

State Monad – Methods (6B)

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

put changes the current state

`put :: s -> State s a`

`put ns = state $ _ -> ((), ns)`

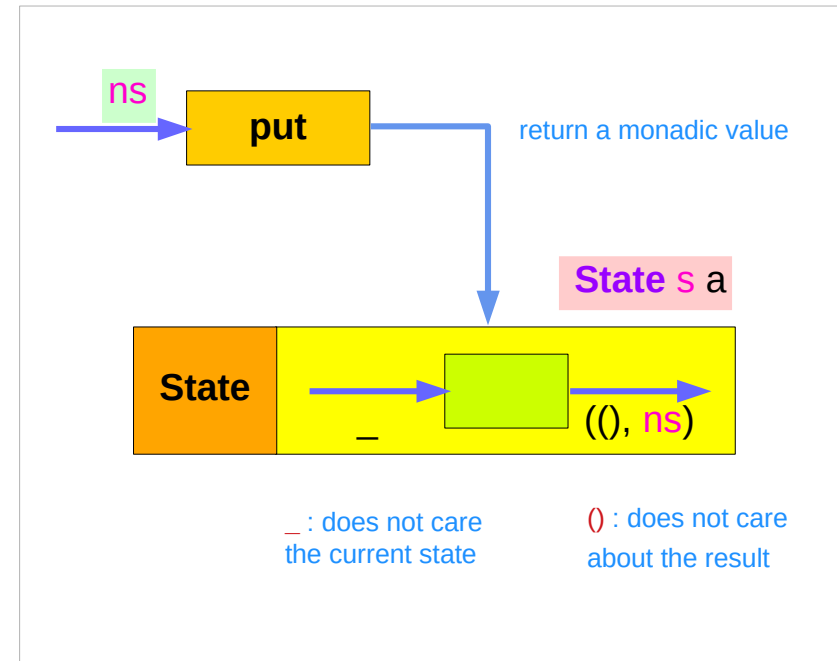
Given a wanted **state new State (ns)**,

put generates a state processor

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

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

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



get gives the current state :

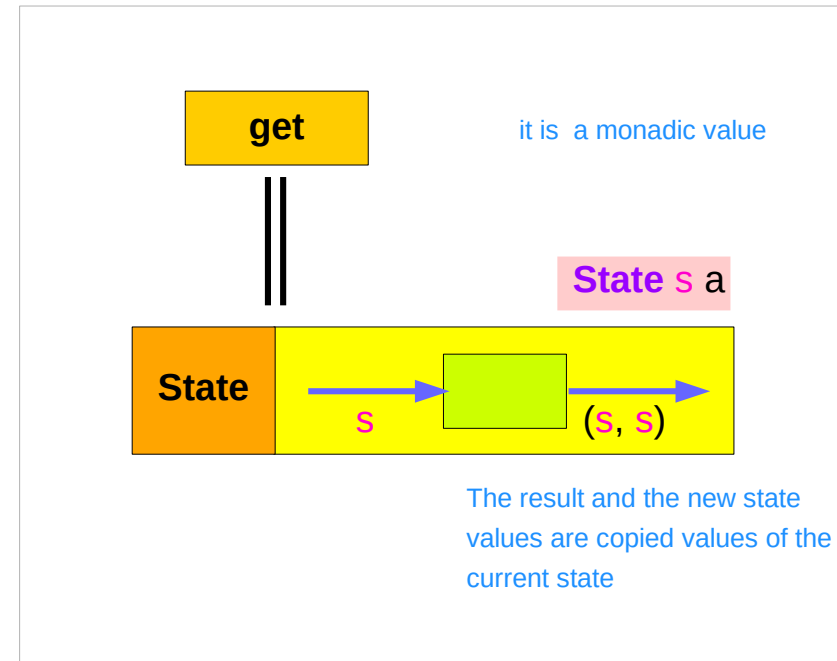
`get :: State s s`

`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

put returns a monadic value

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

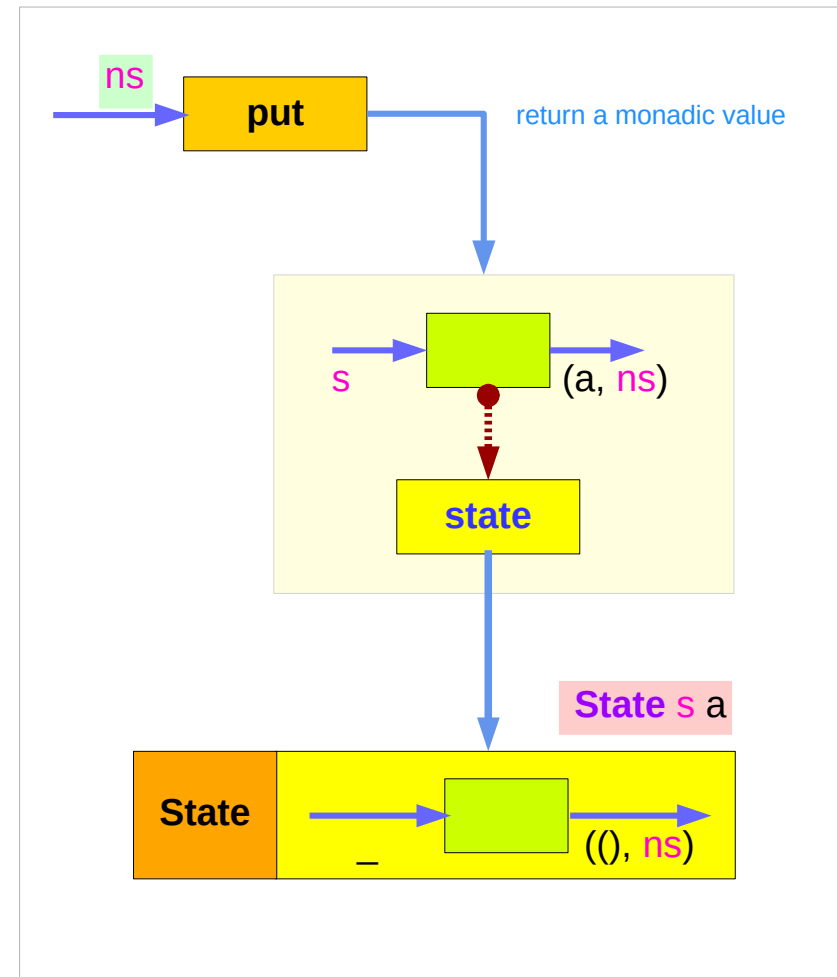
```
put s :: State s a
```

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

```
-- setting a state to newState
```

```
-- regardless of the old state
```

```
-- setting the result to ()
```



