

Hunt the Trevor

Elliot Greenwood



Agenda

1 What is Hunt the Wumpus?

2 What is a Monad?

- Intuition
- Theory

3 Reader

- Motivation
- Definition
- Example

4 State

- Motivation
- Definition
- Example

5 Writer

- Motivation
- Definition
- Example

6 It's Trevor Hunting Time

What is Hunt the Wumpus?

Why are we Hunting Poor Trevor?

You, the brave adventurer, have gotten lost and stumbled into a dark cavern. You can barely see your hand in front of your face. You heard rumours that Trevor, the Wumpus, lived in these here parts.

Ahead of you are:

- 20 Caves, each connected to 3 other caves
- 5 Arrows (a.k.a. 5 attempts to kill Trevor)
- 2 crazy bats (that like to transport you to other caves)
- 2 endless pits (that almost certainly mean death)
- Some great fun from 1973

On your turn you can Move to an adjoining cave, via a dark tunnel, or shoot one of your crooked arrows in the hope of hitting the Wumpus. Fortunately, the bats make a lot of noise, the pits cause an awful draft, & Trevor is quite smelly!

What is a Monad?

Intuition



- A monad abstracts computation away
- It is defined by two functions:
 - ▶ one to take non-monadic value and *return* a monadic one
 - ▶ another to take monadic value and *bind* the underlying value to a function

What is a Monad?

Theory

Functor Definition

$$\text{fmap} :: \mathbf{Functor} \, f \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b$$

Applicative Definition

$$\begin{aligned} \text{pure} &:: \mathbf{Applicative} \, f \Rightarrow a \rightarrow f a \\ (<*>) &:: \mathbf{Applicative} \, f \Rightarrow f(a \rightarrow b) \rightarrow f a \rightarrow f b \end{aligned}$$

What is a Monad?

Theory

Monad Definition

$$\text{return} :: \mathbf{Monad} \ m \Rightarrow a \rightarrow m \ a$$
$$(>>=) :: \mathbf{Monad} \ m \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b$$

- `return` constructs the monad from a value
- `>>=` allows composition of the monad
- *Note: You might see other people say you need `join` to define the monad, providing either `>>=` or `join` is “rigorously” equivalent.*

Reader Monad

Motivation

```
loop :: GameState -> WorldConfig -> IO GameState
loop gs wc = do
  let m = maze wc
  let cave = gCave gs
  let tunnels = sort (m ! cave)

  putStrLn $ Msg.youAreInCave cave
  mapM_ putStrLn $ sense gs wc
  putStrLn $ Msg.tunnelsLeadTo tunnels

  action <- getAction tunnels
  let (gs', logs) = execute action gs wc
  let shouldDebug l = (isDebug wc) || (not $ isPrefixOf "[DEBUG]" l)
  mapM_ putStrLn $ filter shouldDebug logs

  case gameOver gs' of
    Nothing -> loop gs' wc
    Just Win -> putStrLn Msg.win >> return gs'
    Just Lose -> putStrLn Msg.lose >> return gs'
```

- Notice how we must pass this `wc` around
- We must pass it through functions that may never use it

Reader Monad

Type

```
class Reader<TC, TA> {  
  Reader(T runReader) {  
    this.runReader = runReader;  
  }  
}
```

```
loop :: GameState -> WorldConfig -> IO GameState  
sense :: GameState -> WorldConfig -> [String]  
execute :: Action -> GameState -> WorldConfig -> (GameState, [String])  
emptyCave :: GameState -> WorldConfig -> (GameState, Cave)  
anotherCave :: GameState -> WorldConfig -> (GameState, Cave)  
getMoveEvent :: Action -> GameState -> WorldConfig -> Maybe MoveEvent
```

Exercise: Reader Type Definition

```
newtype Reader c a = Reader {runReader :: }
```


Reader Monad

Type

Recall: newtype **Reader** $c\ a = \text{Reader} \{ \text{runReader} :: c \rightarrow a \}$

Exercise: Reader Type Definition

$\text{return} :: a \rightarrow \text{Reader } c\ a$

$\text{return } a =$

$(\gg=) :: \text{Reader } c\ a \rightarrow (a \rightarrow \text{Reader } c\ b) \rightarrow \text{Reader } c\ b$

$x \gg= fn =$

Reader Monad

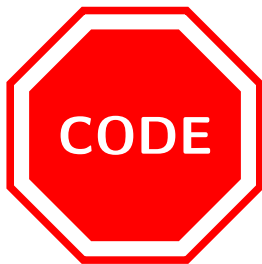
Functions

Homework: Reader Functions

$\text{ask} :: \mathbf{Reader} \ c \ a \rightarrow a$

$\text{asks} :: \mathbf{Reader} \ c \ a \rightarrow (a \rightarrow \mathbf{Reader} \ c \ b) \rightarrow \mathbf{Reader} \ c \ b$

