### Today

- Final overview
- Higher-Order Functions
- Wrap up

### Assignments

- HW10 out
- Exam 3 due
Final Exam Overview

Basics:
- Thursday, May 7th, 3:30–5:30
- Open book and notes
- Worth 20% of final grade
- Zoom and blackboard (must be logged in to zoom to take exam!)

Topics:
- Select (variations of) questions from Exam’s 1–3
- Go over quizzes
- Go over lecture notes
- Additional topics:
  - Haskell user-defined data types
  - Higher order functions (from today)
More on Higher-Order Functions

Recall: A higher order function ...

- is a function that takes functions as arguments
- or returns a function as a result ... e.g., partial application

We'll look at some more examples of higher-order functions

- we've already seen: filter, any/all, map, zipWith ...
- new functions: function composition, foldl, foldr
**Function Composition**

Assume functions $f(x)$ and $g(x)$ (e.g., $f(x) = x + 1$ and $g(x) = x - 1$)

Q: What is $g \circ f$?

- Function composition!

\[(g \circ f)(x) = g(f(x))\]

The ‘.’ function in Haskell implements function composition

```
Prelude> :type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Q: What is this saying?

```
Prelude> :type length
length :: [a] -> Int

Prelude> :type words
words :: String -> [String]
```

Q: What is the type of this expression?

```
Prelude> let f = length . words

Prelude> :type f
f :: String -> Int

Prelude> f "the quick brown fox"
4

Prelude> (length . words) "the quick brown fox"
4
Q: What about this expression?

```haskell
Prelude> :type head . words
head . words :: String -> String
```

```haskell
Prelude> (head . words) "the quick brown fox"
the
```

Q: What about this expression?

```haskell
Prelude> :type words . head
words . head :: [String] -> [String]
```

```haskell
Prelude> (words . head) ["the quick brown fox", "blah"]
["the", "quick", "brown", "fox"]
```

Q: What about this expression?

```haskell
Prelude> :type tail . head
tail . head :: [[a]] -> [a]
```

Q: And this expression?

```haskell
Prelude> :type head . length
... error ...
```

- here the types do not align!
Accumulating values (from the left) with `foldl`

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

- `(a -> b -> a)` is the **step function**
- first `a` is an **accumulator**
- `[b]` are the **input values**
- last `a` is the **accumulated value**
- `new_accumulator = step_function accumulator b_value`

```haskell
Prelude> foldl (+) 0 [1, 2, 3]
6
```
- This just sums up the list of values

**How it works ...**

```
foldl _ acc [] = acc
foldl step acc (x:xs) = foldl step (step acc x) xs
```

```
foldl (-) 9 [5, 3, 1]
==> foldl (-) (9 - 5) [3, 1]
==> foldl (-) (((9 - 5) - 3) - 1) []
==>opathic
```

```haskell
==> 0
```
Accumulating values from the right with foldr

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

Prelude> foldr (+) 0 [1, 2, 3]  -- same as foldl in this case
6

• Similar to foldl but works right-to-left
• new_accumulator = step_function b_value accumulator
• ... where the b_value is from the list

How it works ...

foldr _ acc [] = acc
foldr step acc (x:xs) = step x (foldr step acc xs)

foldr (-) 9 [5, 3, 1]  -- try 3 instead of 9
==&gt; 5 - (foldr (-) 9 [3, 1])
==&gt; 5 - (3 - (foldr (-) 9 [1]))
==&gt; 5 - (3 - (1 - (foldr (-) 9 [])))
==&gt; 5 - (3 - (1 - 9))
==&gt; -6
Many recursive functions follow the fold pattern

\begin{verbatim}
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
    | p x = x : filter p xs
    | otherwise = filter p xs
\end{verbatim}

Q: How can \texttt{filter} be defined using \texttt{foldr}?

\begin{verbatim}
filter' p xs = foldr step [] xs
    where step x acc
        | p x = x : acc
        | otherwise = acc
\end{verbatim}

- For example ...