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

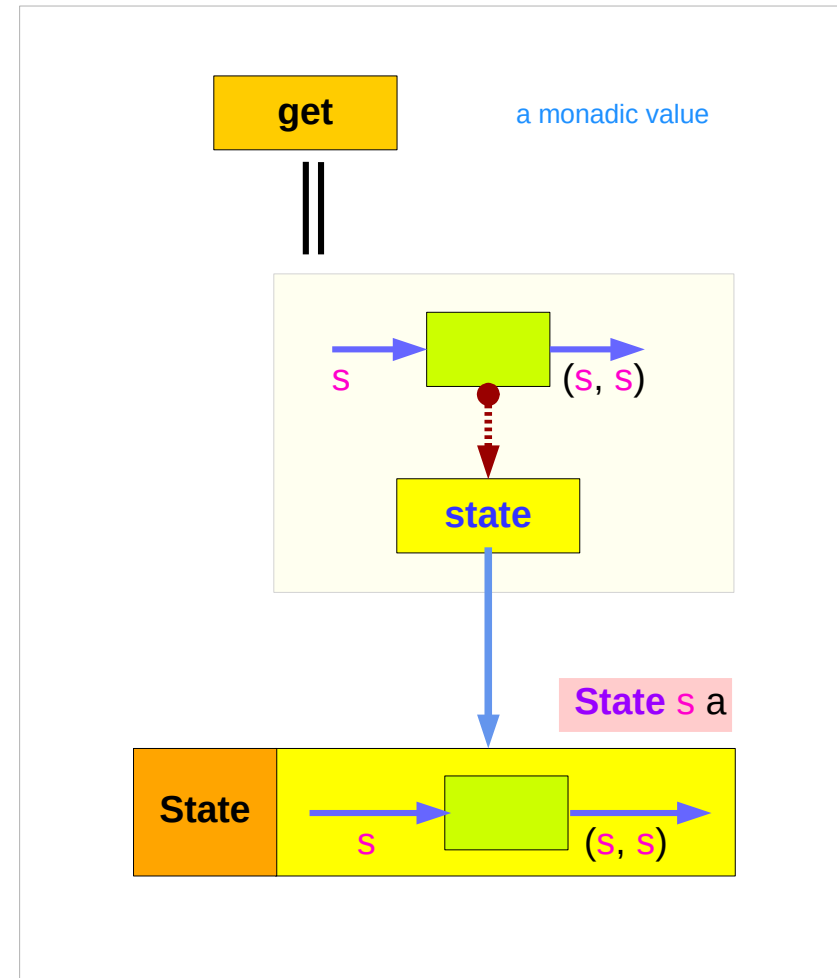
get is a monadic value

`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

Running put

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

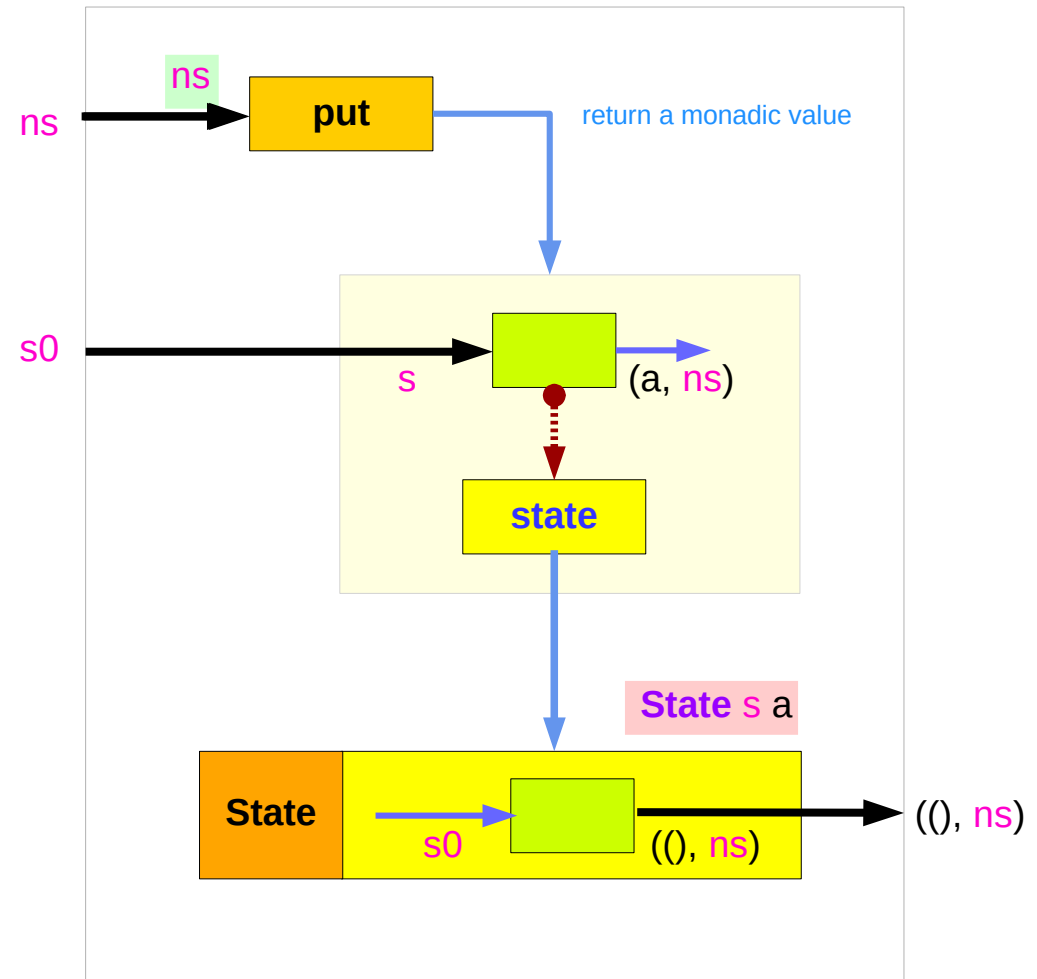
```
put s :: State s a
```

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

```
runState (put ns) s0
```

