Higher-Order Functions

Plan for this week

Last week:

- user-defined *data types*
- manipulating data-types with *pattern matching* and *recursion*
- how to make recursive functions more efficient with *tail recursion*

"Loops"

The long arc of history

Pattern matching is a *very* old PL idea …

- Variants of LISP from 1970 by Fred McBride

… but will ~~finally~~ be added to Python 3.10 (2020)

- https://www.python.org/dev/peps/pep-0622/

```
def make_point_3d(pt):
    match pt:
        case (x, y):
            return Point3d(x, y, 0)
        case (x, y, z):
            return Point3d(x, y, z)
        case Point2d(x, y):
            return Point3d(x, y, 0)
        case Point3d(_, _, _):
            return pt
        case _:
            raise TypeError("not a point we support")
```

Plan for this week

Last week:

- user-defined *data types*
- manipulating data-types with *pattern matching* and *recursion*
- how to make recursive functions more efficient with *tail recursion*

This week:

- code reuse with *higher-order functions* (HOFs)
- some useful HOFs: `map` , `filter` , and `fold`

Recursion is good…

- Recursive code mirrors recursive data
  - Base constructor -> Base case
  - Inductive constructor -> Inductive case (with recursive call)
- But it can get kinda repetitive!

Example: evens

D. R. Y.
↓ ↓ ↓
Don't Repeat Yourself

Let's write a function `evens` :

```
-- evens []        ==> []
-- evens [1,2,3,4] ==> [2,4]

evens       :: [Int] -> [Int]
evens []     = ...
evens (x:xs) = ...
```

Example: four-letter words

Let's write a function `fourChars` :

```
-- fourChars [] ==> []
-- fourChars ["i","must","do","work"] ==> ["must","work"]

fourChars :: [String] -> [String]
fourChars []     = ...
fourChars (x:xs) = ...
```

Yikes! Most Code is the Same!

Lets rename the functions to `foo` :

```
foo []          = []
foo (x:xs)
  | x mod 2 == 0  = x : foo xs
  | otherwise     =     foo xs

foo []          = []
foo (x:xs)
  | length x == 4 = x : foo xs
  | otherwise     =     foo xs
```

Only difference is **condition**

- `x mod 2 == 0` vs `length x == 4`

Moral of the day

D.R.Y. Don't Repeat Yourself!

Can we

- *reuse* the general pattern and
- *plug-in* the custom condition?

Higher-Order Functions

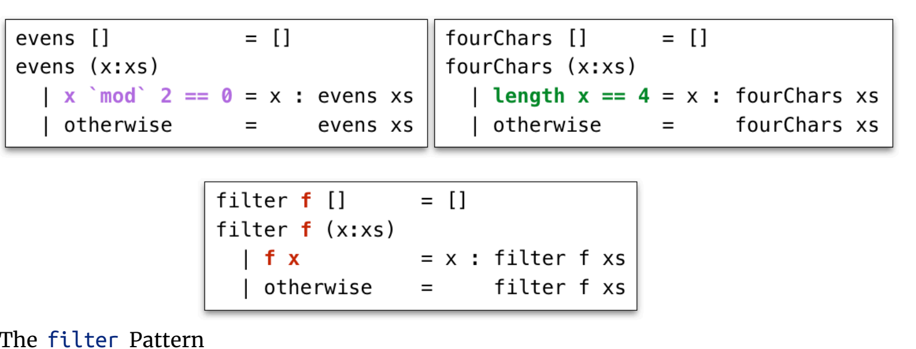
General **Pattern**

- expressed as a *higher-order function*
- takes plugin operations as *arguments*

Specific **Operation**

- passed in as an argument to the HOF

The "filter" pattern



```
evens []            = []
evens (x:xs)
  | x `mod` 2 == 0 = x : evens xs
  | otherwise      =     evens xs
```

```
fourChars []       = []
fourChars (x:xs)
  | length x == 4 = x : fourChars xs
  | otherwise     =     fourChars xs
```

```
filter f []       = []
filter f (x:xs)
  | f x           = x : filter f xs
  | otherwise     =     filter f xs
```

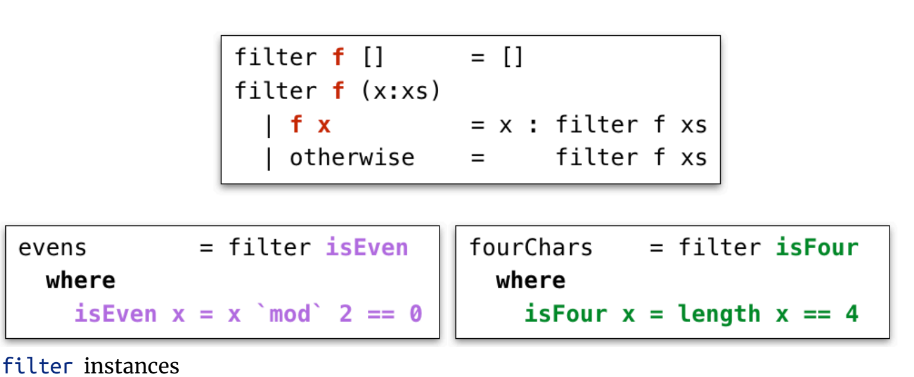
The `filter` Pattern

General Pattern

- HOF `filter`
- Recursively traverse list and pick out elements that satisfy a predicate

Specific Operations

- Predicates `isEven` and `isFour`



```
filter f []       = []
filter f (x:xs)
  | f x           = x : filter f xs
  | otherwise     =     filter f xs
```

```
evens          = filter isEven
  where
    isEven x = x `mod` 2 == 0
```

```
fourChars     = filter isFour
  where
    isFour x = length x == 4
```

`filter` instances

Avoid duplicating code!

QUIZ: What is the type of `filter`?

```
-- evens [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens xs = filter isEven xs
  where
    isEven :: Int -> Bool
    isEven x  =  x `mod` 2 == 0

-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars xs = filter isFour xs
  where
    isFour :: String -> Bool
    isFour x  =  length x == 4
```

So what's the type of `filter` ?

```
{- A -} filter :: (Int -> Bool) -> [Int] -> [Int]

{- B -} filter :: (String -> Bool) -> [String] -> [String]

{- C -} filter :: (a -> Bool) -> [a] -> [a]

{- D -} filter :: (a -> Bool) -> [a] -> [Bool]

{- E -} filter :: (a -> b) -> [a] -> [b]
```

Type of `filter`

```
-- evens [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens xs = filter isEven xs
  where
    isEven :: Int -> Bool
    isEven x  =  x `mod` 2 == 0

-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars xs = filter isFour xs
  where
    isFour :: String -> Bool
    isFour x  =  length x == 4
```

For *any* type `a`

- **Input** a *predicate* `a -> Bool` and *collection* `[a]`
- **Output** a (smaller) *collection* `[a]`

`filter :: (a -> Bool) -> [a] -> [a]`

`filter` *does not care* what the list elements are

- as long as the predicate can handle them

`filter` is **polymorphic** (generic) in the type of list elements

Example: ALL CAPS!

Lets write a function `shout` :

```
-- shout []                    ==> []
-- shout ['h','e','l','l','o'] ==> ['H','E','L','L','O']

shout :: [Char] -> [Char]
shout []     = ...
shout (x:xs) = ...
```

Example: squares

Lets write a function `squares` :

```
-- squares []        ==> []
-- squares [1,2,3,4] ==> [1,4,9,16]

squares :: [Int] -> [Int]
squares []     = ...
squares (x:xs) = ...
```

Yikes, Most Code is the Same

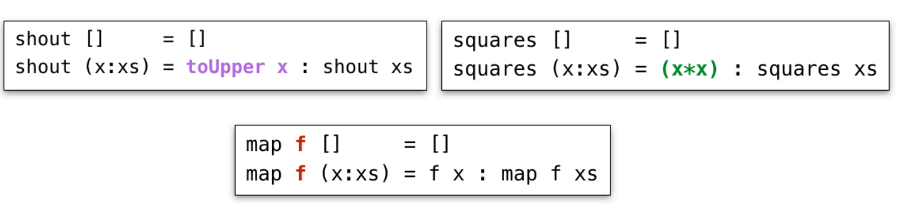
Lets rename the functions to `foo` :