\begin{verbatim}
filter' odd [1,2,3]
==> foldr step [] [1,2,3]
==> step 1 (foldr step [] [2,3])
==> 1 : (foldr step [] [2,3])
==> 1 : (step 2 (foldr step [] [3]))
==> 1 : (foldr step [] [3])
==> 1 : (step 3 foldr step [] [])
==> 1 : (3 : (foldr step [] []))
==> 1 : (3 : [])
\end{verbatim}

Q: How can \texttt{filter} be defined using \texttt{foldl}?

\begin{verbatim}
filter' p xs = foldl step [] xs
    where step acc x
        | p x = acc ++ [x]
        | otherwise = acc
\end{verbatim}
We can also define map using foldr

\[ \text{map'} :: (a \to b) \to [a] \to [b] \]
\[ \text{map'} f \text{ xs} = \text{foldr step \([\] \text{ xs}) \]
\[ \text{where step \(x \text{ ys} = f \ x : \text{ys} \]}

For example ...

\[ \text{map'} \text{ odd \([1,2,3]\]}
\[ == f\text{oldr step \([\] \text{ [1,2,3] \)]}
\[ == \text{step 1 \((\text{foldr step \([\] \text{ [2,3]})\)]}
\[ == \text{odd 1 : (odd 2 : (}\text{step 3 \((\text{foldr step \([\] \text{ []})\)})\)]}
\[ == \text{odd 1 : (odd 2 : (odd 3 : (\text{foldr step \([\] \text{ []})\})\)])\)]
\[ == \text{odd 1 : (odd 2 : (odd 3 : [\] ))\)}\]

Why care about these higher-order functions?

- In general, should use them whenever possible ...
- ... Can make functions easier to understand (shorter)
- ... Well behaved (fewer bugs)
- ... Optimization
List Comprehensions

List comprehensions mimic set definitions (“set builder” notation):

\[ A \times B = \{(a, b) \mid a \in A \land b \in B\} \]

- The cartesian product operation

Using list comprehensions:

\[ \text{cprod } xs \ ys = [(x,y) \mid x \leftarrow xs, y \leftarrow ys] \]

- here \( xs \) and \( ys \) have to be in scope

Another example:

\[ [x \times 2 \mid x \leftarrow \{1..10]\} \]

With a predicate (to filter)

\[ [x \times y \mid x \leftarrow [1..10], y \leftarrow [1..3], \text{even } x] \]

In general, a lot like lambda functions

- but for defining lists “on the fly”
- syntactic sugar for defining lists (list set builders)
- can use anywhere you’d expect a list
Wrapping up

What we didn’t cover

- A lot! ... Haskell has *many* features

- The IO type (recall “purity”)
  - Various I/O operations
  - All the normal things you’d expect to write real apps

- Monads
  - A pattern (data type) to chain together a list of operations
  - Functional machinery to sequence commands
  - The `do` expression is an example
    
    ```haskell
    do input <- readFile inputFile
       putStrLn (function input)
    ```
  - Appears imperative, but is still functional ...

- The **Monad** typeclass has a sequencing function `>>=`

  ```haskell
  Prelude> :type (>>=)
  (>>=) :: (Monad m) => m a -> (a -> m b) -> m b
  ```
  - extracts (unwraps) value on left from the Monad
  - passes it to a function that returns a wrapped result value
• And a `return` function

```haskell
Prelude> :type return
return :: (Monad m) => a -> m a
```
– Takes a value and wraps it into a Monad member

• Here is a (silly) example

```haskell
wrapIt x = Just x

go = wrapIt "hi" >>=
  \v1 -> wrapIt (head v1) >>=
  \v2 -> return v2
```

```haskell
Prelude> go
Just 'h'
```
– `return` for `Maybe` is defined to create a `Just` value
– i.e., `return x = Just x`

• The `do` keyword provides a shorthand ...

```haskell
go' = do
  v1 <- wrapIt "hi"
  v2 <- wrapIt (head v1)
  return v2 # return does the wrapping
```

• Here is a better example using the IO monad

```haskell
do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Nice to meet you " ++ name)
```