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

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



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

Running get

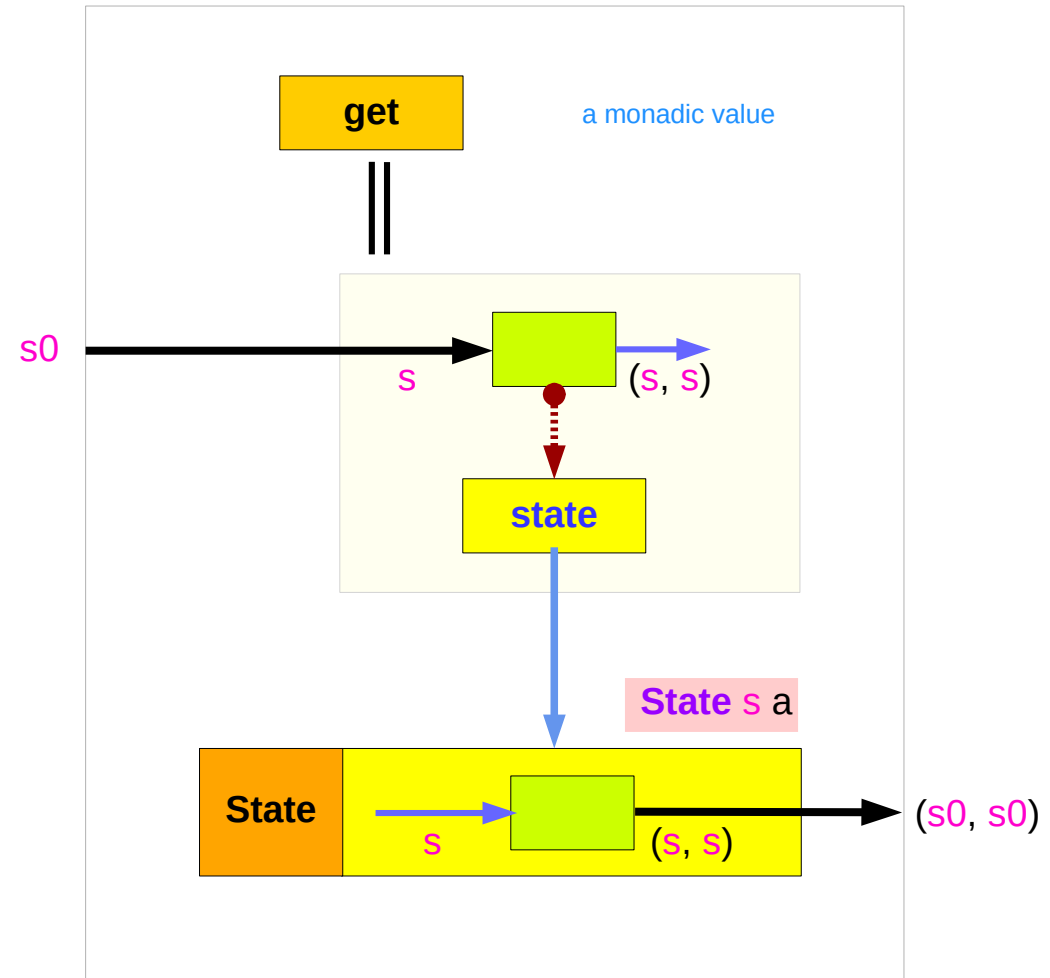
`get :: State s s`

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

`runState (get) s0`

`runState (get) 1`

`(1,1)`



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

Example codes (1)

```
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)
```

```
runState (put 1 >> get >> put 2 >> get ) 0
```

```
(2,2)
```

```
runState (get >>= \n -> put (n+1) >> return n) 0
```

```
(0,1)
```

```
inc = get >>= \n -> put (n+1) >> return n
```

```
runState inc 0
```

```
(0,1)
```

```
runState (inc >> inc) 0
```

```
(1,2)
```

```
runState (inc >> inc >> inc) 0
```

```
(2,3)
```

https://wiki.haskell.org/State_Monad

Example codes (2)

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

```
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
```

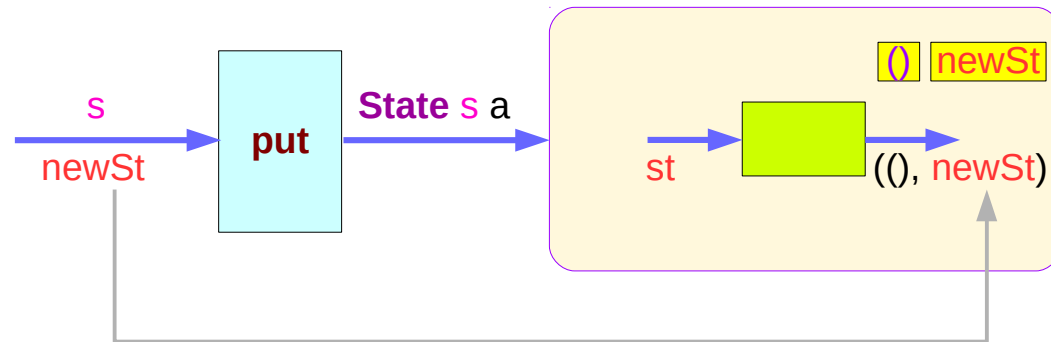
```
execState (gets (+1)) 1
```

```
1
```

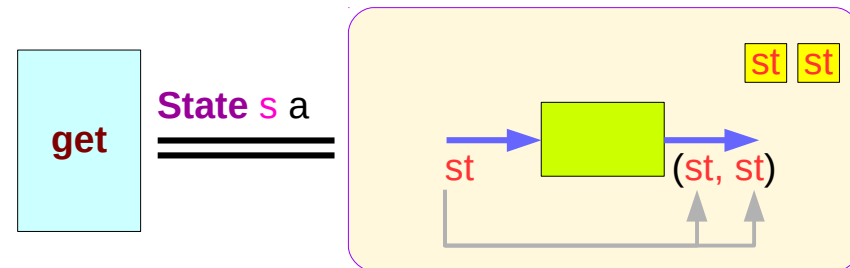
https://wiki.haskell.org/State_Monad

Simple representation 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)
```

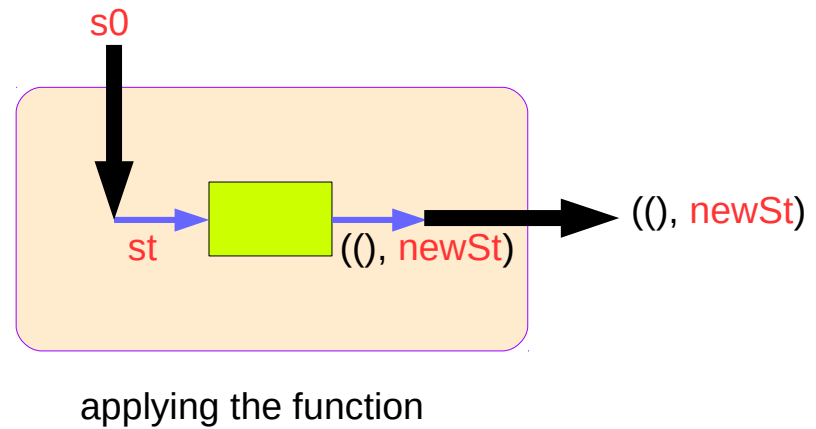


https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Executing the state processor

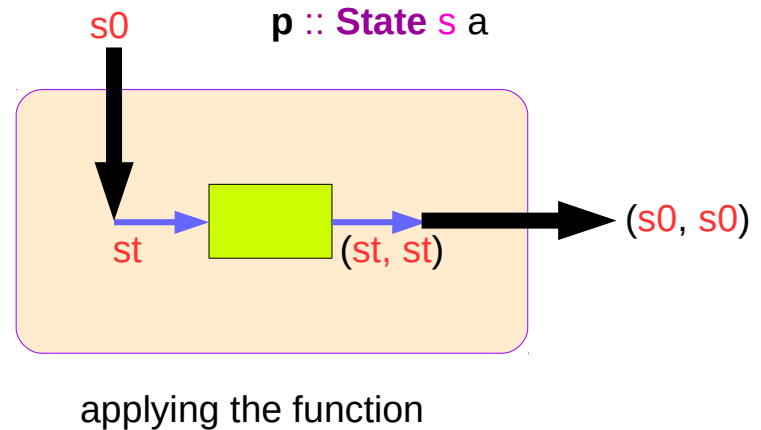
```
put :: s -> State s a
put newSt = state $ \_ -> ((), newSt)

runState (put newSt) s0  →  ((), newSt)
```



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

runState (get) s0  →  (s0, s0)
```



