Outline of Lecture 11

- User-defined types revisited. Recursive datatypes
- Polymorphic datatypes
- More about modules
- Abstract data types

User-defined datatypes revisited

- The general form of datatypes (seen so far):

```
data TypeName =
    Con1 t11 ... t1k1 |
    Con2 t21 ... t2k2 |
    ...
    Conn tn1 ... tnkn
```

  where $\texttt{Con}_\texttt{i}$ are constructors and $\texttt{t}_\texttt{ij}$ are types names

- Each constructor (function) $\texttt{Con}_\texttt{i}$ takes arguments of the types $\texttt{t}_\texttt{i1}$ ... $\texttt{t}_\texttt{ik}_\texttt{i}$ and returns a result of the type `TypeName`

- Two possible extensions: datatypes can be recursive and/or polymorphic

Recursive datatypes

Types are often naturally described in terms of themselves.

- Very simple arithmetic expressions, e.g. (17-5)+11:

```
data Expr = Lit Integer |
  Add Expr Expr |
  Sub Expr Expr
```

  The above expression: `Add (Sub (Lit 17) (Lit 5)) (Lit 11)`

- Trees of integers:

```
data NTree = NilT |
  Node Integer NTree Ntree
```

  A tree is either nil or is given by combining a value and two sub-trees, e.g., `Node 10 NilT NilT`

Recursive datatypes (cont.)

Writing functions (based on primitive recursion) for a recursive datatype:

```
eval ::  Expr -> Integer

eval (Lit n) = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

- At the non-recursive case, base cases (`Lit n` here), the value is given outright
- At the recursive cases, the function value can be based on function applications on the respective sub-expressions
- Recursive evaluation typically matches (follows) the recursive datatype structure

Recursive datatypes (cont.)

Example: the depth of an integer tree

```
depth ::  NTree -> Integer

depth NilT = 0
depth (Node _ t1 t2) p = 1 + max (depth t1) (depth t2)
```

Example: how many times the number occurs in an integer tree

```
occurs ::  NTree -> Integer -> Integer

occurs NilT _ = 0
occurs (Node n t1 t2) p
  | n == p =  1 + (occurs t1 p) + (occurs t2 p)
  | otherwise =  (occurs t1 p) + (occurs t2 p)
```

Mutually recursive types

In describing one type, it is often useful to use others; these in turn may refer back to the original type.
This gives us a pair of **mutually recursive** datatypes

```
data Person = Adult Name Address Bio |
  Child Name
data Bio = Parent String [Person] |
  NonParent String
```

Mutually recursive functions may be needed in those cases:

```
showPerson (Adult nm ad bio) =
  show nm ++ show ad ++ showBio bio
  ...
showBio (Parent st perList)
  st ++ concat (map showPerson perList)
  ...
```

Polymorphic datatypes

- Datatype descriptions can contain the type variables `a`, `b`, `...` defining polymorphic types.
- The type variables appear after the type name, e.g.,

```
data Pairs a = Pr a a
```

- Examples of such pairs: `Pr 2 3 :: Pairs Integer`, or `Pr [] [9] :: Pairs [Int]`, or `Pr [] [] :: Pairs [a]`
- A function testing equality of such pairs:

```
equalPair ::  Eq a => Pairs a -> Bool
equalPair (Pr x y) = (x == y)
```

Polymorphic datatypes (cont.)

- The built-in type of lists can be redefined as follows:

```
infixr 5 :::

data List a = NilL | a :::  (List a)
  deriving (Eq,Ord,Show,Read)
```

- Here **fixity declaration** `infixr 5 :::` defines the fixity strength (5) from the range 0..9 and the kind of associativity (right) for the `:::` operator. The values are the same as it is defined for the pre-defined operator `:` .

Binary trees

- A polymorphic binary tree can be defined as follows:

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

- The functions `depth` and `occurs` are as before
- Example: a function collapsing a tree into a list by traversing the left side first

```
collapse ::  Tree a -> [a]
collapse Nil = []
collapse (Node x t1 t2) =
  collapse t1 ++ [x] ++ collapse t2
```

Binary trees (cont.)

- We can define higher-order functions for such datatypes too
- For instance, a version of the `map` function for trees can be defined as:

