

# Monad P2: State Monad Examples (2C)

---

Copyright (c) 2016 - 2019 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto:youngwlim@hotmail.com).

This document was produced by using LibreOffice.

# Based on

---

Haskell in 5 steps

[https://wiki.haskell.org/Haskell\\_in\\_5\\_steps](https://wiki.haskell.org/Haskell_in_5_steps)

---

# Game Example

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Game Example (1)

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.

'ab' = 0          +1-1=0

'ca' = 1          on, +1

'cabca' = 0      on,+1-1,off

State = (game is on/off, current score) = (Bool, Int)

Example use of **State** monad

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Game Example (2)

```
module StateGame where
import Control.Monad.State

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

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

State s a

when input string is null

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Game Example (3)

```
playGame :: String -> State GameState GameValue
```

```
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)
```

when input string is not null

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Game Example (4)

```
main = print $ evalState (playGame "abcaaacbbcabab") startState
```

a	(off, 0)	→	b	(off, 0)	→	c	(on, 0)	→
a	(on, 1)	→	a	(on, 2)	→	a	(on, 3)	→
c	(off, 3)	→	b	(off, 3)	→	b	(off, 3)	→
c	(on, 3)	→	a	(on, 4)	→	b	(on, 3)	→
b	(on, 2)	→	a	(on, 3)	→	b	(on, 2)	

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)



# Game Example Source Code

```
import Control.Monad.State

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)
  case x of
    'a' | on -> return $ score + 1
    'b' | on -> return $ score - 1
    'c'   -> return $ score
    _     -> return score
  playGame xs

startState = (False, 0::Int)
```

```
*Main> runState (playGame "abcaaacbbcabbab") startState
(2,(True,2))
*Main> execState (playGame "abcaaacbbcabbab") startState
(True,2)
*Main> evalState (playGame "abcaaacbbcabbab") startState
2
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

---

# Incrementer Example

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example (1)

- a concrete and simple example of using the State monad
- **non monadic** version of a very simple state example
- the **State** is an **integer**.
- the **value** will always be the **negative** of of the **state**

```
import Control.Monad.State
```

```
type MyState = Int
```

```
type MyStateMonad = State MyState
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example (2)

```
valFromState :: MyState -> Int
```

```
valFromState s = -s
```

```
nextState :: MyState->MyState
```

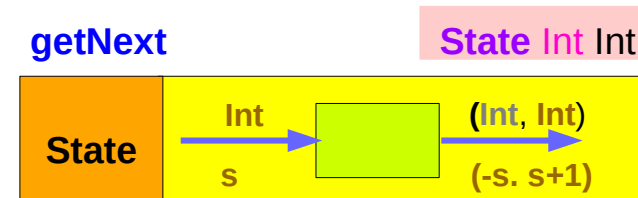
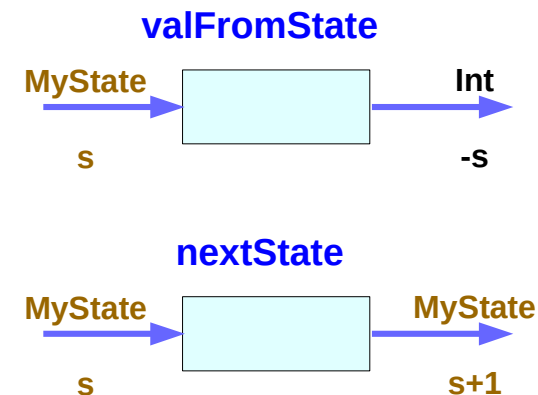
```
nextState x = 1+x
```

```
-- 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'))
```



[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example – (a) inc3

```
-- advance the state three times.
```

```
inc3::MyStateMonad Int
```

```
inc3 = getNext >>= \x ->
```

```
    getNext >>= \y ->
```

```
    getNext >>= \z ->
```

```
    return z
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example – (b) inc3Sugard

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

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example – (c) inc3DiscardedValues

```
-- advance the state three times without inspecting computed values
inc3DiscardedValues::MyStateMonad Int
inc3DiscardedValues = getNext >> getNext >> getNext
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example – (d) inc3DiscardedValuesSugared

```
-- advance the state three times
-- without inspecting computed values with do sugar
inc3DiscardedValuesSugared::MyStateMonad Int
inc3DiscardedValuesSugared = do
    getNext
    getNext
    getNext
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)



