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

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)

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)

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)

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

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

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.

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

- The functional programming style forces you to expose your inputs, outputs and intentions
- Monads help cover them up again
- They can then allow you to seamlessly introduce abstractions, *only where they are needed*, letting you focus on the business logic
- They provide convenient frameworks for effects found in imperative languages [1](e.g. raising exceptions, null checking, random number generators, and, *cough*, I/O)

[1] Wadler, P. (1995), Monads for Functional Programming, in 'Advanced Functional Programming' , Springer, London , pp. 24–52 .

What is a Monad?

Theory

Monad Definition

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

- return constructs the monad from a value
- $\gg\!=$ allows composition of the monad
- *Note: You might see other people say you need* join *to define the monad, providing either* $\gg\!=$ *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

$$\text{newtype } \mathbf{Reader}\ c\ \mathsf{a} = \text{Reader}\ \{\text{runReader} :: \qquad\ \}$$

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

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

Reader Monad

Type

Recall: newtype $\mathbf{Reader}\, c\, a = \text{Reader}\,\{\text{runReader} :: c \to a\}$

Exercise: Reader Type Definition

$$\text{return} :: a \to \mathbf{Reader}\, c\, a$$
$$\text{return}\ a =$$

$$(\gg\!=) :: \mathbf{Reader}\, c\, a \to (a \to \mathbf{Reader}\, c\, b) \to \mathbf{Reader}\, c\, b$$
$$x \gg\!= fn =$$

Reader Monad

Type

Recall: newtype $\mathbf{Reader}\ c\ a = \text{Reader}\ \{\text{runReader} :: c \to a\}$

Exercise: Reader Type Definition

$$\text{return} :: a \to \mathbf{Reader}\ c\ a$$
$$\text{return}\ a = \text{Reader}\ (\lambda \_ \to a)$$

$$(\gg\!=) :: \mathbf{Reader}\ c\ a \to (a \to \mathbf{Reader}\ c\ b) \to \mathbf{Reader}\ c\ b$$
$$x \gg\!= fn =$$

Reader Monad

Type

Recall: newtype $\mathbf{Reader}\ c\ a = \text{Reader}\ \{\text{runReader} :: c \to a\}$

Exercise: Reader Type Definition

$$\text{return} :: a \to \mathbf{Reader}\ c\ a$$
$$\text{return}\ a = \text{Reader}\ (\lambda \_ \to a)$$

$$(\gg\!=) :: \mathbf{Reader}\ c\ a \to (a \to \mathbf{Reader}\ c\ b) \to \mathbf{Reader}\ c\ b$$
$$x \gg\!= fn = \text{Reader}\ (\lambda c \to \text{runReader}\ (fn\ (\text{runReader}\ x\ c))\ c)$$

Reader Monad

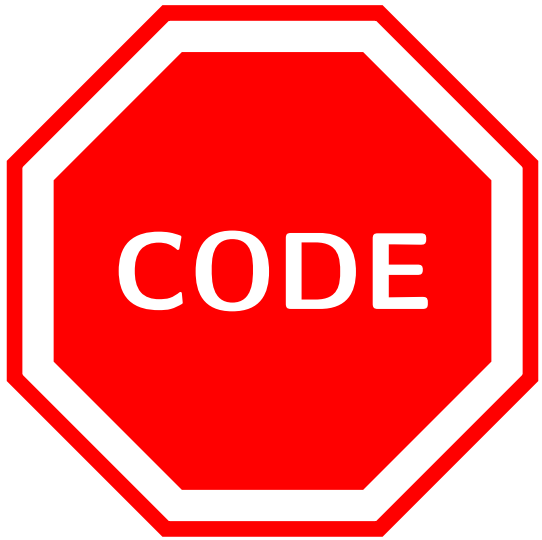
Functions

Homework: Reader Functions

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

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

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


CODE

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

$$\text{newtype } \mathbf{State}\ s\ \mathrm{a} = \text{State } \{\text{runState} :: \quad \}$$

