Today

- Quiz 10
- User-Defined Types (wrap up)
- Higher-Order Functions (cont)

Assignments

- HW 10 out (due Tues)
- Extra credit out (due by end of semester)
- Proj Status Update 2 (due Tue, Apr 23)

Warm Up ...

data List a = Node a (List a)
    | Nil
    deriving (Show, Eq)

Exercise: Write a function to return the last element of a List

removeLast :: List a -> List a
removeLast Nil = error "empty list"
removeLast (Node _ Nil) = Nil
removeLast (Node x tail) = Node x (removeLast tail)
Binary Trees

Exercise: Define a binary tree structured using a parameterized type

```haskell
data Tree a = Node a (Tree a) (Tree a)
            | Nil
    deriving (Show, Eq)

• Again, this is a recursive data structure
```

Exercise: Create a 3-node int and 5-node string binary search tree

```haskell
tree1 = Node 2 (Node 1 Nil) (Node 3 Nil)

subtree1 = Node "b" (Node "a" Nil Nil) (Node "c" Nil Nil)
tree2 = Node "d" subtree1 (Node "e" Nil Nil)
```

Exercise: Write an isEmpty function for trees

```haskell
isEmpty Nil = True
isEmpty _   = False
```

Exercise: Write a size function for the tree (# of nodes)

```haskell
size Nil = 0
size (Node _ l r) = 1 + size l + size r
```

Exercise: Write a height function for the tree (# of levels)

```haskell
-- note max is defined in Haskell
height Nil = 0
height (Node _ l r) = 1 + max (height l) (height r)
```
Example Binary Search Tree Implementation

\[
\text{insert} :: \text{(Ord} \ a) \Rightarrow a \rightarrow \text{Tree} \ a \rightarrow \text{Tree} \ a
\]
\[
\text{insert} \ x \ \text{Nil} = \text{Node} \ x \ \text{Nil} \ \text{Nil}
\]
\[
\text{insert} \ x \ (\text{Node} \ y \ l \ r)
\]
\[
| \ x < y \quad = \text{Node} \ y \ (\text{insert} \ x \ l) \ r
\]
\[
| \ \text{otherwise} \quad = \text{Node} \ y \ l \ (\text{insert} \ x \ r)
\]

\[
\text{find} :: \text{(Ord} \ a) \Rightarrow a \rightarrow \text{Tree} \ a \rightarrow \text{Bool}
\]
\[
\text{find} \ x \ \text{Nil} = \text{False}
\]
\[
\text{find} \ x \ (\text{Node} \ y \ l \ r)
\]
\[
| \ x == y \quad = \text{True}
\]
\[
| \ x < y \quad = \text{find} \ x \ l
\]
\[
| \ \text{otherwise} \quad = \text{find} \ x \ r
\]

\[
\text{delete} :: \text{(Ord} \ a) \Rightarrow a \rightarrow \text{Tree} \ a \rightarrow \text{Tree} \ a
\]
\[
\text{delete} \ x \ \text{Nil} = \text{Nil}
\]
\[
\text{delete} \ x \ (\text{Node} \ y \ l \ r)
\]
\[
| \ x < y \quad = \text{Node} \ y \ (\text{delete} \ x \ l) \ r
\]
\[
| \ x > y \quad = \text{Node} \ y \ l \ (\text{delete} \ x \ r)
\]
\[
| \ x == y \quad = \text{delete'} \ (\text{Node} \ x \ l \ r)
\]

where
\[
\text{delete'} \ (\text{Node} \ _ \ \text{Nil} \ \text{Nil}) = \text{Nil}
\]
\[
\text{delete'} \ (\text{Node} \ _ \ l \ \text{Nil}) = l
\]
\[
\text{delete'} \ (\text{Node} \ _ \ \text{Nil} \ r) = r
\]
\[
\text{delete'} \ (\text{Node} \ x \ l \ r)
\]
\[
\quad = \begin{aligned}
&\text{let} \ y = \text{inorderSucc} \ r \\
&\text{in} \ \text{Node} \ y \ l \ (\text{delete} \ y \ r)
\end{aligned}
\]
\[
\text{inorderSucc} \ (\text{Node} \ x \ \text{Nil} \ _) = x
\]
\[
\text{inorderSucc} \ (\text{Node} \ _ \ l \ _) = \text{inorderSucc} \ l
\]
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 ‘\( \cdot \)’ 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?

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

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

Q: What about this expression?

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?

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

Q: And this expression?

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
```