# Incrementer Example – (e) inc3AlternateResult

```
-- advance state 3 times, compute the square of the state
inc3AlternateResult::MyStateMonad Int
inc3AlternateResult = do getNext
                      getNext
                      getNext
                      s <- get
                      return (s*s)
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example – (f) inc4

```
-- advance state 3 times, ignoring computed value, and then once more
```

```
inc4::MyStateMonad Int
```

```
inc4 = do
```

```
    inc3AlternateResult
```

```
    getNext
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

# Incrementer Example – (g) main

```
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
```

[https://wiki.haskell.org/State\\_Monad](https://wiki.haskell.org/State_Monad)

---

# Roll Dice Example

[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# randomR type signature

```
import System.Random
```

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

(a, a) : range

(a, g) : output

R in randomR

a : type of a random number

g : type of a seed

g : input seed

cabal update

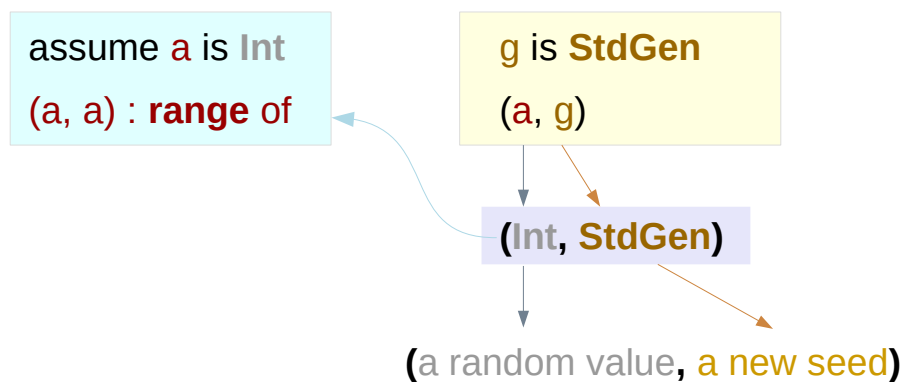
cabal install random

[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# randomR type signature example

`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](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# randomR – updates a seed

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

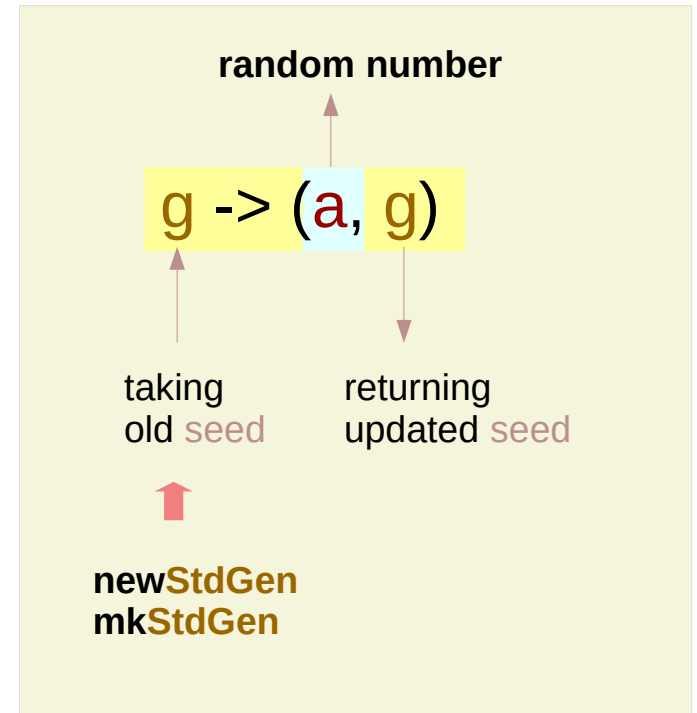
**randomR** updates (generates) a **seed**

**newStdGen** or **mkStdGen** can be used  
to generate an initial input **seed**

the type of **seed** is **StdGen**

the **StdGen** type :

an instance of **RandomGen** and **Random**



[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# randomR – a state processing function