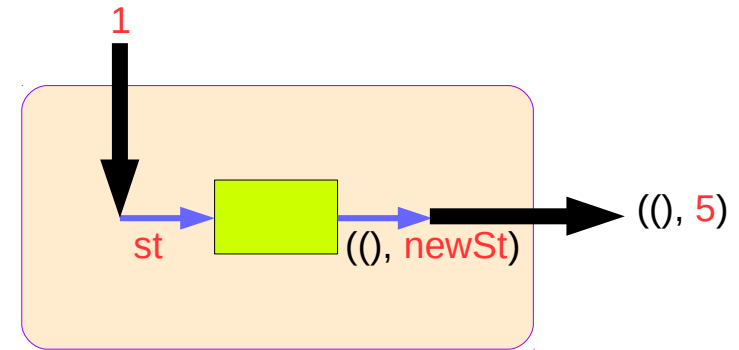
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

State Monad Examples – put

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

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

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



```
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)
```

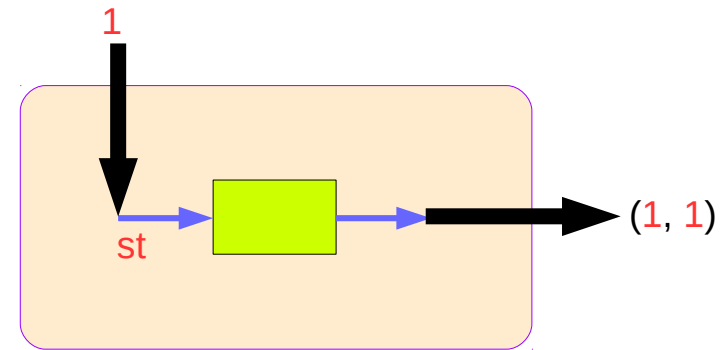
https://wiki.haskell.org/State_Monad

State Monad Examples – get

```
runState get 1
```

```
(1,1)
```

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



```
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) \rightarrow `1 -> (5,1)` -- a way of thinking
get \rightarrow `1 -> (1,1)` -- a way of thinking
(put 5) \rightarrow `1 -> ((),5)` -- a way of thinking

Think an **unwrapped** state processor

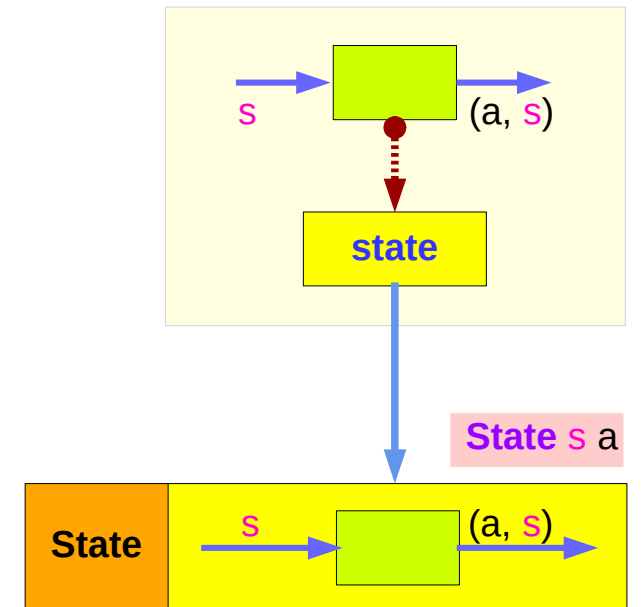
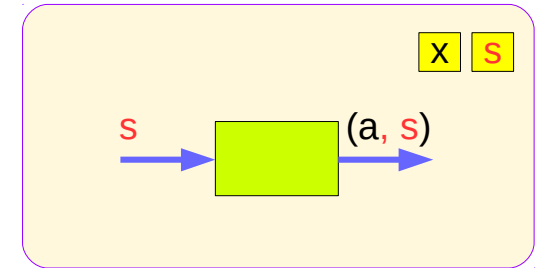
a value of type (`State s a`) is
a **function** from **initial state** `s`
to **final value** `a` and **final state** `s`: `(a,s)`.

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

(return 5) \rightarrow `state (1 -> (5,1))` -- an actual way
get \rightarrow `state (1 -> (1,1))` -- an actual way
(put 5) \rightarrow `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:

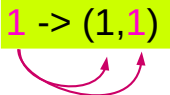
-- ie: (return 5)  1 -> (5,1) -- a way of thinking



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

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

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

```
evalState (modify (+1)) 1 → ()
      → s :: state      fst ((), 2)
```

```
execState (modify (+1)) 1 → 2
      → a :: result     snd ((), 2)
```

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

```
execState (gets (+1)) 1 → 1
      → a :: result     snd (2, 1)
```

modify state (–, f x)

get state (f x, s)

evalState (a, s)

execState (a, s)

(eval, exec)

(get, modify)

https://wiki.haskell.org/State_Monad

Unwrapped Implementation Examples

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

```
return x s = (x,s)
```

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

```
get s = (s,s)
```

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

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

```
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) }
```

(x,s)

(s,s)

((),x)

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

x <- **get**; **put** (f x) - state

x <- **get**; **return** (f x) - result

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

https://wiki.haskell.org/State_Monad

State Monad Examples – put, get, modify

`execState get 0` → 0

set the value of the counter using put:

`execState (put 1) 0` → 1

set the state multiple times:

`execState (do put 1; put 2) 0` → 2

modify the state based on its current value:

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

`execState (do modify (+ 1)) 0` → 1

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

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

A Stateful Computation

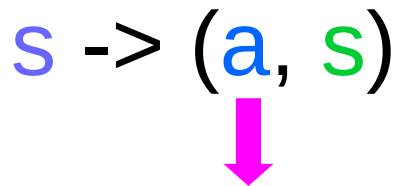
a **stateful computation** is a **function** that takes some **state** and returns a **value** along with some **new state**.

That function would have the following type:

$s \rightarrow (a, s)$

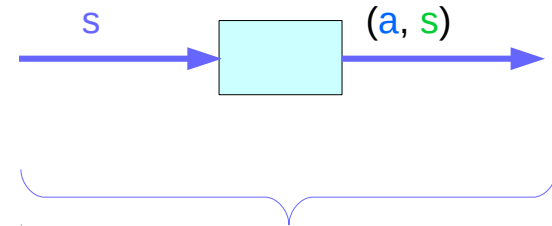
s is the type of the **state** and **a** the **result** of the **stateful computation**.

$s \rightarrow (a, s)$



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

$s \rightarrow (a, s)$



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

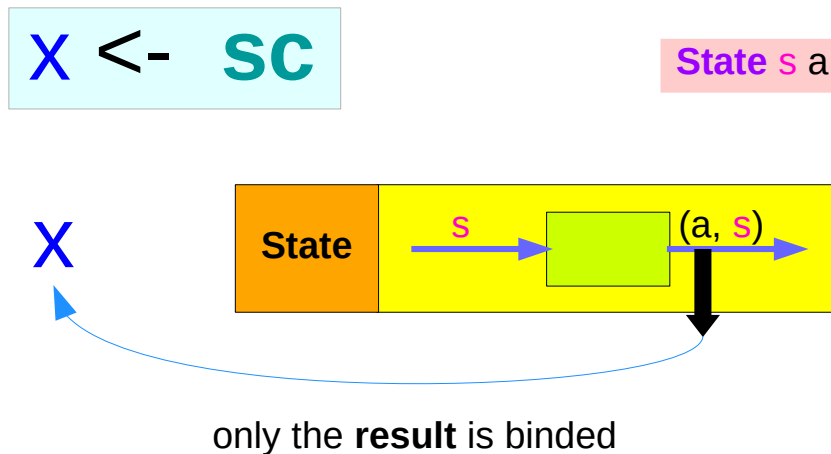
Stateful Computations inside the State Monad

inside a monad,

when **sc** is a **stateful computation**

then the **result** of the stateful computation **sc**

can be assigned to **x**



x <- sc

sc :: State s a

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

~~**x :: State s a**~~

s -> (a, s)

the **result** type

<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 with the type of **State s a**

