

State Monad Methods (3G)

Copyright (c) 2016 - 2018 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.

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

Setting the State : put

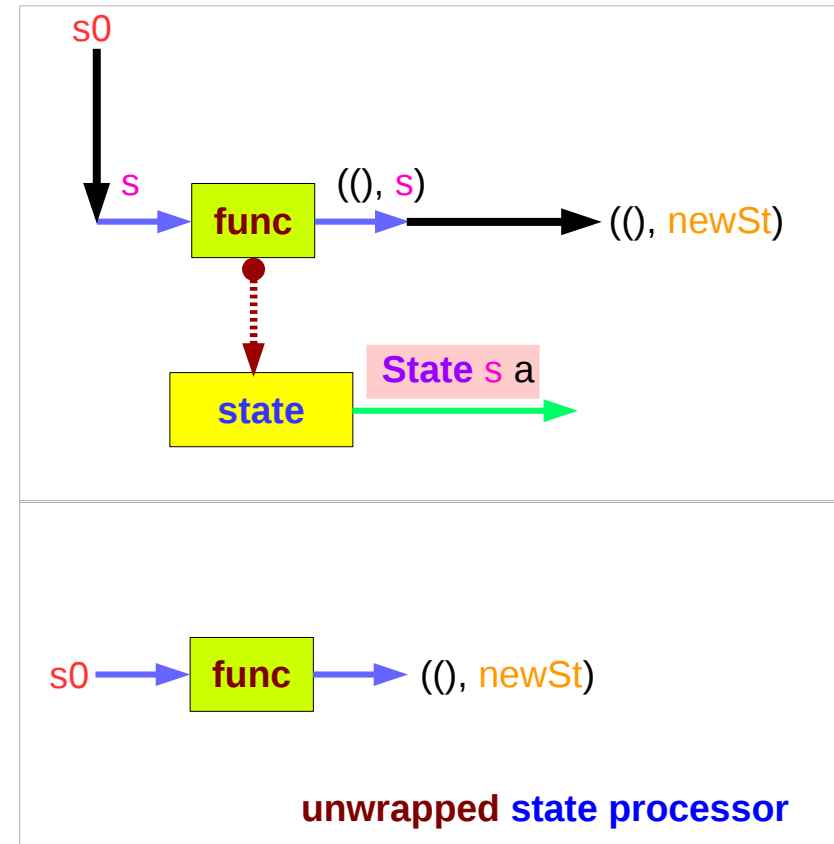
```
put :: s -> State s a  
put newSt = state $ \_ -> ((), newSt)
```

Given a wanted `state newState`,

`put` generates a `state processor`

- ignores whatever the `state` it receives,
- updates the `state` to `newState`
- doesn't care about the `result` of this processor

- all we want to do is to change the `state`
- the tuple will be `((), newState)`
- `()` : the **universal placeholder value**.



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

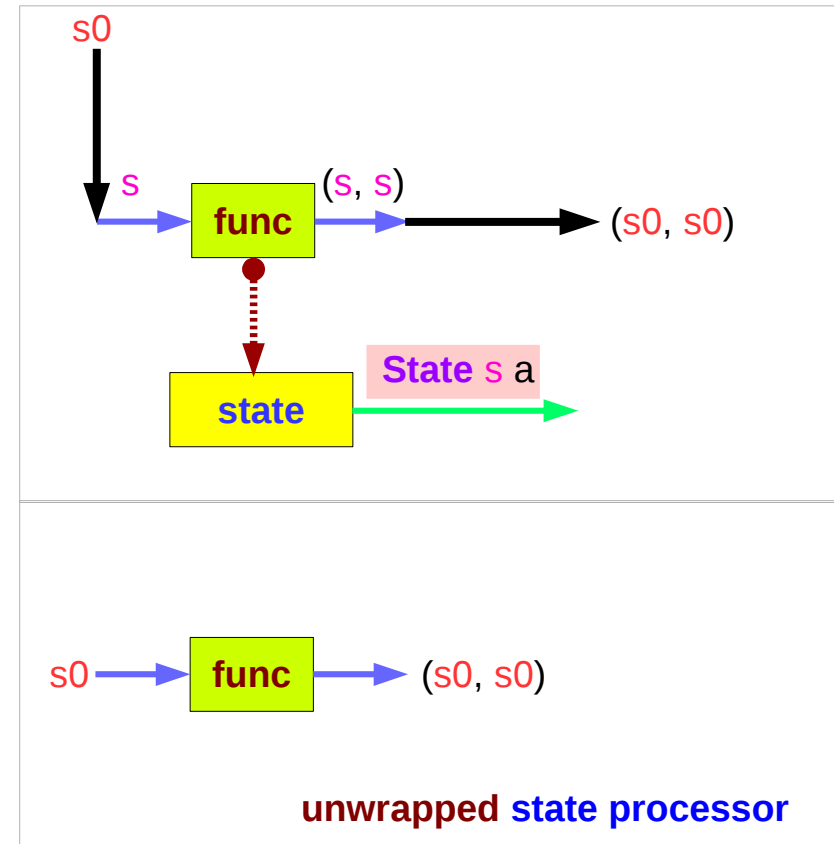
Getting the State : `get`

`get :: State s s`

`get = state $ \s -> (s, s)`

`get` generates a **state processor**

- gives back the **state** `s0`
- as a **result** and as an updated **state** – `(s0, s0)`
- the **state** will remain unchanged
- a copy of the **state** will be made available through the **result** returned



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

Example Codes

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

```
runState get 1
```

```
(1,1)
```

```
runState (return 'X') 1
```

```
('X',1)
```

```
runState get 1
```

```
(1,1)
```

```
runState (put 5) 1
```

```
((),5)
```

```
let postincrement = do { x <- get; put (x+1); return x }
```

```
runState postincrement 1
```

```
(1,2)
```

```
let predecrement = do { x <- get; put (x-1); get }
```

```
runState predecrement 1
```

```
(0,0)
```

```
runState (modify (+1)) 1
```

```
((),2)
```

```
runState (gets (+1)) 1
```

```
(2,1)
```

```
evalState (gets (+1)) 1
```

```
2
```

```
execState (gets (+1)) 1
```

```
1
```

https://wiki.haskell.org/State_Monad

Setting and Getting the State

```
put :: s -> State s a
```

```
put s :: State s a
```

```
put newState = state $ \_ -> ((), newState)
```