```
mapTree ::  (a-> b) -> Tree a -> Tree b
mapTree Nil = Nil
mapTree f (Node x t1 t2) =
  Node (f x) (mapTree f t1) (mapTree f t2)
```

The union type, `Either`

- Datatype definition can take more than one type parameter. In general, we can form a type whose elements come from either `a` or `b`:

```
data Either a b = Left a | Right b
  deriving (Eq,Ord,Show,Read)
```

- To define a function from `Either a b` to, for example, `Int`, we have to deal with two cases

```
fun ::  Either a b -> Int
fun (Left x) = ... x ...
fun (Right y) = ... y ...
```

- Example: a higher-order function on `Either a b`

```
either ::  (a->c) -> (b->c) -> Either a b -> c
either f g (Left x) = f x
either f g (Right y) = g y
```

Modelling program errors

How should a program to deal with a situation which should not occur?

- By generating an exception (the function `error :: String -> a` in Haskell)
- By returning a pre-defined (dummy) value:

```
divide :: Integer -> Integer -> Integer
divide n m
  | (m /= 0)  = n 'div' m
  | otherwise  = 0
```

  The problem: how to distinguish an error from the normal case generating the same value?

- Error datatypes, such as `Maybe`

Modelling program errors (cont.)

Why, instead of a dummy value, not to return an error value as a result?

- We define the datatype:

```
data Maybe a = Nothing | Just a
  deriving (Eq,Ord,Read,Show)
```

  It is essentially the type a with an extra value `Nothing` added

- Redefining `divide`:

```
divide ::  Integer -> Integer -> Maybe Integer
divide n m
  | (m /= 0)  = Just (n 'div' m)
  | otherwise  = Nothing
```

- Finding a value in a list (from the module Data.List):

```
find ::  (a -> Bool) -> [a] -> Maybe a
```

Modelling program errors (cont.)

- In general case, when a function `f` gives an error when some condition `cond` holds:

```
fErr x
  | cond  = Nothing
  | otherwise  = Just (f x)
```

  The results of such functions are now not of the original output type, say `a`, but of type `Maybe a`.

- We can now either transmit error further, trap it, or raise an error
- Example: transmitting the error through `mapMaybe`

```
mapMaybe ::  (a->b) -> Maybe a -> Maybe b

mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)
```

Datatypes and type classes

Building more complex type classes and their instances usually goes hand in hand with declaration of new datatypes

Example: movable geometrical objects

```
data Vector = Vec Float Float

class Movable a where
  move ::  Vector -> a -> a
  reflectX ::  a -> a
  reflectY ::  a -> a
  reflect180 ::  a -> a
  reflect180 = reflectX . reflectY
```

Datatypes and type classes (cont.)

```
data Point = Point Float Float
  deriving Show

instance Movable Point where
  move (Vec v1 v2) (Point c1 c2) =
    Point (c1+v1) (c2+v2)
  reflectX (Point c1 c2) = Point c1 (-c2)
  reflectY (Point c1 c2) = Point (-c1) c2
  rotate180 (Point c1 c2) = Point (-c1) (-c2)
```

Modules: importing and exporting revisited

- There are a number of module controls, declaring which module declarations (functions and types) are visible and can be used when imported
- It is important to know whilst building or using a hierarchy of modules
- Visibility restrictions can be given either in a module declaration (`module M (...)`) or during module import (`import M (...)`, `import M hiding (...)`)

Modules: simple example

```
module Ant where

data Ants = ...
antEater x = ...
```

```
module Bee where
import Ant

beeKeeper x = ...
```

All **visible** definitions from `Ant` can be used in `Bee`

Modules: export controls

We can control what is exported (visible) by following the module name with a list of what is to be exported, e.g.,

