Higher-Order Functions
Today’s Plan

• A little more practice with recursion
  - Base Pattern $\rightarrow$ Base Expression
  - Induction Pattern $\rightarrow$ Induction Expression

• Higher-Order Functions
  - Why “take” and “return” functions?
(* val evens: int list -> int list *)
let rec evens xs = match xs with
| []    -> ...
| x::xs' -> ...

evens [] ===> []
evens [1;2;3;4] ===> [2;4]
(* val evens: int list -> int list *)

let rec evens xs = match xs with
  | [] -> []
  | x::xs' -> if x mod 2 = 0
    then x::(evens xs')
    else (evens xs')

evens [] ===> []

evens [1;2;3;4] ===> [2;4]
bigrams []
  ===> []

bigrams ["cat"; "must"; "do"; "my"; "work"]
  ===> ["do"; "my"]

(* val bigrams: string list -> string list list *)
let rec bigrams xs = match xs with
  | [] -> ...
  | []:xs' -> ...
  | x::xs' -> ...


```ocaml
(* val bigrams: string list -> string list *)
let rec bigrams xs = match xs with
  | [] -> []
  | x::xs' -> if String.length x = 2
            then x::(bigrams xs')
            else (bigrams xs')
```

bigrams []
  ===> []

bigrams ["cat"; "must"; "do"; "my"; "work"]
  ===> ["do"; "my"]
Yuck! Most code is same!
Yuck! Most code is same!
Yuck! Most code is same!
Moral of the Day

“D.R.Y”
Don’t Repeat Yourself!
Moral of the Day

HOFs Allow “Factoring” Into:

General Pattern + Specific Operation
let rec evens xs =
    match xs with
    | []  -> []
    | x::xs' -> if x mod 2 = 0
                then x::(foo xs')
                else (foo xs')

let rec bigrams xs =
    match xs with
    | []  -> []
    | x::xs' -> if length x = 2
                then x::(foo xs')
                else (foo xs')

let rec filter f xs =
    match xs with
    | []  -> []
    | x::xs' -> if f x
                then x::(filter f xs')
                else (filter f xs')

The “filter” pattern
The “filter” pattern
Factor Into Generic + Specific

Specific Operations

let evens xs =
  filter (fun x -> x mod 2 = 0) xs

let bigrams xs =
  filter (fun x -> length x = 2) xs

let rec filter f xs =
  match xs with
  | [] -> []
  | x::xs' -> if f x
    then x::(filter f xs')
    else (filter f xs')
Clicker Question

What is the type of `filter`?

```ocaml
let rec filter f xs =
  match xs with
  | [] -> []
  | x::xs' -> if f x
     then x::(filter f xs')
     else (filter f xs')
```

(a) ('a -> 'b) -> 'a list -> 'a list
(b) ('a -> bool) -> 'a list -> 'a list
(c) (int -> bool) -> int list -> int list
(d) (bool -> bool) -> bool list -> bool list
Clicker Question

What is the type of `filter`?

```ocaml
let rec filter f xs =
  match xs with
  | [] -> []
  | x::xs' -> if f x
    then x::(filter f xs')
    else (filter f xs')
```

(a) ('a -> 'b) -> 'a list -> 'a list
(b) ('a -> bool) -> 'a list -> 'a list
(c) (int -> bool) -> int list -> int list
(d) (bool -> bool) -> bool list -> bool list
(* val listUpper: string list -> string list *)

let rec listUpper xs =
  match xs with
  | [] -> ...
  | x::xs' -> ...

listUpper []
  ===> []

listUpper [“carne”; “asada”]
  ===> [“CARNE”; “ASADA”]
let rec listUpper xs =
    match xs with
    | []   -> []
    | x::xs' -> (uppercase x):::(listUpper xs')

listUpper []
    ===> []

listUpper ["carne"; "asada"]
    ===> ["CARNE"; "ASADA"]
listSquare [] ===> []
listSquare [1;2;3;4;5] ===> [1;4;9;16;25]

(* val listSquare: int list -> int list *)
let rec listSquare xs =
  match xs with
  | []  -> ...
  | x::xs'-> ...
let rec listSquare xs =
  match xs with
  | [] -> []
  | x::xs' -> (x*x)::(listSquare xs')

listSquare [] ===> []
listSquare [1;2;3;4;5] ===> [1;4;9;16;25]
Yuck! Most code is same!

(* val listUpper: string list -> string list *)
let rec listUpper xs =
  match xs with
  | []     -> []
  | x::xs'  -> (uppercase x)::(listUpper xs')

(* val listSquare: int list -> int list *)
let rec listSquare xs =
  match xs with
  | []     -> []
  | x::xs'  -> (x*x)::(listSquare xs')
Yuck! Most code is same!

