Today

- User-Defined Types (wrap up)

Assignments

- HW10 out
- Exam 3 out (due by class next Thurs)

Note on where with guards vs. let

\[
f x \\
| g1 = e1 \\
| g2 = e2 \\
where ... 
\]

\[
f x \\
| g1 = let ... in e1 \\
| g2 = let ... in e2 
\]
More on Parameterized Types

A linked list can be defined using a (recursive) parameterized type

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

- A node value consists of an `a`-value followed by an `a`-list value
- `Nil` is a list "terminator" value
- Uses default implementations of `show` and `(==)`

**Exercise**: 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 :: Num a => List a

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

\begin{verbatim}
isEmpty :: List a -> Bool
isEmpty Nil = True
isEmpty _   = False
\end{verbatim}

\textbf{Exercise}: Write a function to return the length of a \texttt{List}

\begin{verbatim}
listLength :: List a -> Int
listLength Nil = 0
listLength (Node _ tail) = 1 + listLength tail
\end{verbatim}
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

```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 _ 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
```

S. Bowers 5 of 5