Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration `let x = e`
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \)

- Update: \( \rho_1 \cup \rho_2 \) 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

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself
- To evaluate an variable, look it up in \( \rho (v) \)
- To evaluate uses of +, -, etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: `let x = e1 in e2`
  - Eval \( e_1 \) to \( v \), eval \( e_2 \) using \( \{x \rightarrow v\} + \rho \)

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, \( c = \langle x_1, \ldots, x_n \rightarrow b, \rho \rangle \)
3. Match \( (x_1, \ldots, x_n) \) variables in (first) argument with values \( (v_1, \ldots, v_n) \)
4. Update the environment \( \rho \) to \( \rho' = \{x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n\} + \rho \)
5. Evaluate body \( b \) in environment \( \rho' \)

Match Expressions

```ocaml
# let triple_to_pair triple =
  match triple
  with
  (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);
val triple_to_pair : int * int * int -> int * int = <fun>
```

Higher Order Functions

- A function is higher-order if it takes a function as an argument or returns one as a result
- Example:
  ```ocaml
  # 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 = compose f (compose f f);;
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  ```

**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**

- ```
  # thrice add_two 5;;
  - : int = 11
  # thrice add2 5;;
  test
  test
  test
  - : int = 11
  ```

- 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>
  ```

- Compare to lambda lifted version:
  ```
  # let f2 = fun g -> compose plus_two g;;
  val f2 : ('a -> int) -> 'a -> int = <fun>
  ```

- What is the difference?
9/8/15 13

Partial Application and “Unknown Types”

- 'a can only be instantiated once for an expression
  
```
# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
Characters 3-14:
f1 List.length;;
```

This expression has type 'a list -> int but is here used with type int -> int

9/8/15 14

Partial Application and “Unknown Types”

- 'a can be repeatedly instantiated
  
```
# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : '_a list -> int = <fun>
```

9/8/15 15

Recursive Functions

```
# let rec factorial n = 
    if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
# (* rec  is needed for recursive function declarations *)
```

9/8/15 16

Recursion Example

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

```
# let rec nthsq n =         (* rec for recursion *)
    match n              (* pattern matching for cases *)
    with 0 -> 0                  (* base case *)
    | n -> (2 * n -1) + nthsq (n -1);;   (* recursive call *)
val nthsq : int -> int = <fun>
# nthsq 3;;
- : int = 9
```

Structure of recursion similar to inductive proof

9/8/15 17

Recursion and Induction

```
# let rec nthsq n = match n with 0 -> 0
    | n -> (2 * n - 1) + nthsq (n - 1) ;;
```

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- if or match must contain base case
- Failure of these may cause failure of termination

9/8/15 18

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 \(x :: xs\)
    - x is head element, xs is tail list, :: called “cons”
    - Syntactic sugar: \([x]\) == \(x :: [\ ]\)
    - \([x1; x2; \ldots; xn]\] == \(x1 :: x2 :: \ldots :: xn :: [\ ]\)

# 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"]\)
  - 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; 3; 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
        | [] -> 0
        | (a :: bs) -> length bs + 1

What result do we give when \( l \) is empty?

let rec length l =
    match l with
        | [] -> 0
        | (a :: bs) -> length bs + 1
Question: Length of list

Problem: write code for the length of the list

What result do we give when \( l \) is not empty?

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

---

Your turn now

Try Problem 1 on MP2

---

Same Length

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)
```

---

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
### Structural Recursion: List Example

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

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

### Forward Recursion

- 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

### Forward Recursion: Examples

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

### Question

- How do you write length with forward recursion?
  - Let rec length l =
    match l with [] ->
    | (a :: bs) ->
    length bs
Question

- How do you write length with forward recursion?

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

Your turn now

Try Problem 2 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)?
- Then \( h \) can return directly to \( f \) instead of \( g \)

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

