Monads

October 28, 2019

Monads

October 28, 2019

Source

Monads

October 28, 2019

Source

Burritos

- Monads are like burritos

Burritos

- Monads are like burritos
- Monads are not like burritos

Sequencing Actions

1. Get a line
2. Get a line
3. “Return” the lines concatenated together

Sequencing Actions

1. Get a line
2. Get a line
3. “Return” the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. “Return” the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b

           = (++) <$> getLine <*> getLine
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. Print the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      return $ a ++ b

           = (++) <$> getLine <*> getLine
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. Print the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

           = (++) <$> getLine <*> getLine
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. Print the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b
  ```
- How to write this in applicative style?

Sequencing Actions

1. Get a line
2. Get a line
3. Print the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

          = (++) <$> getLine <*> getLine
  ```
    - Actions

Sequencing Actions

1. Get a line
2. Get a line
3. Print the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

          = (++) <$> getLine <*> getLine
  ```
    - What to do with the results

Sequencing Actions

1. Get a line
2. Get a line
3. Print the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

  myAction' = (\x y -> print $ x ++ y)
                   <$> getLine <*> getLine
  ```

Sequencing Actions

1. Get a line
2. Get a line
3. Print the lines concatenated together

-
  ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

  myAction' = (\x y -> print $ x ++ y)
                   <$> getLine <*> getLine
  ```
  - Why doesn't this work?

Sequencing Actions

- `(\x y -> print $ x ++ y) <$> getLine <*> getLine`
  - Get a line a, apply `(\x y -> print $ x ++ y)` to a (to get `(\y -> print $ a ++ y)`), and wrap it up in an `IO` box

Sequencing Actions

- `(\x y -> print $ x ++ y) <$> getLine <*> getLine`
    - Get a line a, apply `(\x y -> print $ x ++ y)` to a (to get `(\y -> print $ a ++ y)`), and wrap it up in an `IO` box
    - Take `(\y -> print $ a ++ y)` out of the box, get another line b, apply `(\y -> print $ a ++ y)` to b (to get `print $ a ++ b`), and wrap it up in another `IO` box

Sequencing Actions

- `(\x y -> print $ x ++ y) <$> getLine <*> getLine`
  - Get a line `a`, apply `(\x y -> print $ x ++ y)` to `a` (to get `(\y -> print $ a ++ y)`), and wrap it up in an `IO` box
  - Take `(\y -> print $ a ++ y)` out of the box, get another line `b`, apply `(\y -> print $ a ++ y)` to `b` (to get `print $ a ++ b`), and wrap it up in another `IO` box
  - We never actually ran `print $ a ++ b`!

Sequencing Actions

- `(\x y -> print $ x ++ y) <$> getLine <*> getLine`
  - Get a line `a`, apply `(\x y -> print $ x ++ y)` to `a` (to get `(\y -> print $ a ++ y)`), and wrap it up in an `IO` box
  - Take `(\y -> print $ a ++ y)` out of the box, get another line `b`, apply `(\y -> print $ a ++ y)` to `b` (to get `print $ a ++ b`), and wrap it up in another `IO` box
  - We never actually ran `print $ a ++ b`!
- `myAction :: IO ()`

Sequencing Actions

- `(\x y -> print $ x ++ y) <$> getLine <*> getLine`
  - Get a line a, apply `(\x y -> print $ x ++ y)` to a (to get `(\y -> print $ a ++ y)`), and wrap it up in an `IO` box
  - Take `(\y -> print $ a ++ y)` out of the box, get another line b, apply `(\y -> print $ a ++ y)` to b (to get `print $ a ++ b`), and wrap it up in another `IO` box
  - We never actually ran `print $ a ++ b`!
- `myAction :: IO ()`
- `myAction' :: IO (IO ())`

Sequencing Actions

- `(\x y -> print $ x ++ y) <$> getLine <*> getLine`
  - Get a line a, apply `(\x y -> print $ x ++ y)` to a (to get `(\y -> print $ a ++ y)`), and wrap it up in an `IO` box
  - Take `(\y -> print $ a ++ y)` out of the box, get another line b, apply `(\y -> print $ a ++ y)` to b (to get `print $ a ++ b`), and wrap it up in another `IO` box
  - We never actually ran `print $ a ++ b`!
- `myAction :: IO ()`
- `myAction' :: IO (IO ())`
  - To run `print $ a ++ b`, we need to take it out of the box

Monads

