Monad P2: State Monad Methods (2B)

Young Won Lim
7/6/19

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

This document was produced by using LibreOffice.

Based on

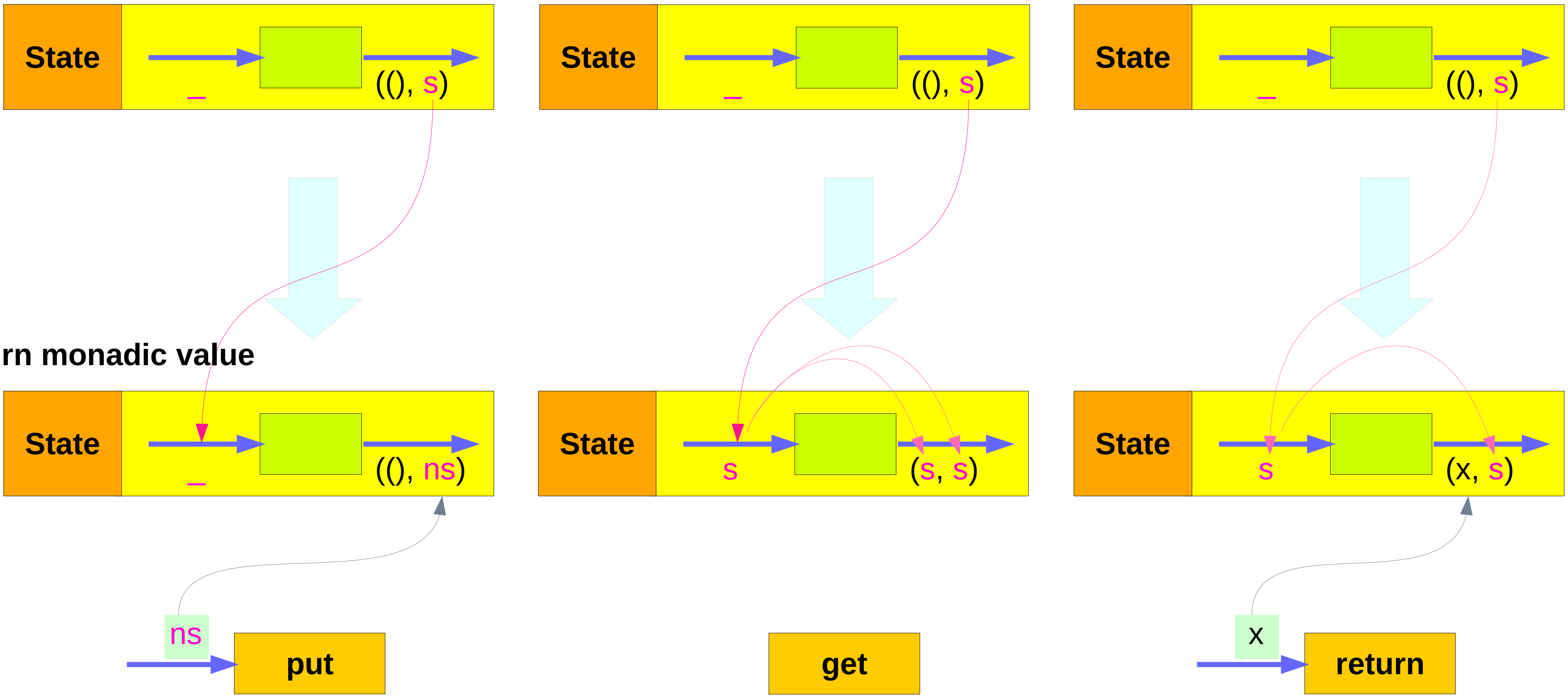
Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