```
# 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 prod_aux list 1;;
val prod : int list -> int = <fun>
```
How do you write length with tail recursion?

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

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

How do you write length with tail recursion?
let length l =
  let rec length_aux list n =
    match list with [] -> n
    | (a :: bs) -> length_aux bs (n + 1)
in

9/8/15 55

Question

How do you write length with tail recursion?
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

9/8/15 56

Your turn now

Try Problem 4 on MP3

9/8/15 57

Mapping Recursion

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

9/8/15 58

Mapping Functions Over Lists

# 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;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
- : int list = [12; 7; 4; 2; 1; 0; 0]

9/8/15 59

Mapping Recursion

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

9/8/15 60
**Folding Recursion**

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

```
# let rec multList list = match list
with [ ] -> 1
| x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

*Computes \(2 \times (4 \times (6 \times 1))\)*

---

**Folding Functions over Lists**

How are the following functions similar?

```
# let rec sumlist list = match list with
[ ] -> 0 | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let rec prodlist list = match list with
[ ] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```

---

**Iterating over lists**

```
# 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 = ()
```

---

**Encoding Recursion with Fold**

```
# let rec append list1 list2 = match list1 with
[ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

<table>
<thead>
<tr>
<th>Base Case</th>
<th>Operation</th>
<th>Recursive Call</th>
</tr>
</thead>
</table>

```
# 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]
```

---

**Folding Recursion**

- multList folds to the right
- Same as:

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

---

**Question**

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

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

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

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

let length list = 
    List.fold_right (fun x -> fun n ->  n + 1) list 0
```

Map from Fold

```ocaml
# let map f list = 
  fold_right (fun x -> fun 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"];;
hithere- : unit = ()
```

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

```ocaml
let rec 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?
```

```ocaml
let length list = 
    List.fold_left (fun n -> fun x -> n + 1) 0 list
```
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 [x1; x2;...;xn] = f(...(f (f a x1) x2)...xn)
```

Recall

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

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

Comparison

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

Comparison

- rev [1,2,3] =
- rev_aux [1,2,3] [ ] =
- rev_aux [2,3] [1] =
- rev_aux [3] [2,1] =
- rev_aux [ ] [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

Continuation Passing Style

- A programming technique for all forms of “non-local” control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it’s a higher-order function version of GOTO

Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure “returns” no result
- Function receiving the result called a continuation
- Continuation acts as “accumulator” for work still to be done

Example of Tail Recursion

# let rec app fl x =
  match fl with
  | [] -> x
  | (f :: rem_fs) -> f (app rem_fs x);
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let app fs x =
  let rec app_aux fl acc =
    match fl with
    | [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
      (fun z -> acc (f z))
in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>

Continuation Passing Style

- Writing procedures so that they take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)
**Example of Tail Recursion & CSP**

```
let app fs x =
    let rec app_aux fl acc=
        match fl with 
        | [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
          (fun z -> acc (f z))
in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

**Continuation Passing Style**

- A compilation technique to implement non-local control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics

**Terms**

- A function is in Direct Style when it returns its result back to the caller.
- A Tail Call occurs when a function returns the result of another function call without any more computations (e.g., tail recursion).
- A function is in Continuation Passing Style when it passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function.

**Example**

- Simple reporting continuation:
  ```
  let report x = (print_int x; print_newline( ));
  val report : int -> unit = <fun>
  ```
- Simple function using a continuation:
  ```
  let plusk a b k = k (a + b)
  val plusk : int -> int -> (int -> 'a) -> 'a = <fun>
  let plusk 20 22 report;;
  42
  - : unit = ()
  ```

**Simple Functions Taking Continuations**

- Given a primitive operation, can convert it to pass its result forward to a continuation.
- Examples:
  ```
  let subk x y k = k(x + y);
  val subk : int -> int -> (int -> 'a) -> 'a = <fun>
  let eqk x y k = k(x = y));
  val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
  let timesk x y k = k(x * y));
  val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
  ```

**Nesting Continuations**

- Examples:
  ```
  let add_three x y z = x + y + z;;
  val add_three : int -> int -> int -> int = <fun>
  let add_three_k x y z k = addk x y z k =
    addk x y (fun p -> addk p z k);;
  val add_three_k : int -> int -> int -> (int -> 'a) -> 'a = <fun>
  ```