ADVANCED

# FUNCTIONAL PROGRAMMING

# OUTLINE

- Type classes for flexibility
- Mutability
- Arrays

# MONAD, STATE, MONADSTATE

- In `mtl`, there are monads, monad transformers, and monad type classes
- All this helps your code to be more readable and maintainable

```
type State s = StateT s Identity

newtype StateT s (m :: Type → Type)

class Monad m ⇒ MonadState s m | m → s where
```

# MONADSTATE TYPE CLASS

- `MonadState` describes the interface of a monad capable of working with state
- You can make your custom monad behave like a state monad by instantiating the class
- What does `m → s` mean?

```
class Monad m ⇒ MonadState s m | m → s where
    get :: m s
    get = state (\s → (s, s))

    put :: s → m ()
    put s = state (\_ → ((), s))

    state :: (s → (a, s)) → m a
    state f = do
      s ← get
      let ~(a, s') = f s
      put s'
      return a
    {-# MINIMAL state | get, put #-}
```

# FUNCTIONAL DEPENDENCIES

- A way to constrain type parameters of a class
- In a multi-parameter type class, it's possible to determine a parameter from others
- Will fail to type check if the uniqueness of the parameter is not guaranteed

```
data Vector = V Int Int deriving (Eq, Show)
data Matrix = M Vector Vector deriving (Eq, Show)

class Mult a b c where
  (***) :: a → b → c
```

# MONADREADER AND MONADWRITER

```haskell
class (Monoid w, Monad m) ⇒
    MonadWriter w m | m → w where
    {-# MINIMAL (writer | tell), listen, pass #-}
    writer :: (a,w) → m a
    writer ~(a, w) = do
      tell w
      return a

    tell   :: w → m ()
    tell w = writer ((),w)

    listen :: m a → m (a, w)
    pass   :: m (a, w → w) → m a
```

```haskell
class Monad m ⇒ MonadReader r m | m → r where
    {-# MINIMAL (ask | reader), local #-}
    ask :: m r
    ask = reader id

    local :: (r → r) → m a → m a

    reader :: (r → a) → m a
    reader f = do
      r ← ask
      return (f r)
```

# RWS

```
module Control.Monad.RWS.Class (
    MonadRWS,
    module Control.Monad.Reader.Class,
    module Control.Monad.State.Class,
    module Control.Monad.Writer.Class,
  ) where

import Control.Monad.Reader.Class
import Control.Monad.State.Class
import Control.Monad.Writer.Class

class (Monoid w, MonadReader r m, MonadWriter w m, MonadState s m)
   ⇒ MonadRWS r w s m | m → r, m → w, m → s
```

# EXERCISE

- Rewrite your implementation for the Imp language to use `RWST` monad transformer
- It should accept a config which states whether you're supposed to give up at the first mistake or continue requesting the user to input a correct int number
- It should write the log of all incorrect attempts

```
type EvalM
  = RWST Config [Error] VarMap (ExceptT Error IO)
```

# OUTLINE

- Type classes for flexibility
- Mutability
- Arrays

# IORef

- Did you miss variables? Now you have them!
- Only exist in the IO context
  - There are `STRef` and other mutable variables in other monads
- Don’t use `IORef` in concurrent code
  - Use `MVar`s instead

```
import Data.IORef

main = do
  x ← newIORef 0
  increment x
  increment x
  counter ← readIORef x
  putStrLn "The counter is"
  print counter

increment :: IORef Int → IO ()
increment ref =
  modifyIORef ref (+1)
```

# IN-PLACE BUBBLE SORT

```ruby
def bubble_sort(list)
  list.each_index do |i|
    (list.length - i - 1).times do |j|
      if list[j] > list[j + 1]
        list[j], list[j + 1] = list[j + 1], list[j]
      end
    end
  end
end
```

# IN-PLACE BUBBLE SORT

```ruby
def bubble_sort(list)
  list.each_index do |i|
    (list.length - i - 1).times do |j|
      if list[j] > list[j + 1]
        list[j], list[j + 1] = list[j + 1], list[j]
      end
    end
  end
end
```

```haskell
bubbleSort :: [Int] → IO [Int]
bubbleSort input = do
  let ln = length input

  xs ← mapM newIORef input

  forM_ [0 .. ln - 1] $ \_ → do
    forM_ [0 .. ln - 2] $ \j → do
      let ix = xs !! j
      let iy = xs !! (j + 1)

      x ← readIORef ix
      y ← readIORef iy

      when (x > y) $ do
        writeIORef ix y
        writeIORef iy x

  mapM readIORef xs
```

# OUTLINE

- Type classes for flexibility
- Mutability
- Arrays

# IOARRAY

- Actual arrays with mutable elements
- `newArray` takes the smallest and the largest indices and initializes the array with the last arg
- `readArray` reads an element of the array
- `writeArray` modifies the element

```
foo :: IO ()
foo = do
  arr ← newArray (1, 10) 13
         :: IO (IOArray Int Int)
  a ← readArray arr 1
  writeArray arr 1 42
  b ← readArray arr 1
  print (a, b)
```