Booleans (aka Truth Values)

```ocaml
# true;;
- : bool = true

# false;;
- : bool = false

// \( \rho_1 = \{ c \to 4, \text{test} \to 3.7, a \to 1, b \to 5 \} \)
# if b > a then 25 else 0;;
- : int = 25
```

Booleans and Short-Circuit Evaluation

```ocaml
# 3 > 1 && 4 > 6;;
- : bool = false

# 3 > 1 || 4 > 6;;
- : bool = true

# (print_string "Hi\n"; 3 > 1) || 4 > 6;;
Hi
- : bool = true

# 3 > 1 || (print_string "Bye\n"; 4 > 6));;
- : bool = true

# not (4 > 6));;
- : bool = true
```

Tuples as Values

```ocaml
// \( \rho_{a} = \{ c \to 4, a \to 1, b \to 5 \} \)
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

// \( \rho_{p} = \{ s \to (5, "hi", 3.2), c \to 4, a \to 1, b \to 5 \} \)
```

Pattern Matching with Tuples

```ocaml
// \( \rho = \{ s \to (5, "hi", 3.2), a \to 1, b \to 5, c \to 4 \} \)

# let (a,b,c) = s;; (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2

# let (a, _, _) = s;;
val a : int = 5

# let x = 2, 9.3;; (* tuples don't require parens in Ocam *)
val x : int * float = (2, 9.3)
```

Nested Tuples

```ocaml
# (*Tuples can be nested *)
# let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float = ((1, 4, 62), ("bye", 15), 73.95)

# (*Patterns can be nested *)
# let (p, (st, _)) = d;;
(* _ matches all, binds nothing *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```
Functions on tuples

```plaintext
let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>

let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
```

Save the Environment!

- A closure is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:

  `< (v1,...,vn) -> exp, ρ >`

- Where ρ is the environment in effect when the function is defined (for a simple function)

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined and recall
  ```plaintext
  let plus_pair (n,m) = n + m;;
  val plus_pair : int * int -> int = <fun>
  ```

- Closure for fun (n,m) -> n + m:
  ```plaintext
  <(n,m) -> n + m, ρ_{plus_pair}>
  ```

- Environment just after plus_pair defined:
  ```plaintext
  \{plus_pair -> <(n,m) -> n + m, ρ_{plus_pair} >} + ρ_{plus_pair}
  ```

Functions with more than one argument

```plaintext
let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>

let t = add_three 6 3 2;;
val t : int = 11
```

Curried vs Uncurried

- Recall
  ```plaintext
  let add_three u v w = u + v + w;;
  val add_three : int -> int -> int -> int = <fun>
  ```

- How does it differ from
  ```plaintext
  let add_triple (u,v,w) = u + v + w;;
  val add_triple : int * int * int -> int = <fun>
  ```

- add_three is curried;
- add_triple is uncurried
Partial application of functions

```ocaml
let add_three x y z = x + y + z;;
```

```ocaml
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

Partial application also called sectioning

Recall: let plus_x = fun x => y + x

```
let x = 12
let plus_x = fun y => y + x
let x = 7
```

Closure for `plus_x`

- When `plus_x` was defined, had environment:
  \[ \rho_{plus_x} = \{ \ldots, x \mapsto 12, \ldots \} \]
- Recall: let `plus_x y = y + x`
  is really let `plus_x = fun y => y + x`
- Closure for `fun y => y + x`:
  \[ <y \mapsto y + x, \rho_{plus_x}> \]
- Environment just after `plus_x` defined:
  \[ \{ plus_x \mapsto <y \mapsto y + x, \rho_{plus_x}>, \rho_{plus_x} \} + \rho_{plus_x} \]

Evaluation

- Running Ocaml source:
  - Parse the program to detect each expression
  - Keep an internal environment at each time step
  - For each expression, interpret the program using the internal function `Eval`
  - Nice property of Ocaml: everything is an expression!