-- setting a state to `newState`

-- regardless of the old state

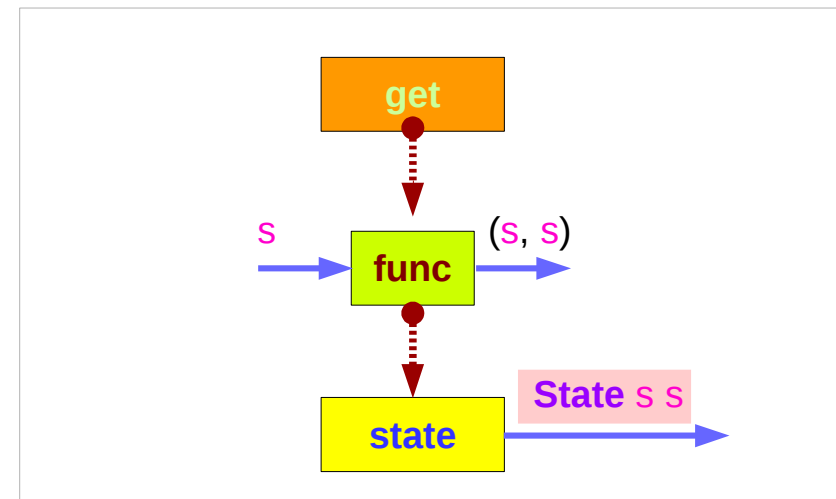
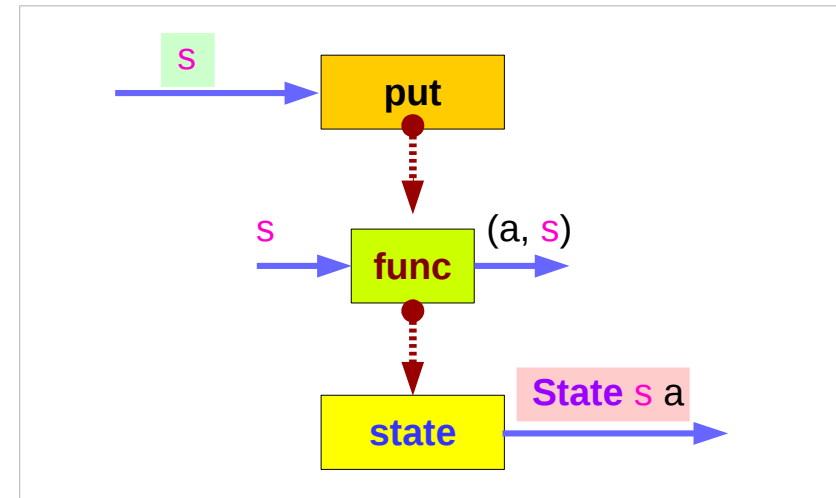
-- setting the result to `()`

```
get :: State s s
```

```
get = state $ \s -> (s, s)
```

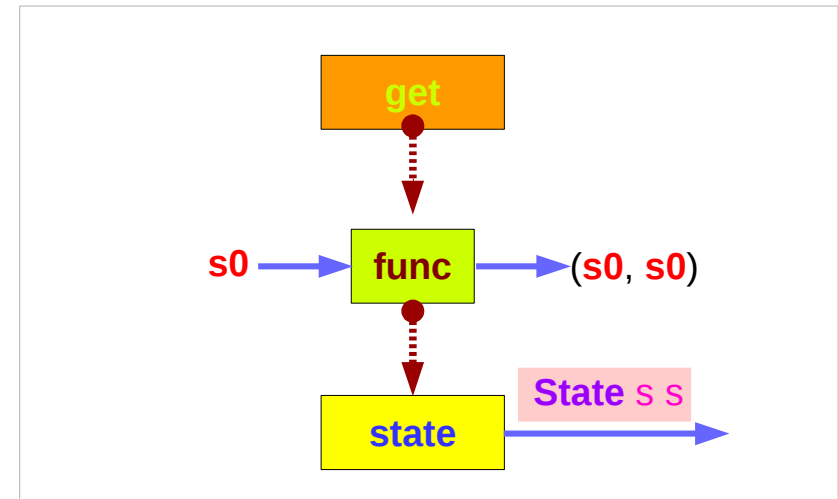
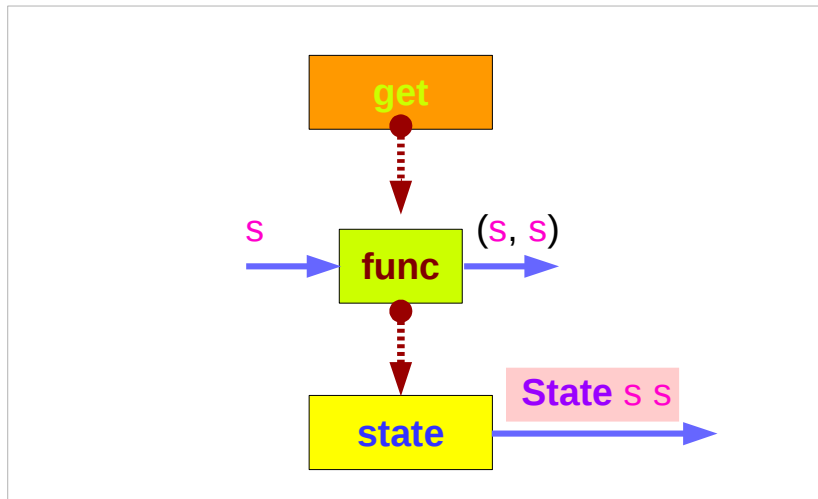
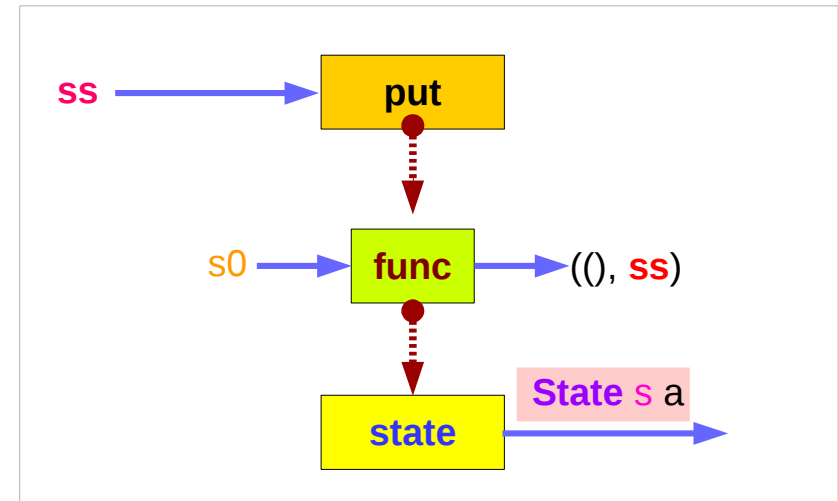
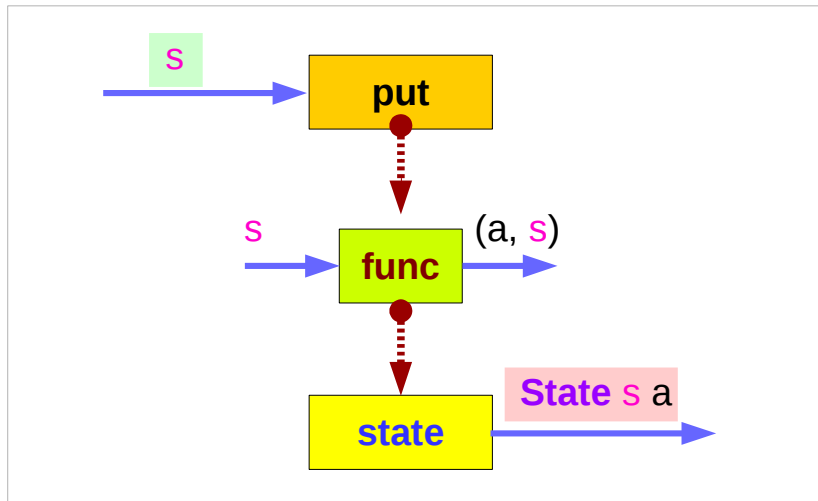
-- getting the current state `s`

