c -> 'a) and -> 'c -> 'b

Thrice

- Recall:
  # let thrice f x = f (f (f x));;
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
- How do you write thrice with compose?
  # let thrice f x = f (f (f x));;
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  How do you write thrice with compose?
  # let thrice f = compose f (compose f f);;
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  Is this the only way?
Partial Application

```
# (+);;
- : int -> int -> int = <fun>
# (+) 2 3;;
- : int = 5
# let plus_two = (+) 2;;
val plus_two : int -> int = <fun>
# plus_two 7;;
- : int = 9
```

Partial application also called sectioning

Lambda Lifting

```
You must remember the rules for evaluation when you use partial application

# let add_two = (+) (print_string "test\n"; 2);;
val add_two : int -> int = <fun>
# let add2 =     (* lambda lifted *)
    fun x -> (+) (print_string "test\n"; 2) x;;
val add2 : int -> int = <fun>
```

Lambda Lifting delayed the evaluation of the argument to (+) until the second argument was supplied

Partial Application and “Unknown Types”

```
Recall compose plus_two:
# let f1 = compose plus_two;;
val f1 : ('_a -> int) -> '_a -> int = <fun>
# let f2 = fun g ->  compose plus_two g;;
val f2 : ('a -> int) -> 'a -> int = <fun>
```

'a can only be instantiated once for an expression

```
# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
Characters 3-14:
    ^^^^^^^^^^^^^
This expression has type 'a list -> int but is here used with type int -> int
```

Partial Application and “Unknown Types”

```
a can be repeatedly instantiated

# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : '_a list -> int = <fun>
```
Your turn now

Try Problem 2 on HW3

Lists
- First example of a recursive datatype (aka algebraic datatype)

- Unlike tuples, lists are homogeneous in type (all elements same type)

Lists
- List can take one of two forms:
  - Empty list, written \([\ ]\)
  - Non-empty list, written \(x \cdot \cdot xs\)
    - \(x\) is head element, \(xs\) is tail list, \(\cdot \cdot\) called "cons"
  - Syntactic sugar: \([x] = x \cdot \cdot [\ ]\)
  - \([x1; x2; ...; xn] = x1 \cdot x2 \cdot ... \cdot xn \cdot [\ ]\)

Lists are Homogeneous

- 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”]]

# 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]

# 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

# let bad_list = [1; 3.2; 7];;
let bad_list = [1; 3.2; 7];;
                             ^^^^
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

# 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]

Functions Over Lists

# 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']

Question: Length of list

Problem: write code for the length of the list

How to start?

let rec length l =
match l with

Question: Length of list

Problem: write code for the length of the list

What patterns should we match against?

let rec length l =
match l with

Question: Length of list
- Problem: write code for the length of the list
  - What patterns should we match against?
  let rec length l =
      match l with [] ->
      | (a :: bs) ->

Question: Length of list
- Problem: write code for the length of the list
  - What result do we give when \( l \) is empty?
  let rec length l =
      match l with [] ->
      | (a :: bs) ->

Question: Length of list
- Problem: write code for the length of the list
  - What result do we give when \( l \) is not empty?
  let rec length l =
      match l with [] -> 0
      | (a :: bs) -> 1 + length bs

Your turn now
Try Problem 1 on MP3
How can we efficiently answer if two lists have the same length?

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

Functions on recursive datatypes (e.g., 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.

In Structural Recursion, split input into components and (eventually) recurse.

Forward Recursion form of Structural Recursion:
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results.
- Wait until whole structure has been traversed to start building answer.

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

Forward Recursion: Examples:

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

```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>

# 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]
```

Question

- How do you write length with forward recursion?

```ocaml
let rec length l =
  match l with [] -> 0
  | (a :: bs) -> 1 + length bs
```

Your turn now

Try Problem 8 on MP3
An Important Optimization

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished.
- What if \( f \) calls \( g \) and \( g \) calls \( h \), but calling \( h \) is the last thing \( g \) does (a tail call)?

Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls.
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls.
- Tail recursion generally requires extra “accumulator” arguments to pass partial results.
- May require an auxiliary function.

