Booleans (aka Truth Values)

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

# false;;
- : bool = false
```

```ocaml
// ρ7 = {c → 4, test → 3.7, a → 1, b → 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
```

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

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

Tuples as Values

```ocaml
// ρ8 = {s → (5, "hi", 3.2), c → 4, test → 3.7, a → 1, b → 5}
# let s = (5, "hi", 3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

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

Pattern Matching with Tuples

```ocaml
// ρ8 = {s → (5, "hi", 3.2), c → 4, test → 3.7, a → 1, b → 5}
# 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 x = 2, 9.3;; (* tuples don't require parens in Ocam *)
val x : int * float = (2, 9.3)
```

```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)
```

```ocaml
# (*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

```ocaml
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

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)

Functions with more than one argument

```ocaml
# 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
# let add_three =
  fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

Curried vs Uncurried

- Recall `val add_three : int -> int -> int -> int = <fun>`
- How does it differ from
  
  ```ocaml
  # 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**
Curried vs Uncurried

```ocaml
# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  ... is applied to too many arguments,
maybe you forgot a `;`
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

Partial application of functions

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

Evaluation uses an environment \( \rho \)

To evaluate a (simple) declaration `let x = e`

- Evaluate expression \( e \) in \( \rho \) to value \( v \)
- Update \( \rho \) with \( x \) \( \mapsto v \): \( \{ x \mapsto v \} + \rho \)

Update: \( \rho_1 + \rho_2 \) has all the bindings in \( \rho_1 \) and all those in \( \rho_2 \) that are not rebound in \( \rho_1 \)

\[
\{ x \mapsto 2, y \mapsto 3, a \mapsto "hi" \} + \{ y \mapsto 100, b \mapsto 6 \} = \{ x \mapsto 2, y \mapsto 3, a \mapsto "hi" \}, b \mapsto 6 \} \]
Given application expression \( f(e_1, \ldots, e_n) \)

In environment \( \rho \), evaluate left term to closure, 
\[ c = \langle (x_1, \ldots, x_n) \rightarrow b, \rho \rangle \]

\((x_1, \ldots, x_n)\) variables in (first) argument

Evaluate \((e_1, \ldots, e_n)\) to value \((v_1, \ldots, v_n)\)

Update the environment \( \rho \) to
\[ \rho' = \{ x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n \} + \rho \]

Evaluate body \( b \) in environment \( \rho' \)

---

Evaluating Application of \( \text{plus}_x \)

Have environment:
\[ \rho = \{ \text{plus}_x \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle, \ldots, y \rightarrow 3, \ldots \} \]

where \( \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)

\[ \text{Eval (plus}_x y, \rho) \] rewrites to
\[ \text{Eval (y + x, } \rho_{\text{plus}_x} \rangle, \) rewrites to
\[ \text{Eval (3 + 12 , } \rho_{\text{plus}_x} \rangle = 15 \]

---

Evaluating Application of \( \text{plus}_\text{pair} \)

Assume environment
\[ \rho = \{ x \rightarrow 3, \ldots, \text{plus}_\text{pair} \rightarrow \langle (n,m) \rightarrow n + m, \rho_{\text{plus}_\text{pair}} \rangle \} + \rho_{\text{plus}_\text{pair}} \]

\[ \text{Eval (plus}_\text{pair} (4,x), \rho) = \]
\[ \text{App (Eval (plus}_\text{pair}, \rho), \text{Eval ((4,x), } \rho) \rangle = \]
\[ \text{Eval (y + x, } \rho_{\text{plus}_\text{pair}} \rangle, \) rewrites to
\[ \text{Eval (3 + 12 , } \rho_{\text{plus}_\text{pair}} \rangle = 7 \]

---

Closure question

If we start in an empty environment, and we execute:
\[ \text{let } f = \text{fun } n \rightarrow n + 5;; \]
\[ (* 0 *) \]
\[ \text{let } \text{pair_map } g (n,m) = (g n, g m);; \]
\[ \text{let } f = \text{pair_map } f;; \]
\[ \text{let } a = f (4,6);; \]

What is the environment at (* 0 *)?

---

Answer

\[ \text{let } f = \text{fun } n \rightarrow n + 5;; \]
\[ \rho_0 = \{ f \rightarrow <n \rightarrow n + 5, \{ \} > \} \]
Answer

\[ \rho_0 = \{ f \rightarrow <n \rightarrow n + 5, \{ \} > \} \]