- How does (Eval expression environment) work:
  - Evaluation uses a starting environment \( \rho \)
  - Define the rules for evaluating declarations, constants, arithmetic expressions, function applications…

Evaluating Declarations

- Evaluation uses a starting environment \( \rho \)
- To evaluate a (simple) declaration `let x = e`
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with the mapping from \( x \) to \( v \): \( \{ x \mapsto v \} + \rho \)

- **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 \mapsto 2, y \mapsto 3, a \mapsto "hi!" \}
  + \{ y \mapsto 100, b \mapsto 6 \}
  = \{ x \mapsto 2, y \mapsto 3, a \mapsto "hi!", b \mapsto 6 \}
  ```

- It is not commutative!
Evaluating Expressions

- Evaluation uses an environment $\rho$

- **A constant** evaluates to itself

- To evaluate an **variable**, look it up in $\rho$ i.e., use $\rho(v)$

- To evaluate uses of $+,$ $-$, etc., first eval the arguments, then do operation

- To evaluate a **local declaration**: let $x = e_1$ in $e_2$
  - Evaluate $e_1$ to $v$, evaluate $e_2$ using $\{x \mapsto v\} + \rho$

Evaluating Application with Closures

- **Function defined as**: let $f \ x_1 \ldots \ x_n = \text{body}$

- **Function application**: $f \ e_1 \ldots \ e_n$

- Evaluation uses the function $\text{App}$:
  - In environment $\rho$, evaluate left term to closure, $c = \langle(x_1,\ldots,x_n) \mapsto \text{body}, \rho\rangle$
  - Evaluate the arguments in the application $e_1 \ldots e_n$ to their values $v_1,\ldots,v_n$ in the environment $\rho$
  - Update the environment $\rho$ to $\rho' = \{x_1 \mapsto v_1,\ldots,x_n \mapsto v_n\} + \rho$

Evaluation of Application of $\text{plus}_x$;

- Have environment:
  
  $$\rho = \{\text{plus}_x \mapsto \langle y \mapsto y + x, \rho_{\text{plus}_x} \rangle \mapsto \ldots, y \mapsto 3, \ldots\}$$

  where $\rho_{\text{plus}_x} = \{x \mapsto 12, \ldots, y \mapsto 24, \ldots\}$

  - Eval ($\text{plus}_x \ y, \rho$) rewrites to $\text{App} (\text{Eval} (\text{plus}_x, \rho), \text{Eval} (y, \rho))$
  - App ($\langle y \mapsto y + x, \rho_{\text{plus}_x} \rangle \mapsto 3$) rewrites to $\text{Eval} (y + x, \rho_{\text{plus}_x})$
  - $\text{Eval} (3 + 12, \rho_{\text{plus}_x}) = 15$

Evaluation of Application of $\text{plus}_\text{pair}$

- Assume environment
  
  $$\rho = \{x \mapsto 3, \ldots, \text{plus}_\text{pair} \mapsto \langle (n,m) \mapsto n + m, \rho_{\text{plus}_\text{pair}} \rangle \mapsto \rho_{\text{plus}_\text{pair}}\}$$

  - Eval ($\text{plus}_\text{pair} (4,x), \rho$) = $\text{App} (\text{Eval} (\text{plus}_\text{pair}, \rho), \text{Eval} ((4,x), \rho))$
  - $\text{App} (\langle (n,m) \mapsto n + m, \rho_{\text{plus}_\text{pair}} \rangle, (4,3))$
  - Eval ($n + m, \{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}$) = $\text{Eval} (4 + 3, \{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}) = 7$

Closure question

- If we start in an empty environment, and we execute:
  
  ```ml
  let f = fun n -> n + 5;;
  (* $\Theta$ *)
  let pair_map g (n,m) = (g n, g m);;
  let f = pair_map f;;
  let a = f (4,6);;
  ```

  What is the environment at ($^\Theta$ $0$ $\Theta$)?

Answer

$$\rho_{\text{start}} = \{\}$$

