Tuples as Values

```ocaml
// \( \rho_0 = \{ c \rightarrow 4, a \rightarrow 1, b \rightarrow 5 \} \)
# let s = (5, "hi", 3.2) ;;
val s : int * string * float = (5, "hi", 3.2)

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

Pattern Matching with Tuples

```ocaml
// \( \rho = \{ s \rightarrow (5, "hi", 3.2), a \rightarrow 1, b \rightarrow 5, c \rightarrow 4 \} \)
# 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 ;;
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 ;;
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 twice x = (x,x) ;;
val twice : 'a -> 'a * 'a = <fun>
# twice 3 ;;
- : int * int = (3, 3)

# twice "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, \rho >
  ```
- Where \( \rho \) is the environment in effect when the function is defined (for a simple function)
Closure for plus_pair

- Assume $\rho_{\text{plus\_pair}}$ was the environment just before 
  plus_pair defined and recall
  - let plus_pair $(n,m) = n + m$;

- Closure for $\text{fun } (n,m) \rightarrow n + m$:
  $(n,m) \mapsto n + m$, $\rho_{\text{plus\_pair}}$

- Environment just after plus_pair defined:
  $(\text{plus\_pair} \mapsto (n,m) \mapsto n + m, \rho_{\text{plus\_pair}})$
  Like set union! (but subtle differences, see slide 17)

Functions with more than one argument

# let add_three $x$ $y$ $z$ = $x$ + $y$ + $z$;
val add_three : int $\rightarrow$ int $\rightarrow$ int $\rightarrow$ 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 $\rightarrow$ int $\rightarrow$ int $\rightarrow$ int = <fun>

Again, first syntactic sugar for second

Curried vs Uncurried

- Recall
  # let add_three $u$ $v$ $w$ = $u$ + $v$ + $w$;;
  val add_three : int $\rightarrow$ int $\rightarrow$ int $\rightarrow$ int = <fun>

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

- add_three is curried;
- add_triple is uncurried

Partial application of functions

let add_three $x$ $y$ $z$ = $x$ + $y$ + $z$;;

# let $h$ = add_three 5 4;;
val $h$ : int $\rightarrow$ int = <fun>

# $h$ 3;;
- : int = 12

# $h$ 7;;
- : int = 16

Partial application also called sectioning

Recall: let plus_x = fun $y$ -> $y$ + $x$
Closure for \texttt{plus\_x}

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

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 (mathematical) function \texttt{Eval}
- Nice property of Ocaml: \texttt{everything is a declaration or an expression!}

- How does \texttt{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 \texttt{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 \(\rho_1\)

\[
\begin{align*}
\{ x \mapsto 2, y \mapsto 3, a \mapsto "hi" \} + \{ y \mapsto 100, b \mapsto 6 \} &= \{ x \mapsto 2, y \mapsto 3, a \mapsto "hi", b \mapsto 6 \}
\end{align*}
\]

Evaluating Expressions (Rules)

- Evaluation uses an environment \(\rho\)
- A constant evaluates to itself
- To evaluate a variable x, look it up in \(\rho\) i.e., use \(\rho(x)\)
- To evaluate tuples, evaluate each tuple element
- To evaluate uses of +, _, etc, first eval the arguments, then do the operation
- To evaluate a local declaration: let \(x = e_1\) in \(e_2\)
  - Evaluate \(e_1\) to v, evaluate \(e_2\) using \(\{x \mapsto v\} + \rho\)
- Function application \((f x)\) -- see next slide

Evaluation of Function Application with Closures

Function \texttt{defined} as: let \(f(x_1, \ldots, x_n) = \text{body}\)

Function application: \(f(e_1, \ldots, e_n)\):

Let us define \texttt{Eval(f (e_1, \ldots, e_n), \rho):}

- In the environment \(\rho\), evaluate the left term (f) to closure, i.e.,
  \(c = <(x_1, \ldots, x_n) \mapsto \text{body}, \rho^c>\)
- Evaluate the arguments in the application \(e_1 \ldots e_n\) to their values \(v_1 \ldots v_n\) in the environment \(\rho\)
- Call helper function \texttt{App(Closure, Value)} to evaluate the function body \(\text{body}\) in the environment \(\rho^c\)
  - Conjoin the mapping of the arguments to values with the environment \(\rho^c\)
  \[
  \rho^\prime = \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \} + \rho^c
  \]
  - The \texttt{App} then calls \texttt{Eval} again for the expressions in \text{body} in the env \(\rho^\prime\)
Evaluation of Application of $\text{plus}_x$:

- Have environment:
  $$\rho = \{ \text{plus}_x \to <y \to y + x, \rho_{\text{plus}_x}>, \ldots, y \to 3, \ldots \}$$
  where $\rho_{\text{plus}_x} = \{ x \to 12, \ldots, y \to 24, \ldots \}$

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

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

- Assume environment
  $$\rho = \{ x \to 3, \ldots, \text{plus}_\text{pair} \to <(n,m) \to n + m, \rho_{\text{plus}_\text{pair}}>, \ldots \}$$

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

Closure question

- If we start in an empty environment, and we execute:
  ```
  let f = fun n -> n + 5;;
  (* 0 *)
  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 (* 0 *)?