-- also setting the result to `s`



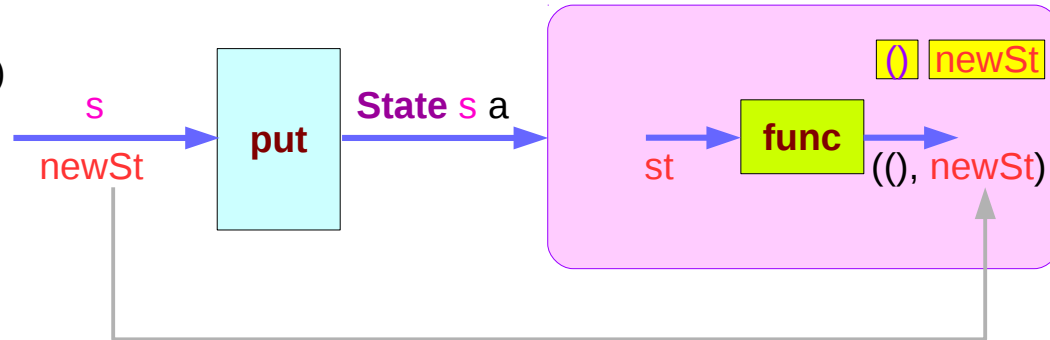
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Types and Values of **put** and **get**

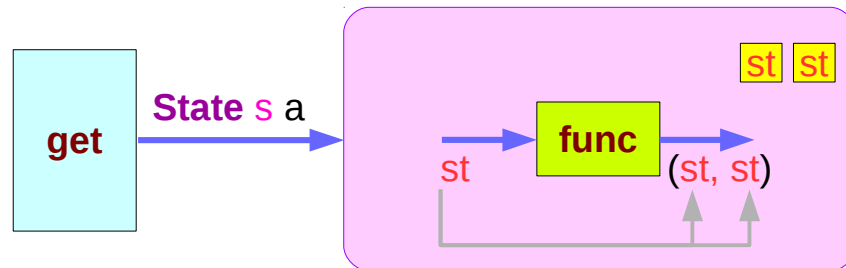


wrapped version of put and get

```
put :: s -> State s a
put s :: State s a
put newSt = state $ \_ -> ((), newSt)
```



```
get :: State s s
get = state $ \s -> (s, s)
```

