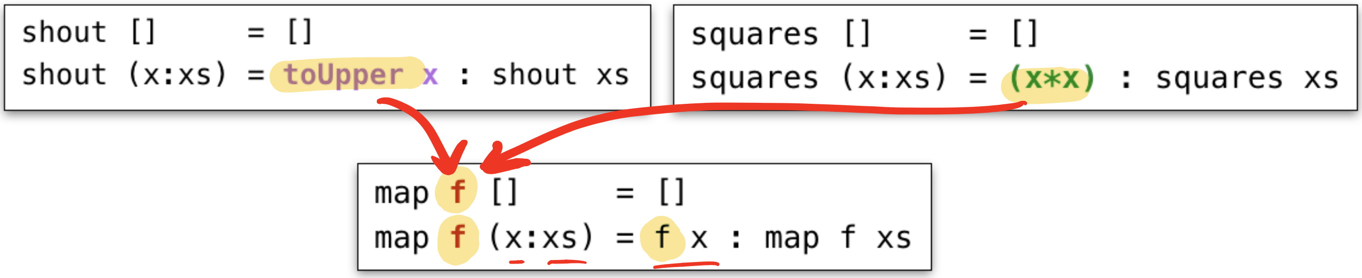
'filter'

```
filter cond [] = []
filter cond (x:xs)
    | cond x    = x : rest
    | otherwis  = rest
  where
    rest        = filter cond xs
```

The "map" pattern



```
shout []     = []
shout (x:xs) = toUpper x : shout xs
```

```
squares []     = []
squares (x:xs) = (x*x) : squares xs
```

```
map f []     = []
map f (x:xs) = f x : map f xs
```

The `map` Pattern

General Pattern

- HOF `map`
- Apply a transformation `f` to each element of a list

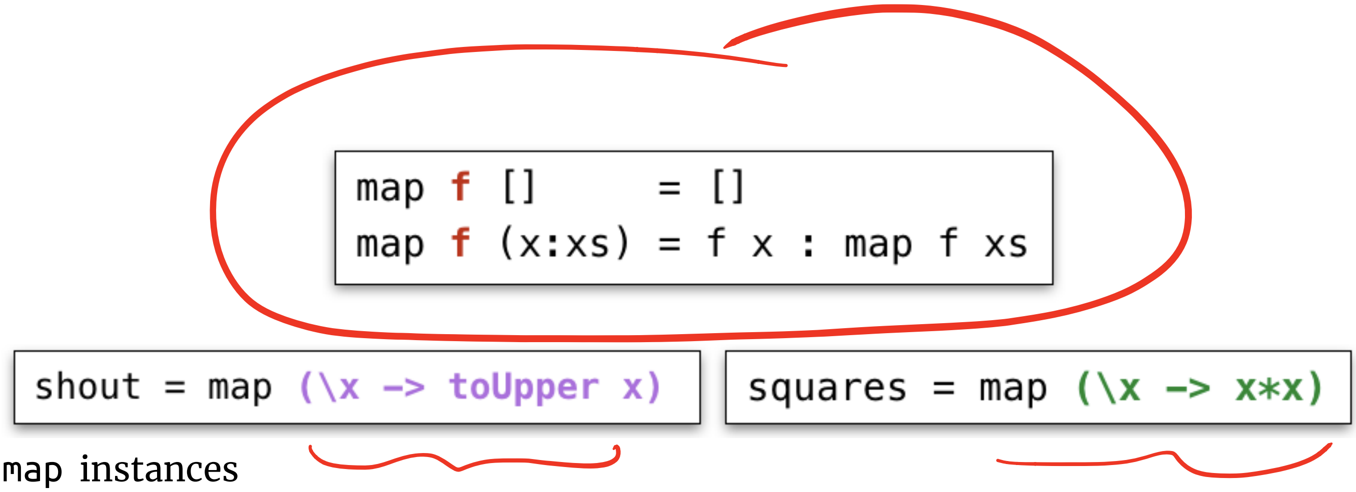
Specific Operations

- Transformations `toUpper` and `\x -> x * x`

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Lets refactor `shout` and `squares`

```haskell
shout   = map ...

squares = map ...
```



```haskell
map f []     = []
map f (x:xs) = f x : map f xs
```

```haskell
shout = map (\x -> toUpper x)
```

```haskell
squares = map (\x -> x*x)
```

