Today

- Heaps
- Higher-order functions

Assignments

- HW-6, R-6 out
- HW-7, R-7 out

Reminders

- Exam 1 on Thurs.
- Proj 1 due date extended to Tues, Mar 15th
More on Higher-Order Functions

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 \( f \circ g \)?

- Function composition!

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

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

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

Q: What about this expression?

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

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

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]
===> foldl (-) (((9 - 5) - 3) - 1) []
===> (((9 - 5) - 3) - 1)
===> 0
Accumulating values from the right with \texttt{foldr}

\texttt{foldr :: (b -> a -> a) -> a -> [b] -> a}

Prelude> \texttt{foldr (+) 0 [1, 2, 3]} \hspace{1em} -- same as \texttt{foldl} in this case
6

\begin{itemize}
  \item Similar to \texttt{foldl} but works right-to-left
  \item \texttt{new\_accumulator = step\_function a\_value accumulator}
  \item \texttt{... where the b\_value is from the list}
\end{itemize}

\textbf{How it works ...}

\begin{verbatim}
foldr _ acc [] = acc
foldr step acc (x:xs) = step x (foldr step acc xs)
\end{verbatim}

\begin{verbatim}
foldr (-) 9 [5, 3, 1] \hspace{1em} -- try 3 instead of 9
==\> 5 - (foldr (-) 9 [3, 1])
==\> 5 - (3 - (foldr (-) 9 [1]))
==\> 5 - (3 - (1 - (foldr (-) 9 [])))
==\> 5 - (3 - (1 - 9))
==\> -6
\end{verbatim}
Many recursive functions follow the fold pattern

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
    | p x = x : filter p xs
    | otherwise = filter p xs

Q: How can filter be defined using foldr?

    filter' p xs = foldr step [] xs
        where step x acc
            | p x = x : acc
            | otherwise = acc

• For example ...

    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 : [])

Q: How can filter be defined using foldl?

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

\begin{verbatim}
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr step [] xs
    where step x ys = f x : ys
\end{verbatim}

For example ...

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

\textbf{Why care about these higher-order functions?}

\begin{itemize}
  \item In general, should use them whenever possible ...
  \item ... Can make functions easier to understand (shorter)
  \item ... Well behaved (fewer bugs)
  \item ... Optimization
\end{itemize}