put, **get**, **return** methods summary

initial monadic value

return monadic value

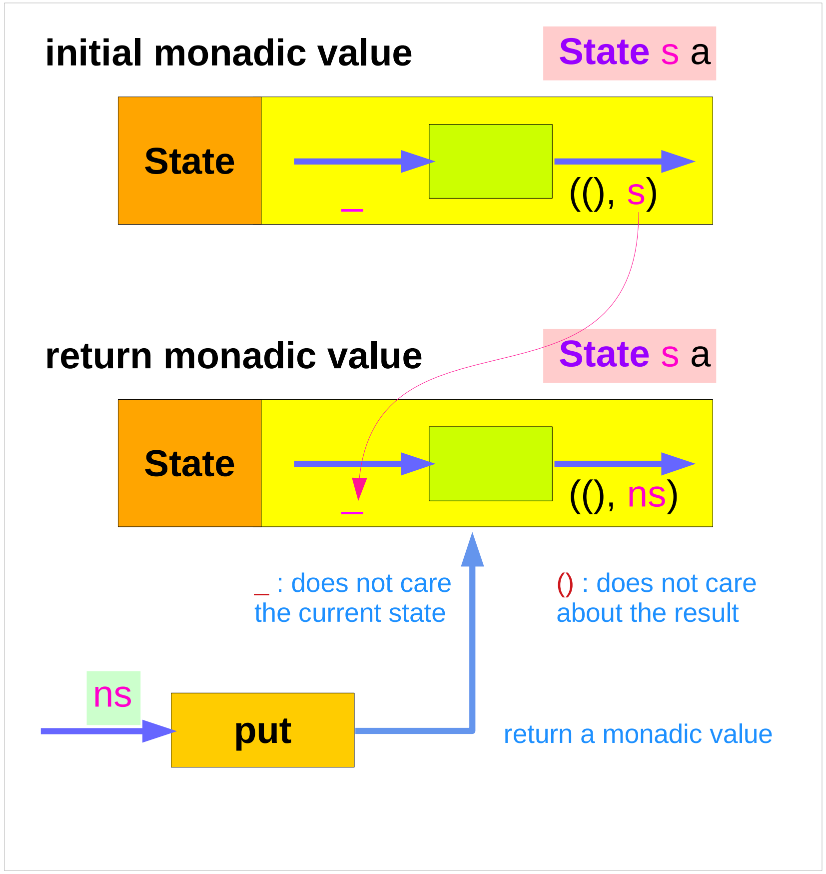


https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

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

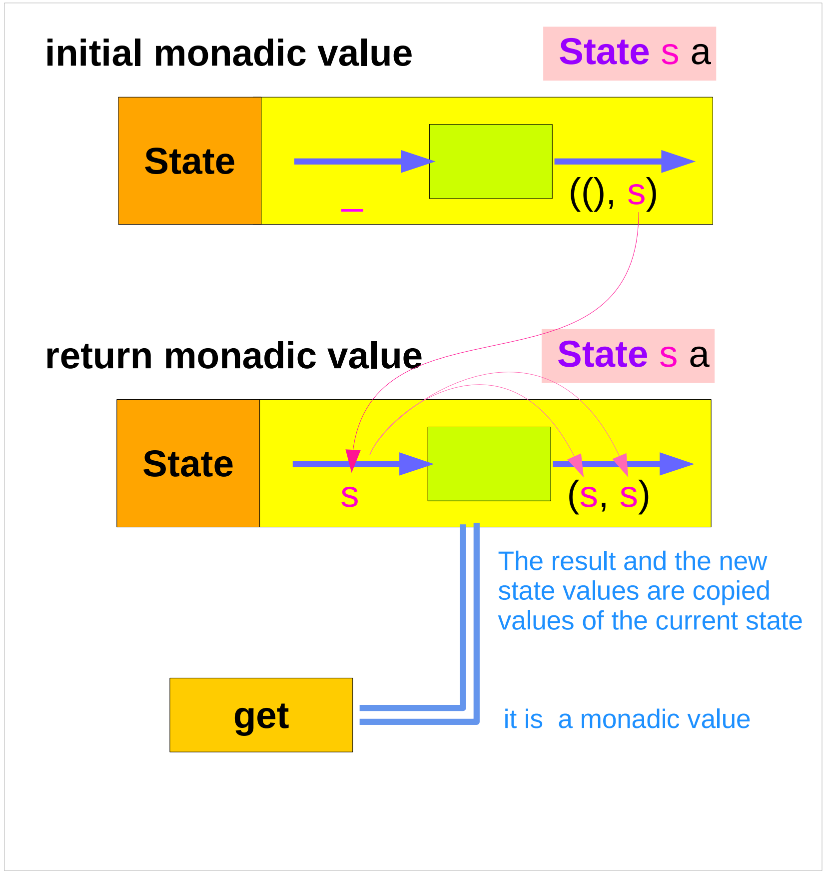
get gives the current state

```
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

return changes the result value

return :: a -> **State** s a

return x = **state** (\s -> (x, s))

giving a value (x) to **return**

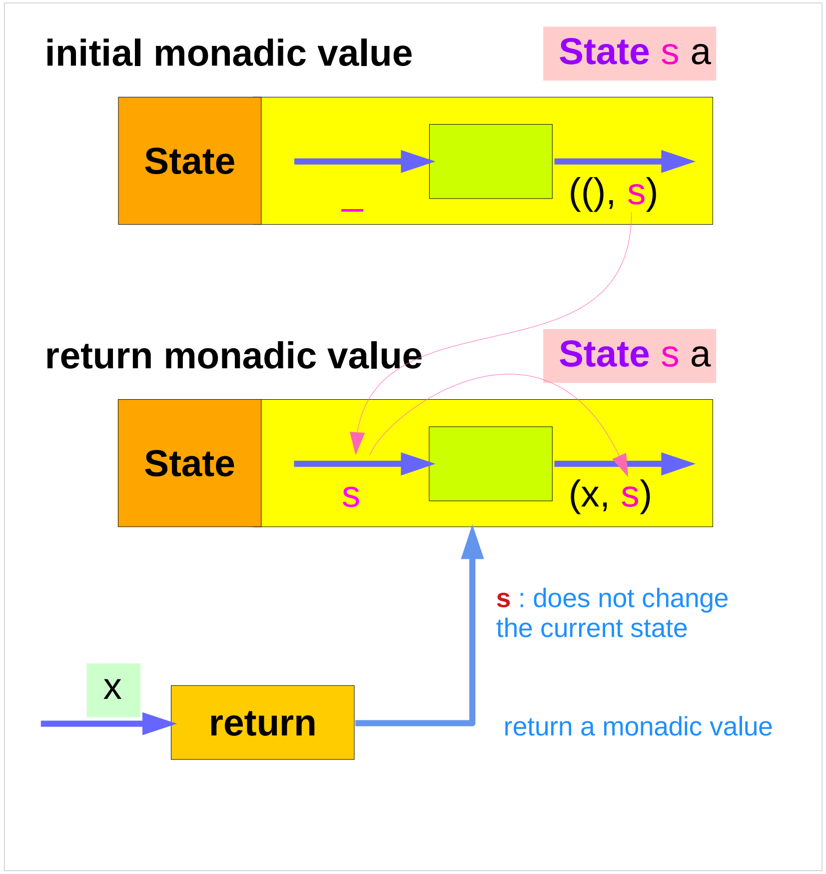
results in a **state processor** function

which takes a state (s) and

returns it unchanged (s),

together with the value x

finally, the function is wrapped up by **state**.



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

put returns a monadic value via state

put :: s **->** **State** s a

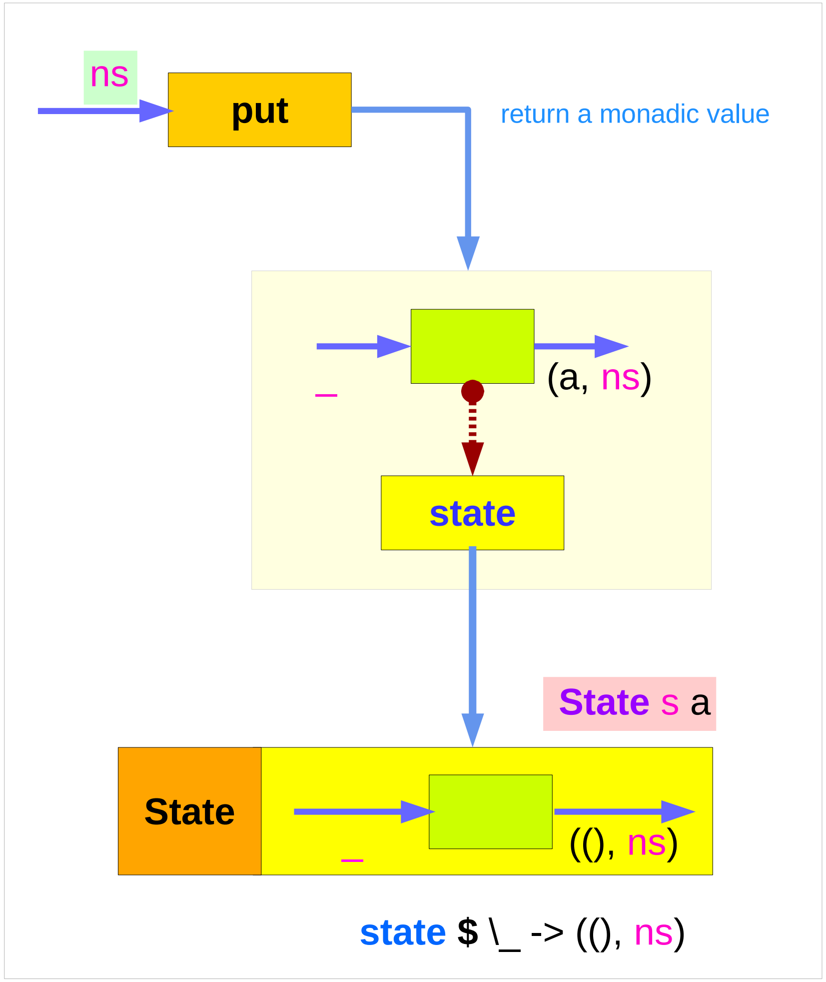
put s **::** **State** s a

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

-- setting a state to ns

-- regardless of the old state

-- setting the result to ()



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

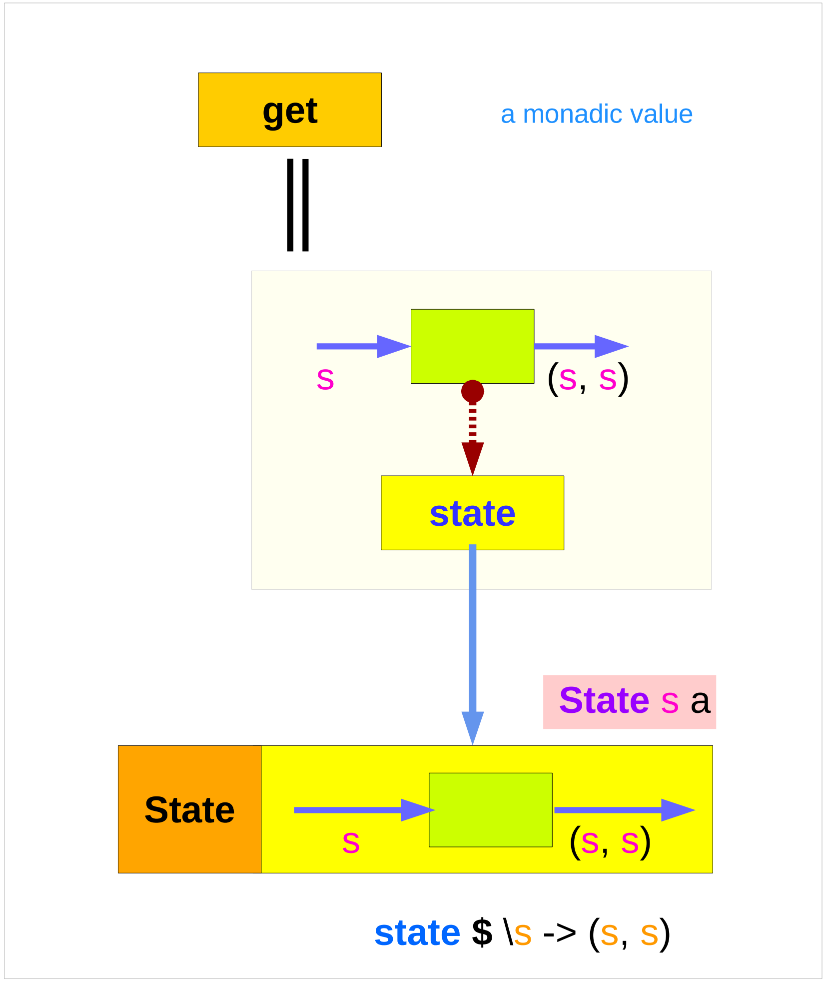
get is a monadic value via state

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

return returns a monadic value via state

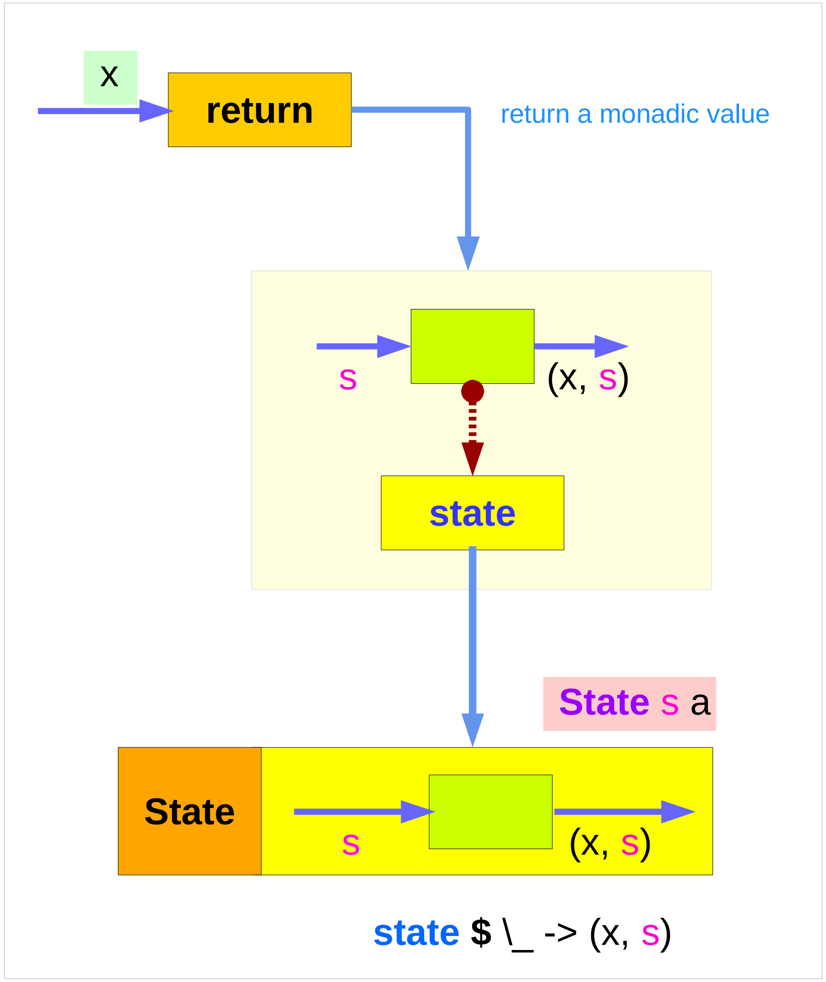
return :: s -> State s a

return s :: State s a