`map` instances

QUIZ

What is the type of `map` ?

```haskell
map f []     = []
map f (x:xs) = f x : map f xs
```

(A) `(Char -> Char) -> [Char] -> [Char]`

(B) `(Int -> Int) -> [Int] -> [Int]`

(C) `(a -> a) -> [a] -> [a]`

(D) `(a -> b) -> [a] -> [b]`

(E) `(a -> b) -> [c] -> [d]`

```
-- For any types `a` and `b`
--   if you give me a transformation from `a` to `b`
--   and a list of `a`s,
--   I'll give you back a list of `b`s
map :: (a -> b) -> [a] -> [b]
```

f xs

Type says it all!

- The only meaningful thing a function of this type can do is apply its first argument to elements of the list
- Hoogle it!

Things to try at home:

- can you write a function `map' :: (a -> b) -> [a] -> [b]` whose behavior is different from `map`?
  ?]
- can you write a function `map' :: (a -> b) -> [a] -> [b]` such that `map' f xs` returns a list whose elements are not in `map f xs`?

X ——— X ——— X

QUIZ

What is the value of `quiz`?

```
map :: (a -> b) -> [a] -> [b]

quiz = map (\(x, y) -> x + y) [1, 2, 3]
```

Int

(Int, Int)

(A) `[2, 4, 6]`

(B) `[3, 5]`

(C) Syntax Error

(D) Type Error

(E) None of the above

Don't Repeat Yourself

Benefits of **factoring** code with HOFs:

- Reuse iteration pattern
    - think in terms of standard patterns
    - less to write / less code to fix /maintain
    - easier to communicate
- Avoid bugs due to repetition

Recall: length of a list

```
-- len []        ==> 0
-- len ["carne","asada"] ==> 2
len :: [a] -> Int
len []     = 0
len (x:xs) = 1 + len xs
```

Recall: summing a list

```
-- sum []        ==> 0
-- sum [1,2,3] ==> 6
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Example: string concatenation

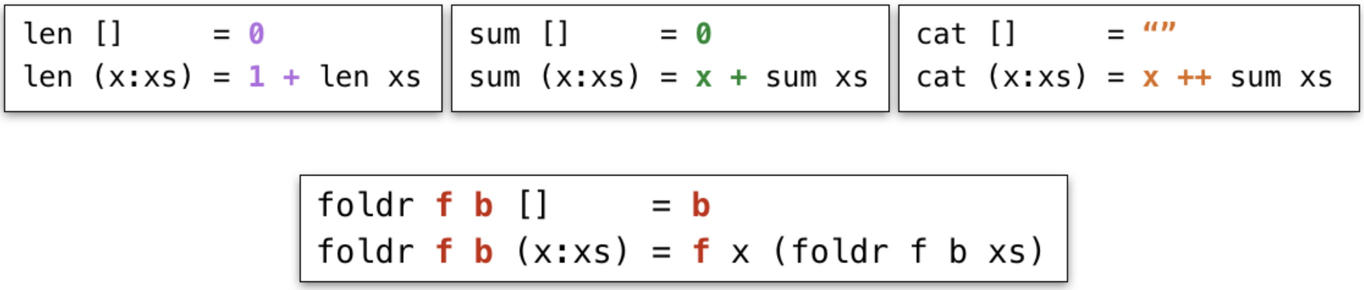
Let’s write a function `cat` :

```
-- cat [] ==> ""
-- cat ["carne","asada","torta"] ==> "carneasadatorta"
cat :: [String] -> String
cat []     = ...
cat (x:xs) = ...
```

Can you spot the pattern?

```haskell
-- len
foo []     = 0
foo (x:xs) = 1 + foo xs

-- sum
foo []     = 0
foo (x:xs) = x + foo xs

-- cat
foo []     = ""
foo (x:xs) = x ++ foo xs



pattern = ...
```

The "fold-right" pattern



```haskell
len []     = 0
len (x:xs) = 1 + len xs
```

```haskell
sum []     = 0
sum (x:xs) = x + sum xs
```

```haskell
cat []     = “”
cat (x:xs) = x ++ sum xs
```

```haskell
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

The `foldr` Pattern

General Pattern

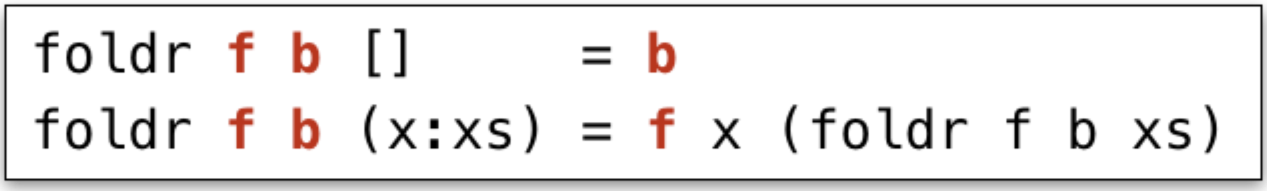
- Recurse on tail
- Combine result with the head using some binary operation

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

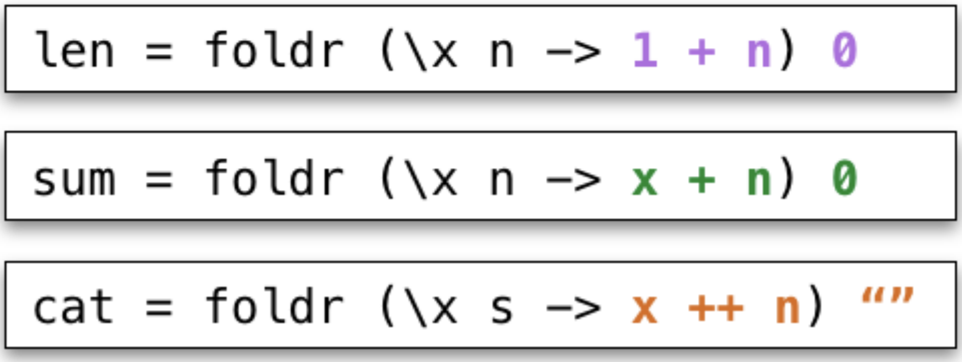
Let’s refactor `sum`, `len` and `cat`:

```
sum = foldr ...  ...

cat = foldr ...  ...

len = foldr ...  ...
```

Factor the recursion out!



```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```



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

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

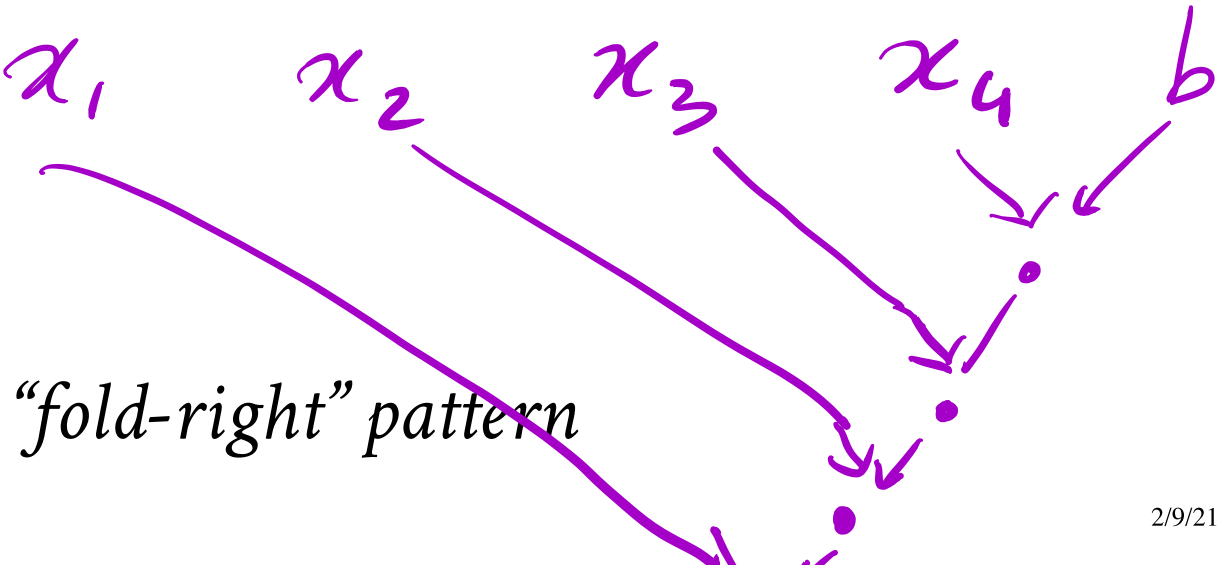
```
cat = foldr (\x s -> x ++ n) “”
```

`foldr` instances

You can write it more clearly as

```
sum = foldr (+) 0
```

```
cat = foldr (++) ""
```



The “fold-right” pattern

```
foldr f b [a1, a2, a3, a4]
  ==> f a1 (foldr f b [a2, a3, a4])
  ==> f a1 (f a2 (foldr f b [a3, a4]))
  ==> f a1 (f a2 (f a3 (foldr f b [a4])))
  ==> f a1 (f a2 (f a3 (f a4 (foldr f b []))))
  ==> f a1 (f a2 (f a3 (f a4 b)))
```

Accumulate the values from the **right**

For example:

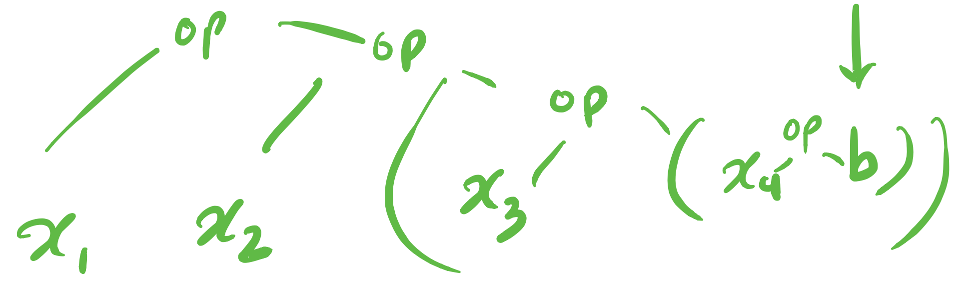
["cat", "dog", "horse"]

"cat" ++ ("dog" ++ ("horse" ++ ""))

```
foldr (+) 0 [1, 2, 3, 4]
  ==> 1 + (foldr (+) 0 [2, 3, 4])
  ==> 1 + (2 + (foldr (+) 0 [3, 4]))
  ==> 1 + (2 + (3 + (foldr (+) 0 [4])))
  ==> 1 + (2 + (3 + (4 + (foldr (+) 0 []))))
  ==> 1 + (2 + (3 + (4 + 0)))
```

$x_1 \text{ `op` } (x_2 \text{ `op` } (x_3 \text{ `op` } (x_4 \text{ `op` } b)))$

$x_1 : (x_2 : (x_3 : (x_4 : [])))$



QUIZ

What does this evaluate to?

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)

quiz = foldr (\x v -> x : v) [] [1,2,3]
```