```
module Bee (beeKeeper, Ants(..), antEater)
where
...
```

In the case of datatypes, the notation `TypeName(..)` indicates that the datatype is exported together with all its constructors (i.e., its implementation structure is revealed)

If `(..)` is omitted, then the datatype acts as an **abstract data type** and can be accessed only via exported operations

Modules: the `Main` module

- Each system of modules should contain a top-level module `Main`, defining the name `main`
- In a compiled system, `main` is the expression that is evaluated first, when the compiled code is run
- In an interpreter like GHCi, it is of less significance
- A module without a header is treated as

```
module Main(main) where
...
```

Modules: import controls

Similarly, we can control what is imported (out of visible module declarations), e.g., choosing not to import antEater

```
import Ant (Ants(..))
...
```

or, alternatively, what names should be hidden, e.g.,

```
import Ant hiding (antEater)
...
```

Abstract datatypes in Haskell

- Abstract datatype: has a clearly defined and agreed **interface** (**signature** of ADT), allowing to separate the tasks of using and implementing the datatype
- As a result, we can modify modify the implementation without having any effect on the user

ADT example: search trees

- A binary search tree is an object of type `Tree a`

  ```
  data Tree a = Nil | Node a (Tree a) (Tree a)
  ```

  whose elements are **ordered**

- The tree `(Node val t1 t2)` is ordered if
  - all values in `t1` are smaller than `val`,
  - all values in `t2` are larger than `val`, and
  - the trees `t1` and `t2` are themselves ordered.

- The operations for building and manipulations such trees must preserve the order. We can ensure that only such approved operations are used by making the type into an abstract data type

ADT example: search trees (cont.)

The signature of the abstract data type for such trees

```
module STree
  (Tree,
   nil,          -- Tree a
   isNil,        -- Tree a -> Bool
   isNode,        -- Tree a -> Bool
   leftSub,      -- Tree a -> Maybe (Tree a)
   rightSub,     -- Tree a -> Maybe (Tree a)
   treeVal,      -- Tree a -> Maybe a
   insertVal,    -- Ord a => a -> Tree a -> Tree a
   deleteVal,    -- Ord a => a -> Tree a -> Tree a
   minTree       -- Ord a => Tree a -> Maybe a
  )
where
```

The constructors for `Tree` datatype are hidden!

ADT example: search trees (cont.)

```
data Tree a = Nil | Node a (Tree a) (Tree a)

nil ::  Tree a
nil = Nil

isNil ::  Tree a -> Bool
isNil Nil = True
isNil _ = False

isNode ::  Tree a -> Bool
isNode Nil = False
isNode _ = True

leftSub ::  Tree a -> Maybe (Tree a)
leftSub (Node _ t1 _) = Just t1
leftSub _ = Nothing
```

ADT example: search trees (cont.)

```
rightSub ::  Tree a -> Maybe (Tree a)
rightSub (Node _ _ t2) = Just t2
rightSub _ = Nothing

treeVal ::  Tree a -> Maybe a
treeVal (Node v _ _) = Just v
treeVal _ = Nothing

minTree ::  Ord a => Tree a -> Maybe a
minTree t
  | isNil t = Nothing
  | isNil t1 = v
  | otherwise = minTree t1
   where
    (Just t1) = leftSub t
    v = treeVal t
```

ADT example: search trees (cont.)

```
insertVal ::  Ord a => a -> Tree a -> Tree a
insertVal val Nil = (Node val Nil Nil)
insertVal val (Node v t1 t2)
  | v==val = Node v t1 t2
  | val < v = Node v (insertVal val t1) t2
  | val > v = Node v t1 (insertVal val t2)

deleteVal ::  Ord a => a -> Tree a -> Tree a
deleteVal val Nil = Nil
deleteVal val (Node v t1 t2)
  | val < v = Node v (deleteVal val t1) t2
  | val > v = Node v t1 (deleteVal val t2)
  | isNil t2 = t1
  | isNil t1 = t2
  | otherwise = joinTrees t1 t2
```

ADT example: search trees (cont.)

```
-- auxiliary function

joinTrees ::  Ord a => Tree a -> Tree a -> Tree a
joinTrees t1 t2 = Node mini t1 newt2
  where
   (Just mini) = minTree t2
   newt2 = deleteVal mini t2
```