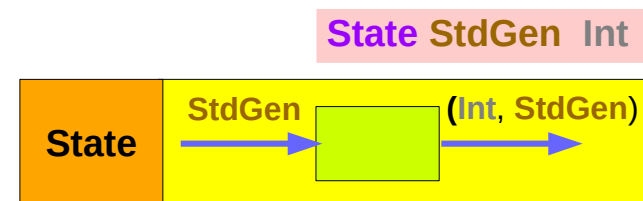
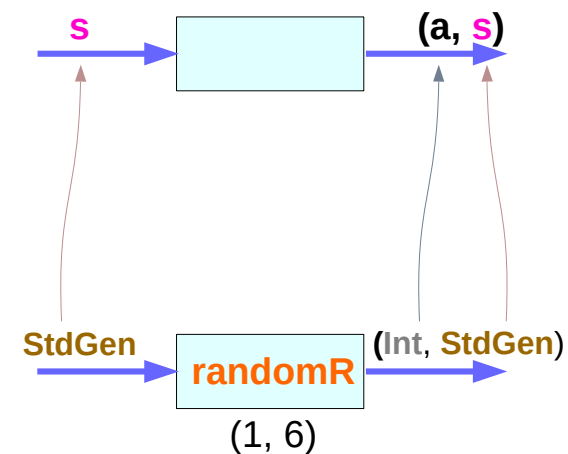
```
import System.Random
```

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

**randomR** is a **state processing function**

```
rollDie :: State StdGen Int
```

```
rollDie = state $ randomR (1, 6)
```



[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)



# randomR – runState, state

```
newtype State s a = State { runState :: s -> (a, s) } (1)
```

```
newtype State s a = state { runState :: s -> (a, s) } (2)
```

```
1) let stst = State { runState = (ly -> (y, y+1)) }
```

```
2) let stst = state (ly -> (y, y+1))
```

```
runState stst (ly -> (y, y+1)) -- no instance error
```

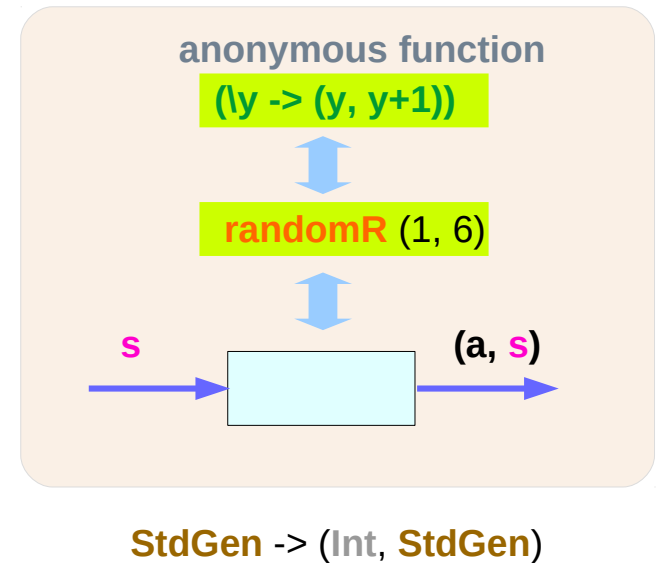
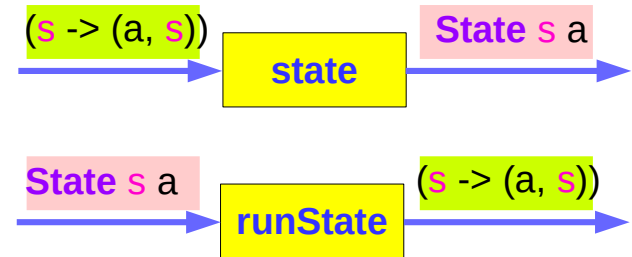
```
runState stst 1 (1, 2)
```

```
rollDie :: State StdGen Int
```

```
rollDie = state $ randomR (1, 6)
```

```
runState rollDie StdGen -> (Int, StdGen)
```

```
runState rollDie (mkStdGen 1) (6, 80028 40692)
```



[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# randomRIO

**randomRIO** returns a random number directly  
in the **IO** monad, without explicitly using a seed of type **StdGen**

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

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

**randomRIO** :: (a, a) -> IO a

**randomIO** :: IO a

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

# Methods of the `random` class

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

```
random :: RandomGen g => g -> (a, g)
```

```
randomRs :: RandomGen g => (a, a) -> g -> [a]
```

```
randoms :: RandomGen g => g -> [a]
```

```
randomRIO :: (a, a) -> IO a
```

```
randomIO :: IO a
```

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

```
randomRIO :: (a, a) -> IO a
```

```
randomIO :: IO a
```

(6, 80028 40692)

6

6

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

# Making a new seed

```
mkStdGen :: Int -> StdGen
```

an alternative way of producing an **initial generator**,  
by mapping an **Int** into a **generator**.  
distinct Int arguments should be likely  
to produce distinct generators.

```
newStdGen :: IO StdGen
```

Applies split to the current **global random generator**,  
updates it with one of the results, and returns the other.

```
mkStdGen 11
```

```
12 1
```

```
newStdGen
```

```
420454863 2147483398
```

actual seed

```
mkStdGen 11
```

```
12 1
```

```
mkStdGen 12
```

```
13 1
```

```
mkStdGen 132
```

```
133 1
```

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

# StdGen Type

```
data StdGen = StdGen !Int32 !Int32      -- 2 32-bit integers
```

```
show :: a -> String
```

```
read :: String -> a
```

**show** converts a data value of two 32-bit number into a **string**

**read** converts a **string** into a data value of two 32-bit number

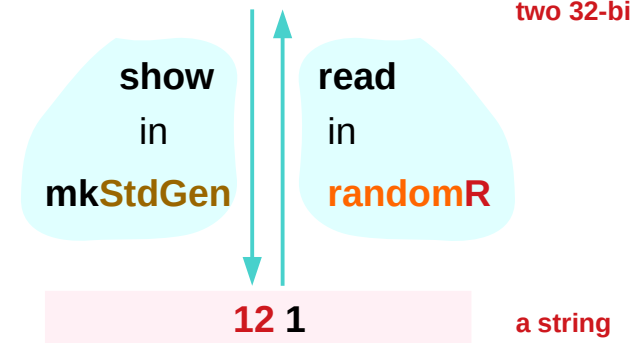
```
read (show g) == g      -- a constraint
```

```
g :: StdGen             -- a seed
```

**reads** may be used to map an arbitrary string (not necessarily one produced by show) onto a **value** of type **StdGen**.

**!** -- a strict declaration

420454863 2147483398 an actual seed  
two 32-bit Int's



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

# Generating a random number (1)

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, **the new seed**)

a random value = 1

**the new seed** =

1012529354 2147442707

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

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

# Generating a random number (2)

```
g = mkStdGen 58
```

```
(x1, y1) = randomR (0,100) g
```

```
x1 → 62
```

random number

```
y1 → 2360826 40692
```

updated seed (two 32bit Int's)

```
g → 59 1
```

string representation

```
randomR (0,100) g → (62,2360826 40692)
```

```
(x1, y1) → (62,2360826 40692)
```

```
x1 → 62
```

```
y1 → 2360826 40692
```

A seed of the type **StdGen**

A seed is generated

by **mkStdGen**

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

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

# Generating a random number (3)

```
randomRIO (1, 6::Int) → 4
```

```
randomRIO (1, 6::Int) → 2
```

```
randomIO :: IO Float → 0.24931645
```

```
randomIO :: IO Float → 0.1068542
```

```
s1 = mkStdGen 55
```

```
(i1, s2) = randomR (1,6::Int) s1
```

```
(i2, _) = randomR (1,6::Int) s2
```

```
i1 → 6
```

```
i2 → 1
```

```
s1 = mkStdGen 55 -- same seed
```

```
(i1, s2) = randomR (1,6::Int) s1
```

```
(i2, _) = randomR (1,6::Int) s2
```

```
i1 → 6
```

```
i2 → 1
```

<https://lotz84.github.io/haskellbyexample/ex/random-numbers>



# Without using IO (1) – use random / randomR

```
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)

( Int      ,      StdGen      )
```

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>

## Without using IO (2)

```
GHCi> let randInt = fst . random $ generator :: Int
```

```
GHCi> randInt
```

```
2092838931
```

```
GHCi> let (randInt, generator') = random generator :: (Int, StdGen)
```

```
GHCi> randInt -- Same value
```

```
2092838931
```

```
GHCi> random generator' :: (Int, StdGen)
```

```
(-2143208520,439883729 1872071452)
```

```
-- Using new generator' returned from "random generator"
```

```
(2092838931,1601120196 1655838864)
```

```
(2092838931,1601120196 1655838864)
```



```
(-2143208520,439883729 1872071452)
```

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

# Dice Example – State

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)**

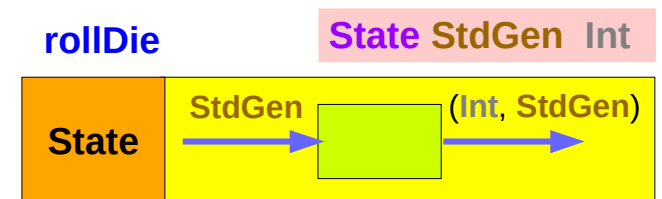
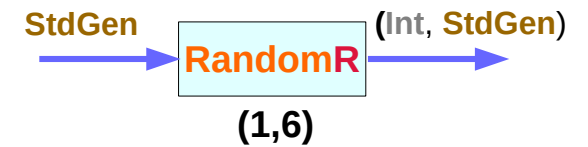
[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# rollDie