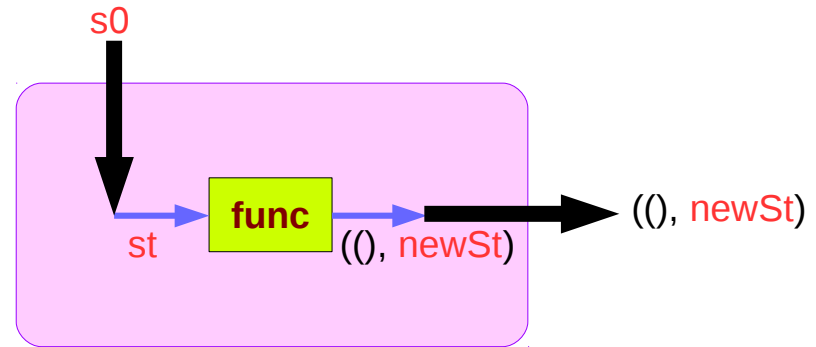


https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Executing the state processor

```
put :: s -> State s a
put newSt = state $ \_ -> ((), newSt)

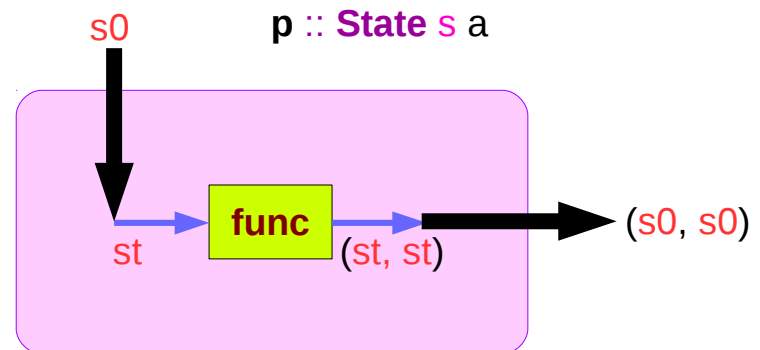
runState (put newSt) s0 -> ((), newSt)
```



applying the function

```
get :: State s s
get = state $ \s -> (s, s)

runState (get) s0 -> (s0, s0)
```



applying the function

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

State Monad Examples – put

```
runState (put 5) 1
```

```
((),5)
```

put

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

Comments:

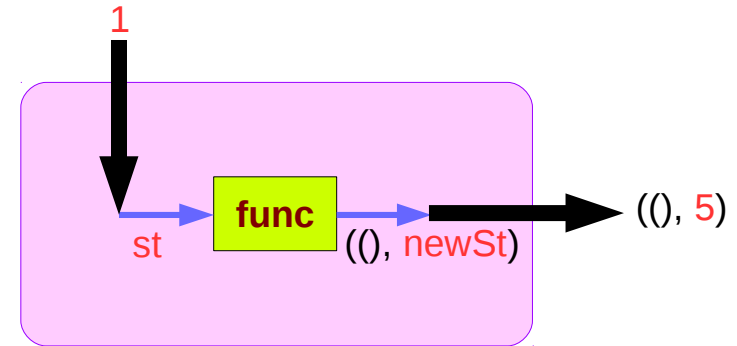
```
put 5 :: State Int ()
```

```
runState (put 5) :: Int -> ((),Int)
```

```
initial state = 1 :: Int
```

```
final value = () :: ()
```

```
final state = 5 :: Int
```



```
put :: s -> State s a
```

```
put newState = state $ \_ -> ((), newState)
```

https://wiki.haskell.org/State_Monad

State Monad Examples – get

```
runState get 1
```

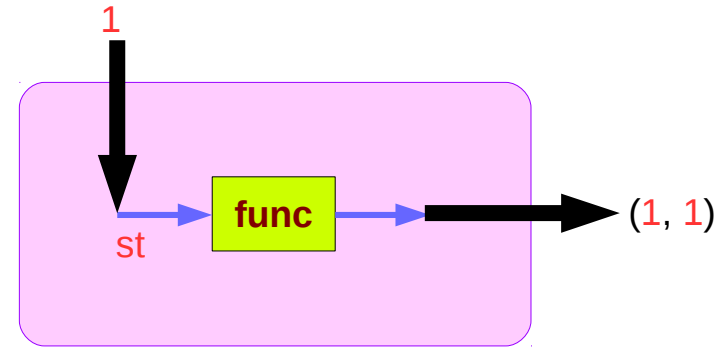
```
(1,1)
```

```
get
```

```
set the result value to the state and leave the state unchanged.
```

Comments:

```
get :: State Int Int
runState get :: Int -> (Int, Int)
initial state = 1 :: Int
final value = 1 :: Int
final state = 1 :: Int
```



```
get :: State s s
get = state $ \s -> (s, s)
```

https://wiki.haskell.org/State_Monad

Think an unwrapped state processor

```
(return 5) → 1 -> (5,1) -- a way of thinking  
get         → 1 -> (1,1) -- a way of thinking  
(put 5)    → 1 -> ((),5) -- a way of thinking
```

Think an **unwrapped** state processor

a value of type (**State s a**) is
a **function** from **initial state s**
to **final value a** and **final state s**: (a,s).

these are usually wrapped,
but shown here unwrapped for simplicity.

```
(return 5) → state(1 -> (5,1)) -- an actual way  
get         → state(1 -> (1,1)) -- an actual way  
(put 5)    → state(1 -> ((),5)) -- an actual way
```

wrapping the state processor

https://wiki.haskell.org/State_Monad

State Monad Examples – return, get, and put

Return leaves the state unchanged and sets the result:

-- ie: `(return 5)` → `1 -> (5,1)` -- a way of thinking

`runState (return 5) 1` → `(5,1)`

Get leaves state unchanged and sets the result to the state:

-- ie: `get` → `1 -> (1,1)` -- a way of thinking

`runState get 1` → `(1,1)`

Put sets the result to () and sets the state:

-- ie: `(put 5)` → `1 -> ((),5)` -- a way of thinking

`runState (put 5) 1` → `((),5)`

https://wiki.haskell.org/State_Monad

State Monad Examples – modify and gets

```
modify :: (s -> s) -> State s ()  
modify f = do { x <- get; put (f x) }
```

```
gets :: (s -> a) -> State s a  
gets f = do { x <- get; return (f x) }
```

```
runState (modify (+1)) 1      (+1) 1 → 2 :: s  
→ ((),2)
```

```
runState (gets (+1)) 1      (+1) 1 → 2 :: a  
→ (2,1)
```

```
evalState (gets (+1)) 1     → :: s state  
→ 2
```

```
execState (gets (+1)) 1     → :: a result  
→ 1
```

https://wiki.haskell.org/State_Monad

```
x <- get; put (f x)  
x <- get; return (f x)
```

- inside a monad instance
- unwrapped implementations of **modify** and **gets**

Unwrapped Implementation – return, get, and put

Return leaves the state unchanged and sets the result:

-- ie: `(return 5)` → `1 -> (5,1)` -- a way of thinking

`return` :: `a -> State s a`

`return x s = (x,s)`

Get leaves state unchanged and sets the result to the state:

