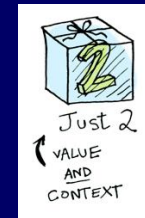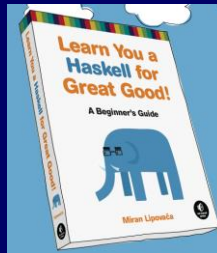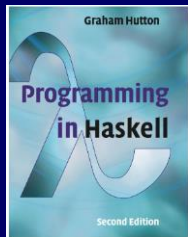# State Monads

Monads

Based on lecture notes by Graham Hutton,
the book "Learn You a Haskell for Great Good",
pictures from Aditya Bhargava, and
https://wiki.haskell.org/State_Monad

Monads

We have seen how Monads work.

There is a nice introduction to Writer, Reader and State Monads here (Aditya Bhargava)

We will look at the State Monad most closely.

State Monad

```
import Control.Monad.State
--will include
newtype State s a = State { runState :: (s -> (a,s)) }
```

There are a few things going on here
- *newtype* is a lot like *data*, except for some details
- what's "in" our data type? It's a function! And we're implicitly defining a function to extract our inner function from the data type: that function is called runState
- It's type is *s -> (a,s)*. Essentially, it's a type for any function that takes some initial state and then returns a tuple of *(regular return value, new state)*.

State Monad – primitives return, runState

> runState (return 'X') 1
('X', 1)

return   --set the result value but leave the state unchanged.
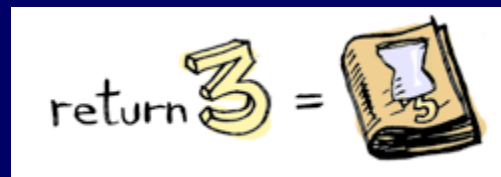
return a = State $ \s -> (a, s)

State Monad - put

put -- set the result value to () and set the state value.

-- ie: (put 5) 1 -> ((),5)

put :: s -> State s

put x s = (() ,x)

State Monad - get

get - set the result value to the state and leave the state unchanged

>runState get 1
(1, 1)

State Monad - helpers

evalState and **execState** just select one of the two values returned by runState.
EvalState returns the final result while execState returns the final state:

```
evalState :: State s a -> s -> a
evalState act = fst . runState act


execState :: State s a -> s -> s
execState act = snd . runState act
```

State Monad –simple example

```
import Control.Monad.State

greeter :: State String String
greeter = do
        name <- get
        put "Some State"
        return ("hello, " ++ name ++ "!")



>runState greeter "Mairead"
("hello, Mairead!","Some State")
```

State Monad - parse example

```haskell
import Control.Monad.State
-- Example use of State monad
- Passes a string of dictionary {a,b,c}
-- Game is to produce a number from the string.
-- By default the game is off, a c toggles the
-- game on and off. A 'a' gives +1 and a b gives -1.
-- E.g -- 'ab' = 0 -- 'ca' = 1 -- 'cabca' = 0
-- State = game is on or off & current score
--         = (Bool, Int)
```

State Monad - parse example

```haskell
type GameValue = Int
type GameState = (Bool, Int)
playGame :: String -> State GameState GameValue
playGame [] = do
       (_, score) <- get
         return score
playGame (x:xs) = do
        (on, score) <- get
        case x of
                'a' | on -> put (on, score + 1)
                'b' | on -> put (on, score - 1)
                'c' -> put (not on, score)
                _ -> put (on, score)
        playGame xs
startState = (False, 0)
main = print $ evalState (playGame "abcaaacbbcabbab") startState
```

State Monad – stack example

```haskell
import Control.Monad.State
type Stack = []

pop :: State Stack Int
pop = state $ \(x:xs) -> (x, xs)


push :: Int -> State Stack ()
push a = state $ \xs -> ((), a:xs)
```

State Monad – stack example

```
stackManip :: State Stack Int
stackManip = do
        push 3
        a <- pop
        pop
```

Run as

```
>runState stackManip [1,2,3]
(1,[2,3])
```

State Monad – stack example

```
stackStuff :: State Stack ()
stackStuff = do
        a <- pop
        if a == 5
                then push 5
                else do
                        push 3
                        push 8
```

Run as

```
>runState stackStuff [9,0,2,1,0]
(1,[2,3])
```

State Monad – simple state

```haskell
-- a concrete and simple example of using the State monad
import Control.Monad.State
-- the State is an integer.
-- the value will always be the negative of the state
type MyState = Int

valFromState :: MyState -> Int
valFromState s = -s
nextState :: MyState->MyState
nextState x = 1+x
```

State Monad – simple state

MyStateMonad **=** State MyState *-- this is it, the State transformation. Add 1 to the state, return -1*the state as the computed value.*

getNext **::** MyStateMonad Int

getNext **=** state (\st **-> let** st' **=** nextState(st) **in**
                                  (valFromState(st'),st'))

-- advance the state three times.

inc3**::**MyStateMonad Int

inc3 **=**   getNext >>= \x **->**

       getNext >>= \y **->**

       getNext >>= \z **->**

       return z

State Monad – simple state

```
-- advance the state three times with do sugar
 inc3Sugared :: MyStateMonad Int
inc3Sugared = do
          x <- getNext
          y <- getNext
          z <- getNext
          return z
```

State Monad – simple state

-- *advance the state three times without inspecting computed values*

inc3DiscardedValues**::**MyStateMonad Int

inc3DiscardedValues **=** getNext >> getNext >> getNext

-- *advance the state three times without inspecting computed values with do sugar*

inc3DiscardedValuesSugared**::**MyStateMonad Int

inc3DiscardedValuesSugared **= do**

getNext

getNext

getNext

State Monad – simple state

-- *advance state 3 times, compute the square of the state*

inc3AlternateResult**::** MyStateMonad Int

inc3AlternateResult **= do**   getNext

        getNext

        getNext

        s**<-**get

        return (s*s)

State Monad – simple state

-- advance state 3 times, ignoring computed value, and

--then once more

inc4**::**MyStateMonad Int

inc4 **= do** inc3AlternateResult

      getNext

main **= do** print (evalState inc3 0) *-- -3*

      print (evalState inc3Sugared 0) *-- -3*

      print (evalState inc3DiscardedValues 0) *-- -3*

      print (evalState inc3DiscardedValuesSugared 0) *-- -3*

      print (evalState inc3AlternateResult 0) *-- 9*

      print (evalState inc4 0) *-- -4*