Notes from 11/2

```
import Control.Monad.State
```

TBD: this needs work

Monads

A type is a monad `m` if it provides the following two operations:

```
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b   -- aka "bind"

returnSt a s = (a,s)
bindSt f g s0 = (b, s2)
  where (a, s1) = f s0
        (b, s2) = g a s1

Nothing >>= \_ -> Nothing

Just 14 >>= \_ -> Nothing

Just 14 >>= \a -> Just(a*2)

[4,5,6] >>= \a -> [a*2]

[4,5,6] >>= \a -> [a*2, a+5, 9]

xs >>= \a -> xs >>= \b -> xs >>= \c -> if c*c == a*a + b*b then [(a,b,c)] else []

"ABCD" >>= \c -> [succ c, pred c]

twiceM =
  do a <- [4,5,6]
     return (a*2)

pythag =
  do a <- [1..100]
     b <- [1..a]
     c <- [1..100]
     if c*c == a*a + b*b
     then return (a,b,c)
     else []
```

State monad

When we worked with threading state through a calculation or traversal, we defined a state function with a type like `s -> (r, s)` where `s` is the state and `r` is the result. It turns out that this function type is *itself* a monad.

Substitute `m a` in the monad operations with `s -> (a, s)` and you get:

```
return :: a -> s -> (a, s)
(>>=) :: (s -> (a, s)) -> (a -> s -> (b, s)) -> s -> (b, s)
```

We can actually write these functions, but we'll name them a little differently so they don't conflict with the real ones:

```
returnState :: a -> s -> (a,s)
returnState result state = (result, state)

bindState :: (s -> (a, s)) -> (a -> s -> (b, s)) -> s -> (b, s)
bindState action1 action2 state0 = (result2, state2)
  where (result1, state1) = action1 state0
        (result2, state2) = action2 result1 state1
```

Notice the usual careful state-passing in the where clause of the `bindState` function. That little bit of state-passing logic turns is sufficient to encode everything we need to build more elaborate stateful functions.

For example, the function `threeRandoms` becomes:

```
threeRandoms :: Seed -> ([Integer], Seed)
threeRandoms =
  rand `bindSt` \r1 ->
  rand `bindSt` \r2 ->
  rand `bindSt` \r3 ->
  returnState [r1, r2, r3]
```

This has the same type as the `threeRandoms` we did list week, but all the state-passing is hidden. It's done transparently by `bindState` and `returnState`, and we get exactly the same results.

```
ghci> threeRandoms (Seed 3453)
([58034571,429459059,225867046],Seed {unSeed = 225867046})

data Seed = Seed { unSeed :: Integer }
  deriving (Eq, Show)
```

```
threeRandsSt :: State Seed [Integer]
threeRandsSt =
  do r1 <- randSt
     r2 <- randSt
     r3 <- randSt
     return [r1,r2,r3]

randSt :: State Seed Integer
randSt = do
  Seed s <- get
  let s' = (s * 16807) `mod` 0x7FFFFFFF
  put (Seed s')
  return s'

rand :: Seed -> (Integer, Seed)
rand (Seed s) = (s', Seed s')
  where
    s' = (s * 16807) `mod` 0x7FFFFFFF

main = putStrLn "OK"
```

"do" notation

Reader monad

Substitute `r -> a` for `m a`.

```
returnRd :: a -> r -> a
returnRd a r = a

bindRd :: (r -> a) -> (a -> r -> b) -> r -> b
bindRd f g r = g (f r) r

ask :: r -> r
ask r = r
```

`r` Represents an *environment* – a value that is always available to you, but you don't have to explicitly pass around.

Reader example: do some calculations on a list. Then take modulo N, where N is from the environment.

```
calc :: Integer -> Integer -> Integer
calc x = blop x `bindRd` \y -> returnRd (sum y)
```

```
blop :: Integer -> Integer -> [Integer]
blop x =
   cran x `bindRd` \y ->
   ask `bindRd` \n ->
   returnRd $ map (`mod` n) y

cran :: Integer -> Integer -> [Integer]
cran x = returnRd $ map (2^) [1..x]
```

Writer monad

Substitute `(a,w)` for `m a`.

```
returnW :: Monoid w => a -> (a,w)
returnW a = (a, mempty)

bindW :: Monoid w => (a,w) -> (a -> (b,w)) -> (b,w)
bindW (a,w1) f = (b, mappend w1 w2)
  where (b,w2) = f a

tell :: Monoid w => w -> ((),w)
tell w = ((), w)

addEvens xs = sum $ filter even xs
```

I'd like this function to log the odd numbers that it skips when filtering.

```
filterW :: (a -> Bool) -> [a] -> ([a],[a])
filterW f [] = returnW []
filterW f (x:xs) =
  filterW f xs `bindW` \ys ->
  if f x
  then returnW (x:ys)
  else tell [x] `bindW` \() ->
       returnW ys

addEvensW xs =
  filterW even xs `bindW` \ys -> returnW (sum ys)
```