-- ie: `get` → `1 -> (1,1)` -- a way of thinking

`get` :: `State s s`

`get s = (s,s)`

Put sets the result to () and sets the state:

-- ie: `(put 5)` → `1 -> ((),5)` -- a way of thinking

`put` :: `s -> State s ()`

`put x s = ((),x)`

https://wiki.haskell.org/State_Monad

`(x,s)`

`(s,s)`

`((),x)`

- inside a monad instance
- unwrapped implementations of `return`, `get`, and `put`

Default Implementations

```
class Monad m => MonadState s m | m -> s where
```

```
-- | Return the state from the internals of the monad.
```

```
get :: m s
```

```
get = state (\s -> (s, s))
```

```
-- | Replace the state inside the monad.
```

```
put :: s -> m ()
```

```
put s = state (\_ -> ((), s))
```

```
-- | Embed a simple state action into the monad.
```

```
state :: (s -> (a, s)) -> m a
```

```
state f = do
```

```
  s <- get
```

```
  let ~(a, s') = f s
```

```
  put s'
```

```
  return a
```

<https://stackoverflow.com/questions/23149318/get-put-and-state-in-monadstate>

Default Implementations

The definitions of `get`, `put`, `state` in the class declaration are the default implementations, which are meant to be overridden in actual instances of the class.

In this way, the dead loop does not happen:

- **put** and **get** in terms of **state**
- **state** in terms of **put** and **get**

<https://stackoverflow.com/questions/23149318/get-put-and-state-in-monadstate>

Default Implementations

MonadState s m is a typeclass constraint, not a type.

get :: MonadState s m => m s

this signature says that for some monad **m**
storing some state of type **s**,
get is an action in **m**
that returns a value of type **s**.

This is abstract

The more concrete (less-overloaded) version of State

get :: State s s

put :: s -> State s ()

<https://stackoverflow.com/questions/25438575/states-put-and-get-functions>

Default Implementations

`execState get 0` \Rightarrow 0

We can set the value of the counter using `put`:

`execState (put 1) 0` \Rightarrow 1

We can set the state multiple times:

`execState (do put 1; put 2) 0` \Rightarrow 2

And we can modify the state based on its current value:

`execState (do x <- get; put (x + 1)) 0` \Rightarrow 1

This combination of `get` and `put` is common enough to have its own name, `modify`:

`execState (do modify (+ 1)) 0` \Rightarrow 1

`execState (do modify (+ 2); modify (* 5)) 0` \Rightarrow 10

<https://stackoverflow.com/questions/25438575/states-put-and-get-functions>

Default Implementations

MonadState is the class of types that are monads with state.

State is an instance of that class:

instance MonadState s (State s) where

get = Control.Monad.Trans.State.get

put = Control.Monad.Trans.State.put

So are StateT (the state monad transformer, which adds state to another monad) and various others.

This overloading was introduced so that if you're using a stack of monad transformers, you don't need to explicitly lift operations between different transformers.

If you're not doing that, you can use the simpler operations from transformers.

<https://stackoverflow.com/questions/25438575/states-put-and-get-functions>

The Result of a Stateful Computation

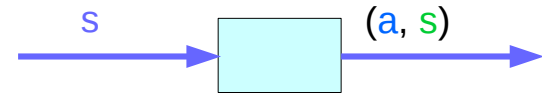
a **stateful computation** is a **function** that takes some **state** and returns a **value** along with some **new state**.

That function would have the following type:

$s \rightarrow (a, s)$

s is the type of the **state** and a the **result** of the **stateful computation**.

$s \rightarrow (a, s)$



a function is an executable data when executed, a result is produced **action, execution, result**

$s \rightarrow (a, s)$
↓

<http://learnyouahaskell.com/for-a-few-monads-more>

Stateful Computations Inside the State Monad

inside a monad,
when `sc` is a **stateful computation**
then the result of the stateful computation `sc`
can be assigned to `x`

```
x <- sc
```

`s` \rightarrow `(a, s)`
↓
the result type

`sc` :: `State s a`

`x` :: `a` (the execution result of `sc`)

<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

get inside the State Monad

inside the State monad,

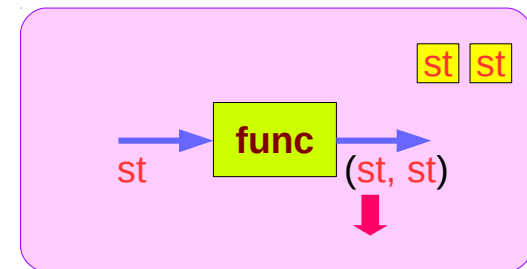
get returns the current monad instance
whose type is State s a

```
x <- get
```

the stateful computation is performed
over the current monad instance returned by get

the result of the stateful computation is st::s
thus x will get the st

$x :: a$ the execution result of get



<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Getting the current state inside the State Monad

inside the State monad,

get returns the current monad instance
whose type is State s a

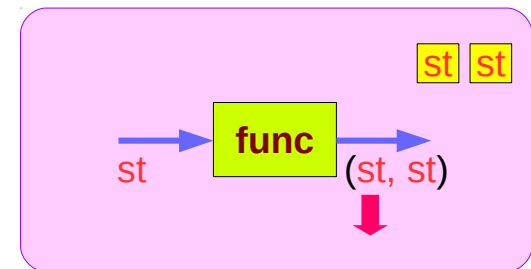
to get the current state st, do

```
s <- get
```

s will have the value of the current state st

this is like **evalState** is called with the current monad instance

- **get**
- **current monad instance**
- **stateful computation**
- **result :: s**