```
return x = state $ \_ -> (x, s)
```

-- do not change a state s

-- setting the result to x



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

Threading **put** via **runState**

put :: s **->** **State** s a

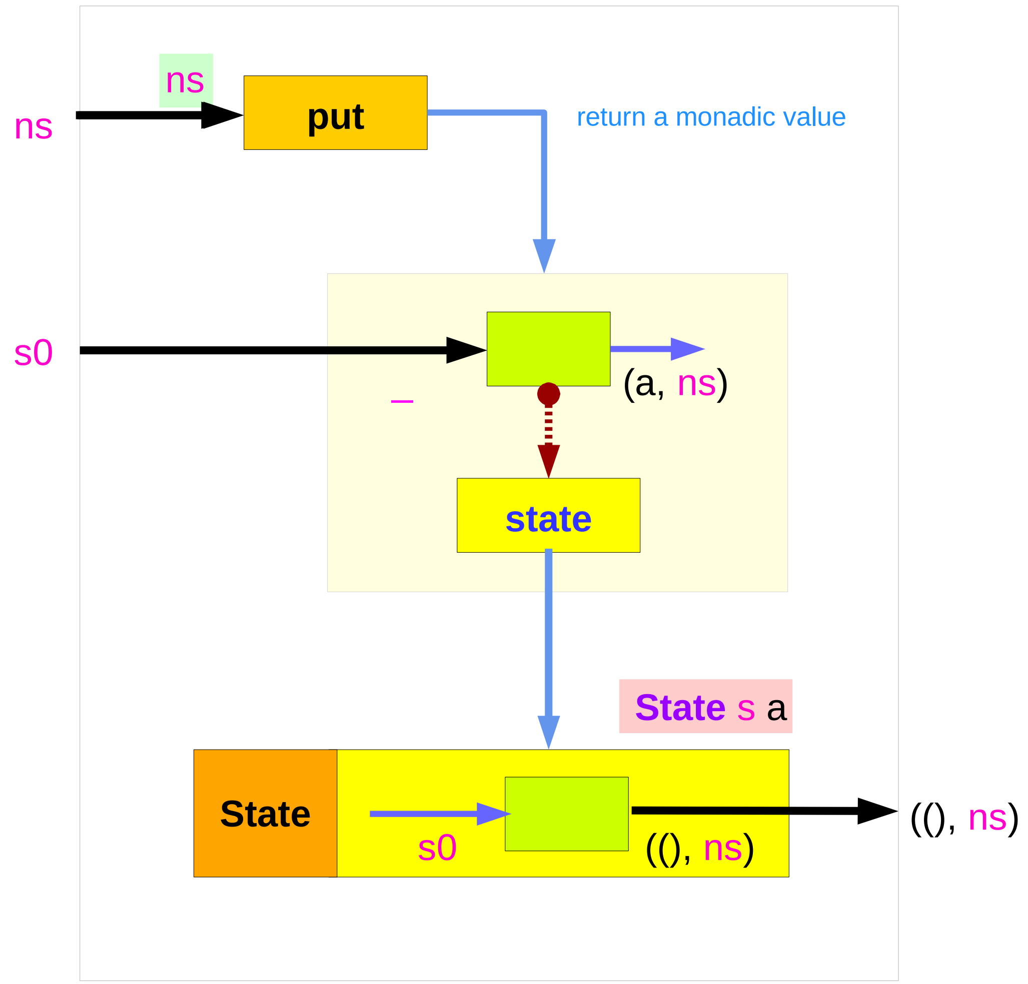
put s **::** **State** s a

put newState = **state $** _ -> ((), newState)

runState (put ns**)** **s0**

runState (put 5) **1**

((),5)



Initial state s0 can be supplied either by **runState**

or by the initial **monadic value**

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

Running **get** via **runState**

get :: State s s

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

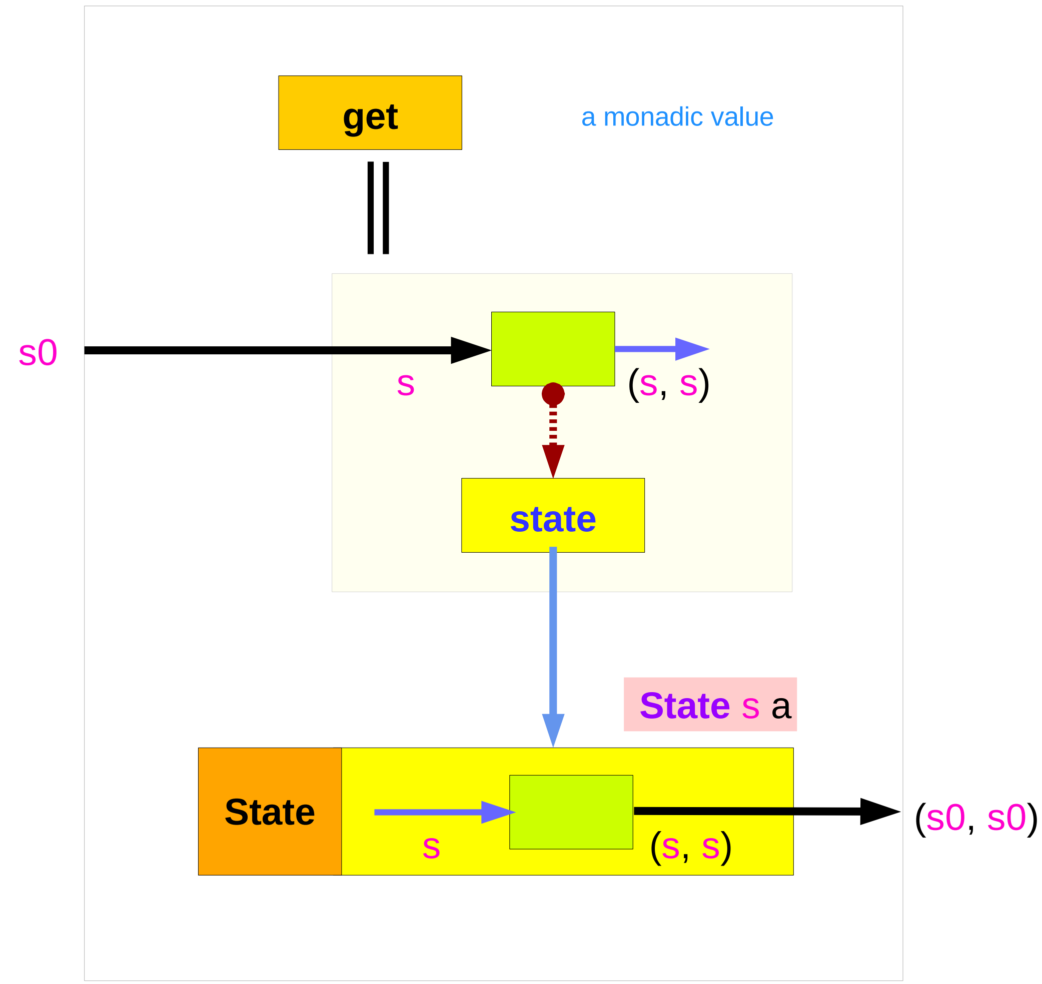
runState (get) s0

runState (get) 1

(1,1)

Initial state s0 can be supplied either by **runState**

or by the initial **monadic value**



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

Running return via runState

return :: s -> State s a

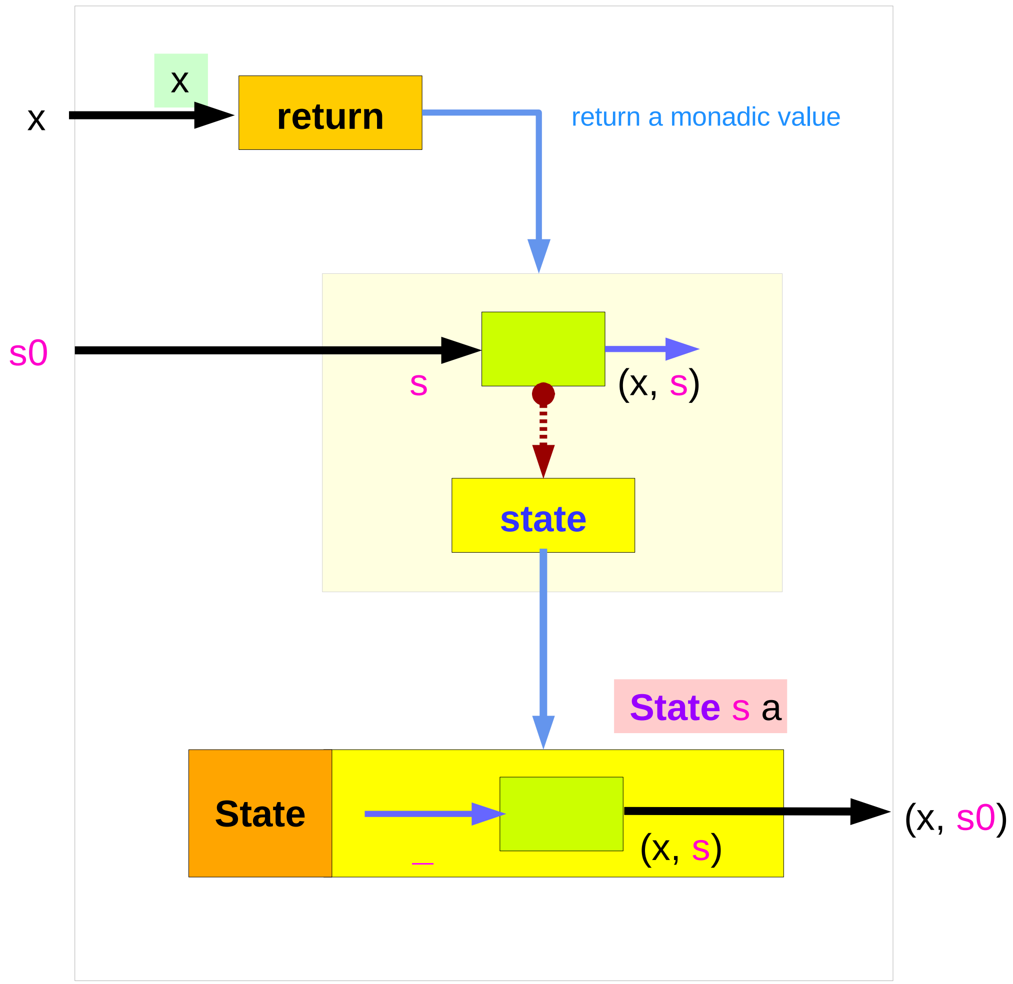
return s :: State s a

return x = state $ _ -> (x, s)

runState (return x) s0

runState (return 3) 1

(3,1)



Initial state s0 can be supplied either by **runState**

or by the initial **monadic value**

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)

get : (1,1) → (1,2)

let predecrement = do { x <- get; put (x-1); get }
 runState predecrement 1
(0,0)

(1,_) → get : (0, 0)
```

