Monads

Why programmers care

by David Darais

Why bother?

- Who uses monads?
- Why use monads?
- Do we *need* monads?
- Will I use monads after learning about them?
- What do monads have to do with...
    - Monoids? Functors? Category Theory?

What Monads do

Maybe (Option in ML)

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  case lookup “Bob” of
    Nothing -> Nothing
    Just p ->
      case favoriteColor p of
        Nothing -> Nothing
        Just c ->
          case teamOfColor c of
            Nothing -> Nothing
            Just t -> Just t
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam = do
  p <- lookup “Bob”
  c <- favoriteColor p
  t <- sportsTeamOfColor c
  return t
```

What Monads do

State

```
genThree :: Gen -> ([Num], Gen)
getThree g =
  let
    (n, g1)  = nextGen g
    (n1, g2) = nextGen g1
    (n2, g3) = nextGen g2
  in ([n, n1, n2], g3)
```

```
genThree :: Gen -> ([Num], Gen)
genThree = do
  n <- nextGen
  n1 <- nextGen
  n2 <- nextGen
  return [n, n1, n2]
```

What Monads do

Identity

```
area :: Rectangle -> Num
area r =
  let
    w = width r
    h = height r
  in (w * h)
```

```
area :: Rectangle -> Num
area = do
  w <- width r
  h <- height r
  return (w * h)
```

Monad is just two functions

- `(>>=)  :: m a -> (a -> m b) -> m b`
    - `(some people call this “shove” or “bind”)`
- `return :: a -> m a`

What Monads do

Maybe (Option in ML)

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  case lookup “Bob” of
    Nothing -> Nothing
    Just p ->
      case favoriteColor p of
        Nothing -> Nothing
        Just c ->
          case teamOfColor c of
            Nothing -> Nothing
            Just t -> Just t
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam = do
  p <- lookup “Bob”
  c <- favoriteColor p
  t <- sportsTeamOfColor c
  return t
```

What Monads do

Maybe (Option in ML)

```
lookup :: String -> Maybe Person
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  case lookup “Bob” of
    Nothing -> Nothing
    Just p ->
      case favoriteColor p of
        Nothing -> Nothing
        Just c ->
          case teamOfColor c of
            Nothing -> Nothing
            Just t -> Just t
```

```
favoriteColor :: Person -> Maybe Color
```

```
teamOfColor :: Color -> Maybe Team
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam = do
  p <- lookup “Bob”
  c <- favoriteColor p
  t <- sportsTeamOfColor c
  return t
```

What Monads do

Maybe (Option in ML)

```
bobsFavoriteTeam :: Maybe Te
bobsFavori
  case lo
    Nothing -> Nothing
    Just p ->
      case favoriteColor p of
        Nothing -> Nothing
        Just c ->
          case teamOfColor c of
            Nothing -> Nothing
            Just t -> Just t
```

```
lookup :: String -> Maybe Person
```

```
favoriteColor :: Person -> Maybe Color
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam = do
  p <- lookup “Bob”
  c <- favoriteColor p
  t <- sportsTeamOfColor c
  return t
```

```
teamOfColor :: Color -> Maybe Team
```

What Monads do

State

```
genThree :: Gen -> ([Num], Gen)
getThree g =
  let
    (n, g1)  = nextGen g
    (n1, g2) = nextGen g1
    (n2, g3) = nextGen g2
  in ([n, n1, n2], g3)
```

```
genThree :: Gen -> ([Num], Gen)
genThree = do
  n <- nextGen
  n1 <- nextGen
  n2 <- nextGen
  return [n, n1, n2]
```

What Monads do

State

```
nextGen :: Gen -> (Num, Gen)
```

```
genThree :: Gen -> ([Num], Gen)
getThree g =
  let
    (n, g1)  = nextGen g
    (n1, g2) = nextGen g1
    (n2, g3) = nextGen g2
  in ([n, n1, n2], g3)
```

```
genThree :: Gen -> ([Num], Gen)
genThree = do
  n <- nextGen
  n1 <- nextGen
  n2 <- nextGen
  return [n, n1, n2]
```

What Monads do

State

```
genThree :: Gen -> ([Num], Gen)
getThree g =
  let
    (n, g1)  = nextGen g
    (n1, g2) = nextGen g1
    (n2, g3) = nextGen g2
  in ([n, n1, n2], g3)
```

```
nextGen :: Gen -> (Num, Gen)
```

```
genThree :: Gen -> ([Num], Gen)
genThree = do
  n <- nextGen
  n1 <- nextGen
  n2 <- nextGen
  return [n, n1, n2]
```

Maybe Monad

```
data Maybe = Nothing | Just a

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

Nothing >>= f = Nothing

(Just a) >>= f = f a

return :: a -> Maybe a

return x = Maybe x
```

Desugaring “do”

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam = do
  p <- lookup “Bob”
  c <- favoriteColor p
  t <- sportsTeamOfColor c
  return t
```

⇨

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p -> do
    c <- favoriteColor p
    t <- sportsTeamOfColor c
    return t)
```

Desugaring “do”

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p -> do
    c <- favoriteColor p
    t <- sportsTeamOfColor c
    return t)
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p ->
  favoriteColor p >>= (\c -> do
    t <- sportsTeamOfColor c
    return t))
```

Desugaring “do”

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p ->
  favoriteColor p >>= (\c -> do
    t <- sportsTeamOfColor c
    return t))
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p ->
  favoriteColor p >>= (\c ->
  sportsTeamOfColor c >>= (\t -> do
    return t)))
```

Desugaring “do”

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p ->
  favoriteColor p >>= (\c ->
  sportsTeamOfColor c >>= (\t -> do
    return t)))
```

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p ->
  favoriteColor p >>= (\c ->
  sportsTeamOfColor c >>= (\t ->
    return t)))
```

Desugaring “do”

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam =
  lookup “Bob” >>= (\p ->
  favoriteColor p >>= (\c ->
  sportsTeamOfColor c >>= (\t ->
    return t)))
```

=

```
bobsFavoriteTeam :: Maybe Team
bobsFavoriteTeam = do
  p <- lookup “Bob”
  c <- favoriteColor p
  t <- sportsTeamOfColor c
  return t
```

Identity Monad

```
data Identity a = Identity a

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

(Identity a) >>= f = f a

return :: a -> m a

return a = Identity a
```

Desugaring “do”

```
area :: Rectangle -> Num
area = do
  w <- width r
  h <- height r
  return (w * h)
```

➡

```
area :: Rectangle -> Num
area =
  width r >>= (\w -> do
    h <- height r
    return (w * h))
```

Desugaring “do”

```
area :: Rectangle -> Num
area =
  width r >>= (\w -> do
    h <- height r
    return (w * h))
```

```
area :: Rectangle -> Num
area =
  width r >>= (\w ->
  height r >>= (\h ->
    return (w * h))
```

What’s Next?

- Functors and Monoids (useful like monads)
- Monad Transformers (necessary)
  - A way to compose multiple monads
- Arrows (really cool)
  - Also generalizes boilerplate
  - All monads are arrows

Building the State Monad

- State Monad in Scheme
- DFS state passing style
- DFS monad style