<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

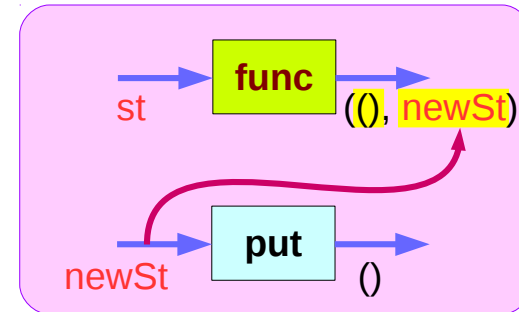
put and get inside State Monad

```
put :: s -> State s a  
put newSt = state $ \_ -> ((), newSt)
```

```
put :: s -> ()
```

the result type :: ()

stateful computation of put

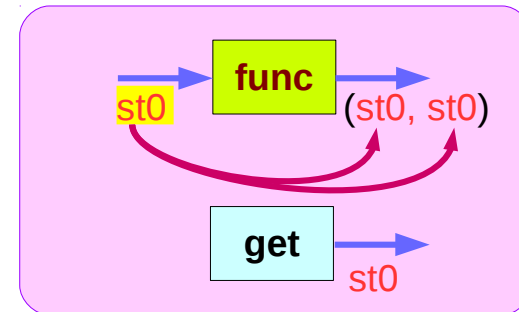


```
get :: State s s  
get = state $ \s -> (s, s)
```

```
get :: s
```

the result type :: s

stateful computation of get



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

Inside Functions and runState Functions

Most monads are equipped with some "run" functions such as `runState`, `execState`, and so forth.

But, frequent calling such functions inside the monad shows that the functionality of the monad does not fully exploited

```
s0 <- get
let (a,s1) = runState p s0
put s1
```

```
-- read the state of the current instance
-- pass the state to p, get new state
-- save new state
```

```
let p = state (\y -> (y, y+1))
```

```
a <- p
```

```
-- the stateful computation p updates the state to s1
-- the result of the state returned is assigned to a
```

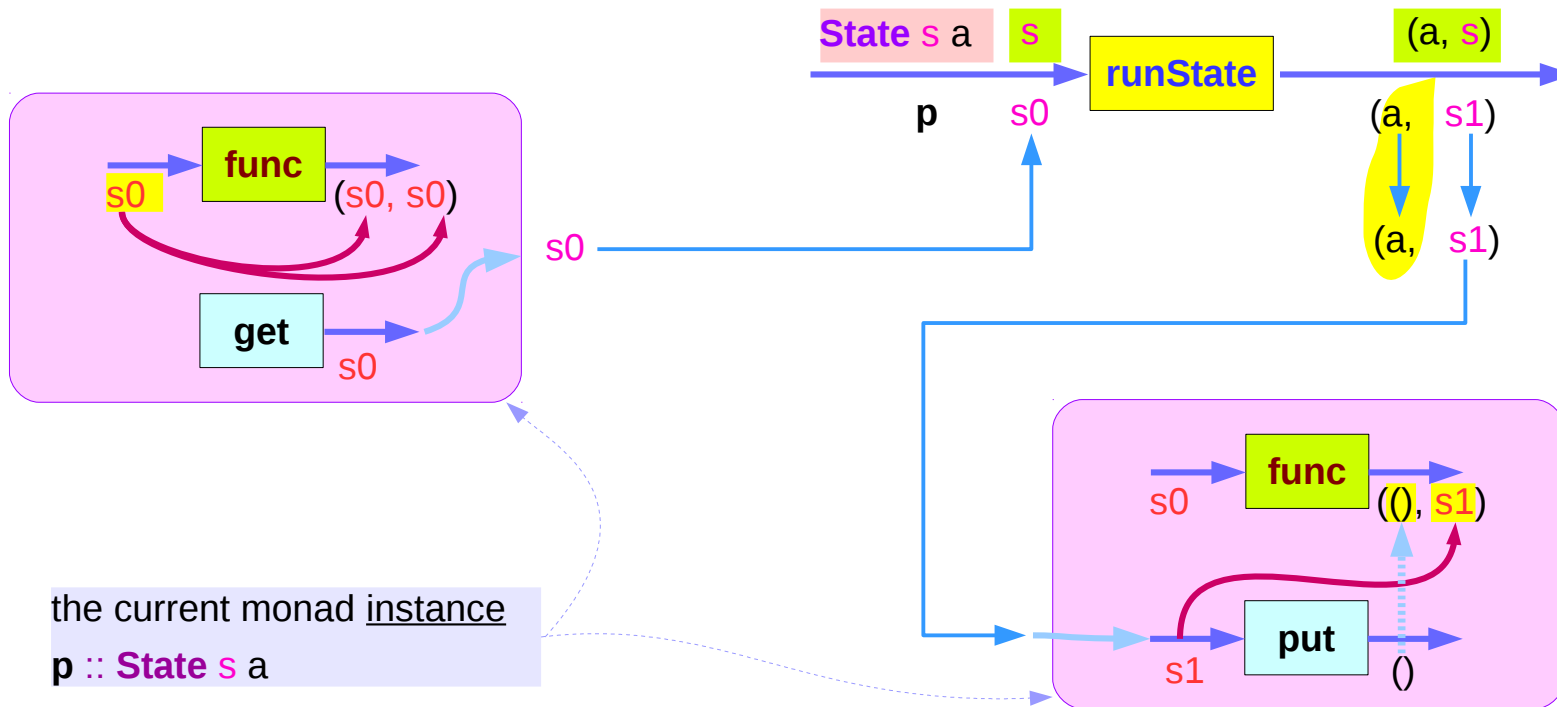
<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Redundant computation examples (1)

```
s0 <- get  
let (a,s1) = runState p s0  
put s1
```

the same binding variable a
the same state $s1$

```
a <- p
```



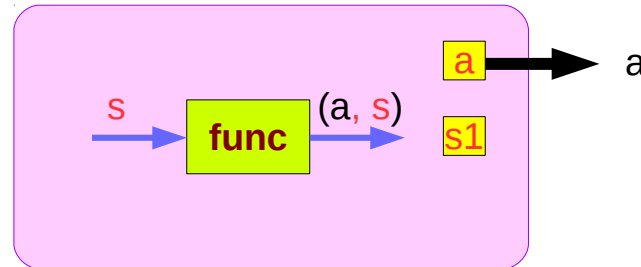
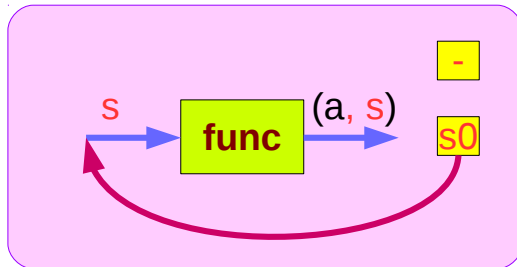
<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Redundant computation examples (2)

```
a <- p
```