```ml
let f = fun n -> n + 5;;
\rho_0 = \{f \mapsto <n \mapsto n + 5, \{\}}\}
```
Closure question

If we start in an empty environment, and we execute:

```plaintext
let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
(* 1 *)
let f = pair_map f;;
let a = f (4,6);;
```

What is the environment at (* 1 *)?

Answer

ρ₀ = \{ f → <n → n + 5, {}> \}
let pair_map g (n,m) = (g n, g m);;
ρ₁ = {
    f → <n → n + 5, {}>,
    pair_map →
        g → (fun (n,m) → (g n, g m)),
        \{ f → <n → n + 5, {}>\}
    }
let f = pair_map f;;

Closure question

If we start in an empty environment, and we execute:

```plaintext
let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
(* 2 *)
let a = f (4,6);;
```

What is the environment at (* 2 *)?

Evaluate pair_map f

ρ₀ = \{ f → <n → n + 5, {}> \}
ρ₁ = \{ f → <n → n + 5, {}>,
    pair_map →
        g → (fun (n,m) → (g n, g m)),
        \{ f → <n → n + 5, {}>\}\}
let f = pair_map f;;
Eval(pair_map f, ρ₁) =

Evaluate pair_map f

ρ₀ = \{ f → <n → n + 5, {}> \}
ρ₁ = \{ f → <n → n + 5, {}>,
    pair_map →
        g → (fun (n,m) → (g n, g m)),
        \{ f → <n → n + 5, {}>\}\}
let f = pair_map f;;
Eval(pair_map f, ρ₁) =
App(<g→fun (n,m)→(g n, g m)>, ρ₀, <n→n+5, {}>) =
Evaluate pair_map f

\[ \rho_0 = \{ f \rightarrow \text{cn} \rightarrow n + 5, \{ \} \} \]
\[ \rho_1 = \{ f \rightarrow \text{cn} \rightarrow n + 5, \{ \}, \text{pair_map} \rightarrow \text{g} \rightarrow \text{fun} \ (n, m) \rightarrow (g \ n, g \ m), \rho_0 \} \]

let f = pair_map f;;

Eval(pair_map f, \rho_1) =

\[ \text{App} (\text{g} \rightarrow \text{fun} \ (n, m) \rightarrow (g \ n, g \ m), \rho_0, \{ f \rightarrow \text{cn} \rightarrow n + 5, \{ \} \}) = \]

let a = f (4,6);;

What is the environment at (\# 3 #)?

Closure question

- If we start in an empty environment, and we execute:
  
  let f = fun => n + 5;;
  let pair_map g (n,m) = (g n, g m);;
  let f = pair_map f;;
  let a = f (4,6);;

What is the environment at (\# 3 #)?

Evaluate f (4,6);;

\[ \rho_2 = \{ f \rightarrow <(n, m) \rightarrow (g \ n, g \ m), \}
\[ \{ g \rightarrow \text{cn} \rightarrow n + 5, \{ \} \}, \]
\[ f \rightarrow \text{cn} \rightarrow n + 5, \{ \} \} \}
\[ \text{pair_map} \rightarrow \text{g} \rightarrow \text{fun} \ (n, m) \rightarrow (g \ n, g \ m), \}
\[ \{ f \rightarrow \text{cn} \rightarrow n + 5, \{ \} \} \}

let a = f (4,6);;

Eval(f (4,6), \rho_2) =

9/5/2017 31

9/5/2017 32

9/5/2017 33

9/5/2017 34

9/5/2017 35

9/5/2017 36
Evaluate $f(4,6)$;

\[
\text{Eval}(f(n, m), \{ n \mapsto 4, m \mapsto 6 \}) = (9, 11)
\]
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.

Match Expressions

