Next

• More on recursion

• Higher-order functions
  - taking and returning functions

• Along the way, will see map and fold

Tail Recursion: Factorial

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

How does it execute?

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 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!

```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(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

```ocaml
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 =

fold, the general helper func!

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

What’s 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

(* 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

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

let concat = fold (^) "";;

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

Examples of fold

```ocaml
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

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

More recursion: interval

```ocaml
(* return a list that contains the integers i through j inclusive *)
let rec interval i j =
```
let rec interval i j = 
  if i > j 
  then [] 
  else i::(interval (i+1) j);;

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

let rec interval i j = 
  interval_init i j (fun x -> x);;

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

let interval = interval_init (fun x -> x);;
**Function Currying**

In general, these two are equivalent:

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

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**

Consider the following:

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

Could have done:  
let lt (x,y) = x<y;

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

**map**

```ml
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 = fun x - (f1 (f2 x));;

(* another way of writing it *)

let compose f1 f2 x = f1 (f2 x);;
Exercise 1 Solution

```ocaml
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

```ocaml
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
```

Exercise 2 Solution

```ocaml
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

Different way of thinking

"Free your mind"  
- Morpheus

- **Different way of thinking about computation**
- **Manipulate the manipulators**