Where are we at?

- Haskell **basics**:
  - Functions, lists, pattern matching, recursion, cases, `let`/`where`
- Language features for **productive Haskell**:
  - Datatypes, typeclasses, monads, IO, modules

- Writing **parallel** programs in Haskell:
  - Laziness, `seq`/`par`, ThreadScope, monads for parallelism
- The final project!

Where are we at?

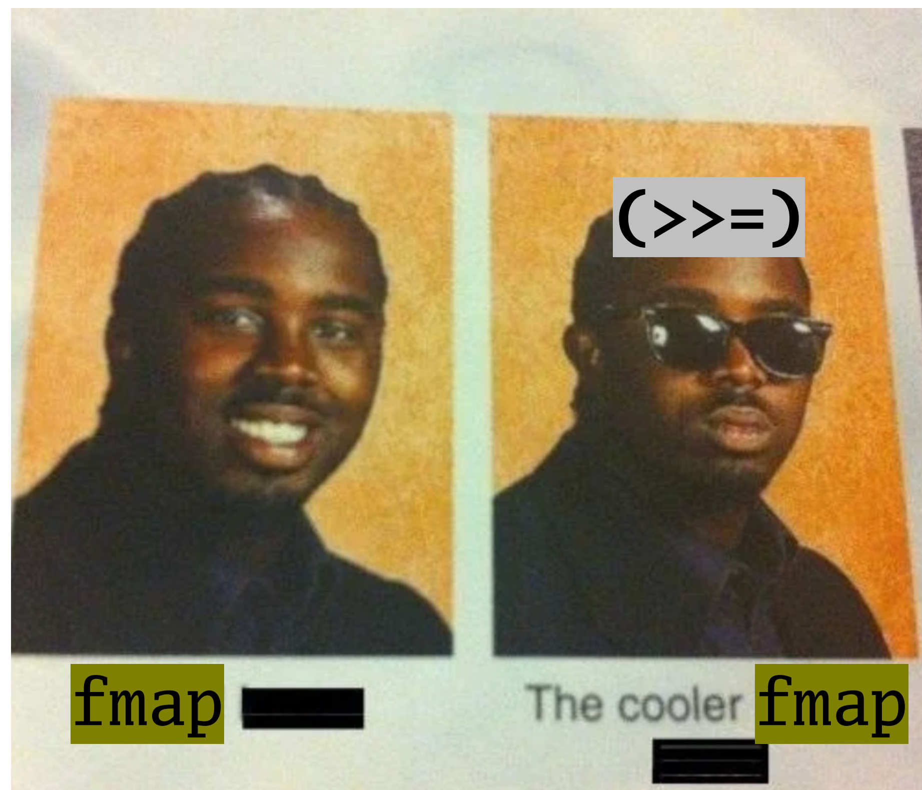
- Haskell **basics**:
  - Functions, lists, pattern matching, recursion, cases, `let`/`where`
- Language features for **productive Haskell**:
  - Datatypes, typeclasses, **monads**, IO, modules

- Writing **parallel** programs in Haskell:
  - Laziness, `seq`/`par`, ThreadScope, monads for parallelism
- The final project!