- Wikipedia: Throughout this article C denotes a category. A monad on C consists of an endofunctor $T\colon C \to C$ together with two natural transformations: $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).

Monads

- Wikipedia: Throughout this article C denotes a category.
  A monad on C consists of an endofunctor
  $T\colon C \to C$ together with two natural transformations:
  $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and
  $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).
    - Remember categories:

Monads

- Wikipedia: Throughout this article C denotes a category.
  A monad on C consists of an endofunctor
  $T\colon C \to C$ together with two natural transformations:
  $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and
  $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).
  - Remember categories:
    - category = objects + morphisms
      - objects = types
      - morphisms = functions

Monads

- Wikipedia: Throughout this article C denotes a category. A monad on C consists of an endofunctor $T\colon C \to C$ together with two natural transformations: $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).
  - endofunctor = functor that maps a category to that same category

Monads

- Wikipedia: Throughout this article C denotes a category. A monad on C consists of an endofunctor $T\colon C \to C$ together with two natural transformations: $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).
  - endofunctor = functor that maps a category to that same category
    - Our only category is `Hask`, so all functors are endofunctors

Monads

- Wikipedia: Throughout this article C denotes a category.
  A monad on C consists of an endofunctor
  `T` together with two natural transformations:
  $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and
  $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).
  - natural transformation $=$ morphism of functors

Monads

- Wikipedia: Throughout this article C denotes a category.
  A monad on C consists of an endofunctor
  `T` together with two natural transformations:
  $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and
  $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).
  - natural transformation = morphism of functors
    - Let us call η `unit` (or `return`), and μ `join`

Monads

- Wikipedia: Throughout this article C denotes a category. A monad on C consists of an endofunctor `T` together with two natural transformations: $\eta\colon 1_C \to T$ (where 1_C denotes the identity functor on C) and $\mu\colon T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C).
  - natural transformation = morphism of functors
    - Let us call η `unit` (or `return`), and μ `join`
      - If Haskell syntax allowed it, we could say `return :: Identity -> T` and `join :: T`^{2} `-> T`

Monads

- Throughout this article C denotes a category. A monad on C consists of an endofunctor `T` together with two natural transformations: `return ::  a -> T a` and `join ::  T (T a) -> T a`.

Sequencing Actions

- `myAction' :: IO (IO ())`

Sequencing Actions

- `myAction' :: IO (IO ())`
- `join myAction' :: IO ()`

Sequencing Actions

- `myAction' :: IO (IO ())`
- `join myAction' :: IO ()`
- `Prelude Control.Monad> join myAction'`

Sequencing Actions

- `myAction' :: IO (IO ())`
- `join myAction' :: IO ()`
- `Prelude Control.Monad> join myAction'`
  `the_`

Sequencing Actions

- `myAction' :: IO (IO ())`
- `join myAction' :: IO ()`

-
  ```
  Prelude Control.Monad> join myAction'
  the_
  dog
  ```

Sequencing Actions

- `myAction' :: IO (IO ())`
- `join myAction' :: IO ()`

-
  ```
  Prelude Control.Monad> join myAction'
  the_
  dog
  "the_dog"
  ```

Monads

-
  ```
  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
  ```

Monads

-
  ```
  class (Applicative m) => 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
  ```
  - Since GHC v7.10, `Applicative` is a superclass of `Monad`

Monads

-
  ```
  class (Applicative m) => 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
  ```
  - What happened to `join`? What are `(>>=)`, `(>>)`, and `fail` doing here?

Monads

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

Monads

- `(>>=) ::  m a -> (a -> m b) -> m b`
- `(=<<) = flip (>>=)`

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

Monads

- `(>>=) ::  m a -> (a -> m b) -> m b`
- `(=<<) = flip (>>=)`

  ```
  (=<<) ::    (a -> m b) -> m a -> m b
  ```
- `(<*>) ::  f (a ->   b) -> f a -> f b`

Monads

- `(>>=) ::  m a -> (a -> m b) -> m b`
- `(=<<) = flip (>>=)`

  `(=<<) ::    (a -> m b) -> m a -> m b`
- `(<*>) ::  f (a ->   b) -> f a -> f b`
- `(<$>) ::    (a ->   b) -> f a -> f b`

Monads

- `(>>=) ::  m a -> (a -> m b) -> m b`
- `(=<<) = flip (>>=)`

  ```
  (=<<) ::    (a -> m b) -> m a -> m b
  ```
- `(<*>) ::  f (a ->   b) -> f a -> f b`
- `(<$>) ::    (a ->   b) -> f a -> f b`

  - `(=<<)` (and `(>>=)`) are maps for monadic functions

Monads

- `(>>=) ::  m a -> (a -> m b) -> m b`
- `(=<<) = flip (>>=)`

  ```
  (=<<) ::    (a -> m b) -> m a -> m b
  ```
- `(<*>) ::  f (a ->   b) -> f a -> f b`
- `(<$>) ::    (a ->   b) -> f a -> f b`

    - `(=<<)` (and `(>>=)`) are maps for monadic functions
        - Functions that create their own boxes

Monads

- `(>>=) ::  m a -> (a -> m b) -> m b`
- `(=<<) = flip (>>=)`

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

- `(<*>) ::  f (a ->   b) -> f a -> f b`
- `(<$>) ::    (a ->   b) -> f a -> f b`

  - `(=<<)` (and `(>>=)`) are maps for monadic functions
    - Functions that create their own context

Monads

- `g >>= f = join (fmap f g) ::  m a -> (a -> m b) -> m b`

Monads

- `g >>= f = join (fmap f g) :: m a -> (a -> m b) -> m b`
  - `f :: a -> m b` is a monadic function

Monads

- `g >>= f = join (fmap f g) :: m a -> (a -> m b) -> m b`
  - `f :: a -> m b` is a monadic function
  - `fmap f` lifts it to type `m a -> m (m b)`

Monads

- `g >>= f = join (fmap f g) :: m a -> (a -> m b) -> m b`
  - `f :: a -> m b` is a monadic function
  - `fmap f` lifts it to type `m a -> m (m b)`
  - `g :: m a` is a value of type a in a box

Monads

- `g >>= f = join (fmap f g) :: m a -> (a -> m b) -> m b`
  - `f :: a -> m b` is a monadic function
  - `fmap f` lifts it to type `m a -> m (m b)`
  - `g :: m a` is a value of type `a` in a box
  - `fmap f g :: m (m b)` outputs a value of type `b` in two nested boxes

Monads

- `g >>= f = join (fmap f g) :: m a -> (a -> m b) -> m b`
    - `f :: a -> m b` is a monadic function
    - `fmap f` lifts it to type `m a -> m (m b)`
    - `g :: m a` is a value of type `a` in a box
    - `fmap f g :: m (m b)` outputs a value of type `b` in two nested boxes
    - `join (fmap f g)` extracts a monadic value of type `m b` from the outermost box

Monads

- `g >>= f = join (fmap f g) :: m a -> (a -> m b) -> m b`
  - `f :: a -> m b` is a monadic function
  - `fmap f` lifts it to type `m a -> m (m b)`
  - `g :: m a` is a value of type `a` in a box
  - `fmap f g :: m (m b)` outputs a value of type `b` in two nested boxes
  - `g >>= f` extracts a value of type `a` from `g` and feeds it to `f` to get a monadic value of type `m b`

Monads

- `g >>= f = join (fmap f g) :: m a -> (a -> m b) -> m b`
  - `f :: a -> m b` is a monadic function
  - `fmap f` lifts it to type `m a -> m (m b)`
  - `g :: m a` is a value of type a in a box
  - `fmap f g :: m (m b)` outputs a value of type b in two nested boxes
  - `g >>= f` extracts a value of type a from g and feeds it to `f` to get a monadic value of type `m b`
- `join x = x >>= id`

Monads

-
  ```
  class (Applicative m) => 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
  ```
  - Shorthand for when we don't need to bind the value inside x to evaluate y

Monads

-
  ```
  class (Applicative m) => 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
  ```
    - Error handler for pattern matching in do expressions