-- the stateful computation **p** updates the state to **s1**
-- the result of the state returned is assigned to **a**

p :: State s a



<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Example of collecting returned values (1)

Inferred Function Type

```
collectUntil :: Monad State t m => (t -> Bool) -> m a -> m [a]
```

`m` → `State t`

Specific Function Type

```
collectUntil :: (t -> Bool) -> State t a -> State t [a]
```

<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Example of collecting returned values (2)

```
collectUntil f comp = do
```

```
  st <- get
```

```
  if f st then return [ ]
```

```
  else do
```

```
    x <- comp
```

```
    xs <- collectUntil f comp
```

```
    return (x:xs)
```

```
-- Get the current state
```

```
-- If it satisfies predicate, return
```

```
-- Otherwise...
```

```
-- Perform the computation s
```

```
-- Perform the rest of the computation
```

```
-- Collect the results and return them
```

```
comp :: State s a
```

```
st  :: s
```

```
x   :: a
```

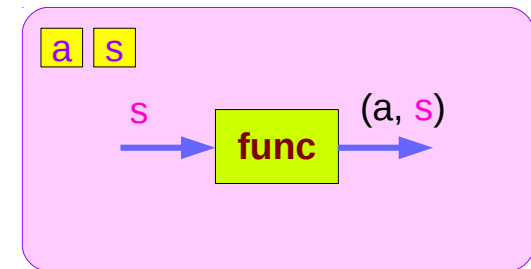
```
xs  :: [a]
```

```
simpleState = state (\x -> (x,x+1))
```

```
*Main> evalState (collectUntil (>10) simpleState) 0
```

```
[0,1,2,3,4,5,6,7,8,9,10]
```

```
simpleState :: State s a
```



<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Example of collecting returned values (3)

```
collectUntil f comp = do
  st <- get
  if f st then return []
  else do
    x <- comp -- stateful computation
    xs <- collectUntil f comp
    return (x:xs)
```

```
simpleState = state (\x -> (x,x+1))
```

```
*Main> evalState (collectUntil (>10) simpleState) 0
[0,1,2,3,4,5,6,7,8,9,10]
```

get	st ← 0	comp :	0 → (0, 1)	x ← 0
get	st ← 1	comp :	1 → (1, 2)	x ← 1
get	st ← 2	comp :	2 → (2, 3)	x ← 2
get	st ← 3	comp :	3 → (3, 4)	x ← 3
get	st ← 4	comp :	4 → (4, 5)	x ← 4
get	st ← 5	comp :	5 → (5, 6)	x ← 5
get	st ← 6	comp :	6 → (6, 7)	x ← 6
get	st ← 7	comp :	7 → (7, 8)	x ← 7
get	st ← 8	comp :	8 → (8, 9)	x ← 8
get	st ← 9	comp :	9 → (9, 10)	x ← 9
get	st ← 10	comp :	10 → (10, 11)	x ← 10

<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Example of collecting returned values (5)

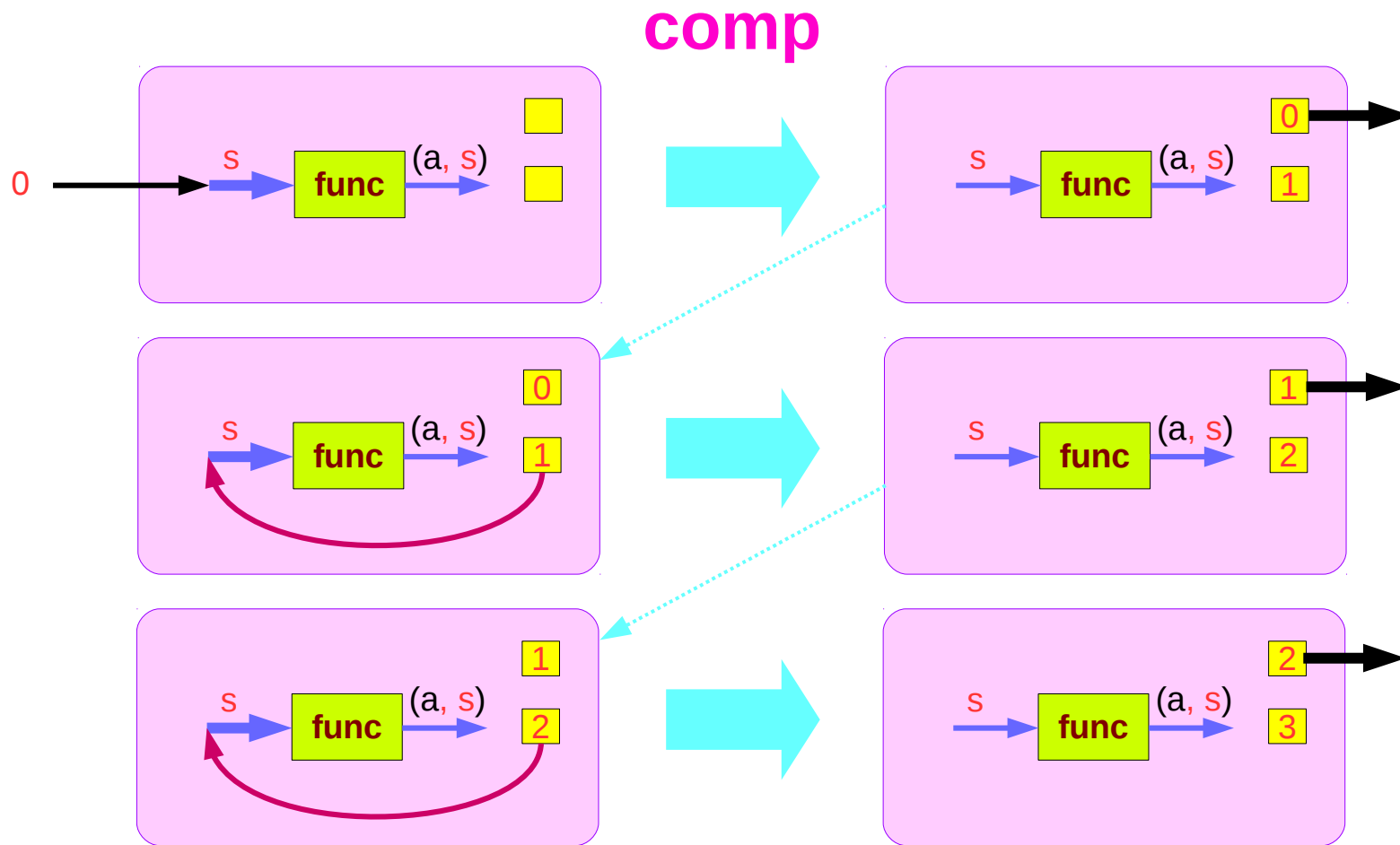
```
collectUntil f comp = do
  st <- get
  if f st then return [ ] ----- return State t [a] type
  else do
    x <- comp -- stateful computation
    xs <- collectUntil f comp
    return (x:xs) ----- return State t [a] type
```