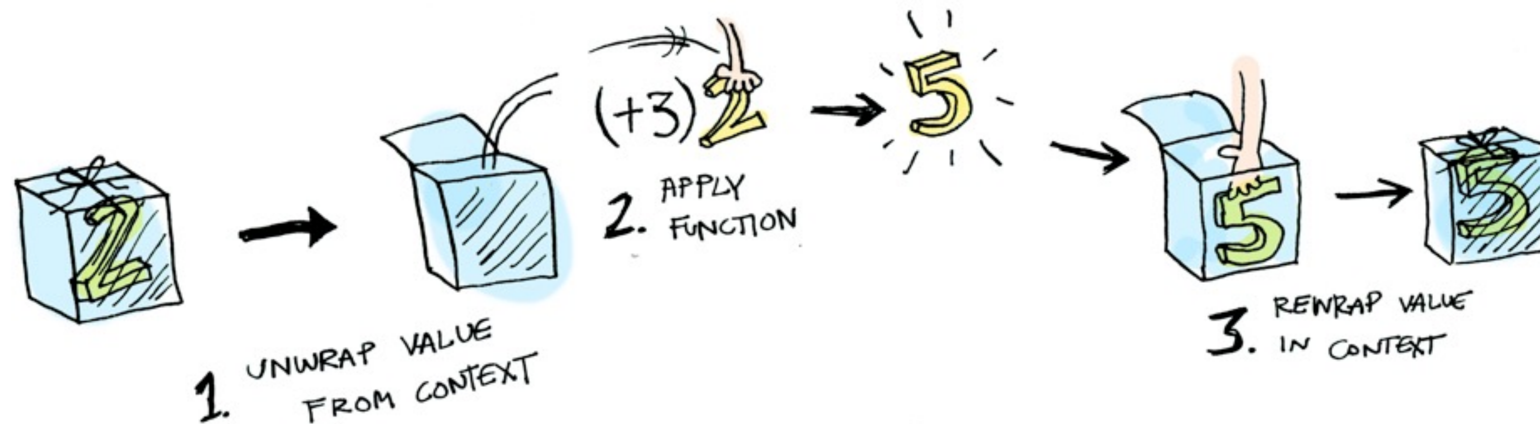
Monads: A review

- `Monad` is a typeclass like `Eq` or `Functor`
  - Must be a `Functor` (and `Applicative`)
  - Must define the `>>=` (bind) operator

What is (>>=)?



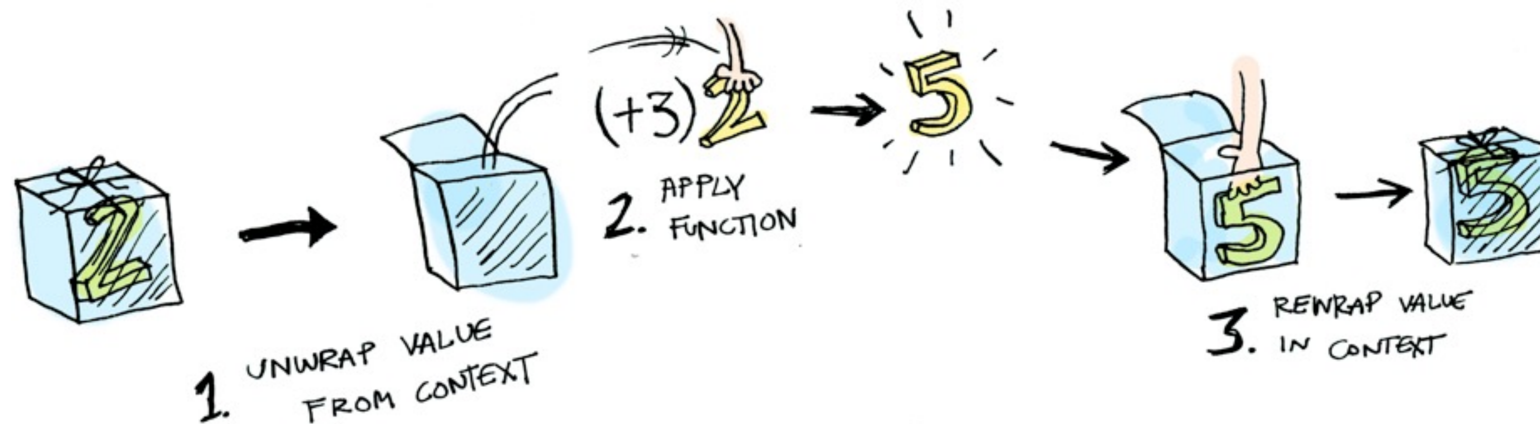
What is (>>=)?




```
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
fmap (\x -> x + 2) (Just 3) = Just 5
fmap (\x -> x + 2) Nothing  = Nothing
```