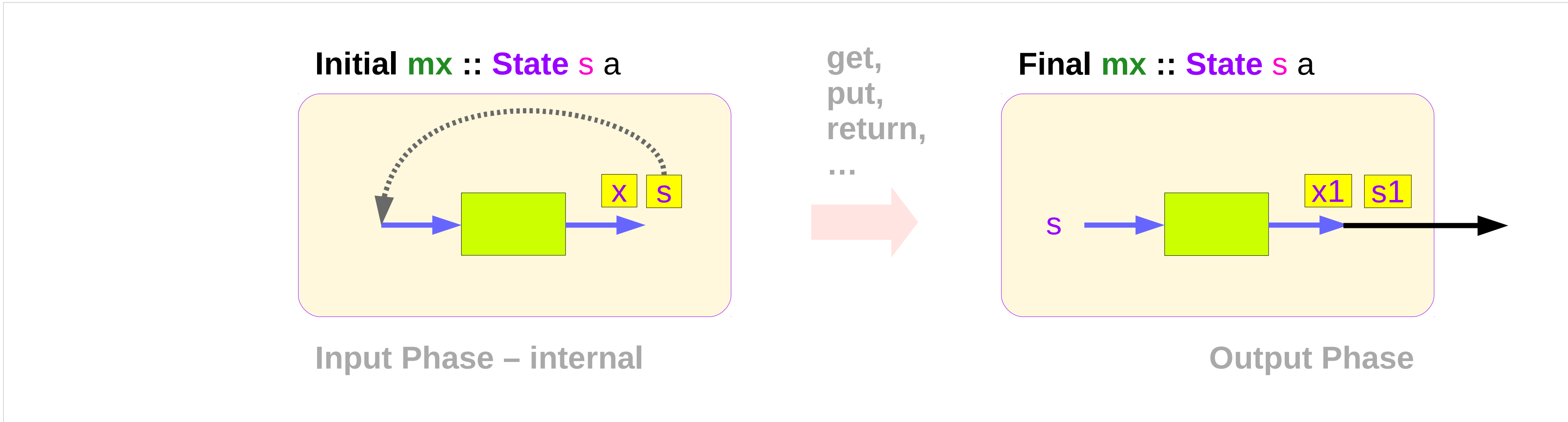
```
runState (modify (+1)) 1
((),2)
runState (gets (+1)) 1
(2,1)
evalState (gets (+1)) 1
2
execState (gets (+1)) 1
1
```