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

randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)
randomR (1, 6) :: StdGen -> (Int, StdGen)

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



[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# rollDie with get, put, return

```
rollDie :: State StdGen Int
```

```
rollDie = state $ randomR (1, 6)
```

```
rollDie :: State StdGen Int
```

```
rollDie = do
```

```
  gen <- get
```

```
  let (val, nGen) = randomR (1,6) gen
```

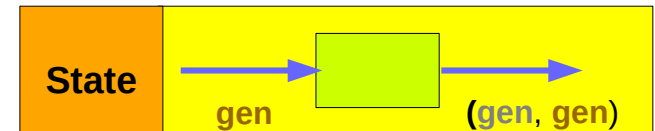
```
  put nGen
```

```
  return value
```

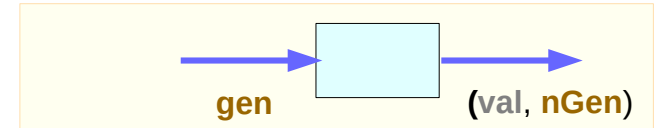
```
GHCi> evalState rollDie (mkStdGen 0)
```

```
6
```

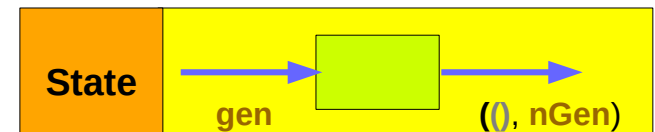
gen <- get



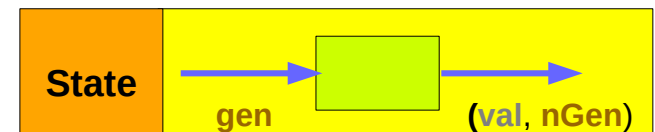
let (val, nGen) = randomR (1,6) gen



put nGen



return value



[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

# Clumsy rollDice 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>

# rollDice (1)

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

```
GHCi> evalState rollDice (mkStdGen 123)
(6,1)
```

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

[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

## rollDice (2)

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

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

That function rolls two dice.

```
rollDice = liftA2 (,) (randomRIO (1,6)) (randomRIO (1,6))
```

[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)



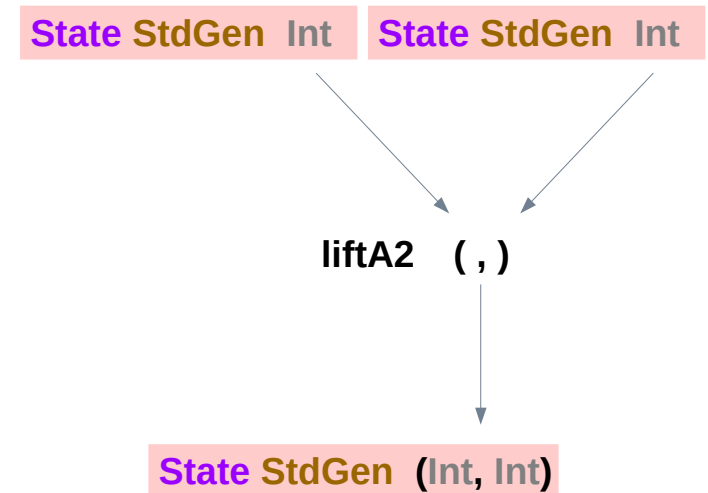
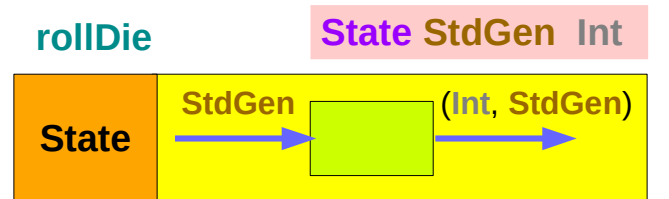
# rollDice (3)

`liftA2` makes the two-argument function `(,)` work within a **monad** or **applicative** functor, `IO`, `rollDie`.

`(,)` is the non-infix version of the tuple constructor.  
the two die rolls will be returned as a **tuple** in `IO`

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

```
rollDice = (,) <$> rollDie <*> rollDie
```



[https://en.wikibooks.org/wiki/Haskell/Understanding\\_monads/State](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>