Next

- More on recursion
- Higher-order functions
  - taking and returning functions
- Along the way, will see map and fold
Tail Recursion: Factorial

```ml
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n-1);;
```

How does it execute?

let rec fact n =
  if n <= 0
  then 1
  else n * fact (n-1);;

fac 3;;
Tail recursion

- Tail recursion:
  - recursion where all recursive calls are immediately followed by a return
  - in other words: not allowed to do anything between recursive call and return
Tail recursive factorial

let fact x =
let fact x =
    let rec helper x curr =
        if x <= 0
        then curr
        else helper (x - 1) (x * curr)
    in
    helper x 1;;
How does it execute?

```ocaml
let fact x =  
    let rec helper x curr =  
        if x <= 0  
            then curr  
            else helper (x - 1) (x * curr)  
    in  
    helper x 1;;

fact 3;;
```
Tail recursion

• Tail recursion:
  - for each recursive call, the value of the recursive call is immediately returned
  - in other words: not allowed to do anything between recursive call and return

• Why do we care about tail recursion?
  - it turns out that tail recursion can be optimized into a simple loop
Compiler can optimize!

let fact x =
  let rec helper x curr =
    if x <= 0
    then curr
    else helper (x - 1) (x * curr)
  in
  helper x 1;;

fact(x) {
  curr := 1;
  while (1) {
    if (x <= 0)
      then { return curr }
    else { x := x - 1; curr := (x * curr) }
  }
}
Tail recursion summary

- Tail recursive calls can be optimized as a jump

- Part of the language specification of some languages (ie: you can count on the compiler to optimize tail recursive calls)
max function

let max x y = if x < y then y else x;;

(* return max element of list l *)
let list_max l =
let max x y = if x < y then y else x;;

(* return max element of list l *)
let list_max l =
  let rec helper curr l =
    match l with
      [] -> curr
    | h::t -> helper (max curr h) t
  in
  helper 0 l;;
concat function

(* concatenate all strings in a list *)
let concat l =
concat function

(* concatenate all strings in a list *)

let concat l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (curr ^ h) t
  in
  helper "" l;;
let list_max l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (max h curr) t
  in helper 0 l;;

let concat l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (curr ^ h) t
  in helper "" l;;
fold, the general helper func!

(* to help us see the pattern: *)

let list_max l =
  let rec helper curr l =
    match l with
    | [] -> curr
    | h::t -> helper (max h curr) t
  in helper 0 l;;

(* fold, the coolest function there is! *)

let rec fold f curr l =
(* fold, the coolest function there is! *)

let rec fold f curr l =
    match l with
    | [] -> curr
    | h::t -> fold f (f curr h) t;;
fold

(* fold, the coolest function there is! *)
let rec fold f curr l =
    match l with
    [] -> curr
    | h::t -> fold f (f curr h) t;;
Examples of fold

```ml
let list_max =

let concat =

let multiplier =
```
Examples of fold

```ml
let list_max = fold max 0;;

let concat = fold (^) "";;

let multiplier = fold (*) 1;;
```
Examples of fold

```
let fact n = 
multiplier (interval 1 n);;
```

Notice how all the recursion is buried inside two functions: interval and fold!
Examples of fold

```ocaml
let cons x y = y :: x;;
let f = fold cons [];;
(* same as: *)
  let f l = fold cons [] l *)
```
Examples of fold

let cons x y = y::x;;
let f = fold cons [];;;
(* same as:
  let f l = fold cons [] l *)

[
 ] ➔ cons ➔ [v₁] ➔ cons ➔ [v₂; v₁] ➔ cons ➔ [v₃; v₂; v₁]

v₁ :: []

v₁ :: [v₁]

v₂ :: [v₁]

v₃ :: [v₂; v₁]
More recursion: interval

(* return a list that contains the integers i through j inclusive *)
let rec interval i j =
(* return a list that contains the integers i through j inclusive *)

let rec interval i j =
  if i > j
  then []
  else i::(interval (i+1) j);;
interval function with init fn

(* return a list that contains the elements f(i), f(i+1), ... f(j) *)

let rec interval_init i j f =
interval function with init fn

(* return a list that contains the elements f(i), f(i+1), ... f(j) *)

let rec interval_init i j f =
  if i > j
  then []
  else (f i)::(interval_init (i+1) j f);;
interval function again

(* our regular interval function in terms of the one with the init function *)
let rec interval i j =
interval function again

(* our regular interval function in terms of the one with the init function *)
let rec interval i j =
  interval_init i j (fun x -> x);;
Interval function yet again!

(* let's change the order of parameters... *)
let rec interval_init f i j =
  if i > j
  then []
  else (f i)::(interval_init f (i+1) j);;

(* now can use currying to get interval function! *)
let interval = interval_init (fun x -> x);;
Function Currying

In general, these two are equivalent:

```plaintext
let f = fun x1 -> ... -> fun xn -> e
```

```plaintext
let f x1 ... xn = e
```

Multiple argument functions by returning a function that takes the next argument
- Named after a person (Haskell Curry)
Function Currying vs tuples

Tuple version:

let \( f (x_1, \ldots, x_n) = e \)

Curried version:

let \( f x_1 \ldots x_n = e \)
Function Currying vs tuples

Consider the following:

```plaintext
let lt x y = x < y;
```

Could have done:

```plaintext
let lt (x,y) = x<y;
```

- But then no “testers” possible

In general: Currying allows you to set just the first n params (where n smaller than the total number of params)
map

(* return the list containing f(e) for each element e of l *)
let rec map f l =
(* return the list containing f(e) for each element e of l *)

let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t) ;;
let incr x = x+1;;

let map_incr = map incr;;
map_incr (interval (-10) 10);;
composing functions

\[(f \circ g) (x) = f(g(x))\]

(* return a function that given an argument x applies f2 to x and then applies f1 to the result*)

let compose f1 f2 =
composing functions

\[(f \circ g) (x) = f(g(x))\]

(* return a function that given an argument x applies f2 to x and then applies f1 to the result*)
let compose f1 f2 = fun x -> (f1 (f2 x));;

(* another way of writing it *)
let compose f1 f2 x = f1 (f2 x);;
let map_incr_2 = compose map_incr map_incr;;
map_incr_2 (interval (-10) 10);;

let map_incr_3 = compose map_incr map_incr_2;;
map_incr_3 (interval (-10) 10);;

let map_incr_3_pos = compose pos_filer map_incr_3;;
map_incr_3_pos (interval (-10) 10);;
(compose map_incr_3_pos pos_filer) (interval (-10) 10);;
Higher-order functions!

```ocaml
let map_incr_2 = compose map_incr map_incr;;
map_incr_2 (interval (-10) 10);;

let map_incr_3 = compose map_incr map_incr_2;;
map_incr_3 (interval (-10) 10);;

let map_incr_3_pos = compose pos_filer map_incr_3;;
map_incr_3_pos (interval (-10) 10);;
(compose map_incr_3 pos_filer) (interval (-10) 10);;
```

Instead of manipulating lists, we are manipulating the list manipulators!
let rec filter f l =  
    match l with  
      | [] -> []  
      | h::t -> let t' = filter f t in  
          if f h then h::t' else t'  

let neg f x = not (f x)  

let partition f l = (filter f l, filter (neg f) l)  

This implementation is not ideal, since it unnecessarily processes the list twice. Rewrite partition so that it is a single call to fold_left, so the input list is processed only once. Recall:

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
Exercise 1

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

let partition f l =
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

let partition f l =
    let fold_fn (pass,passnot) x =
        if f x then (pass@[x], passnot)
        else (pass, passnot@[x])
    in
    List.fold_left fold_fn ([],[]) l;;
Exercise 2

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val map : ('a -> 'b) -> 'a list -> 'b list

Implement map using fold:

let map f l =
Exercise 2 Solution

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val map : ('a -> 'b) -> 'a list -> 'b list

Implement map using fold:

let map f l =
  List.fold_left (fun acc x -> acc@[f x]) [] l
Benefits of higher-order functions

Identify common computation “patterns”

- **Iterate** a function over a set, list, tree ...
- **Accumulate** some value over a collection

Pull out (factor) “common” code:
- **Computation Patterns**
- **Re-use** in many different situations
Higher-order funcs enable **modular** code

- Each part only needs **local** information

**Data Structure**

- Client
- Uses list

**Data Structure**

- Library
- list

**Uses** meta-functions:

- map, fold, filter

With locally-dependent funs

- (lt h), square etc.

**Provides** meta-functions:

- map, fold, filter

to traverse, accumulate over

lists, trees etc.

Meta-functions don’t need client

info
Different way of thinking

“Free your mind”
-Morpheus

• Different way of thinking about computation
• Manipulate the manipulators