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

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

Lets **refactor** into the **common pattern**

```
pattern = ...
```

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`

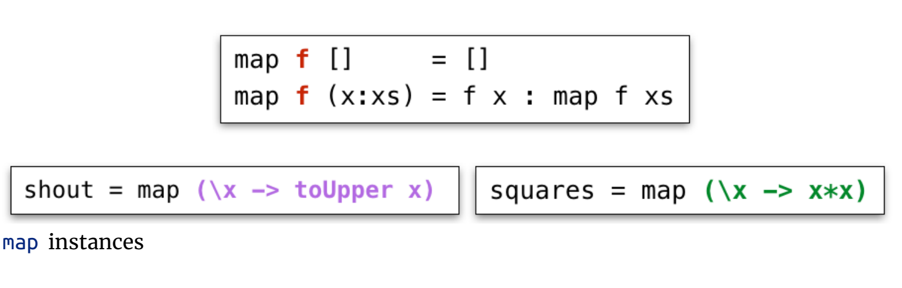
```
map f []     = []
map f (x:xs) = f x : map f xs
```

Lets refactor `shout` and `squares`

```
shout   = map ...
squares = map ...
```

toUpper

(\x -> x * x)



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

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

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

`map` instances

QUIZ

ords :: [Char] -> [Int]
ords xs = map ord xs

What is the type of `map` ?

```
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]
```

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` ?

QUIZ

What is the value of `quiz`?

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

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

a ≡ (Int, Int)
b ≡ Int
arg1
arg2
[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
  - 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

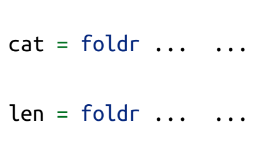
Let's write a function `cat`:

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

Can you spot the pattern?

```
-- 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 = ...
```

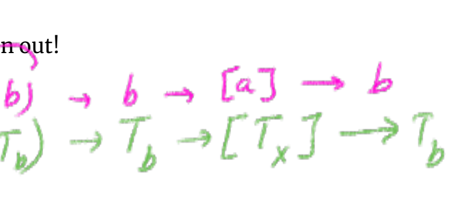
The "fold-right" pattern



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

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

```
cat []     = ""
cat (x:xs) = x ++ sum xs
```



```
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 ...  ...
```

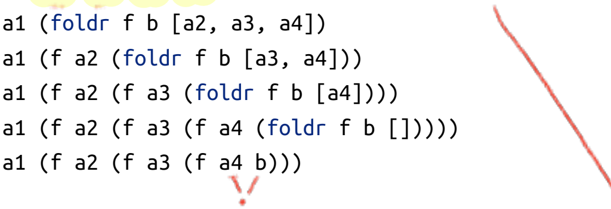
Factor the recursion out!

$(a \to b \to b) \to b \to [a] \to b$
$(T_x \to T_b \to T_b) \to T_b \to [T_x] \to T_b$

$x :: T_x$
$xs :: [T_x]$
$f :: \underbrace{T_x \to T_b \to T_b}_{T_f}$

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

T_x T_b



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

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

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

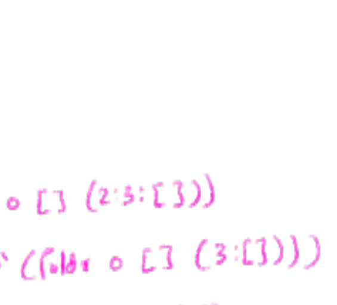
`foldr` instances

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:

```
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)))
```

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]
```

op
b

(A) Type error

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

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

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

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

→ 1 \`op\` (foldr o [] (2:3:[]))
→ 1 \`op\` (2 \`o\` (foldr o [] (3:[])))
→ 1 \`op\` (2 \`op\` (3 \`op\` ([])))
1 : 2 : [3]

```
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]
```

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

len = foldr ? 0
(1+)
(\x → 1+x)

QUIZ

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**?

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

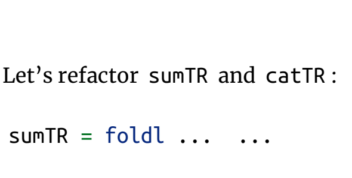
```
catTR                  ["carne", "asada", "torta"]

  ==> 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?

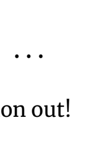
```
-- sumTR
foo xs                 = helper 0 xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc + x) xs


-- 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!

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

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

Higher Order Functions (HOF)

Iteration patterns over collections:

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

= 'op' (a → b → b)

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

2004

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