[illegible]

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

[illegible]

1. If we map the id function over a functor, the functor that we get back should be the same as the original functor.

```haskell
fmap id = id
```

2. Composing two functions and then mapping the resulting function over a functor should be the same as first mapping one function over the functor and then mapping the other one.

```haskell
fmap (f . g) = fmap f . fmap g
fmap (f . g) F = fmap f (fmap g F) -- where F is any functor
```

[illegible]

[illegible]

[illegible]

```haskell
instance Functor [] where
    fmap = map
```

[illegible]

```
> fmap (*2) [1..3]
[2,4,6]
```

[illegible]

[illegible]

```haskell
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

[illegible]

```
> fmap (*2) (Just 200)
Just 400
> fmap (*2) Nothing
Nothing
```

[illegible]

[illegible]

```haskell
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

[illegible]

```
> fmap (*10) $ Left "you can ignore me"
Left "you can ignore me"
> fmap (*10) $ Right 10
Right 100
```

[illegible]

<$>

[illegible]

```haskell
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

[illegible]

```
> (\x -> x + 3) <$> Just 5
Just 8
> fmap (\x -> x + 3) $ Just 5
Just 8
> fmap (\x -> x + 3) [1, 2, 3]
[4,5,6]
> (\x -> x + 3) <$> [1, 2, 3]
[4,5,6]
> (*10) <$> Right 10
Right 100
```

[illegible]

[illegible]

```haskell
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

[illegible]

1. `pure f <*> x = fmap f x`
2. `pure id <*> v = v`
3. `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
4. `pure f <*> pure x = pure (f x)`
5. `u <*> pure y = pure ($ y) <*> u`

[illegible]

[illegible]

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

[illegible]

```
> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

[illegible]

[illegible]

```
instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

[illegible]

```
> import Control.Applicative (ZipList(ZipList,getZipList))
> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
[101,102,103]
> getZipList $ (+) <$> ZipList [1,2,3,4] <*> ZipList [100,100,100]
[101,102,103]
```

[illegible]

[illegible]

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

[illegible]

```
> Just (+1) <*> Just 4
Just 5
> Just (+1) <*> Nothing
Nothing
```

`((->) r)`

`(->) r a` could be rewritten as `r -> a` .

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

. ...

```
> f = fmap (\x1 x2 -> "3x is " ++ show x1 ++ " and 4x is " ++ show x2) (3*) <*> (4*)
> :t f
f :: (Show a, Num a) => a -> [Char]
> f 10
"3x is 30 and 4x is 40"
> f 100
"3x is 300 and 4x is 400"
> pure 3 "blah"
3
```

.

. .. .

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

. ...

```
> concatInputLines = (++) <$> getLine <*> getLine
> concatInputLines
First
Second
"FirstSecond"
```

. - ....

liftA2

. .. .

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

. ...

```
> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
> liftA2 (\x y -> show x ++ " is spelled " ++ y) (Just 3) (Just "three")
Just "3 is spelled three"
```

- - -----

. .. .

A `Monoid` instance must be a concrete type, such as `Maybe Int`.

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

. ....

1. mempty `mappend` x = x
2. x `mappend` mempty = x
3. (x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)

[illegible]

[illegible]

```haskell
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

[illegible]

```haskell
> mappend "one " "two"
"one two"
```

[illegible]

[illegible]

```haskell
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

[illegible]

```
> import Control.Monad.RWS (Sum(Sum))
> import Control.Monad.RWS (Product(Product))
> Product 3 `mappend` Product 9
Product {getProduct = 27}
> mconcat . map Sum $ [1,2,3]
Sum {getSum = 6}
```

?▯▯▯▯▯?▯?

. .. .

```
instance Monoid Any where
    mempty = Any False
    Any x `mappend` Any y = Any (x || y)

instance Monoid All where
    mempty = All True
    All x `mappend` All y = All (x && y)
```

. ...

```
> import Control.Monad.RWS (All(All), Any(Any))
> mconcat . map Any $ [True, True, True]
Any {getAny = True}
> mconcat . map Any $ [True, True, False]
Any {getAny = True}
> mconcat . map Any $ [False, False, False]
Any {getAny = False}
> mconcat . map All $ [True, True, True]
All {getAll = True}
> mconcat . map All $ [True, True, False]
All {getAll = False}
```

. .. .

. .. .

```
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` y = y
    GT `mappend` _ = GT
```

. ...

```
> import Data.Monoid
> lengthCompare x y = (length x `compare` length y) `mappend`  (x `compare` y)
> lengthCompare "zen" "ants"
LT
> lengthCompare "zen" "ant"
GT
```

- - - - - -

. .. - - - - - - - -

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

. . . . .

1. `return x >>= f` is the same damn thing as `f x`.
2. `m >>= return` is no different than just `m`.
3. `(m >>= f) >>= g` is just like doing `m >>= (\x -> f x >>= g)`.

.. . .

. . . . . .

. .. .

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

. ...

```
> return "WHAT" :: Maybe String
Just "WHAT"
> Just 9 >>= \x -> return (x*10)
Just 90
> Nothing >>= \x -> return (x*10)
Nothing
```

. ..

. .. .

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

. ...

```
> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

. - ....

<==<

. .. .

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

. ...

```
> let f x = [x,-x]
> let g x = [x*3,x*2]
> let h = f <=< g
> h 3
[9,-9,6,-6]
```

do ....

In a `do` expression, every line is a monadic value. To inspect its result, we use `<-`.

. ...

```
foo :: Maybe String
foo = Just 3   >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))

-- is equivalent to

foo' :: Maybe String
foo' = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

- - - - - - -

. .. .

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }

instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f = let (Writer (y, v')) = f x in Writer (y, v `mappend` v')
```

[illegible]

```haskell
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Got number: " ++ show x])

multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    return (a*b)
```

In console:

```
> import Control.Monad.Writer
> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

[illegible]

[illegible]

```haskell
newtype State s a = State { runState :: s -> (a,s) }

instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                        (State g) = f a
                                    in  g newState
```

```
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.State

type Stack = [Int]

pop :: State Stack Int
pop = state $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = state $ \xs -> ((),a:xs)

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

In console:

```
> runState stackManip [5,8,2,1]
(5,[8,2,1])
```