(A) Type error

(B) `[1,2,3]`

(C) `[3,2,1]`

(D) `[[3],[2],[1]]`

(E) `[[1],[2],[3]]`

$$x_1 : (x_2 : (x_3 : (x_4 : [])))$$

$$x_1 : (x_2 : (x_3 : (x_4 : [])))$$

```
foldr (:) [] [1,2,3]
  ==> (:) 1 (foldr (:) [] [2, 3])
  ==> (:) 1 ((:) 2 (foldr (:) [] [3]))
  ==> (:) 1 ((:) 2 ((:) 3 (foldr (:) [] [])))
  ==> (:) 1 ((:) 2 ((:) 3 []))
  ==  1 : (2 : (3 : []))
  ==  [1,2,3]
```

~~QUIZ~~ HW 'EXERCISE'

What is the most general type of `foldr`?

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

(A) `(a -> a -> a) -> a -> [a] -> a`

(B) `(a -> a -> b) -> a -> [a] -> b`

(C) `(a -> b -> a) -> b -> [a] -> b`

(D) `(a -> b -> b) -> b -> [a] -> b`

(E) `(b -> a -> b) -> b -> [a] -> b`

Tail Recursive Fold

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

Is `foldr` **tail recursive**?

NOT TR

What about tail-recursive versions?

Let’s write tail-recursive `sum`!

```
sumTR :: [Int] -> Int
sumTR = ...
```

Lets run `sumTR` to see how it works

```
sumTR [1,2,3]
  ==> helper 0 [1,2,3]
  ==> helper 1   [2,3]    -- 0 + 1 ==> 1
  ==> helper 3     [3]    -- 1 + 2 ==> 3
  ==> helper 6      []    -- 3 + 3 ==> 6
  ==> 6
```

Note: `helper` directly returns the result of recursive call!

Let's write tail-recursive `cat`!

```
catTR :: [String] -> String
catTR = ...
```

Lets run `catTR` to see how it works

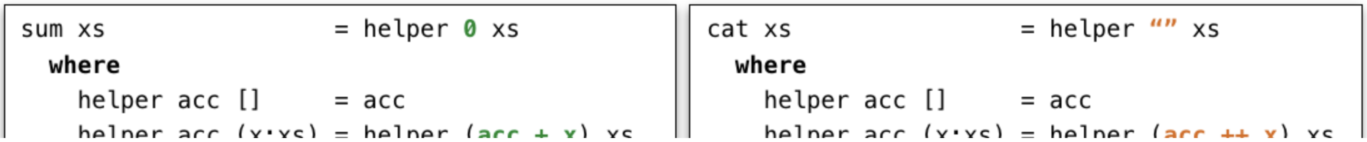
```
catTR                  ["carne", "asada", "torta"]

  ==> helper ""        ["carne", "asada", "torta"]

  ==> helper "carne"            ["asada", "torta"]

  ==> helper "carneasada"                ["torta"]

  ==> helper "carneasadatorta"                  []

  ==> "carneasadatorta"
```

Note: `helper` directly returns the result of recursive call!

Can you spot the pattern?

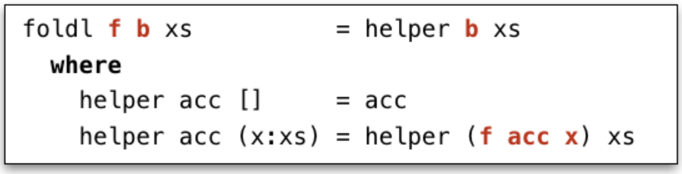
```
-- sumTR
foo xs                = helper 0 xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc + x) xs



-- catTR
foo xs                = helper "" xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc ++ x) xs



pattern = ...
```