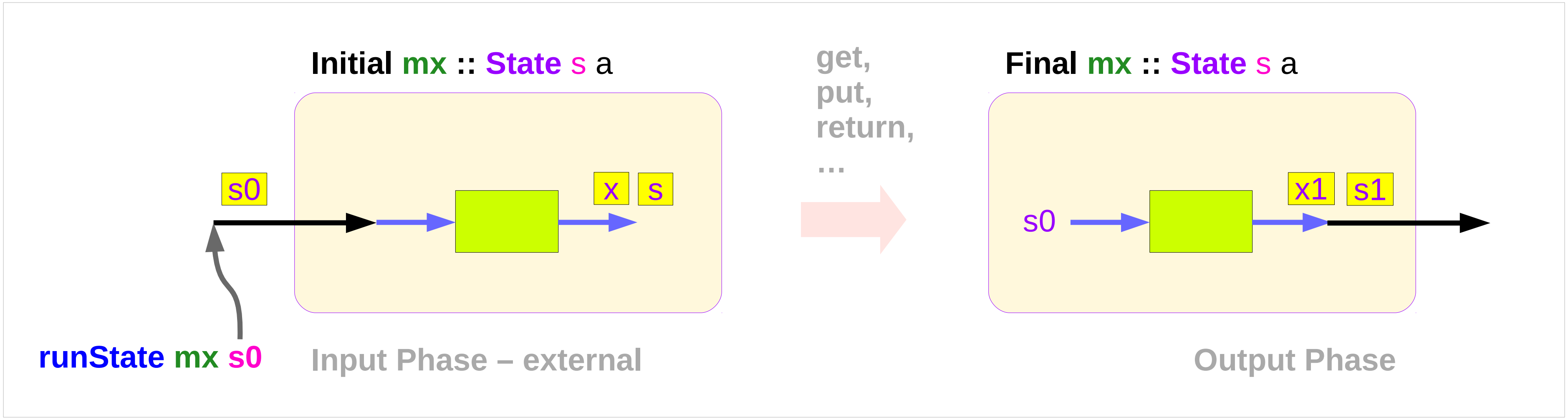
evalState	**(a, s) computes the result**
execState	**(a, s) updates state**

modify state	**((), f x)**
get state	**(f x, s)**

https://wiki.haskell.org/State_Monad

Think two phases (input, output)

