CS 11 Haskell track: lecture 6

- This week:
  - Modules
  - Arrays
  - More Monads
    - MonadPlus
  - Wrapping up
Modules

- Haskell modules much more conservative than ocaml's module system
- Much of the work of e.g. functors done by type classes
- Consequently, modules are rather simple
module Tree (Tree (Leaf, Branch), fringe) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right
module Tree ( Tree(Leaf, Branch), fringe ) where

... 

- This means that this module explicitly exports
  - the Tree datatype
  - the fringe function
  - nothing else
- If written as:
  module Tree where ...
- then everything in module is exported
module Main where
import Tree ( Tree(Leaf, Branch), fringe )
main = print (fringe (Branch (Leaf 1) (Leaf 2)))

- If the second line was just
  import Tree
- then everything exported from Tree module would be imported
Avoiding name clashes (1)

- By default, imported names dumped into local namespace
- What if two modules are used which share names?
- Can explicitly qualify names during import
Avoiding name clashes (2)

module Main where
  import Tree ( Tree(Leaf, Branch), fringe )
  import qualified Fringe ( fringe )

- Module Fringe contains a function fringe which has same name as Tree module's fringe function
- Qualifying means refer to second fringe as Fringe.fringe
import qualified ... as ...

- Can rename the qualifier of a module by using the `as` syntax
  
  ```
  import qualified VeryLongModuleName as V
  ```

- Watch out for this:
  
  ```
  import Foobar as F
  ```

- Brings in all names from `Foobar` with and without qualification (why would you want this?)
hiding declarations

- Can selectively hide some names upon import with a `hiding` declaration:

- Assume module A exports x and y
  
  ```
  import A           -- x and y imported
  import A hiding y  -- x only
  import qualified A hiding y -- A.x only
  ```
Modules and instances

- Instance declarations not explicitly imported/exported
  - modules export all instance declarations
Arrays

- Haskell arrays are functional
  - no in-place update in standard Arrays
  - though some mutable array types in ghc libraries (not covered here)
- Arrays require an `Ix` (indexing) type to represent indices (usually just `Int`
Array indices

class (Ord a) => Ix a where
  range :: (a, a) -> [a]
  index :: (a, a) -> a -> Int
  inRange :: (a, a) -> a -> Bool

range (0,4) => [0,1,2,3,4]
range ((0,0), (1,2)) => [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)]
**Array indices**

```haskell
class (Ord a) => Ix a where
  range :: (a, a) -> [a]
  index :: (a, a) -> a -> Int
  inRange :: (a, a) -> a -> Bool

index (1,9) 2 => 1
index ((0,0), (1,2)) (1,1) => 4
```
Creating arrays

array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b

squares = array (1,100) [(i, i*i) | i <- [1..100]]
Accessing array elements

squares ! 8 => 64
bounds squares => (1,100)
Example

fibs :: Int -> Array Int Int
fibs n = a
  where a =
    array (0, n)
      [(0, 1), (1, 1)] ++
      [[(i, a!(i-2) + a!(i-1)) | i <- [2..n]]]
"Modifying" array elements

(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b

squares_bad = squares // [(8, 63)]
squares_bad ! 8 => 63

- Creates a new array, not modifying in place
- Other ways to actually modify in place
  - but need to be in e.g. IO monad
MonadPlus

- Many Monads have a notion of
  - a "zero" element
  - some kind of "addition" of monadic objects
- This is captured in the MonadPlus class

class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
MonadPlus instances

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` ys = ys
  xs `mplus` ys = xs

instance MonadPlus [] where
  mzero = []
  mplus = (+++)
Where to now? (1)

- Lots of information on the web
  - www.haskell.org
  - www.haskell.org/ghc

- Haskell mailing lists:
  - www.haskell.org/haskellwiki/Mailing_Lists
  - haskell mailing list
  - haskell-cafe mailing list
Where to now? (2)

- Lots of interesting paper collections
- I particularly recommend Phil Wadler's papers:
  - http://homepages.inf.ed.ac.uk/wadler/
- Good examples:
  - "Imperative Functional Programming"
  - "Monads for Functional Programming"
  - "Comprehending monads"