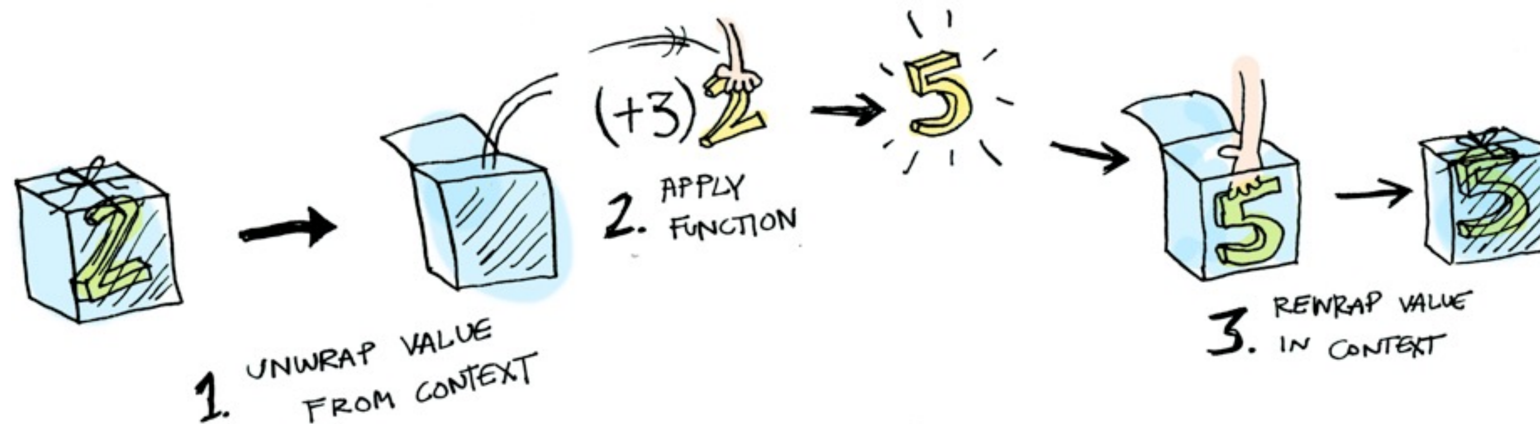
What is (>>=)?




```
fmap :: Functor f => (a -> b) -> f a -> f b
```

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

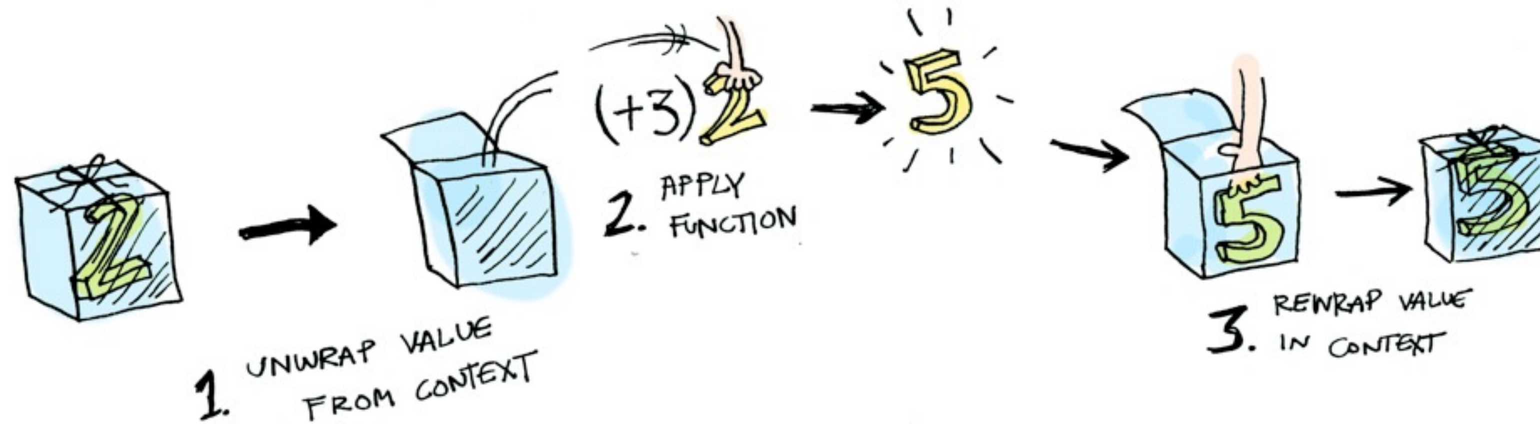
What is (>>=)?




```
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
(>>=) :: Monad f => f a -> (a -> f b) -> f b
```