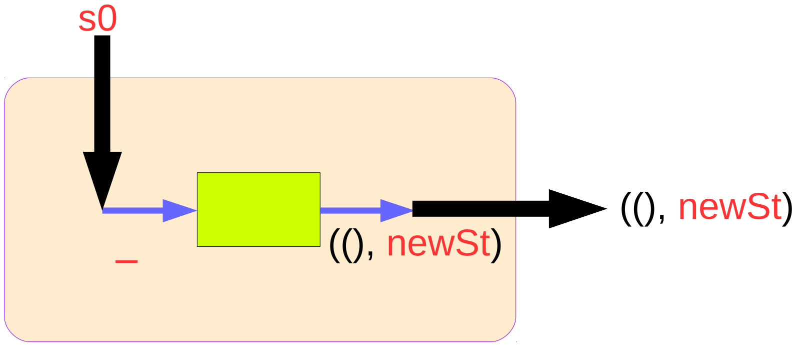




Executing the state processor – **put**

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

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



applying the function

```
runState (put 5) 1

  ((),5)
```

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

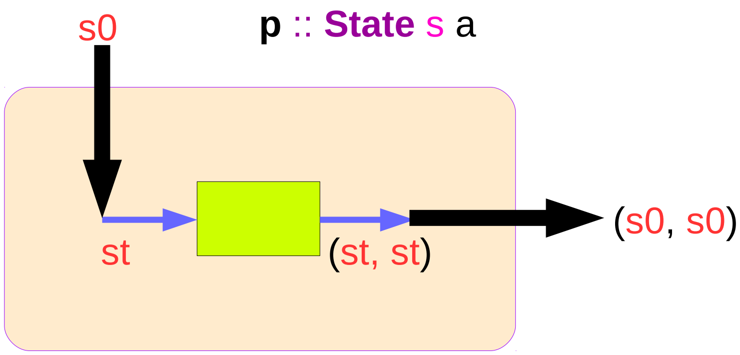
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Executing the state processor – **get**

get :: State s s

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

runState (**get**) s0 ⇒ (s0, s0)



applying the function

runState get 1

(1,1)

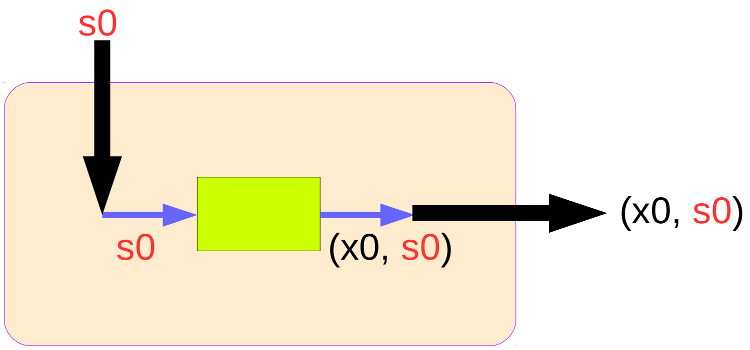
set the result value to the state and leave the state unchanged.

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

Executing the state processor – **return**

```
return :: s -> State s a
return x = state $ s -> (x, s)
```

runState **(return** x0**)** s0 ⇒ (x, s)



applying the function

runState **return** 3 1

(3,1)

set the new result value and leave the state unchanged.

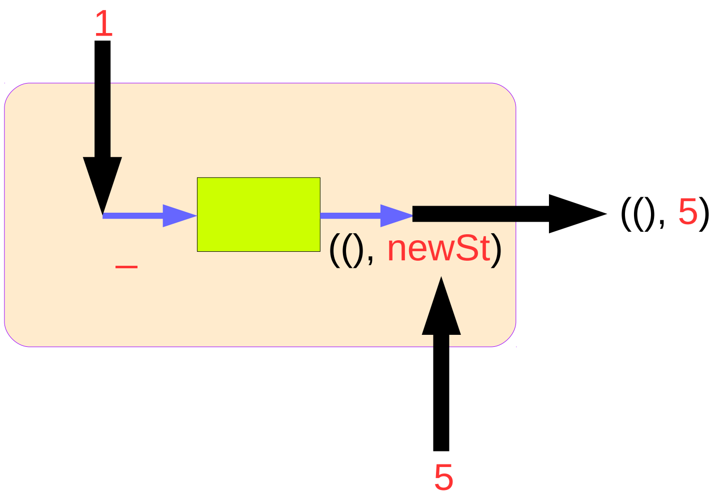
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

State Monad Examples – put

runState (**put** 5) 1

(( ),5)

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



put :: s -> State s a

put newState = **state $** _ -> ((), newState)

put 5 :: **State** Int ()

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

initial state = 1 :: Int

final value = () :: ()