(* val foo: int list -> int list *)
let rec foo xs =
    match xs with
    | []     -> []
    | x::xs'  -> (x*x)::(foo xs')

(* val foo: string list -> string list *)
let rec foo xs =
    match xs with
    | []     -> []
    | x::xs'  -> (uppercase x)::(foo xs')
Yuck! Most code is same!

(* val foo: string list -> string list *)
let rec foo xs =
    match xs with
    | []    -> []
    | x::xs'-> (uppercase x)::(foo xs')

(* val foo: int list -> int list *)
let rec foo xs =
    match xs with
    | []    -> []
    | x::xs'-> (x*x)::(foo xs')
“D.R.Y”
Don’t Repeat Yourself!
The “iteration” pattern
The “iteration” pattern

```ocaml
let rec listUpper xs =  
  match xs with
  | []    -> []
  | x::xs'->  
    (uppercase x)::(listUpper xs')

let rec listSquare xs =  
  match xs with
  | []    -> []
  | x::xs'->  
    (x*x)::(listSquare xs')

let listUpper xs =  
  map uppercase xs

let rec map f xs =  
  match xs with
  | []    -> []
  | x::xs' ->  
    (f x)::(map f xs')
```
The “iteration” pattern

```ml
let rec listUpper xs =
  match xs with
  | []  -> []
  | x::xs'->
    (uppercase x)::(listUpper xs')

let listUpper =
  map uppercase

let rec listSquare xs =
  match xs with
  | []  -> []
  | x::xs'->
    (x*x)::(listSquare xs')

let listSquare =
  map (fun x -> x*x)

let rec map f xs =
  match xs with
  | []  -> []
  | x::xs' -> (f x)::(map f xs')
```
Factor Into Generic + Specific

```ocaml
let listSquare = map (fun x -> x * x)
let listUpper = map uppercase
```

Generic “iteration” pattern

```ocaml
let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')
```
Clicker Question

What is the type of map?

let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')

(a) ('a -> 'b) -> 'a list -> 'b list
(b) (int -> int) -> int list -> int list
(c) (string -> string) -> string list -> string list
(d) ('a -> 'a) -> 'a list -> 'a list
(e) ('a -> 'b) -> 'c list -> 'd list
What is the type of `map`?

```occam
let rec map f xs =
  match xs with
  | []     -> []
  | x::xs' -> (f x)::(map f xs')
```

(a) `(a -> b) -> 'a list -> 'b list`

Type says it all!

- Apply `f` to each element in input list
- Return a list of the results
What does this evaluate to?

```
map (fun (x, y) -> x+y) [1;2;3]
```

(a) [2;4;6]
(b) [3;5]
(c) Syntax Error
(d) Type Error
“Factored” code:

- Avoid bugs due to repetition
- Fix bug in one place!

Made Possible by Higher-Order Functions!
Recall: len

(* 'a list -> int *)

let rec len xs =
  match xs with
  | [] -> 0
  | x::xs' -> 1 + len xs'

len [] ===> 0
len ["carne"; "asada"] ===> 2
Recall: sum

(* int list -> int *)

let rec sum xs =
    match xs with
    | [] -> 0
    | x::xs' -> x + len xs'

sum [] ===> 0
sum [10;20;30] ===> 60
Write: concat

(* string list -> string *)
let rec concat xs =
  match xs with
  | []    -> ...
  | x::xs' -> ...

concat []
  ===> ""

concat ["carne"; "asada"; "torta"]
  ===> "carneasadatorta"
Write: concat

(* string list -> string *)
let rec concat xs =
  match xs with
  | [] -> ""
  | x::xs' -> x^(concat xs')

concat []
  ======> ""

concat ["carne"; "asada"; "torta"]
  ======> "carneasadatorta"
let rec foldr f b xs =
    match xs with
    | []    -> b
    | x::xs' -> f x (foldr f b xs')

let rec len xs =
    match xs with
    | []    -> 0
    | x::xs'-> 1+ (len xs')

let rec sum xs =
    match xs with
    | []    -> 0
    | x::xs'-> x+ (sum xs')

let rec concat xs =
    match xs with
    | []    -> ""
    | x::xs'-> x^ (concat xs')

What’s the Pattern?
let rec len xs =
  match xs with
  | []   -> 0
  | x::xs' -> 1 + (len xs')

let rec sum xs =
  match xs with
  | []   -> 0
  | x::xs' -> x + (sum xs')

let rec concat xs =
  match xs with
  | []   -> ""
  | x::xs' -> x ^ (concat xs')

let len =
foldr (fun x n -> n+1) 0

let sum =
foldr (fun x n -> x+n) 0

let concat =
foldr (fun x n -> x^n) ""
let rec foldr f b xs =
  match xs with
  | []   -> b
  | x::xs' -> f x (foldr f b xs')

let len =
  foldr (fun x n -> n+1) 0

let sum =
  foldr (fun x n -> x+n) 0

let concat =
  foldr (fun x n -> x^n) ""
Clicker Question

What does this evaluate to?

