Lecture 11.

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

• Nondeterministic computation.

N.B. [ ] a = [a]

```
instance Monad [] where
    -- return :: a -> [a]
    return x = [x]
    -- (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)
```

$\underbrace{x}_{a}$, $\underbrace{[x]}_{[a]}$

$\underbrace{xs}_{[a]}$, $\underbrace{f}_{a \to [b]}$, $\underbrace{\text{concat}(\underbrace{\text{map } f \text{ } xs}_{[[b]]})}_{[b]}$

map :: (a -> b) -> [a] -> [b]

concat :: [[a]] -> [a]

List comprehension style:

```
triples :: [Triple]          type Triple = (Int, Int, Int)
triples = [ (a, b-a, n-b) | n <- [0..]
                            a <- [0..n]
                            b <- [a..n] ]
```

Monad style:

```
triples = do  n <- [0..]
              a <- [0..n]
              b <- [a..n]
              return (a, b-a, n-b)
```

State Monad

$x = x + 1$

```
newtype State s a = State (s -> (a, s))
```

↑ like "data", exept ⊥ are shared

$s \to (a, s)$

```
[0..] >>= \n -> [0..n] >>= \a -> [a..n] >>= b ->
              return (a, b-a, n-b)
```

↑ desugared version of the monad style

s stores the state / variables

instance Monad (State s) where

return :: $a \to$ State s a

return $\underbrace{x}_{a}$ = State $(\underbrace{\lambda s \to (x, s)}_{s \to (a, s)})$

$f :: s \to (a, s)$

$f\ s = (x, s)$

alternatively, we could write

$f = \lambda s \to (x, s)$

written '\' in ASCII

runState :: State s a $\to (s \to (a, s))$

runState (State f) = f

alternatively, we can write:

newtype State s a = State {runState :: $(s \to (a, s))$}

$(\gg\!=)$ :: State s a $\to$ (a $\to$ State s b) $\to$ State s b

$\underbrace{mx}_{\text{State s a}} \gg\!= \underbrace{f}_{a \to \text{State s b}} = \underbrace{\text{State}\ \_\_\_}_{\text{State s b}}$

$$\underbrace{mx}_{\text{State } s\ a} \gg= \underbrace{f}_{a \to \text{State } s\ b} = \text{State}\ \underbrace{\overbrace{\left(\lambda s \to \overbrace{\qquad}^{(b,s)}\right)}^{s \to (b,s)}}_{\text{State } s\ b}$$

$$\text{let } (\underbrace{x}_{a}, \underbrace{s'}_{s}) = \overbrace{\underbrace{(\text{runState } mx)}_{s \to (a,s)}\ s}^{(a,s)} \text{ in}$$

$$\underbrace{\underbrace{\overbrace{\text{runState } (\underbrace{f\ x}_{\text{State } s\ b})\ s'}^{(b,s)}}_{}}_{\substack{s \to (b,s) \\ (b,s)}}$$

So, putting the pieces together,

$$mx \gg= f = \text{State } (\lambda s \to \text{let } (x, s') = \text{runState } mx\ s \text{ in}$$
$$\text{runState } (f\ x)\ s'\ )$$

Morally, `int x ;`
`x = x + 1;` in C

is like doing

```
incr :: State Int ()
incr = do  x <- get
           put (x+1)
```

in Haskell

```
get :: State s s
get = State (λs -> (s, s))

put :: s -> State s ()
put s' = State (λs -> ((), s'))
```