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

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

State Monad

Type

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

Exercise: State Type Definition

$$\mathrm{return} :: a \to \mathbf{State}\, s\, a$$
$$\mathrm{return}\ a =$$

$$(\gg\!=) :: \mathbf{State}\, s\, a \to (a \to \mathbf{State}\, s\, b) \to \mathbf{State}\, s\, b$$
$$x \gg\!= fn =$$

State Monad

Type

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

Exercise: State Type Definition

$$\text{return} :: a \to \mathbf{State}\, s\, a$$
$$\text{return}\ a = \text{State}\ (\lambda s \to (s, a))$$

$$(\gg\!=) :: \mathbf{State}\, s\, a \to (a \to \mathbf{State}\, s\, b) \to \mathbf{State}\, s\, b$$
$$x \gg\!= fn =$$

State Monad

Type

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

Exercise: State Type Definition

$$\mathrm{return} :: a \to \mathbf{State}\, s\, a$$
$$\mathrm{return}\ a = \mathrm{State}\ (\lambda s \to (s, a))$$

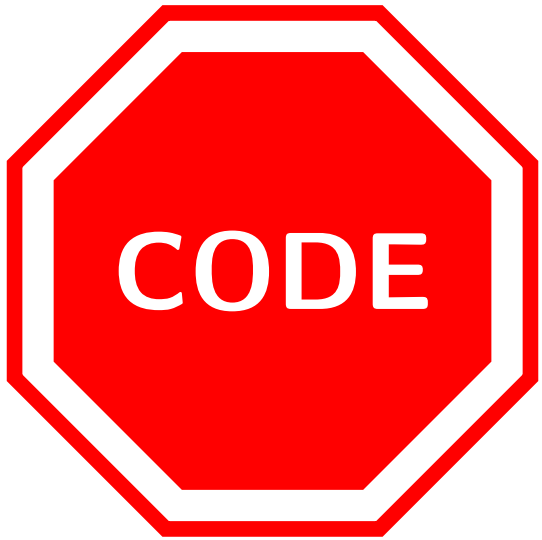
$$(\gg\!=) :: \mathbf{State}\, s\, a \to (a \to \mathbf{State}\, s\, b) \to \mathbf{State}\, s\, b$$
$$x \gg\!= fn = \mathrm{State}\ (\lambda s \to \mathbf{let}\ (s', a) = \mathrm{runState}\ x\ s \ \mathbf{in}\ \mathrm{runState}\ (fn\ a)\ s')$$

State Monad

Functions

Homework: State Functions

$$\begin{aligned}
\text{get} &:: \mathbf{State}\, s\, s \\
\text{gets} &:: (s \to a) \to \mathbf{State}\, s\, a \\
\text{put} &:: s \to \mathbf{State}\, s\, () \\
\text{modify} &:: (s \to s) \to \mathbf{State}\, s\, ()
\end{aligned}$$


CODE

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

$$\text{writer}\ \mathbf{Writer}\ [\ell]\ \text{a} = \text{Writer}\ \{\text{runWriter} :: \quad \}$$

Writer Monad

Type

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

Exercise: Writer Type Definition

$$\text{writer}\ \mathbf{Writer}\ [\ell]\ \text{a} = \text{Writer}\left\{\text{runWriter} :: ([\ell], \text{a})\right\}$$

Writer Monad

Type

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

Exercise: Writer Type Definition

$$\mathsf{return} :: a \to \mathbf{Writer}\,[\ell]\,a$$

$$\mathsf{return}\ a =$$

$$(\gg\!=) :: \mathbf{Writer}\,[\ell]\,a \to (a \to \mathbf{Writer}\,[\ell]\,b) \to \mathbf{Writer}\,[\ell]\,b$$