```
x <- get
```

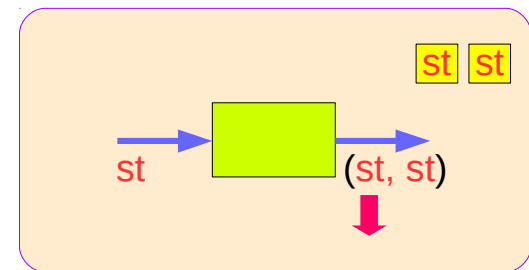
the **stateful computation** is performed
over the current monad instance returned by **get**

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

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

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

x :: a the execution result of **get**



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

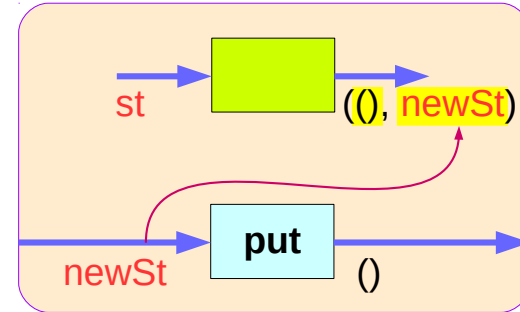
put and get inside State Monad

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

```
put :: s -> ()
the result type :: ()
```

-- a way of thinking

stateful computation of put



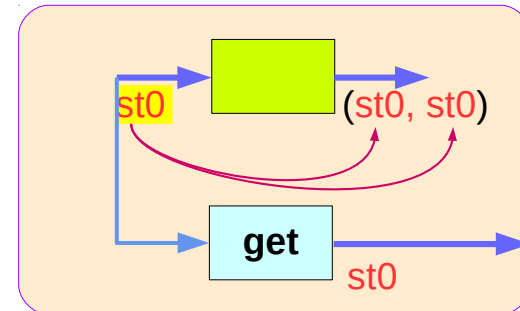
```
get :: State s s
get = state $ \s -> (s, s)
```

```
get :: s
the result type :: s
```

-- a way of thinking

-- no such a function

stateful computation of get



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

run functions inside a Monad

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 -- read the state of the current instance
let (a,s1) = runState p s0 -- pass the state to p, get new state
put s1 -- save new state
return a
```

||

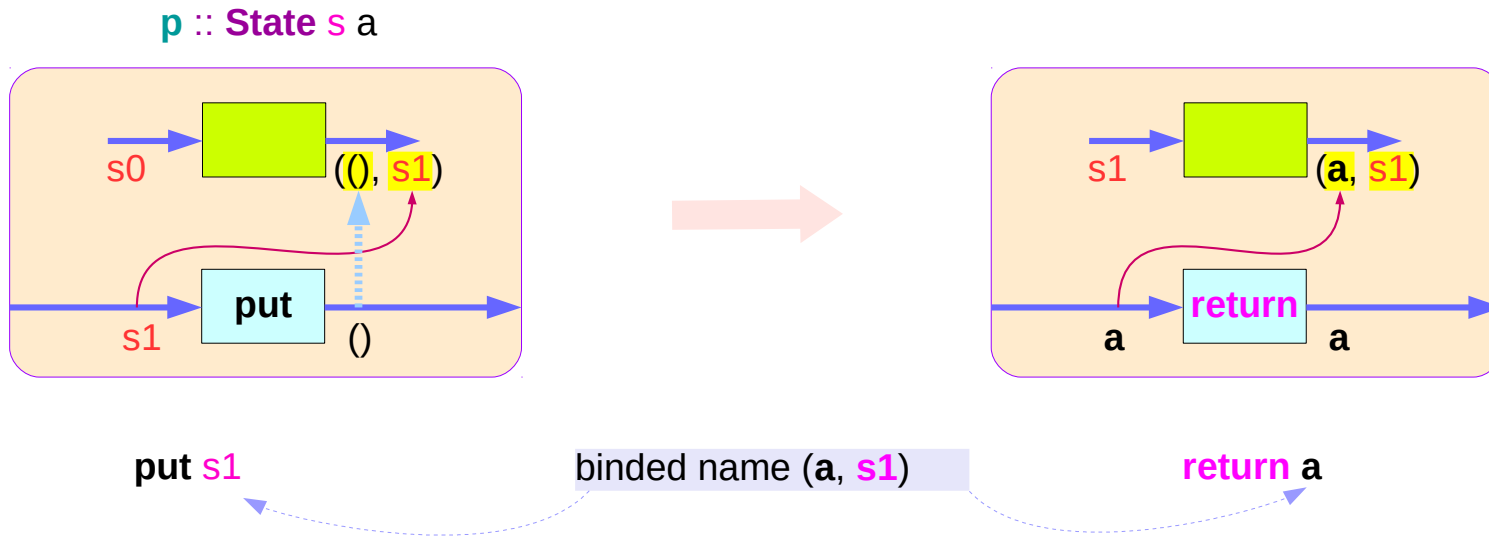
```
a <- p -- the stateful computation p updates the state to s1
-- the result of the state returned is assigned to a
```

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

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

Redundant computation examples (2)

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



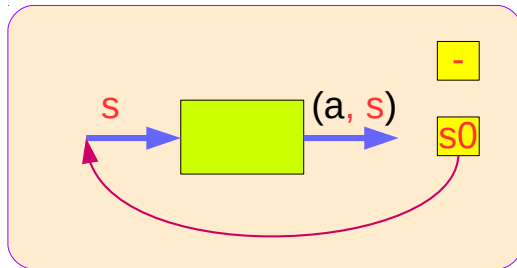
<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Redundant computation examples (3)

```
a <- p
```

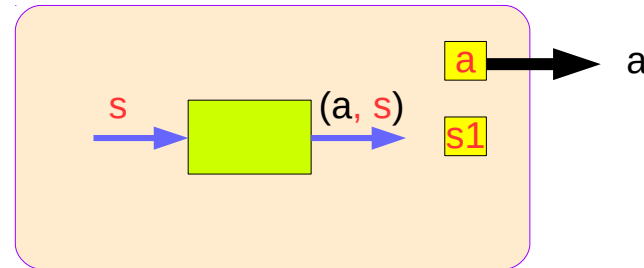
-- the stateful computation **p** updates the state to **s1**
-- the result of the state returned is assigned to **a**

p :: State s a



stateful computation **p**

runState p s0



return the result **a**

return a

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

Counter Example

```
import Control.Monad.State.Lazy

tick :: State Int Int
tick = do n <- get
         put (n+1)
         return n

plusOne :: Int -> Int
plusOne n = execState tick n

plus :: Int -> Int -> Int
plus n x = execState (sequence $ replicate n tick) x
```