Example of Tail Recursion

```ocaml
# let rec prod l = 
  match l with 
  | [] -> 1 
  | x :: rem -> x * prod rem;;
val prod : int list -> int = <fun>
# let prod list = 
  let rec prod_aux l acc = 
    match l with 
    | [] -> acc 
    | y :: rest -> prod_aux rest (acc * y) (* Uses associativity of multiplication *) 
  in prodAux list;;
val prod : int list -> int = <fun>
```

Question

- How do you write length with tail recursion?

```ocaml
let length l = 
  let rec length_aux list n = 
  in
```
How do you write length with tail recursion?

```ocaml
let rec length_aux list n =
  match list with [] -> n
  | (a :: bs) -> length_aux bs (n + 1)
in
```

9/11/14 60
Your turn now

Try Problem 10 on MP3

Mapping 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>
# 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

- One common form of structural recursion applies a function to each element in the structure

```ocaml
# let rec doubleList list = match list 
  with [ ] -> [] 
  | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

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>
# 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 sumList list = match list with 
  [ ] -> 0 
  | x::xs -> x + sumList xs;;
val sumList : int list -> int = <fun>
# sumList [2;4;6];;
- : int = 48
```

Computes $(2 * (4 * (6 * 1)))$
Iterating over lists

```ocaml
# let rec fold_right f list b = 
  match list 
  with [] -> b 
  | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# fold_right 
  (fun s -> fun () -> print_string s) 
  ['hi'; 'there'] 
  ();;
therehi- : unit = ()
```

Folding Recursion

- multList folds to the right
- Same as:

```ocaml
# let multList list =
  List.fold_right 
  (fun x -> fun p -> x * p)
  list 1;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

Encoding Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with 
  [] -> list2 | x::xs -> append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# let append list1 list2 = 
  fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Question

- How do you write length with fold_right, but no explicit recursion?

```ocaml
let rec length l = 
  match l with 
  [] -> 0 | (a :: bs) -> 1 + length bs
```

Map from Fold

```ocaml
# let map f list =
  fold_right (fun x y -> f x :: y) list [ ];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map ((+)1) [1;2;3];;
- : int list = [2; 3; 4]
```

- Can you write fold_right (or fold_left) with just map? How, or why not?
Iterating over lists

```ocaml
# let rec fold_left f a list =
  match list
  with [] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# fold_left
  (fun () -> print_string) ()
['hi'; "there"];;
```

Encoding Tail Recursion with fold_left

```ocaml
# let prod list = let rec prod_aux l acc =
  match l with [] -> acc
  | (y :: rest) -> prod_aux rest (acc * y)
  in prod_aux list 1;;
val prod : int list -> int = <fun>

# prod [4;5;6];;
- : int = 120
```

Question

let length l =
  let rec length_aux list n =
  match list with [] -> n
  | (a :: bs) -> length_aux bs (n + 1)
in length_aux l 0

How do you write length with fold_left, but no explicit recursion?

Folding

```ocaml
# let rec fold_left f a list = match list
  with [] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

fold_left f a [x_1; x_2;...;x_n] = f(…(f (f a x_1) x_2)…)x_n

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

fold_right f [x_1; x_2;...;x_n] b = f x_1(f x_2(…(f x_n b)…))
```

Recall

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

poor_rev x@ [x] = x
```

What is its running time?
Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

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

Tail Recursion - Example

```ocaml
# let rec rev_aux list revlist = 
  match list with [] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list -> 'a list = <fun>

# let rev list = rev_aux list [ ];
val rev : 'a list -> 'a list = <fun>
```

What is its running time?

Comparison

- poor_rev [1,2,3] =
- (poor_rev [2,3]) @ [1] =
- (((poor_rev [3]) @ [2]) @ [1]) =
- ((([] @ [3]) @ [2]) @ [1]) =
- ([3] @ [2]) @ [1] =
- (3::([ ] @ [2])) @ [1] =
- [3,2] @ [1] =
- 3 :: ([ ] @ [1]) =
- 3 :: (2::([ ] @ [1])) = [3, 2, 1]

Folding - Tail Recursion

- # let rev list =
-     fold_left
-     (fun l -> fun x -> x :: l)  //comb op
-     []                     //accumulator cell
-     list

Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
- Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition