

# COMP90048: Workshop 4

Eleanor McMurtry, University of Melbourne



# Higher Order Functions

# Intro

- In Haskell, functions are *first-class*: they are perfectly good values in their own right.
- We'll see how we can use this to our advantage, similar to function pointers in C.

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```



# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

# Filtering lists

```
even :: Integral a => a -> Bool
```

```
even x = x `mod` 2 == 0
```

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter even list
```

```
Prelude> onlyEvens [4, 5, 8, 7, 5, 3, 2]
```

```
[4, 8, 2]
```

# Lambda functions

- Short, anonymous functions that won't be reused
- Named after the *lambda calculus*

# Lambda functions

```
onlyEvens :: Integral a => [a] -> [a]
```

```
onlyEvens list = filter (\x -> x `mod` 2 == 0) list
```

# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```



# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```



**section:** shorthand for `(\x -> x^2)`

# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```

```
Prelude> squared [1, 2, 3, 4]
```

# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```

```
Prelude> squared [1, 2, 3, 4]
```



1

# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```

```
Prelude> squared [1, 2, 3, 4]
```



1 4

# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```

```
Prelude> squared [1, 2, 3, 4]
```

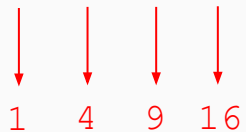
1 4 9

# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```

```
Prelude> squared [1, 2, 3, 4]
```



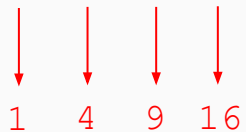
↓	↓	↓	↓
1	4	9	16

# Mapping lists

```
squared :: Integral a => [a] -> [a]
```

```
squared list = map (^2) list
```

```
Prelude> squared [1, 2, 3, 4]
```



↓	↓	↓	↓
1	4	9	16

```
[1, 4, 9, 16]
```

# Folding lists together

```
sum :: Num a => [a] -> a
```

```
sum list = foldr (\x acc -> acc + x) list
```



# Folding lists together

```
sum :: Num a => [a] -> a
```

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



**accumulator:** final result is accumulated ("built up")

# Folding lists together

```
sum :: Num a => [a] -> a
```

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



starting value

# Folding lists together

```
sum :: Num a => [a] -> a
```

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

```
Prelude> sum [1, 2, 3]
```

# Folding lists together

```
sum :: Num a => [a] -> a
```

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

```
Prelude> sum [1, 2, 3]
```


0

# Folding lists together

```
sum :: Num a => [a] -> a
```

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

```
Prelude> sum [1, 2, 3]
```



3 + 0

# Folding lists together

```
sum :: Num a => [a] -> a
```

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

```
Prelude> sum [1, 2, 3]
```

← 2 + 3 + 0

# Folding lists together

```
sum :: Num a => [a] -> a
```

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

```
Prelude> sum [1, 2, 3]
```

 1 + 2 + 3 + 0

# Folding lists together

```
sum :: Num a => [a] -> a
```

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

```
Prelude> sum [1, 2, 3]
```

 1 + 2 + 3 + 0

6



# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

"a"

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

"x" ++ "a"

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

```
"y" ++ ("x" ++ "a")
```

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

`"y" ++ ("x" ++ "a")`  
• associates rightwards

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

`"y" ++ ("x" ++ "a")`

- associates rightwards

`"a"`

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

`"y" ++ ("x" ++ "a")`

- associates rightwards

`"a" ++ "x"`

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

`"y" ++ ("x" ++ "a")`

- associates rightwards

`("a" ++ "x") ++ "y"`



# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

`"y" ++ ("x" ++ "a")`

- associates rightwards

`("a" ++ "x") ++ "y"`

- associates leftwards

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"axy"
```

`"y" ++ ("x" ++ "a")`

- associates rightwards

`("a" ++ "x") ++ "y"`

- associates leftwards

- Most commonly use foldr (can terminate on infinite lists – *why?*)

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

`"y" ++ ("x" ++ "a")`  
• associates rightwards

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"Axy"
```

`("a" ++ "x") ++ "y"`  
• associates leftwards

- Most commonly use foldr (can terminate on infinite lists – *why?*)... but can stack overflow.

# Left vs right folds

```
Prelude> foldr (++) "a" ["x", "y"]
```

```
"xya"
```

```
Prelude> foldl (++) "a" ["x", "y"]
```

```
"Axy"
```

- Most commonly use foldr (can terminate on infinite lists – *why?*)... but can stack overflow.
- foldl' is **strict** and can be faster than foldr

Try these techniques out in Grok  
Module 5.