What is (>>=)?



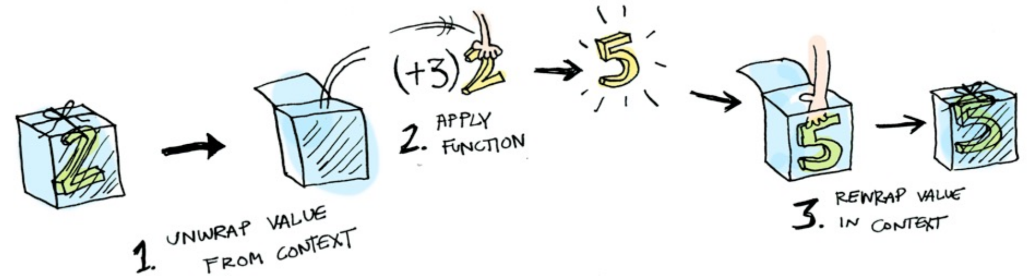

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

What is (>>=)?

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value
3. Rewrap value in context



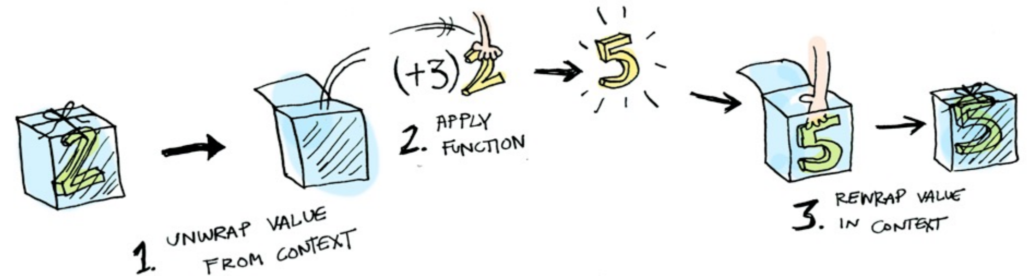

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. Rewrap value in context?

What is (>>=)?

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value
3. Rewrap value in context



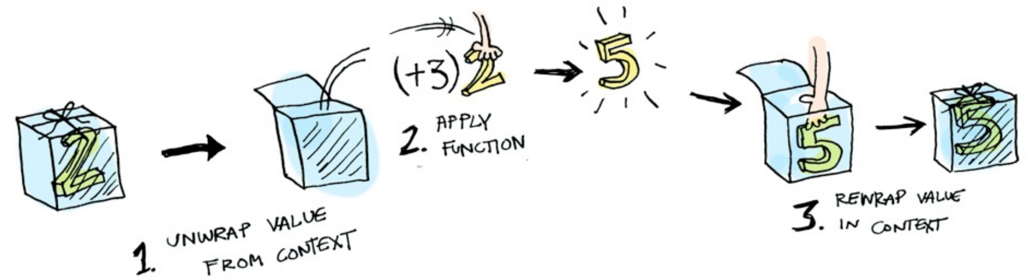

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. ~~Rewrap value in context~~

What is (>>=)?

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value
3. Rewrap value in context




```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. **Merge** contexts
4. Rewrap value in context

What is (>>=)?

```
fmap          (\x -> x + 2)          (Just 3) = Just 5
fmap          (\x -> x + 2)          Nothing  = Nothing
```

What is (>>=)?

```
fmap          (\x -> x + 2)            (Just 3) = Just 5
fmap          (\x -> x + 2)            Nothing  = Nothing

(flip (>>=)) (\x -> Just (x + 2)) (Just 3) = Just 5
(flip (>>=)) (\x -> Nothing)       (Just 3) = Nothing
(flip (>>=)) (\x -> Just (x + 2)) Nothing  = Nothing
```

What is (>>=)?

```
fmap          (\x -> x + 2)           (Just 3) = Just 5
fmap          (\x -> x + 2)           Nothing  = Nothing

(flip (>>=)) (\x -> Just (x + 2)) (Just 3) = Just 5
(flip (>>=)) (\x -> Nothing)         (Just 3) = Nothing
(flip (>>=)) (\x -> Just (x + 2)) Nothing  = Nothing

Just 3 >>= \x -> Just (x + 2)                 = Just 5
```

What is (>>=)?

```
fmap          (\x -> x + 2)          (Just 3) = Just 5
fmap          (\x -> x + 2)          Nothing  = Nothing

(flip (>>=)) (\x -> Just (x + 2)) (Just 3) = Just 5
(flip (>>=)) (\x -> Nothing)       (Just 3) = Nothing
(flip (>>=)) (\x -> Just (x + 2)) Nothing  = Nothing

Just 3 >>= \x -> Just (x + 2)                  = Just 5
Just 3 >>= \x -> Just (x + 2) >>= \y -> Just (y + 3)
                                               = Just 8
```

Monad introduction from Stephen

https://www.youtube.com/watch?v=_Gk_lwhJMzk

Where else do we see this pattern?

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. **Merge** contexts
4. Rewrap value in context

```
newtype Logged a = Logged (a, [String])
evalA :: Aexpr -> Logged Int
evalA (Negate e) = …
```

Where else do we see this pattern?

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

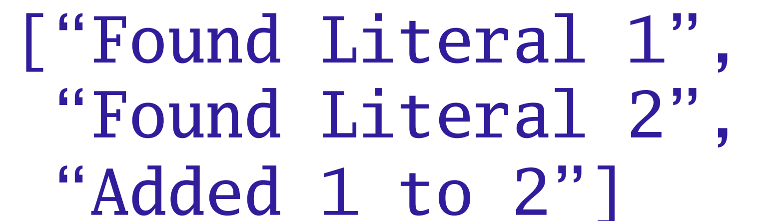
1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. **Merge** contexts
4. Rewrap value in context

```
newtype Logged a = Logged (a, [String])
evalA :: Aexpr -> Logged Int
evalA (Negate e) = …
```

Where else do we see this pattern?

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. **Merge** contexts
4. Rewrap value in context




```
newtype Logged a = Logged (a, [String])
evalA :: Aexpr -> Logged Int
evalA (Negate e) = …
```

Where else do we see this pattern?

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. **Merge** contexts
4. Rewrap value in context

```
negate i
```

```
["Negated 3"]
```

```
["Found Literal 1",
 "Found Literal 2",
 "Added 1 to 2"]
```

```
newtype Logged a = Logged (a, [String])
evalA :: Aexpr -> Logged Int
evalA (Negate e) = …
```

Where else do we see this pattern?

```
flip (>>=) :: Monad f => (a -> f b) -> f a -> f b
```

1. Unwrap value from context
2. Apply function, produces new value **with** more context
3. **Merge** contexts
4. Rewrap value in context

```
(++)
```

```
negate i
```

```
["Negated 3"]
```

```
["Found Literal 1",
 "Found Literal 2",
 "Added 1 to 2"]
```

```
newtype Logged a = Logged (a, [String])
evalA :: Aexpr -> Logged Int
evalA (Negate e) = …
```