$$x \gg\!= fn =$$

Writer Monad

Type

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

Exercise: Writer Type Definition

$$\mathsf{return} :: a \rightarrow \mathbf{Writer}\,[\ell]\,a$$

$$\mathsf{return}\ a = \mathsf{Writer}\ ([\,], a)$$

$$(\gg\!=) :: \mathbf{Writer}\,[\ell]\,a \rightarrow (a \rightarrow \mathbf{Writer}\,[\ell]\,b) \rightarrow \mathbf{Writer}\,[\ell]\,b$$

$$x \gg\!= fn =$$

Writer Monad

Type

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

Exercise: Writer Type Definition

$$\begin{aligned}
&\mathsf{return} :: a \rightarrow \mathbf{Writer}\,[\ell]\,a \\
&\mathsf{return}\ a = \mathsf{Writer}\ ([\,], a)
\end{aligned}$$

$$\begin{aligned}
&(\gg\!=) :: \mathbf{Writer}\,[\ell]\,a \rightarrow \\
&\qquad\qquad (a \rightarrow \mathbf{Writer}\,[\ell]\,b) \rightarrow \mathbf{Writer}\,[\ell]\,b \\
&x \gg\!= fn = \mathsf{Writer}\ (\mathbf{let}\ \ (l, a) = \mathsf{runWriter}\ x \\
&\qquad\qquad\qquad\qquad\quad (l', b) = \mathsf{runWriter}\ (fn\ a) \\
&\qquad\qquad\qquad\quad \mathbf{in}\ (l +\!\!+ l', b))
\end{aligned}$$

Writer Monad

Monoid

Let: writer $\mathbf{Writer}\,\ell\,a = \text{Writer}\,\{\text{runWriter} :: (\ell, a)\}$

Exercise: Writer Type Definition

$$\begin{aligned}
\text{return} &:: a \rightarrow \mathbf{Writer}\,\ell\,a \\
\text{return}\ a &= \text{Writer}\ (\text{mempty}, a)
\end{aligned}$$

$$\begin{aligned}
(\gg\!=) &:: \mathbf{Writer}\,\ell\,a \rightarrow \\
&\qquad (a \rightarrow \mathbf{Writer}\,\ell\,b) \rightarrow \mathbf{Writer}\,\ell\,b \\
x \gg\!= fn &= \text{Writer}\ (\mathbf{let}\ \ (l, a) = \text{runWriter}\ x \\
&\qquad\qquad\qquad\quad (l', b) = \text{runWriter}\ (fn\ a) \\
&\qquad\qquad\quad \mathbf{in}\ (\text{mappend}\ l\ l', b))
\end{aligned}$$

Writer Monad

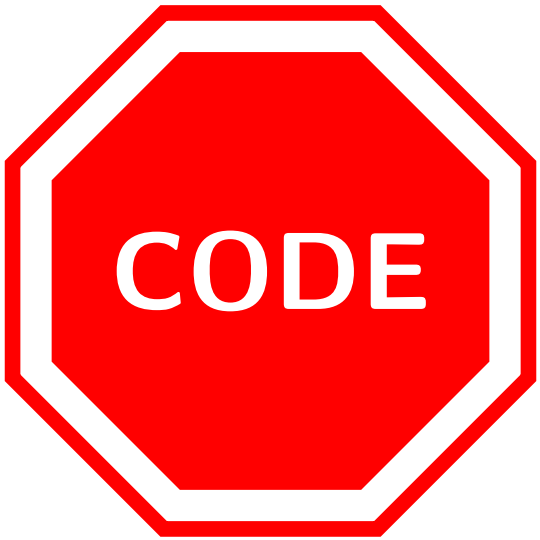
Functions

Homework: Writer Functions

$$\text{tell} :: \ell \rightarrow \mathbf{Writer}\,\ell\,()$$

$$\text{censor} :: (\ell \rightarrow \ell) \rightarrow \mathbf{Writer}\,\ell\,a \rightarrow \mathbf{Writer}\,\ell\,a$$

$$\text{listen} :: \mathbf{Writer}\,\ell\,a \rightarrow \mathbf{Writer}\,\ell\,(a, \ell)$$


CODE

That's all Folks!