let pair_map g (n,m) = (g n, g m);;

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

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;;
(* 2 *)
let a = f (4,6);;
What is the environment at (* 2 *)?

Evaluate pair_map f

\[ \rho_0 = \{ f \rightarrow <n \rightarrow n + 5, \{ \} > \} \]
\[ \rho_1 = \{ \text{pair_map} \rightarrow <g \rightarrow \text{fun (n,m) -> (g n, g m),} \\
    \text{f} \rightarrow <n \rightarrow n + 5, \{ \} >, \\
    \text{f} \rightarrow <n \rightarrow n + 5, \{ \} > \} \]
Eval(pair_map f, \rho_1) =
App (<g \rightarrow \text{fun (n,m) -> (g n, g m),} \\
    \text{f} \rightarrow <n \rightarrow n + 5, \{ \} >, \\
    \text{f} \rightarrow <n \rightarrow n + 5, \{ \} > \})
Answer

ρ₁ = {pair_map →
      <g → fun (n,m) -> (g n, g m), (f → <n → n + 5, ( )>)>,
      f → <n → n + 5, ( )>)}

let f = pair_map f;;

ρ₂ = {f → <(n,m) → (g n, g m),
            {g → <n → n + 5, ( )>,
             f → <n → n + 5, ( )>}>}
      pair_map → <g → fun (n,m) -> (g n, g m),
                         {f → <n → n + 5, ( )>}>}

let a = f (4,6);;

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);;
(* 3 *)

What is the environment at (* 3 *)?

Final Evaluation?

ρ₂ = {f → <(n,m) → (g n, g m),
      {g → <n → n + 5, ( )>,
       f → <n → n + 5, ( )>}>}

let a = f (4,6);;

Evaluate f (4,6);;

ρ₂ = {f → <(n,m) → (g n, g m),
      {g → <n → n + 5, ( )>,
       f → <n → n + 5, ( )>}>}

Pair_map → <g → fun (n,m) -> (g n, g m),
              {f → <n → n + 5, ( )>}>}

let a = f (4,6);; Eval(f (4,6), ρ₂) =

Evaluate f (4,6);;}

ρ₂ = {f → <(n,m) → (g n, g m),
      {g → <n → n + 5, ( )>,
       f → <n → n + 5, ( )>}>}

Pair_map → <g → fun (n,m) -> (g n, g m),
              {f → <n → n + 5, ( )>}>}

Eval(f (4,6), ρ₂) =

App<(n,m) -> (g n, g m), {g -> <n -> n + 5, ( )>},
    f -> <n -> n + 5, ( )>}>)

(4,6) =

Eval((g n, g m), {n -> 4, m -> 6}) +
{g -> <n -> n + 5, ( )>,
 f -> <n -> n + 5, ( )>}) =
(App<n -> n + 5, ( )>, 4),
App (n -> n + 5, ( )>, 6)) =
Evaluate \( f(4,6) \);

\[
\begin{align*}
\text{App}(\langle n \rightarrow n + 5, \{ \rangle, 4), \\
\text{App}(\langle n \rightarrow n + 5, \{ \rangle, 6)) = \\
(\text{Eval}(n + 5, \{n \rightarrow 4\} + \{\}), \\
\text{Eval}(n + 5, \{n \rightarrow 6\} + \{\})) = \\
(\text{Eval}(4 + 5, \{n \rightarrow 4\} + \{\}), \\
\text{Eval}(6 + 5, \{n \rightarrow 6\} + \{\})) = (9, 11)
\end{align*}
\]

Functions as arguments

\[
\begin{align*}
\text{# let thrice } f \ x = f (f (f \ x));; \\
\text{val thrice : ('a -> 'a) -> 'a -> 'a = <fun>} \\
\text{# let } g = \text{thrice } \text{plus_two};; \\
\text{val } g : \text{int} -> \text{int} = \text{<fun>} \\
\text{# } g \ 4;; \\
\ - : \text{int} = 10 \\
\text{# thrice } (\text{fun } s -> "\text{Hi! } ^{s} \text{Good-bye!}");; \\
\text{- : string} = "\text{Hi! Hi! Hi! Good-bye!}" \\
\end{align*}
\]

Higher Order Functions

- A function is higher-order if it takes a function as an argument or returns one as a result.
- Example:
  \[
  \begin{align*}
  \text{# let compose } f \ g = \text{fun } x -> f (g \ x);; \\
  \text{val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>} \\
  \end{align*}
  \]
  - 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:
  \[
  \begin{align*}
  \text{# let thrice } f \ x = f (f (f \ x));; \\
  \text{val thrice : ('a -> 'a) -> 'a -> 'a = <fun>} \\
  \text{# How do you write thrice with compose?} \\
  \end{align*}
  \]

- How do you write thrice with compose?
  \[
  \begin{align*}
  \text{# let thrice } f = \text{compose } f (\text{compose } f \ f);; \\
  \text{val thrice : ('a -> 'a) -> 'a -> 'a = <fun>} \\
  \text{# Is this the only way?} \\
  \end{align*}
  \]

Lambda Lifting

- You must remember the rules for evaluation when you use partial application
  \[
  \begin{align*}
  \text{# let } \text{add}_2 = (+) (\text{print_string } "\text{test}\n"; 2);; \\
  \text{test} \text{add}_2 : \text{int} -> \text{int} = \text{<fun>} \\
  \text{# let } \text{add2} = \text{(*) lambda lifted *)} \\
  \text{fun } x -> (+) (\text{print_string } "\text{test}\n"; 2) \ x;; \\
  \text{val add2 : int} -> \text{int} = \text{<fun}> \\
  \end{align*}
  \]
Lambda Lifting

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