```
# 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>
```

Each clause: pattern on left, expression on right
• Each x, y has scope of only its clause
• Use first matching clause

Recursion Example

```
# let rec nthsq n =      (* rec for recursion *)
match n      (* pattern matching for cases *)
with 0      (* base case *)
| n       (* recursive case *)
+ nthsq (n - 1) ;; (* recursive call *)

val nthsq : int -> int
= <fun>
# nthsq 3;;
- : int = 9
```

Structure of recursion similar to inductive proof

Recursion and Induction

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

Base case is the last case; it stops the computation
Recurisve call must be to arguments that are somehow smaller - must progress to base case
**if or match must contain base case (!!!)**
• Failure of selecting base case will cause non-termination
• But the program will crash because it exhausts the stack!

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”

How we typically write them (syntactic sugar):

- \([x]\) == \( x :: [] \)
- \([x_1; x_2; \ldots; x_n]\) == \( x_1 :: x_2 :: \ldots :: x_n :: [] \)

```
let fib5 = [8; 5; 3; 2; 1; 1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
```

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

```
let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
```

```
# (8::5::3::2::1::[ ]) = fib5;;
- : bool = true
```

```
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```

List are Homogeneous

```
# 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. \(["hi"; "there"]; ["wahcha"]; [ ]; ["doin"]\)

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

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

Functions Over Lists
Question: Length of list

- Problem: write code for the length of the list
  - How to start?
  
  ```
  let length l =
  ```

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 [] ->
      | (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 [] -> 0
    | (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) ->
  ```
Question: Length of list

- Problem: write code for the length of the list
- What result do we give when \( l \) is not empty?

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

Same Length

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

```ml
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 Over Lists

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

Iterating over lists

```ml
# 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"];;
val hitheri : unit = ()
```

Iterating over lists

```ml
# 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"]
  ();
val therehi : unit = ()
```
**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
# let rec length list =  
  match list with  
  | [] -> 0         (* Nil case *)  
  | x :: xs -> 1 + length xs; (* 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 xs**

**Forward Recursion**

- In **Structural Recursion**, split input into components and (eventually) recurse
- **Forward Recursion** is a 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) -> 2 * 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>
```

**Encoding Recursion with Fold**

```ocaml
# let rec append list1 list2 = match list1 with
  | [] -> list2 | x :: xs -> x :: append xs 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]
```

**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>
```
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]
```

- Same function, but no recursion

Folding Recursion

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

```ocaml
let rec multList list = match list with [] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
```

```ocaml
# multList [2;4;6];;
- : int = 48
```

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

How long will it take?

Common big-\(O\) times:

- Constant time \(O(1)\)
  - input size doesn’t matter
- Linear time \(O(n)\)
  - \(2x\) input size \(\Rightarrow 2x\) time
- Quadratic time \(O(n^2)\)
  - \(3x\) input size \(\Rightarrow 9x\) time
- Exponential time \(O(2^n)\)
  - Input size \(n+1\) \(\Rightarrow 2x\) time

Linear Time

- Expect most list operations to take linear time \(O(n)\)
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: `multList`, `append`
- Integer example: `factorial`

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>
```
**Exponential running time**

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

---

**Exponential running time**

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

---

**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

---

**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?**
Folding Functions over Lists

How are the following functions similar?

```ocaml
# 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
```

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 list -> 'a
     = <fun>
fold_left f a [x1; x2;...;xn] = f(...(f (f a x1) x2)...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 [x1; x2;...;xn] b = f x1(f x2 (...(f xn b)...))
```

Folding - Forward Recursion

```ocaml
# let sumlist list = fold_right (+) list 0;;
val sumlist : int list -> int = <fun>

# sumlist [2;3;4];;
- : int = 9

# let prodlist list = fold_right ( * ) list 1;;
val prodlist : int list -> int = <fun>

# prodlist [2;3;4];;
- : int = 24
```

Folding - Tail Recursion

```ocaml
- # 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