A function to increment a counter.

tick :

- a monadic value itself
- ~~a function returning a monadic value~~

Add one to the given number using the state monad:

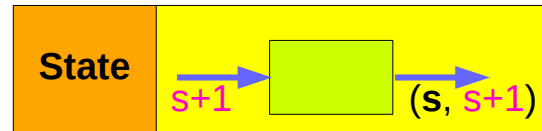
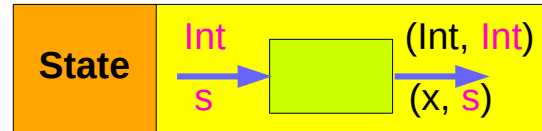
A contrived addition example. Works only with positive numbers:

<https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html>

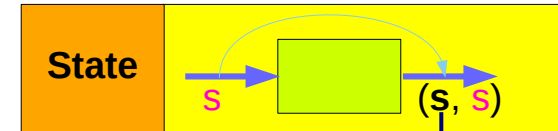
Counter Example – tick

```
tick :: State Int Int
tick = do n <- get
         put (n+1)
         return n
```

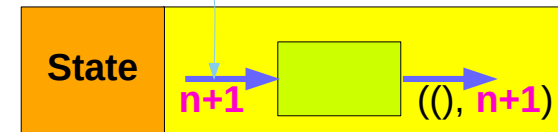
tick :: State Int Int



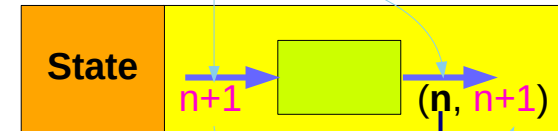
get



put (n+1)



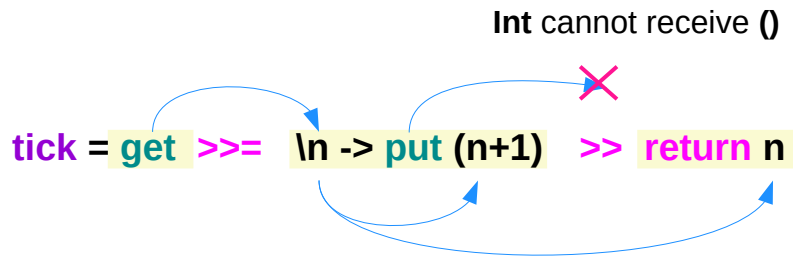
return n



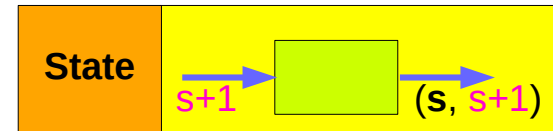
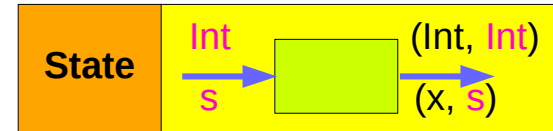
<https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html>

Counter Example – tick without **do**

```
tick :: State Int Int
tick = do n <- get
         put (n+1)
         return n
```



```
tick :: State Int Int
```

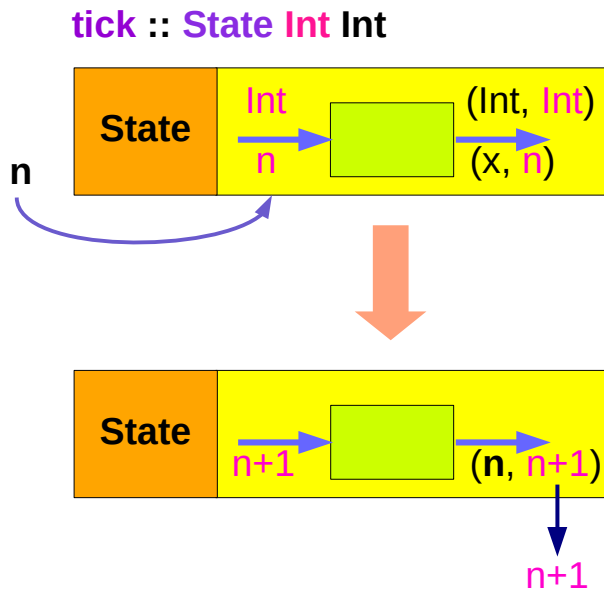


<https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html>

Counter Example – incrementing

```
tick :: State Int Int
tick = do n <- get
         put (n+1)
         return n
```

```
plusOne :: Int -> Int
plusOne n = execState tick n
```



<https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html>

Counter Example – using sequence

```
plus :: Int -> Int -> Int
```

```
plus n x = execState (sequence $ replicate n tick) x
```

```
sequence $ [tick1, tick2, ..., tickn]
```

```
runState (sequence $ [tick, tick]) 3 → ([3,4],5)
```

```
runState (sequence $ [tick, tick, tick]) 3 → ([3,4,5],6)
```

```
execState (sequence $ [tick, tick, tick]) 3 → 6
```

```
evalState (sequence $ [tick, tick, tick]) 3 → [3,4,5]
```

<https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html>

replicate

```
replicate :: Int -> a -> [a]
```

replicate **n** **x** is a list of length **n** with **x** the value of every element.

```
replicate 3 5
```

```
[5,5,5]
```

```
replicate 5 "aa"
```

```
["aa","aa","aa","aa","aa"]
```

```
replicate 5 'a'
```

```
"aaaaa"
```

http://zvon.org/other/haskell/Outputprelude/replicate_f.html

sequence

```
sequence :: Monad m => [m a] -> m [a]
```

evaluate **each action** in the sequence from left to right,
and collect the **results**.

```
runState (sequence [get, return 3, return 4 ]) 1
```

```
([1,3,4],1)
```

```
runState get 1      (1,1)      result: 1
```

```
runState (return 3) 1 (3,1)      result: 3
```

```
runState (return 4) 1 (4,1)      result: 4
```

<http://derekwyatt.org/2012/01/25/haskell-sequence-over-functions-explained/>

Example of collecting returned values

```
collectUntil f comp = do
  st <- get           -- Get the current state
  if f st then return [] -- If it satisfies predicate, return
  else do            -- Otherwise...
    x <- comp        -- Perform the computation s
    xs <- collectUntil f comp -- Perform the rest of the computation
    return (x:xs)   -- Collect the results and return them
```

`comp` :: State s a

`st` :: s

`x` :: a

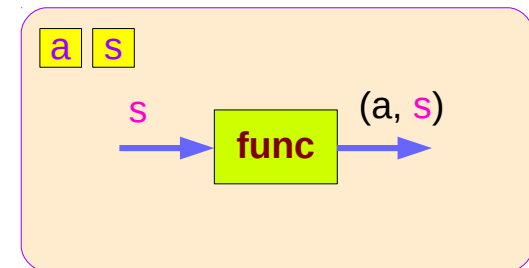
`xs` :: [a]

```
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 – stateful computations