final state = 5 :: Int

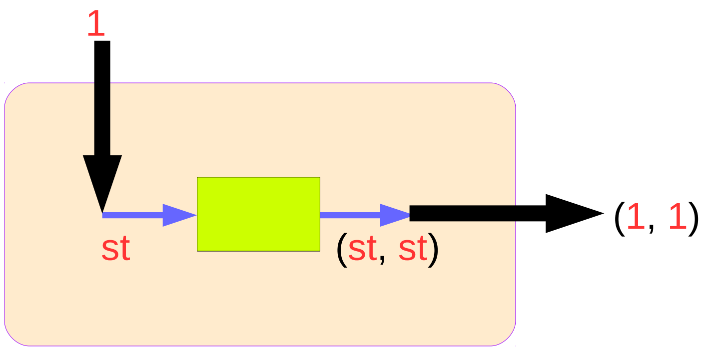
https://wiki.haskell.org/State_Monad

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)
```

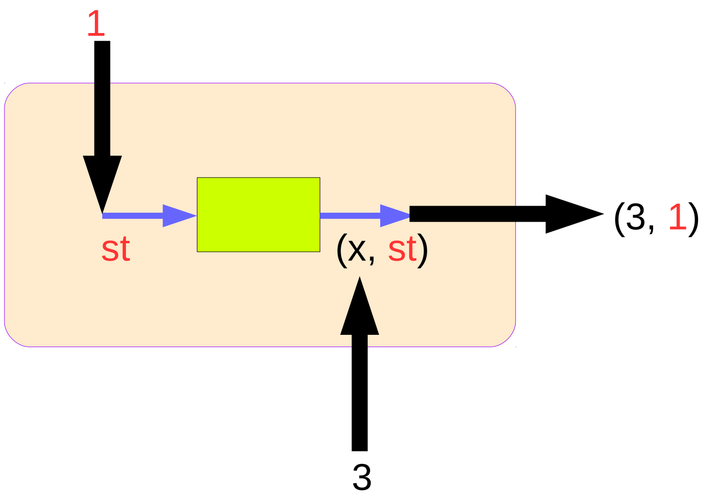
https://wiki.haskell.org/State_Monad

State Monad Examples – return

runState return 3 1

(3,1)

set the new result value and leave the state unchanged.



return :: Int -> State Int Int

runState return 3 :: Int -> (Int, Int)

initial state = 1 :: Int

final value = 3 :: Int

final state = 1 :: Int

return :: s -> State s a

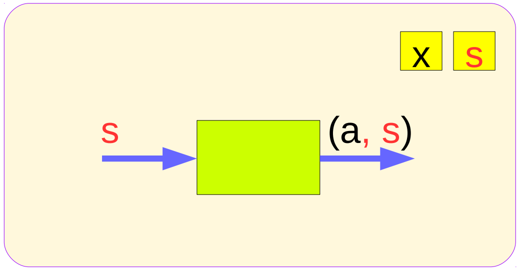
return x **=** **state** **$** s -> (x, s)

https://wiki.haskell.org/State_Monad

Think an unwrapped state processor

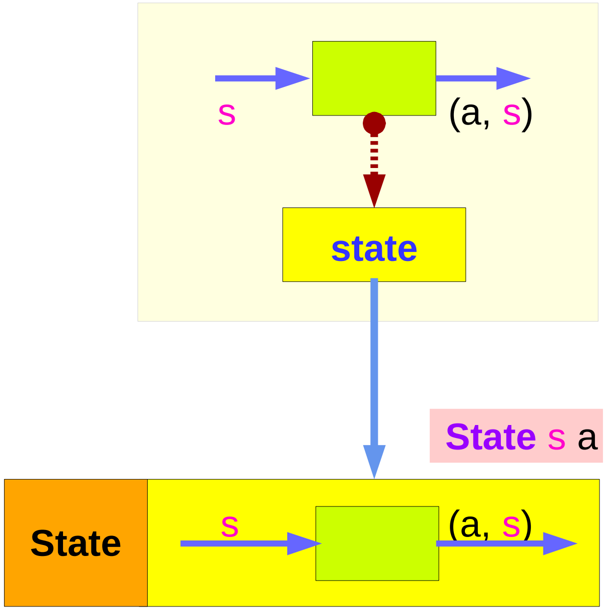
(**return** 5) ➡ 1 -> (5,1) -- a way of thinking
get ➡ 1 -> (1,1) -- a way of thinking
(**put** 5) ➡ 1 -> ((),5) -- a way of thinking

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 impl
get ➡ **state**(1 -> (1,1)) -- an actual impl
(**put** 5) ➡ **state**(1 -> ((),5)) -- an actual implementation

wrapping the state processor

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) }
```

- inside a monad instance
- unwrapped implementations

return → **(x,s)**

get → **(s, s)**

put → **((),s)**

modify

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

gets

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

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)

https://wiki.haskell.org/State_Monad

modify state ((), f x)

get state (f x, s)

evalState (a, s)

computes the result

execState (a, s)

updates state

(a , s)

(eval, exec)

(get, modify)

State Monad Examples – put, get, modify

`execState get 0`	0	(0,0)
set the value of the counter using put:		
`execState (put 1) 0`	1	((),1)
set the state multiple times:		
`execState (do put 1; put 2) 0`	2	((),1) → ((), 2)
modify the state based on its current value:		
`execState (do x <- get; put (x + 1)) 0`	1	(0,0) → ((),1)
`execState (do modify (+ 1)) 0`	1	((),1)
`execState (do modify (+ 2); modify (* 5)) 0`	10	((),2) → ((),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**.

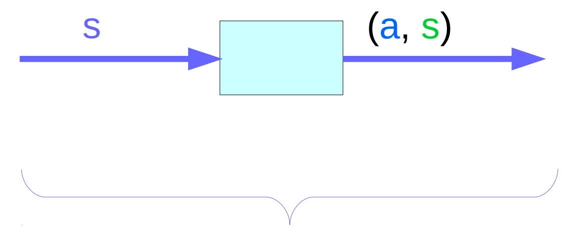
That function would have the following type:

s -> (a,s)

s is the type of the **state** and

a the **result** of the **stateful computation**.

s -> (a, s)



a <u>function</u> is an executable <u>data</u>

when <u>executed</u>, a <u>result</u> is produced

action, execution, result

s -> (a, s)

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

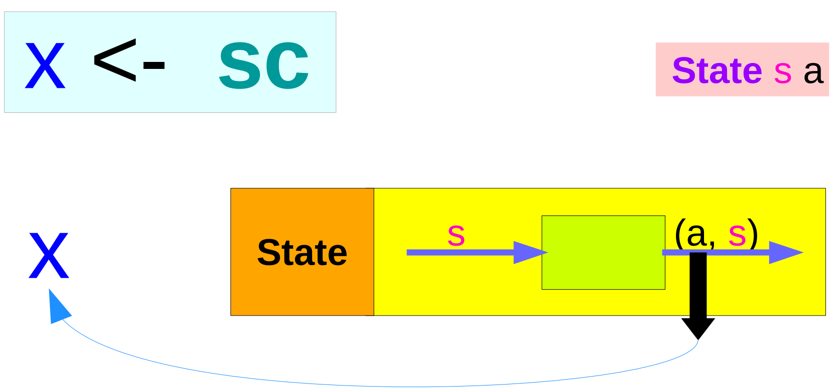
Stateful Computations inside the State Monad

inside a monad,

when **sc** is a stateful computation

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

can be assigned to **x**



only the **result** is bound

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 **State** monadic value whose new state and result values are the current state value

x <- get

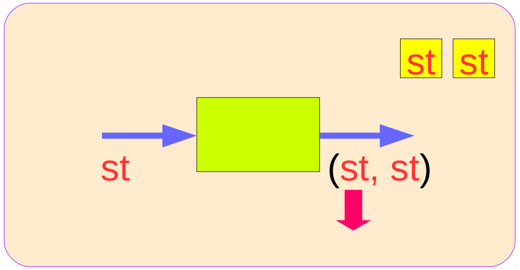
the stateful computation is performed over the monadic value returned by **get**

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**
- **State monadic value**
- **stateful computation**
- **result :: s**

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



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

put inside State Monad

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

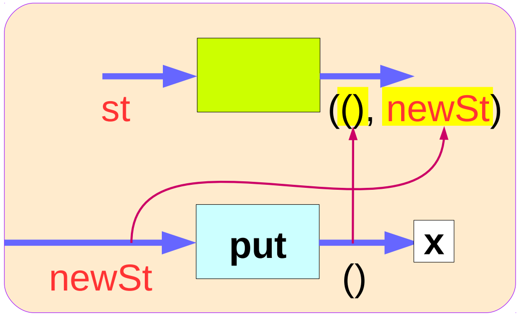
in **x <- put newSt**

put :: s -> ()

the result type :: ()

-- a way of thinking

stateful computation of **put**



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

get inside State Monad

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

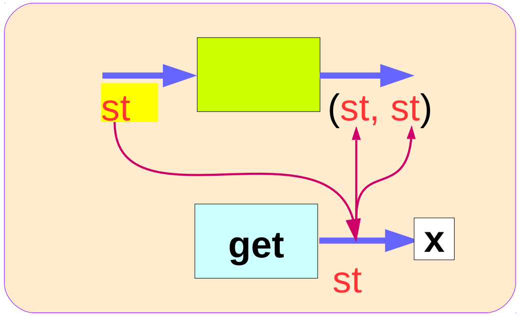
in **x <- get**

get :: s

the result type :: s

-- a way of thinking

stateful computation of **get**



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

return inside State Monad

put :: s **->** **State** s a

put newSt = **state** **$** _ -> ((), newSt)