nesting do statements is possible
if they are within the same monad

enables **branching** within one do block,
as long as both branches of the **if statement**
results in the same monadic type.

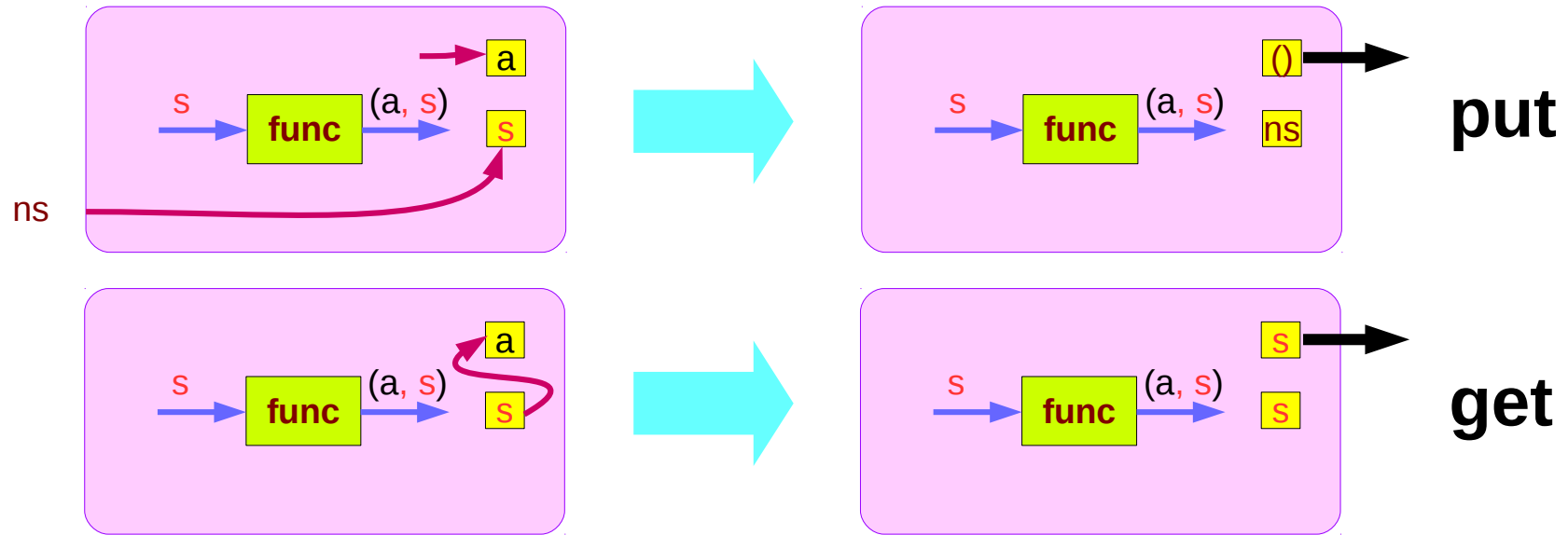
<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Stateful Computation of **comp**



<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Stateful Computations of `put` & `get`



<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Another example of collecting returned values (1)

```
collectUntil :: (s -> Bool) -> State s a -> State s [a]
collectUntil f s = step
  where
    step = do a <- s
            liftM (a :) continue
    continue = do s' <- get
                if f s' then return [] else step
```

```
simpleState = state (\x -> (x,x+1))
```

```
*Main> evalState (collectUntil (>10) simpleState) 0
[0,1,2,3,4,5,6,7,8,9,10]
```

<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Another example of collecting returned values (2)

```
collectUntil :: (s -> Bool) -> State s a -> State s [a]
collectUntil f s = step
  where
    step = do a <- s
            liftM (a :) continue
    continue = do s' <- get
                if f s' then return [] else step
```

Since a is part of the result
in both branches of the 'if'

```
collectUntil f comp = do
  st <- get
  if f st then return [ ]
  else do
    x <- comp -- stateful computation
    xs <- collectUntil f comp
    return (x:xs)
```

<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

liftM and mapM

```
liftM    :: (Monad m) => (a -> b)  -> m a -> m b
mapM     :: (Monad m) => (a -> m b) -> [a]  -> m [b]
```

liftM lifts a function of type `a -> b` to a monadic counterpart.

mapM applies a function which yields a monadic value to a list of values,
yielding list of results embedded in the monad.

```
> liftM (map toUpper) getLine
```

```
Hallo
```

```
"HALLO"
```

```
> :t mapM return "monad"
```

```
mapM return "monad" :: (Monad m) => m [Char]
```

<https://stackoverflow.com/questions/5856709/what-is-the-difference-between-liftm-and-mapm-in-haskell>

References

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