$\text{local} :: (c \rightarrow c') \rightarrow \mathbf{Reader} \ c' \ a \rightarrow \mathbf{Reader} \ c \ a$



State Monad

Motivation

```
loop :: (World m, MonadIO m) => GameState -> m GameState
loop gs = do
  m <- asks maze
  let cave = gCave gs
  let tunnels = sort (m ! cave)

  putStrLn $ Msg.youAreInCave cave
  sense gs >=> traverse_ putStrLn
  putStrLn (Msg.tunnelsLeadTo tunnels)

  action <- getAction tunnels
  d <- asks isDebug
  let shouldDebug l = d || (not $ isPrefixOf "[DEBUG]" l)
  (gs', logs) <- execute action gs
  traverse_ putStrLn $ filter shouldDebug logs

case gameOver gs' of
  Nothing -> loop gs'
  Just Win -> putStrLn Msg.win >> return gs'
  Just Lose -> putStrLn Msg.lose >> return gs'
```

- Notice how we must pass this `gs*` around
- Again, we must pass it through functions that may never use it
- And this time we get back a new version which we must remember to use so we do not lose state

State Monad

Type

```
runWumpus :: MonadIO m => GameState -> WorldConfig -> m (GameState, ())
loop :: (World m, MonadIO m) => GameState -> m (GameState, ())
execute :: World m => Action -> GameState -> m (GameState, [String])
emptyCave :: World m => GameState -> m (GameState, Cave)
anotherCave :: World m => GameState -> m (GameState, Cave)
```

Exercise: Reader Type Definition

```
newtype State s a = State { runState ::
```

State Monad

Type

Recall: newtype $\mathbf{State} \, s \, a = \mathbf{State} \{ \text{runState} :: s \rightarrow (s, a) \}$

Exercise: State Type Definition

$\text{return} :: a \rightarrow \mathbf{State} \, s \, a$

$\text{return} \, a =$

$(\gg=) :: \mathbf{State} \, s \, a \rightarrow (a \rightarrow \mathbf{State} \, s \, b) \rightarrow \mathbf{State} \, s \, b$

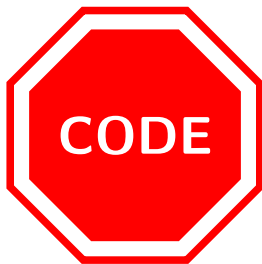
$x \gg= fn =$

State Monad

Functions

Homework: State Functions

```
get :: State s s
gets :: (s → a) → State s a
put :: s → State s ()
modify :: (s → s) → State s ()
```



Writer Monad

Motivation

```
execute :: (Game m) => Action -> m [String]
execute a@(Move c) = do
  eCave <- emptyCave

  updateHistory a
  let logs = ["[DEBUG] Updated cave to Cave " ++ show c]
  modify (\s -> s { gCave = c })
  let logs' = ("[DEBUG] Updated history with action " ++ show a) : logs
  getMoveEvent a >>= \case
    Just Wumpus -> (modify $ \s -> s { gameOver = Just Lose }) >> (return $ Msg.encounterWumpus : logs')
    Just Pit    -> (modify $ \s -> s { gameOver = Just Lose }) >> (return $ Msg.losePits : logs')
    Just Bat    -> (modify $ \s -> s { gCave = eCave }) >> (return $ Msg.encounterBats : ("[DEBUG] Updated hi
    Nothing     -> return logs'
```

- Notice how we must add onto this list of strings
- Functions don't care about the logs from what proceeded them, so why should we have to know about them

Writer Monad

Type

```
sense :: (Game m) => m ([String], ())  
execute :: (Game m) => Action -> m ([String], ())
```

Exercise: Writer Type Definition

writer **Writer** $[\ell]$ a = Writer {runWriter :: }

Writer Monad

Type

Recall: writer **Writer** $[\ell] a = \text{Writer} \{ \text{runWriter} :: ([\ell], a) \}$

Exercise: Writer Type Definition

$\text{return} :: a \rightarrow \text{Writer} [\ell] a$

$\text{return } a =$

$(\gg=) :: \text{Writer} [\ell] a \rightarrow \hookrightarrow$

$(a \rightarrow \text{Writer} [\ell] b) \rightarrow \text{Writer} [\ell] b$

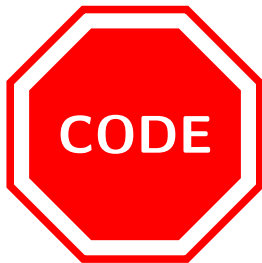
$x \gg= fn =$

Writer Monad

Functions

Homework: Writer Functions

```
tell ::  $\ell \rightarrow \mathbf{Writer} \ell ()$   
censor ::  $(\ell \rightarrow \ell) \rightarrow \mathbf{Writer} \ell a \rightarrow \mathbf{Writer} \ell a$   
listen ::  $\mathbf{Writer} \ell a \rightarrow \mathbf{Writer} \ell (a, \ell)$ 
```



That's all Folks!