```
collectUntil f comp = do
  st <- get
  if f st then return []
  else do
    x <- comp
    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

stateful computation

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

Example of collecting – the return type

```
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.

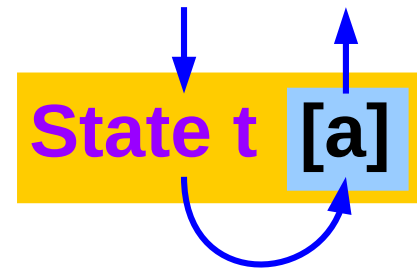
<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

Example of collecting – another stateful computation

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

```
return :: State t [a] type
collectUntil f comp :: State t [a] type
xs <- collectUntil f comp -- stateful computation
xs :: [a]
```

$t \rightarrow ([a], t)$
↓
the result type



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

Example of collecting – the function type

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]
```

```
(>10) :: (t -> Bool)
```

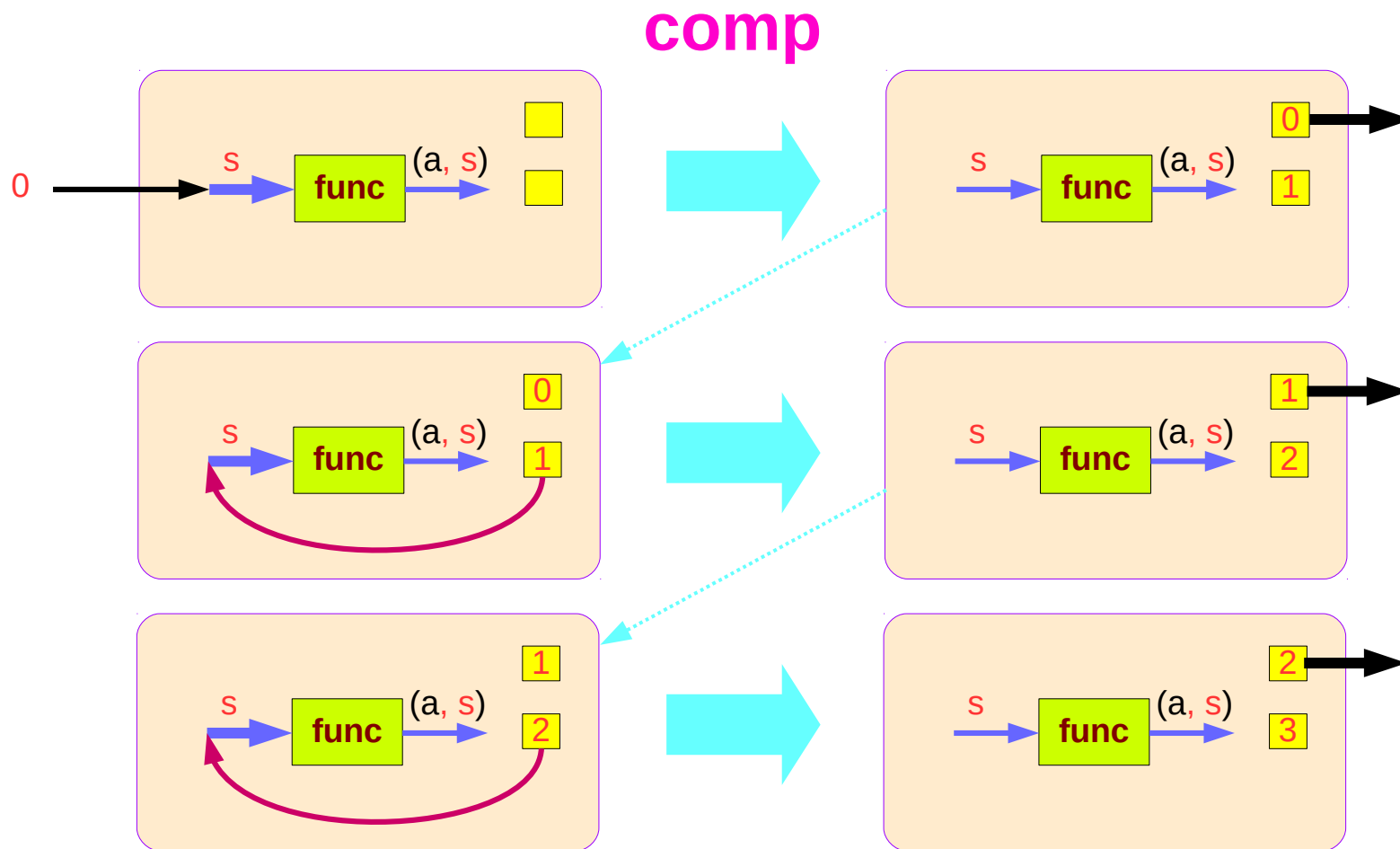
```
SimpleState :: State t a
```

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

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

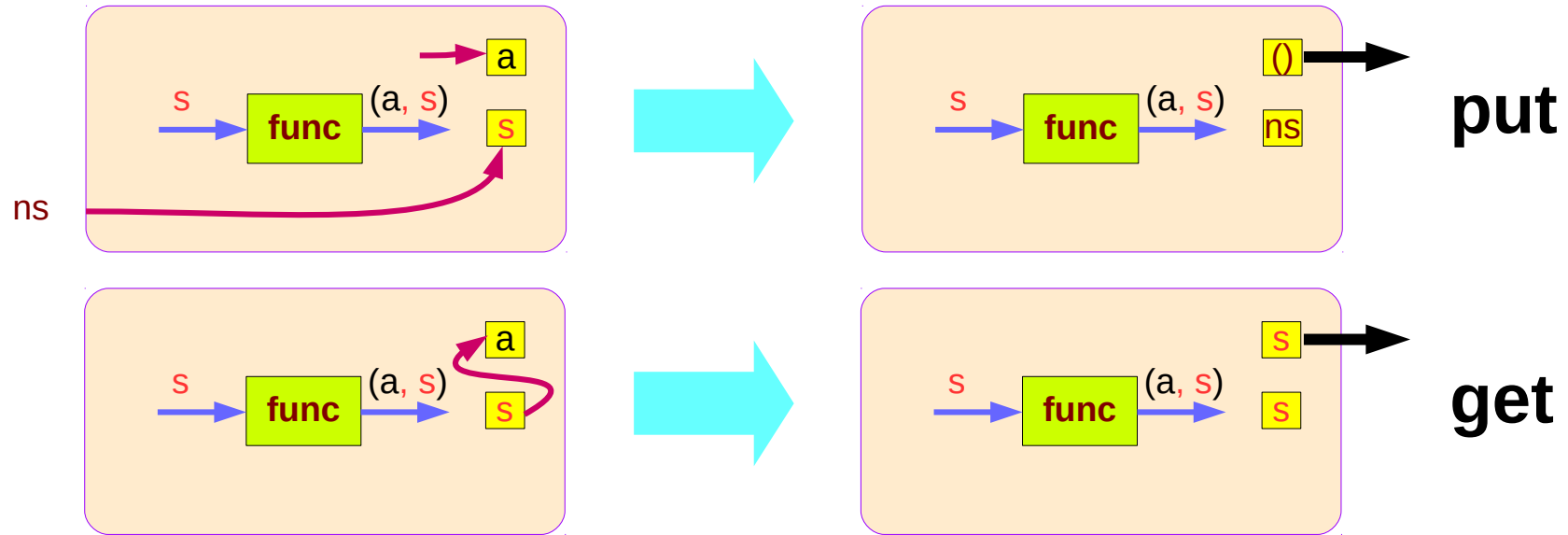
<https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell>

