Today

- User-defined types (cont)
- Higher-order functions

Assignments

- R-10, HW-10 out (due Thurs)
- HW-11 out (due next Thurs)
More on Parameterized Types

Exercise 1: Define a linked list structure using a parameterized type

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

- This is a “recursive” data structure
- Can use with pattern matching
- Uses default implementations of `show` and `(==)

Exercise 2: Create a 3-element list of strings and a 4-element list of ints

```haskell
list1 = Node "foo" (Node "bar" (Node "baz" Nil))  
list2 = Node 1 (Node 2 (Node 3 (Node 4 Nil)))
```

Q: What are the types of the two lists?

```haskell
:type list1  
list1 :: List [Char]

:type list2  
list2 :: List Integer

:type Node 1 (Node 2 Nil)  
Node 1 (Node 2 Nil) :: Num a => List a
```
We can use pattern matching to define List functions:

```haskell
isEmpty :: List a -> Bool
isEmpty Nil = True
isEmpty _ = False
```

**Exercise:** Write a function to return the length of a List

```haskell
listLength :: List a -> Int
listLength Nil = 0
listLength (Node _ tail) = 1 + listLength tail
```

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

```haskell
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 1: 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 2: 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 3: Write an isEmpty function for trees

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

Exercise 4: Write a size function for the tree (# of nodes)

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

Exercise 5: 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

```haskell
insert :: (Ord a) => a -> Tree a -> Tree a
insert x Nil = Node x Nil Nil
insert x (Node y l r)
  | x < y    = Node y (insert x l) r
  | otherwise = Node y l (insert x r)

find :: (Ord a) => a -> Tree a -> Bool
find x Nil = False
find x (Node y l r)
  | x == y    = True
  | x < y    = find x l
  | otherwise = find x r

delete :: (Ord a) => a -> Tree a -> Tree a
delete x Nil = Nil
delete x (Node y l r)
  | x < y    = Node y (delete x l) r
  | x > y    = Node y l (delete x r)
  | x == y    = delete' (Node x l r)

where
  delete' (Node _ Nil Nil) = Nil
  delete' (Node _ l Nil) = l
  delete' (Node _ Nil r) = r
  delete' (Node x l r) =
    let y = inorderSucc r
    in Node y l (delete y r)
inorderSucc (Node x Nil _) = x
inorderSucc (Node _ l _) = inorderSucc l
```
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 \( g \circ f \)?

- 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?

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!