State Monad Example (3H)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Some Examples (1)

```
module StateGame where

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)
```

https://wiki.haskell.org/State_Monad

Some Examples (2)

```
type GameValue = Int
type GameState = (Bool, Int)

playGame :: String -> State GameState GameValue
playGame []     = do
    (_, score) <- get
    return score
```

https://wiki.haskell.org/State_Monad

Some Examples (3)

```
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
```

https://wiki.haskell.org/State_Monad

Dice Examples

The result type : **Int** dice : a number between 1 and 6

The state type : a pseudo-random generator of type **StdGen**

the type of the **state processors** will be

State StdGen Int	**State s a**
StdGen -> (Int, StdGen)	**s -> (a, s)**

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

randomR – a state processing function

randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)

randomR (1, 6) :: StdGen -> (Int, StdGen)

assume a is Int (a, a) : range

and g is StdGen a seed

the StdGen type : an instance of RandomGen

randomR a state processing function

A seed of the type StdGen

A new seed is generated

by newStdGen

(Int, StdGen)

(a random value, a new seed)

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Generating a random number

to generate a random number with a seed
type **StdGen** must be used,
randomR is used to generate a number
newStdGen is used to create a new seed
(this will have to be done in IO).

```
import System.Random
g <- newStdGen
randomR (1, 10) g
(1,1012529354 2147442707)
```

A seed of the type **StdGen**
A new seed is generated
by **newStdGen**

The result of **randomR** is a tuple
(a random value, a new seed)

https://stackoverflow.com/questions/8416365/generate-a-random-integer-in-a-range-in-haskell

randomRIO

Otherwise, you can use **randomRIO**
to get a random number directly in the **IO** monad,
Without explicitly using a seed of type **StdGen**

```
import System.Random
randomRIO (1, 10)
6
```

https://stackoverflow.com/questions/8416365/generate-a-random-integer-in-a-range-in-haskell

randomR

```
randomR (1, 6) :: StdGen -> (Int, StdGen)

rollDie :: State StdGen Int
rollDie = state $ randomR (1, 6)
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

randomR

```
import Control.Monad.Trans.State
import System.Random

-- The StdGen type we are using is an instance of RandomGen.
randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)

randomR (1, 6) :: StdGen -> (Int, StdGen)
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

randomR

```
rollDice :: State StdGen (Int, Int)
rollDice = liftA2 (,) rollDie rollDie

GHCi> evalState rollDice (mkStdGen 666)
 (6,1)
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

rollDice

```
rollDice :: State StdGen (Int, Int)
rollDice = liftA2 (,) rollDie rollDie
```

That function rolls two dice.
Here, **liftA2** is used to make the two-argument function **(,)** work within a monad or applicative functor, in this case **IO**.
It can be easily defined in terms of **(<*>)**:

```
liftA2 f u v = f <$> u <*> v
```

As for (,), it is the non-infix version of the tuple constructor.
That being so, the two die rolls will be returned as a tuple in **I**

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Removing IO

```
randomR (1, 6) :: StdGen -> (Int, StdGen)

GHCi> :m System.Random
GHCi> let generator = mkStdGen 0          -- "0" is our seed
GHCi> :t generator
generator :: StdGen
GHCi> generator
1 1
GHCi> :t random
random :: (RandomGen g, Random a) => g -> (a, g)
GHCi> random generator :: (Int, StdGen)
(2092838931,1601120196 1655838864)
```

A seed of the type **StdGen**

A new seed is generated

by **newStdGen**

https://stackoverflow.com/questions/8416365/generate-a-random-integer-in-a-range-in-haskell

Dice without IO

GHCi> **randomR (1,6) (mkStdGen 0)**
(6, 40014 40692)

The resulting tuple combines
the result of throwing a single die with a new generator.
A simple implementation for throwing two dice is then:

```
clumsyRollDice :: (Int, Int)
clumsyRollDice = (n, m)
    where
    (n, g) = randomR (1,6) (mkStdGen 0)
    (m, _) = randomR (1,6) g
```

A seed of the type **StdGen**
A new seed is generated
by **newStdGen**

https://stackoverflow.com/questions/8416365/generate-a-random-integer-in-a-range-in-haskell

randomR

```
rollDie :: State StdGen Int
rollDie = state $ randomR (1, 6)

rollDie :: State StdGen Int
rollDie = do generator <- get
             let (value, newGenerator) = randomR (1,6) generator
             put newGenerator
             return value
```

```
GHCi> evalState rollDie (mkStdGen 0)
6
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf