

Write: evens

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

evens [] ==> []

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

Write: evens

```
(* 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]

Write: fourLetters

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

fourLetters []

====> []

fourLetters ["cat"; "must"; "do"; "work"]

====> ["must"; "work"]

```
(* 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')
```

```
(* fourLetters: string list -> string list *)  
let rec fourLetters xs = match xs with  
  | []           -> []  
  | x::xs'      -> if length x = 4  
                    then x::(fourLetters xs')  
                    else (fourLetters xs')
```

Yuck! Most code is same!

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

```
(* fourLetters: string list -> string list *)  
let rec foo xs = match xs with  
  | []           -> []  
  | x::xs'      -> if length x = 4  
                    then x::(foo xs')  
                    else (foo xs')
```

Yuck! Most code is same!

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

```
let rec fourLetters xs =  
  match xs with  
  | []      -> []  
  | x::xs'  -> if length x = 4  
                then x::(foo xs')  
                else (foo xs')
```

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

The “filter” pattern

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

```
let rec fourLetters xs =
  match xs with
  | []      -> []
  | x::xs'  -> if length x = 4
                then x::(foo xs')
                else (foo xs')
```

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

```
let fourLetters xs =
  filter (fun x -> length x = 4) xs
```

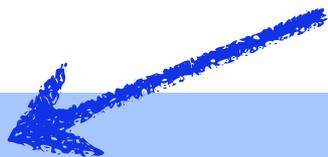
```
let rec filter f xs =
  match xs with
  | []      -> []
  | x::xs'  -> if f x
                then x::(filter xs')
                else (filter xs')
```

The “filter” pattern

Factor Into Generic + Specific

Specific Operations

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



```
let fourLetter xs =  
  filter (fun x -> length x = 4) xs
```



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

Generic “filter” pattern

Write: listUpper

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

`listUpper []` =====> []

`listUpper ["carne"; "asada"]` =====> ["CARNE"; "ASADA"]

Write: listUpper

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

`listUpper []` =====> []

`listUpper ["carne"; "asada"]` =====> ["CARNE"; "ASADA"]

Write: listSquare

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

listSquare [] ==> []

listSquare [1;2;3;4;5] ==> [1;4;9;16;25]

Write: listSquare

```
(* int list -> int list *)  
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!

```
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')
```

What's the Pattern?

```
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')
```

What's the Pattern?

```
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')
```

“Refactor” Pattern

```
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 rec pattern ...
```

“Refactor” Pattern

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

“Refactor” Pattern

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

```
let listUpper xs = map (fun x -> uppercase x) xs
```

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

“Refactor” Pattern

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

```
let listUpper = map uppercase
```

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

“Refactor” Pattern

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

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

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

Factor Into Generic + Specific

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

```
let listUpper = map uppercase
```



Specific Op

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

Generic “iteration” pattern

Q: 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`

Q: 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

Type says it all !

- Apply “f” to each element in input list
- Return a list of the results

Don't Repeat Yourself!

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

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"

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')
```

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

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

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

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

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

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

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

“fold” Pattern

```
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
```

Specific Op

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

Write: concat (TR)

```
let concat xs = ...
```

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

`concat ["carne"; "asada"; "torta"]`
====> "carneasadatorta"

Write: concat

```
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: sum (TR)

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

sum [] ==> 0

sum [10;20;30] ==> 60

What's the Pattern?

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

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

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

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

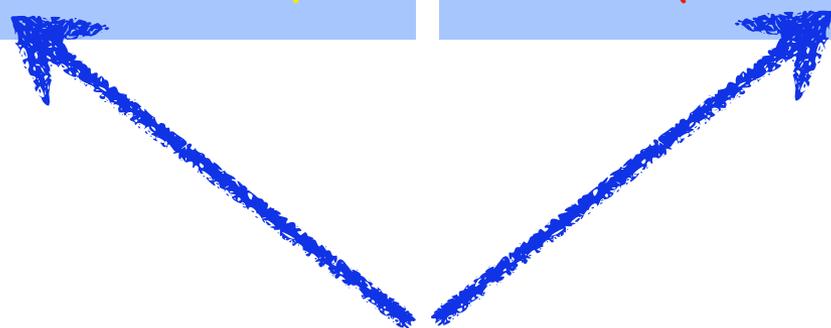
```
let foldl f b xs =  
  let rec helper res = function  
    | []      -> res  
    | x::xs' -> helper (f res x) xs'  
  in helper b xs
```

“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 =  
  foldl (fun res x -> res + x) 0
```

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

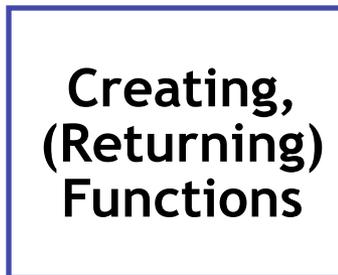


Specific Op

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*

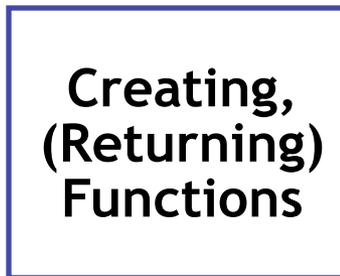


*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*

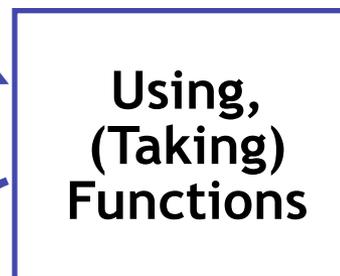
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*



*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*

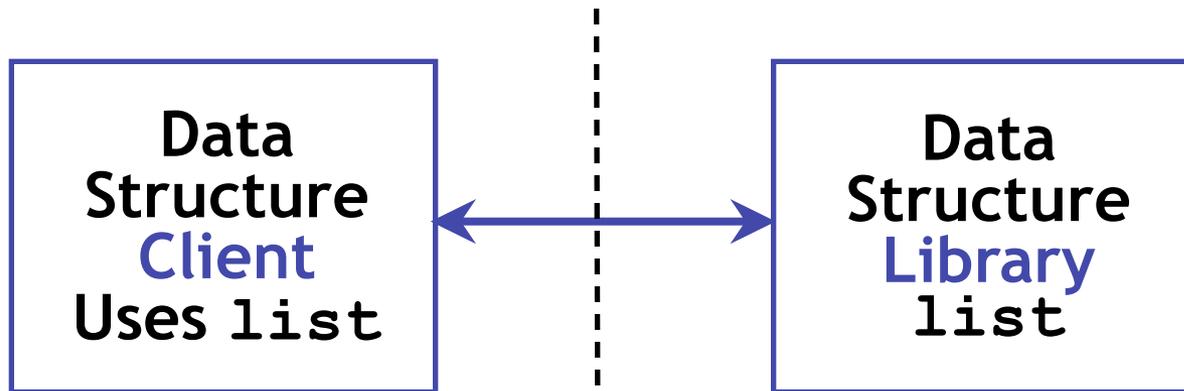


*Compose Functions:
Flexible way to build
Complex functions
from primitives.*

Funcs taking/returning funcs

Higher-order funcs enable **modular** code

- Each part only needs **local** information



Uses meta-functions:

`map, fold, filter`

With locally-dependent funcs

`(lt h), square` etc.

Without requiring Implement.
details of data structure

Provides meta-functions:

`map, fold, filter`

to traverse, accumulate over
lists, trees etc.

Meta-functions don't need client
info (tester ? accumulator ?)