Class 4: Higher-Order Patterns

February 13

```
func (\x -> length xss …) …

vs.

func (\x -> height …) …
  where
    height = length xss
```

questions from homework

Q: is the second version more performant?

A: no need to worry about optimizations for this class.
the answer in this specific case is *probably* yes, by a little.
but the answer more generally is quite interesting!

questions from homework

```
fib 0 = 0
fib 1 = 1
fib x = fib (x - 1) + fib (x - 2)
```

questions from homework

```haskell
result1 = fib 30 + fib 30

result2 = x + x
  where
    x = fib 30

result3 = x + x
  where
    x :: Int
    x = fib 30
```

questions from homework

Q: how much does good Haskell code make use of map, filter, fold?

A: they are actually used frequently when working with lists!
many situations call for a more general or a more specific function.
but the general principle remains: break down the problem into
high-level *transformations* instead of low-level details.

questions from homework

Q: when are parentheses needed?

A: in general, parentheses in Haskell are used to *remove ambiguity*.
the most common case of this is to group together multiple space-separated components into a single function argument.

questions from homework

```
ex1 (x : xs) (y : ys) = …

ex2 = f (3 + 4) (g 3)
```

questions from homework

Q: how are anonymous functions used? when are they necessary?

A: anonymous functions allow us to construct a function (often to be used as an argument to another function) *without* having to give it a name

questions from homework

```haskell
add3All :: [Int] -> [Int]
add3All xs = map (\x -> x + 3) xs
```

questions from homework

our first higher-order functions

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

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

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

idiom:
Functions can be used without mentioning their arguments.

idiom:
Functions can be used without mentioning their arguments.

conceptually, this makes sense because
Functions don’t always have to do. They can just be.

practically, the benefit of this is
We can reason at a higher level of abstraction.

idiom:
Functions can be used without mentioning their arguments.

conceptually, this makes sense because
Functions don't always have to do. They can just be.

technique #1:
Use function names directly.

```haskell
absAll :: [Int] -> [Int]
absAll xs = map (\x -> abs x) xs
```

examples

```haskell
absAll :: [Int] -> [Int]
absAll xs = map abs xs
```

examples

```haskell
plus :: Int -> Int -> Int

sum :: [Int] -> Int
sum xs = foldr (\x acc -> plus x acc) 0 xs
```

examples

```
plus :: Int -> Int -> Int

sum :: [Int] -> Int
sum xs = foldr plus 0 xs
```

examples

```haskell
(+) :: Int -> Int -> Int

sum :: [Int] -> Int
sum xs = foldr (\x acc -> x + acc) 0 xs
```

examples

```haskell
(+) :: Int -> Int -> Int

sum :: [Int] -> Int
sum xs = foldr (\x acc -> (+) x acc) 0 xs
```

examples

```haskell
(+) :: Int -> Int -> Int

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

examples

try locally!

```haskell
and :: [Bool] -> Bool
and xs = foldr (\x acc -> x && acc) True xs
```

```haskell
and :: [Bool] -> Bool
and xs = foldr (&&) True xs
```

examples

function composition

```
compose :: (b -> c) -> (a -> b) -> (a -> c)
```

dot operator

```haskell
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g =
```

dot operator

```haskell
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x ->
```

dot operator

```haskell
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x)
```

dot operator

```haskell
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

dot operator

technique #2:

Leverage function composition.

```haskell
desort :: [Int] -> [Int]
desort xs = reverse (sort xs)
```

examples

```haskell
desort :: [Int] -> [Int]
desort = reverse . sort
```

examples

```haskell
oddOnly :: [Int] -> [Int]
oddOnly xs = filter (\x -> not (even x)) xs
```

examples

```haskell
oddOnly :: [Int] -> [Int]
oddOnly xs = filter (not . even) xs
```

try locally!

```
evenOdds :: [Int] -> Bool
evenOdds xs = even (length (oddOnly xs))
```

```haskell
evenOdds :: [Int] -> Bool
evenOdds = even . length . oddOnly
```

examples

```
evenOdds :: [Int] -> Bool
evenOdds = even . length . oddOnly
```

```
Int -> Bool   [Int] -> Int   [Int] -> [Int]
```

examples

Does this type check?

```
f :: Int -> Int
g :: Int -> Bool

f . g
```

Does this type check?

```
f :: Bool -> Int
g :: Int -> Bool

f . g
```

Does this type check?

```
f :: Int -> Bool
g :: Int -> Int

f . g
```

```haskell
desort :: [Int] -> [Int]
desort = reverse . sort
```

