Comp 205: Comparative Programming Languages

More User-Defined Types

- Tree types
- Catamorphisms

Lecture notes, exercises, etc., can be found at:
www.csc.liv.ac.uk/~grant/Teaching/COMP205/

Recursive Type Definitions

Some recursively-defined types:

```haskell
data Nat = Zero | Succ Nat
data List a = Nil | Cons a (List a)
data BTree a = Leaf a |
  Fork (BTree a) (BTree a)
```

Recursive Functions

Some recursive functions on Nat:

```haskell
value :: Nat -> Int
value Zero = 0
value (Succ n) = 1 + value n
```

More Recursive Functions

Some more recursive functions on Nat:

```haskell
data Nat = Zero | Succ Nat
nplus :: Nat Nat -> Nat
nplus m Zero = m
nplus m (Succ n) = Succ (nplus m n)
ntimes :: Nat Nat -> Nat
ntimes m Zero = Zero
ntimes m (Succ n) = nplus m (ntimes m n)
```

Recursive Functions on BTrees

Some recursive functions on BTrees:

```haskell
sumAll :: BTree Int -> Int
sumAll (Leaf n) = n
sumAll (Fork t1 t2) = (sumAll t1)+(sumAll t2)
leaves :: BTree a -> [a]
leaves (Leaf x) = [x]
leaves (Fork t1 t2) = (leaves t1)++(leaves t2)
```
Recursive Functions on BTrees

```haskell
data BTree a = Leaf a | Fork (BTree a) (BTree a)

bTree_Map :: (a -> b) -> (BTree a) -> (BTree b)
bTree_Map f (Leaf x) = Leaf (f x)
bTree_Map f (Fork t1 t2) = Fork (bTree_Map f t1) (bTree_Map f t2)
```

Catamorphisms

```haskell
bTree_Reduce :: (a -> b) -> (b -> b -> b) -> (BTree a) -> b
bTree_Reduce f g (Leaf x) = f x
bTree_Reduce f g (Fork t1 t2) = g (bTree_Reduce f g t1) (bTree_Reduce f g t2)
```

Using Reduce

The recursive functions above can all be expressed as reductions:

```haskell
sumAll = bTree_Reduce id (+)
    where
        id x = x
leaves = bTree_Reduce (:[[]]) (++)
bTree_Map f = bTree_Reduce (Leaf.f) Fork
```

Replacing Constructors

Summary

- Recursive types
- Catamorphisms

Next: Infinite lists and lazy evaluation