Code (Tuesday Week 6)

Dice Example

```
d6 :: [Int]
d6 = [1,2,3,4,5,6]

twoD6 :: [(Int, Int)]
twoD6 = (,) <$> d6 <*> d6


game :: [(Int,Int)]
game = twoD6 >>= \(d1,d2) ->
          if abs (d1 - d2) < 2 then
             d6 >>= \d2' -> pure (d1, d2')
          else
             pure (d1,d2)

game' :: [(Int,Int)]
game' = do
  (d1,d2) <- twoD6
  if abs (d1 - d2) < 2 then do
     d2' <- d6
     pure (d1,d2')
  else do
     pure (d1,d2)


score :: (Int, Int) -> Int
score (d1,d2) = abs (d1 - d2)

scores :: [Int]
scores = fmap score game
```

Student Database Example

```
students = [(3253158, "Liam")
           ,(4444444, "Mort Deathington")
           ,(8888888, "Rich Moneybags")
           ]

-- lookup :: [(a,b)] -> a -> Maybe b
```

```haskell
findNames :: [Int] -> Maybe [String]
findNames [] = Just []
findNames (z:zs) = lookup z students
               >>= \n ->
                 findNames zs
                   >>= \ns ->
                     pure (n : ns)

findNames' :: [Int] -> Maybe [String]
findNames' [] = Just []
findNames' (z:zs) = do
   n  <- lookup z students
   ns <- findNames zs
   pure (n : ns)
```

Arbitrary Search Trees

```haskell
import Test.QuickCheck
data Tree = Leaf
          | Branch Int Tree Tree
          deriving (Show,Eq)

instance Arbitrary Tree where
  -- arbitrary :: Gen Tree
  arbitrary = arbitrary >>= \min
               -> arbitrary >>= \(Positive max')
                 -> searchTrees min (min + max')
   where
     searchTrees :: Int -> Int -> Gen Tree
     searchTrees min max
           | min < max   = oneof [ leafGen
                                 , branchGen
                                 ]
           | otherwise   = leafGen
       where
         leafGen :: Gen Tree
         leafGen = pure Leaf

         branchGen :: Gen Tree
         branchGen = choose (min,max)
                       >>= \n ->
                         Branch n <$> searchTrees min (n -1)
                                  <*> searchTrees n max
{- with do notation:
instance Arbitrary Tree where
  -- arbitrary :: Gen Tree
```

```
  arbitrary = do
    min <- arbitrary
    Positive max' <- arbitrary
    searchTrees min (min + max')
   where
     searchTrees :: Int -> Int -> Gen Tree
     searchTrees min max
           | min < max    = oneof [ leafGen
                                  , branchGen
                                  ]
           | otherwise    = leafGen
       where
         leafGen :: Gen Tree
         leafGen = pure Leaf

         branchGen :: Gen Tree
         branchGen = do
           n <- choose (min,max)
           Branch n <$> searchTrees min (n -1)
                    <*> searchTrees n max
-}
```

Basic Instances

Most of this code duplicates the standard library and Prelude, so won't compile.

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f (Just x) = Just (f x)
maybeMap f Nothing  = Nothing

instance Functor [ ] where
  fmap = map

instance Functor Maybe where
  fmap = maybeMap


instance Functor ((->) x) where

  -- fmap :: (a -> b) -> f a -> f b
  -- so for this type, f is (x ->) so:
  -- fmap :: (a -> b) -> (x -> a) -> (x -> b)
  fmap = (.)

instance Functor ((,) x) where
  -- fmap :: (a -> b) -> (x,a) -> (x,b)
  fmap f (x,a) = (x, f a)
```

```haskell
-- remember (3,2) == (,) 3 2


-- we can write functions:
-- toString :: Int -> String
-- as
-- toString :: (->) Int String


instance Applicative Maybe where
  -- pure :: a -> Maybe a
  pure a = Just a

  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  Nothing <*> arg = Nothing
  Just f  <*> Nothing = Nothing
  Just f  <*> Just a  = Just (f a)




{- Proof of Functor Laws for all applicatives
   where
    fmap f x = pure f <*> x

1. fmap id x == id x

2.  fmap f (fmap g x) == fmap (f . g) x

-- Proof of 1)
pure id <*> x == x -- Identity

-- Proof of 2)
pure f <*> (pure g <*> x)
== -- composition (backwards)
pure (.) <*> pure f <*> pure g <*> x
== -- homomorphism
pure ((.) f) <*> pure g <*> x
== -- homomorphism
pure (f . g) <*> x

-}

-- This instance is what Haskell actually uses for lists.
instance Applicative [ ] where
  []     <*> as = []
  (f:fs) <*> as = map f as ++ (fs <*> as)
```

```
  pure a = [a]

-- This instance is put behind ZipList in the standard library
instance Applicative [ ] where
  []     <*> as = []
  fs     <*> [] = []
  (f:fs) <*> (a:as) = f a : (fs <*> as)

  pure a = a : pure a

instance Applicative ((->) x) where

  pure :: a -> (x -> a)
  pure a x = a

  -- (<*>) :: f (a -> b) -> f a -> f b
  -- (<*>) :: (x -> a -> b) -> (x -> a) -> (x -> b)
  (f <*> a) = \x -> f x (a x)





 instance Monad Maybe where
   (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
   Just a >>= f = f a
   Nothing >>= f = Nothing

 instance Monad [ ] where
   (>>=) :: [a] -> (a -> [b]) -> [b]
   (>>=) as f = concatMap f as

 instance Monad ((->) x) where
   (>>=) :: (x -> a) -> (a -> x -> b) -> (x -> b)
   (>>=) xa axb = \x -> axb (xa x) x


{-
We can make an applicative operator given a monad, by writing:

(<*>) :: m (a -> b) -> m a -> m b
mf <*> mx = mf >>= \f ->
              mx >>= \x ->
                pure (f x)

-}
```