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

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

```
collectUntil f comp = step
```

```
where
```

```
step = do a <- comp -- updating stateful computation
```

```
liftM (a :) continue
```

```
continue = do comp' <- get -- current state getting stateful computation
```

```
if f comp' 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 – lifting to merge

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

```
(:) :: a -> [a] -> [a]
```

```
(++) :: [a] -> [a] -> [a]
```

```
(:) :: a -> [a] -> [a]
```

```
LiftM (:) :: a -> State s [a] -> State s [a]
```

```
(a :) :: [a] -> [a]
```

```
LiftM (a :) :: State s [a] -> State s [a]
```

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

Another example – the return type

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

```
collectUntil f comp = step
```

```
where
```

```
step = do a <- comp
```

-- updating stateful computation

```
liftM (a :) continue
```

```
continue = do comp' <- get
```

-- current state getting stateful computation

```
if f comp' 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 – sequence comparison

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

update and **check** the current state
And then **merge**

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

a is the common part of both
'then' part and 'else' part

```
collectUntil f comp = do
  st <- get           -- current state check
  if f st then return []
  else do
    x <- comp         -- update & result return
    xs <- collectUntil f comp
  return (x:xs)
```

check the current state
then **update** and **merge**

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

Another example – merge comparison

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

```
collectUntil f comp = do
  st <- get                    -- current state check
  if f st then return []
  else do
    x <- comp                  -- update & result return
    xs <- collectUntil f comp
    return (x:xs)
```

update and **check** the current state
And then **merge**

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

a is the common part of both
'then' part and 'else' part

check the current state
then **update** and **merge**

<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>

Monad typeclass and Instances

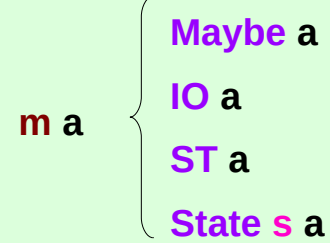
```
class Monad m where
```

```
  return :: a -> m a
```

```
  (>>=) :: m a -> (a -> m b) -> m b
```

```
  (>>) :: m a -> m b -> m b
```

```
  fail :: String -> m a
```



A diagram showing the type `m a` on the left, followed by a large right-facing curly brace. To the right of the brace are four monad instances: `Maybe a`, `IO a`, `ST a`, and `State s a`.

```
instance Monad Maybe where
```

```
  return x = Just x
```

```
  Nothing >>= f = Nothing
```

```
  Just x >>= f = f x
```

```
  fail _ = Nothing
```

```
instance Monad IO where
```

```
  m >> k = m >>= \_ -> k
```

```
  return = ...
```

```
  (>>=) = ...
```

```
  fail s = ...
```

Default Implementations in `MonadState s m`

```
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
```

The `mtl` package

`Control.Monad.State.Class` module

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

No dead loop in the default implementation

the definitions of **get**, **put**, **state** in the **Monad class declaration**

- the default implementations,
- to be overridden in actual **instances** of the class.

the dead loop in the default definition does not happen:

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

* minimal definition is *either* both of **get** and **put** or just **state**

```
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
```

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

Functional Dependency | (vertical bar)

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

functional dependencies

to constrain the parameters of type classes. s and m

s can be determined from m , $m \rightarrow s$

so that s can be the return type **State** $s \rightarrow s$

but m can not be the return type

in a multi-parameter type class,

one of the parameters can be determined from the others,

so that the parameter determined by the others can be the return type

but none of the argument types of some of the methods.

```
class Monad m where
```

```
return :: a -> m a
```

```
(>>=) :: m a -> (a -> m b) -> m b
```

```
(>>) :: m a -> m b -> m b
```

```
fail :: String -> m a
```

$m\ a$

Maybe a

IO a

ST a

State s a

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

Typeclass MonadState s

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

MonadState s

a typeclass

instance MonadState s MM where ...

its type instance itself does not specify values

MonadState s m =>

- can be used as class constraint
- all the **Monad m**
which supports *state operations* with state of type **s**.

:t get

:t put

s ← m functional dependencies

m á **State s** → **s**

state operations
defined in the
typeclass definition

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

Types of `get` and `put`

`:t get` ▶ `get :: MonadState s m => m s`

for all `Monad m` which supports *state operations* over state of type `s`,
we have a value of type `m s` - that is,
the monad operation which yields the current state

`get :: m s`

`:t put` ▶ `put :: MonadState s m => s -> m ()`

a function that takes a value of type `s`
and returns a polymorphic value
representing any `Monad m`
which supports state operations over a state of type `s`

`put :: s -> m ()`

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

Instances of `MonadState s m`

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

The `mtl` package

`Control.Monad.State.Class` module

```
instance Monad m => MonadState s (Lazy.StateT s m) where ...
instance Monad m => MonadState s (Strict.StateT s m) where ...
instance MonadState s m => MonadState s (ContT r m) where ...
instance MonadState s m => MonadState s (ReaderT r m) where ...
instance (Monoid w, MonadState s m) => MonadState s (Lazy.WriterT w m) where ...
instance (Monoid w, MonadState s m) => MonadState s (Strict.WriterT w m) where ...
```

`m`

`Lazy.StateT s m`
`Strict.StateT s m`
`ContT r m`
`ReaderT r m`
`Lazy.WriterT w m`
`Strict.WriterT w m`

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

Instances of the typeclass `MonadState s`

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

```
instance MonadState s (State s) where
  get = Control.Monad.Trans.State.get
  put = Control.Monad.Trans.State.put
```

```
instance MonadState s (StateT s) where
  get = Control.Monad.Trans.State.get
  put = Control.Monad.Trans.State.put
```

`State s` is an instance of that typeclass:

`StateT s` is an instance of that typeclass:
(the `state monad transformer`
which adds `state` to another monad)

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

Overloading get and put

```
instance MonadState s (State s) where
  get = Control.Monad.Trans.State.get
  put = Control.Monad.Trans.State.put
```

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

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

The **mtl** package provides **auto-lifting**

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

Typeclass Constrain **MonadState s m** (1)

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

```
get :: MonadState s m => m s
```

for some monad **m**

storing some state of type **s**,

get is an action in **m**

that returns a value of type **s**.

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

Typeclass Constrain `MonadState s m` (2)

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

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

for some monad `m`

`put` is an action in `m`

storing the given state of type `s`,

but returns nothing `()`.

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

References

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