Answer

- If we start in an empty environment, and we execute:
  ```
  let f = fun n -> n + 5;;
  let pair_map g (n,m) = (g n, g m);
  let a = f (4,6);;
  ```
  What is the environment at (* 1 *)?
Closure question

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

```
let f = fun n -> 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`**

| $\rho_0$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |
| $\rho_1$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |
| $\rho_2$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |
| $\rho_3$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |

```
let f = pair_map f;;
```

```
Eval(pair_map f, $\rho_1$) =
```

---

**Answer**

| $\rho_0$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |
| $\rho_1$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |
| $\rho_2$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |
| $\rho_3$ | $\{ f \rightarrow \langle n \rightarrow n + 5, \{ \rangle \rangle \}$ |

```
let f = pair_map f;;
```

```
Eval(pair_map f, $\rho_1$) =
```

---

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

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

  ```
  let f = fun n -> 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 *)?

Final Evaluation?

- Evaluation:

  ```
  let a = f (4,6);;
  ```

  Evaluating `f (4,6)` yields:

  ```
  Eval(f (4,6), ρ_2) =
  ```

  Finally:

  ```
  ρ_3 = {a -> (9, 11)} + ρ_2
  ```
**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:
  ```ocaml
  # let thrice f x = f (f (f x));
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  ```

- How do you write thrice with compose?
  ```ocaml
  # let thrice f = compose f (compose f f);
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  ```

**Lambda Lifting**

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

  ```ocaml
  # let add_two = (+) (print_string "test\n"; 2);
  val add_two : int -> int -> int = <fun>
  ```

  ```ocaml
  # let add2 = (+) (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.

```ocaml
  # thrice add_two 5;
  . : int = 11
  ```

```ocaml
  # thrice add2 5;
  test
  test
  test
  test
  . : int = 11
  ```
Reminder: Pattern Matching with Tuples

```ml
# 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

(*Patterns can be nested *)
# let (p, (st, _), _) = d;;
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

Match Expressions

```ml
# 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>
```

Recursive Functions

```ml
# 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

```ml
Compute \( n^2 \) recursively using:
\[
\frac{n^2}{2} = (2 \cdot n - 1) + (n - 1)^2
\]

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

# nthsq 3;;
- : int = 9
```

Recursion and Induction

```ml
# 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
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- if or match must contain the 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):
  - \([x] == x :: [\ ]\)
  - \([x_1; x_2; …; x_n] == x_1 :: x_2 :: … :: x_n :: [\ ]\)

List 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
```

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>

(* fib5 = [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]
```

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. \(["hi; "there"]\)

```ocaml
3 is invalid because of the last pair
```

Functions Over Lists

```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 silly;;
- : string list = ["there"; "there"; "hi"; "hi"]
```
Question: Length of list

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

```ocaml
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?

```ocaml
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 empty?

```ocaml
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?

```ocaml
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?

**Tactics:**

| First list is empty: then true if second list is empty else false
| First list in not empty: then if second list empty return false, or otherwise compare whether the sublists (after the first element) have the same length

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 | _ -> false)
  | (x::xs) -> (match list2 with | [] -> false | _ -> same_length xs (y::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;;
val it : int list = [10; 7; 5; 4; 3; 3]

# map (fun x -> x - 1) fib6;;
val it : 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 it : unit = ()
```

Iterating over lists

```ml
# let rec fold_right f b list =
    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 it : 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
  | x :: xs -> 1 + length xs;; (* Nil 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 -> 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>

# 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]

# let rec append_alt list1 list2 =
  fold_right (fun x y -> x :: y) list1 list2;;
val append_alt : 'a list -> '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>
```

Mapping Recursion

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

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

# double_list [2;3;4];;
- : int list = [4; 6; 8]
```
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>

# 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>

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

- Computes (2 * (4 * (6 * 1)))

How long will it take?

Common big-O times:
- Constant time $O(1)$
- Linear time $O(n)$
  - 2x input size ⇒ 2x time
- Quadratic time $O(n^2)$
  - 3x input size ⇒ 9x time
- Exponential time $O(2^n)$
  - Input size $n+1$ ⇒ 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

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>
```

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>
```

Tail Recursion - Example

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

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

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

# 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 = <fun>
fold_right f [x_1; x_2;...;x_n] b = f x_1(f x_2(...(f x_n b)...
```

Folding

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