```
'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
```

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

Partial Application and “Unknown Types”

```
Partial Application and “Unknown Types”
```

```
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 * int = <fun>
```

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

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 *)
Recursion Example

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

```ocaml
# let rec nthsq n = (* rec for recursion *)
  match n              (* pattern matching for cases *)
  with 0 -> 0                  (* base case *)
  | n -> (2 * n -1)           (* 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

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

- 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

Lists

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

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

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

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
```

```ocaml
# let bad_list = [1; 3.2; 7];;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```
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

# 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]

# 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 length l =
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
      | [] -> 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 = [10; 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) ();

hithere- : 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"] ();
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.

```ml
# 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
```

Structural Recursion : List Example

- 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>
```

```ocaml
# 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>
```

```ocaml
# 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>
```

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

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;3;4];; 
- : int = 48
```

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

How long will it take?

- Remember the big-O notation from CS 225 and CS 273
- Question: given input of size \( n \), how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power

Common big-O times:
- Constant time \( O(1) \)
  - input size doesn’t matter
- Linear time \( O(n) \)
  - double input ⇒ double time
- Quadratic time \( O(n^2) \)
  - double input ⇒ quadruple time
- Exponential time \( O(2^n) \)
  - increment input ⇒ double 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)?

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 = <fun>

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

Comparison

```ocaml
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 :: ([2] @ [1]) =
3 :: (2:: ([ ] @ [1])) = [3, 2, 1]
```
Comparison

- \( \text{rev} \ [1,2,3] = \)
- \( \text{rev} \ _{aux} \ [1,2,3] \ [ \ ] = \)
- \( \text{rev} \ _{aux} \ [2,3] \ [1] = \)
- \( \text{rev} \ _{aux} \ [3] \ [2,1] = \)
- \( \text{rev} \ _{aux} \ [ \ ] \ [3,2,1] = [3,2,1] \)

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 -> '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)...))
```

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 rec 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