The “fold-left” pattern



```
sum xs                = helper 0 xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc + x) xs
```

```
cat xs                = helper “” xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc ++ x) xs
```



```
foldl f b xs            = helper b xs
  where
    helper acc []       = acc
    helper acc (x:xs) = helper (f acc x) xs
```

The `foldl` Pattern

General Pattern

- Use a helper function with an extra accumulator argument
- To compute new accumulator, combine current accumulator with the head using some binary operation

```
foldl f b xs          = helper b xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (f acc x) xs
```

Let’s refactor `sumTR` and `catTR` :

```
sumTR = foldl ...  ...
```

```
catTR = foldl ...  ...
```

Factor the tail-recursion out!

QUIZ

What does this evaluate to?

```
foldl f b xs          = helper b xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (f acc x) xs

quiz = foldl (\xs x -> x : xs) [] [1,2,3]
```

(A) Type error

(B) `[1,2,3]`

(C) `[3,2,1]`

(D) `[[3],[2],[1]]`

(E) `[[1],[2],[3]]`

```
foldl f b (x1: x2: x3 : [])
  ==> helper b (x1: x2: x3 : [])
  ==> helper (f x1 b)  (x2: x3 : [])
  ==> helper (f x2 (f x1 b))  (x3 : [])
  ==> helper (f x3 (f x2 (f x1 b)))  []
  ==> ( x3 :  (x2 : (x1 : [])))
```

The "fold-left" pattern

```
foldl f b                          [x1, x2, x3, x4]
  ==> helper b                     [x1, x2, x3, x4]
  ==> helper (f b x1)                  [x2, x3, x4]
  ==> helper (f (f b x1) x2)               [x3, x4]
  ==> helper (f (f (f b x1) x2) x3)            [x4]
  ==> helper (f (f (f (f b x1) x2) x3) x4)  []
  ==> (f (f (f (f b x1) x2) x3) x4)
```

Accumulate the values from the **left**

For example:

```
foldl (+) 0                    [1, 2, 3, 4]
  ==> helper 0                 [1, 2, 3, 4]
  ==> helper (0 + 1)              [2, 3, 4]
  ==> helper ((0 + 1) + 2)           [3, 4]
  ==> helper (((0 + 1) + 2) + 3)        [4]
  ==> helper ((((0 + 1) + 2) + 3) + 4)  []
  ==> ((((0 + 1) + 2) + 3) + 4)
```

Left vs.Right

```
foldl f b [x1, x2, x3]  ==> f (f (f b x1) x2) x3  -- Left
```

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

For example:

```
foldl (+) 0 [1, 2, 3]  ==> ((0 + 1) + 2) + 3  -- Left
```

```
foldr (+) 0 [1, 2, 3]  ==> 1 + (2 + (3 + 0))  -- Right
```

Different types!

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b  -- Left

foldr :: (a -> b -> b) -> b -> [a] -> b  -- Right
```

Higher Order Functions

Iteration patterns over collections:

- **Filter** values in a collection given a *predicate*
- **Map** (iterate) a given *transformation* over a collection
- **Fold** (reduce) a collection into a value, given a *binary operation* to combine results

HOFs can be put into libraries to enable modularity

- Data structure **library** implements `map` , `filter` , `fold` for its collections
    - generic efficient implementation
    - generic optimizations: `map f (map g xs) --> map (f.g) xs`
- Data structure **clients** use HOFs with specific operations
    - no need to know the implementation of the collection

Crucial foundation of

- “big data” revolution e.g. *MapReduce*, *Spark*, *TensorFlow*
- “web programming” revolution e.g. *Jquery*, *Angular*, *React*

(https://ucsd-cse130.github.io/wi21/feed.xml) (https://twitter.com/ranjitjhala)
(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)

Generated by Hakyll (http://jaspervdj.be/hakyll), template by Armin Ronacher (http://lucumr.pocoo.org), suggest improvements here (https://github.com/ucsd-progsys/liquidhaskell-blog/).