\[
\text{foldr (fun x n -> x::n)} \ [\ ] \ [1;2;3]
\]

(a) [1; 2; 3]
(b) [3; 2; 1]
(c) []
(d) [[3]; [2]; [1]]
(e) [[[1]; [2]; [3]]
“fold-right” pattern

let rec foldr f b xs =
  match xs with
  | [] -> b
  | x::xs' -> f x (foldr f b xs')

foldr f b [x1;x2;x3]
====> f x1 (foldr f b [x2;x3])
====> f x1 (f x2 (foldr f b [x3]))
====> f x1 (f x2 (f x3 (foldr f b [])))
====> f x1 (f x2 (f x3 (b)))
A function is “tail recursive” if:

- all recursive calls are immediately followed by return
- that is, each recursive call is in “tail position”
- so cannot do anything between call and return

Why do we care?

- Compiler can transform recursion into a loop
- You write readable code
- Compiler optimizes into fast code!
“fold-right” pattern

```haskell
let rec foldr f b xs =
    match xs with
    | []    -> b
    | x::xs' -> f x (foldr f b xs')
```

foldr f b [x1;x2;x3]

`====> f x1 (foldr f b [x2;x3])`

`====> f x1 (f x2 (foldr f b [x3]))`

`====> f x1 (f x2 (f x3 (foldr f b [])))`

`====> f x1 (f x2 (f x3 (b)))`

**Not Tail Recursive!**
Write: Tail-recursive concat

let concat xs = ...

concat []
  ===> ""

concat ["carne"; "asada"; "torta"]
  ===> "carneasadatorta"
let concat xs =
  let rec helper res = function
  | [] -> res
  | x::xs' -> helper (res^x) xs'
in helper "" xs

helper "" ["carne"; "asada"; "torta"]
====> helper "carne" ["asada"; "torta"]
====> helper "carneasada" ["torta"]
====> helper "carneasadatorta" []
====> "carneasadatorta"
Write: Tail-recursive sum

```plaintext
let sum xs = ...
```

sum [] ===> 0

sum [10;20;30] ===> 60
let sum xs =
  let rec helper res = function
  | []    -> res
  | x::xs' -> helper (res+x) xs'
in helper 0 xs

helper 0 [10; 100; 1000]
====> helper 10 [100; 1000]
====> helper 110 [1000]
====> helper 1110 []
====> 1110
“Accumulation” Pattern

let foldl f b xs =
    let rec helper res = function
    | []    -> res
    | x::xs' -> helper (f res x) xs'
  in helper b xs

let sum xs =
    let rec helper res = function
    | []    -> res
    | x::xs'-> helper (res + x) xs'
  in helper 0 xs

let sum xs =
    foldl (fun res x -> res + x) 0

let concat xs =
    let rec helper res = function
    | []    -> res
    | x::xs'-> helper (res ^ x) xs'
  in helper "" xs

let concat xs =
    foldl (fun res x -> res ^ x) ""
"Accumulation" Pattern

```
let foldl f b xs =
  let rec helper res = function
  | [] -> res
  | x::xs' -> helper (f res x) xs'
  in helper b xs
```

Specific Op

```
let sum xs =
  foldl (fun res x -> res + x) 0

let sum xs =
  foldl (fun res x -> res ^ x) ""
```
Clicker Question

What does this evaluate to?

```ocaml
let foldl f b xs =
  let rec helper res = function
  | [] -> res
  | x::xs' -> helper (f res x) xs'
  in helper b xs
```

foldl (fun res x -> x::res) [] [1;2;3]

(a) [1; 2; 3]
(b) [3; 2; 1]
(c) []
(d) [[3]; [2]; [1]]
(e) [[1]; [2]; [3]]
Another fun function: “pipe”

```plaintext
let pipe x f = f x

let (|>) x f = f x
```

Compute sum of squares of numbers in a list:

```plaintext
let sumOfSquares xs =
    xs |> map (fun x -> x * x)
    |> foldl (+) 0
```

Tail Rec ?
Higher-Order Functions

Identify common computation “patterns”

• **Filter** values in a set, **list**, tree ...
• **Iterate/Map** a function over a set, **list**, tree ...
• **Accumulate/Fold** some value over a collection

Pull out (factor) boilerplate code:

• **Computation Patterns**
• **Re-use** in many different situations
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?
  - Data structure (list, tree, etc.) library provides meta-functions (map, fold, filter, etc.) to traverse in a generic way
  - Data structure client uses meta-functions with application-specific details
  - “MapReduce” = “MapFold”