upon closer examination

partial application

```
f :: Int -> Int -> Int
```

currying

```
f :: Int -> Int -> Int

f :: Int -> (Int -> Int)
```

currying

```
f 3 4
```

currying

```
f 3 4

 (f 3) 4
```

currying

```haskell
f :: Int -> Int -> Int
f x y = 2 * x + y
```

currying

```
f :: Int -> (Int -> Int)
f x = \y -> 2 * x + y
```

currying

```haskell
f :: (Int -> (Int -> Int))
f = \x -> (\y -> 2 * x + y)
```

currying

function type arrows associate to the **right**

```
a -> b -> c -> d
```

same as

```
a -> (b -> (c -> d))
```

function applications associate to the **left**

```
func 3 4 5
```

same as

```
((func 3) 4) 5
```

Are these the same?

```
Int -> Int -> Int

(Int -> Int) -> Int
```

Are these the same?

```
(Int -> Int) -> Int -> Int

(Int -> Int) -> (Int -> Int)
```

Does this type check?

```
f :: Int -> Int
g :: Int -> Int

f g
```

Does this type check?

```
f :: Int -> Int -> Int
g :: Int -> Int

f g
```

Does this type check?

```
f :: (Int -> Int) -> Int
g :: Int -> Int

f g
```

Does this type check?

```
f :: (Int -> Int) -> (Int -> Int)
g :: Int -> Int

f g 3
```

Does this type check?

```
f :: (Int -> Int) -> (Int -> Int)
g :: Int -> Int

(f g) 3
```

Does this type check?

```
f :: (Int -> Int) -> (Int -> Int)
g :: Int -> Int

f (g 3)
```

```haskell
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x)
```

```haskell
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)
```

currying

```
plus :: Int -> Int -> Int

plus3 :: ???
plus3 = plus 3
```

partial application

```haskell
plus :: Int -> Int -> Int

plus3 :: Int -> Int
plus3 = plus 3
```

partial application

```haskell
plus :: Int -> Int -> Int

plus3 :: Int -> Int
plus3 = plus 3

result :: Int
result = plus3 4
```

partial application

technique #3:
Leverage partial application.

```haskell
absAll :: [Int] -> [Int]
absAll xs = map abs xs
```

examples

```haskell
absAll :: [Int] -> [Int]
absAll = map abs
```

examples

try locally!

```haskell
and :: [Bool] -> Bool
and xs = foldr (&&) True xs
```

```haskell
and :: [Bool] -> Bool
and = foldr (&&) True
```

examples

```haskell
add3All :: [Int] -> [Int]
add3All xs = map (\x -> plus 3 x) xs
```

examples

```haskell
add3All :: [Int] -> [Int]
add3All = map (\x -> plus 3 x)
```

examples

```
add3All :: [Int] -> [Int]
add3All = map (plus 3)
```

examples

```haskell
add3All :: [Int] -> [Int]
add3All = map (\x -> 3 + x)
```

examples

```haskell
add3All :: [Int] -> [Int]
add3All = map (3 +)
```

examples

```
(\x -> 3 + x) same as (3 +)
(\x -> x + 3) same as (+ 3)
```

operator sections

try locally!

```haskell
greaterThan100 :: [Int] -> [Int]
greaterThan100 = filter (\x -> x > 100)
```

examples

```haskell
greaterThan100 :: [Int] -> [Int]
greaterThan100 = filter (> 100)
```

examples

All together now!

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\x acc -> f x : acc) [] xs
```

examples

```haskell
map :: (a -> b) -> [a] -> [b]
map f = foldr (\x acc -> f x : acc) []
```

examples

```
map :: (a -> b) -> [a] -> [b]
map f = foldr ((:) . f) []
```

```
youngNames :: [Person] -> [String]
youngNames xs =
   map name (filter (\x -> age x <= 18) xs)
```

```haskell
youngNames :: [Person] -> [String]
youngNames =
   map name . filter (\x -> age x <= 18)
```

```haskell
youngNames :: [Person] -> [String]
youngNames =
   map name . filter ((<= 18) . age)
```

wholemeal programming

```haskell
func :: [Int] -> Int
func [] = 1
func (x : xs)
  | x > 3      = (x + 7) * func xs
  | otherwise = func xs
```

```haskell
func :: [Int] -> Int
func [] = 1
func (x : xs)
  | x > 3     = (x + 7) * func xs
  | otherwise = func xs
```

```
func :: [Int] -> Int
func [] = 1
func (x : xs)
  | x > 3      = (x + 7) * func xs
  | otherwise = func xs
```

```haskell
func :: [Int] -> Int
func [] = 1
func (x : xs)
  | x > 3     = (x + 7) * func xs
  | otherwise = func xs
```

```haskell
func :: [Int] -> Int
func = product . map (+ 7) . filter (> 3)
```