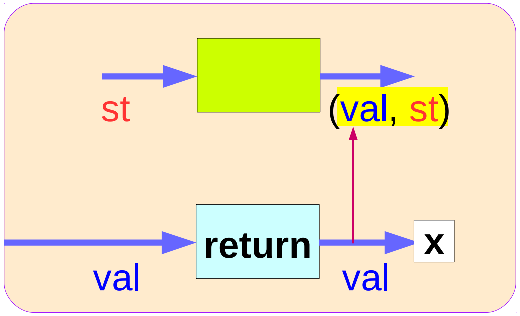
in **x <- return val**

return :: s **->** s

the result type :: s

-- a way of thinking

stateful computation of **put**



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

run functions inside a Monad

Most monads have some "***run***" **functions**

such as **runState**, **execState**, and so forth.

frequent calling such functions *inside* the monad

indicates that the **functionality** of the monad does not fully exploited

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

-- 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

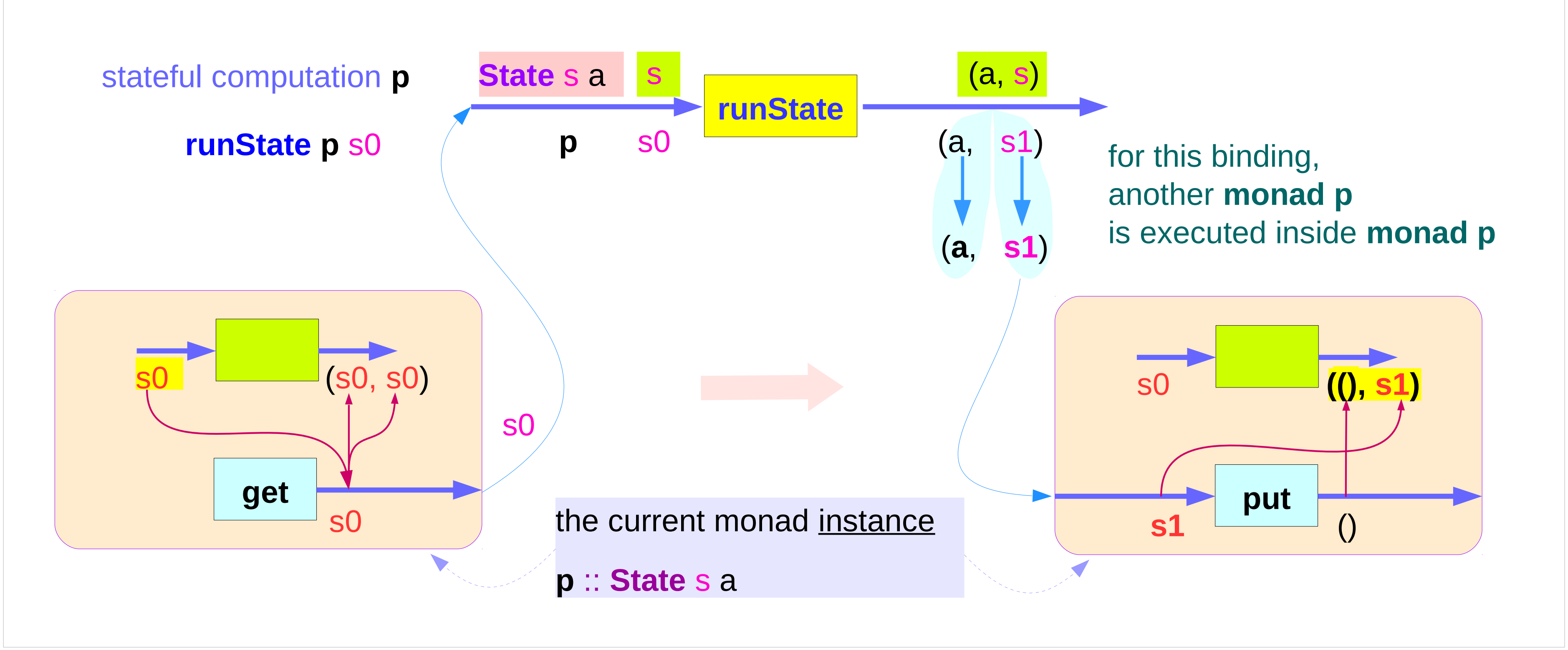
Redundant computation examples (1)

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

the same binding variable **a**

the same state s1

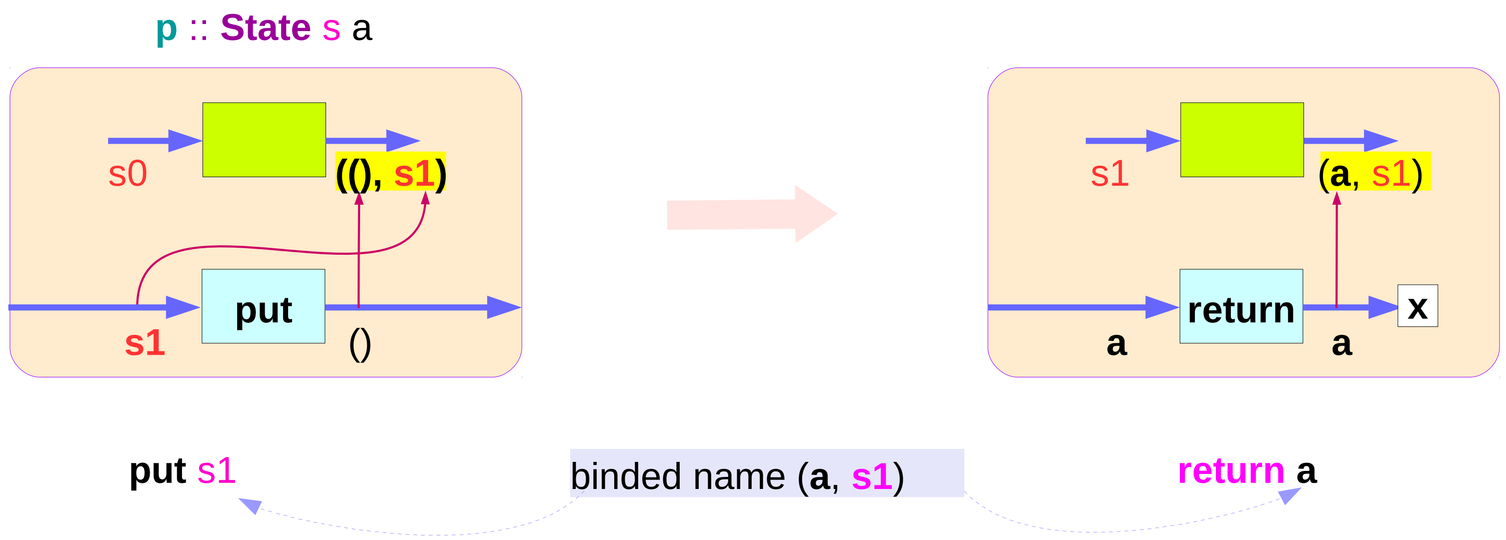
a <- p



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

Redundant computation examples (2)

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



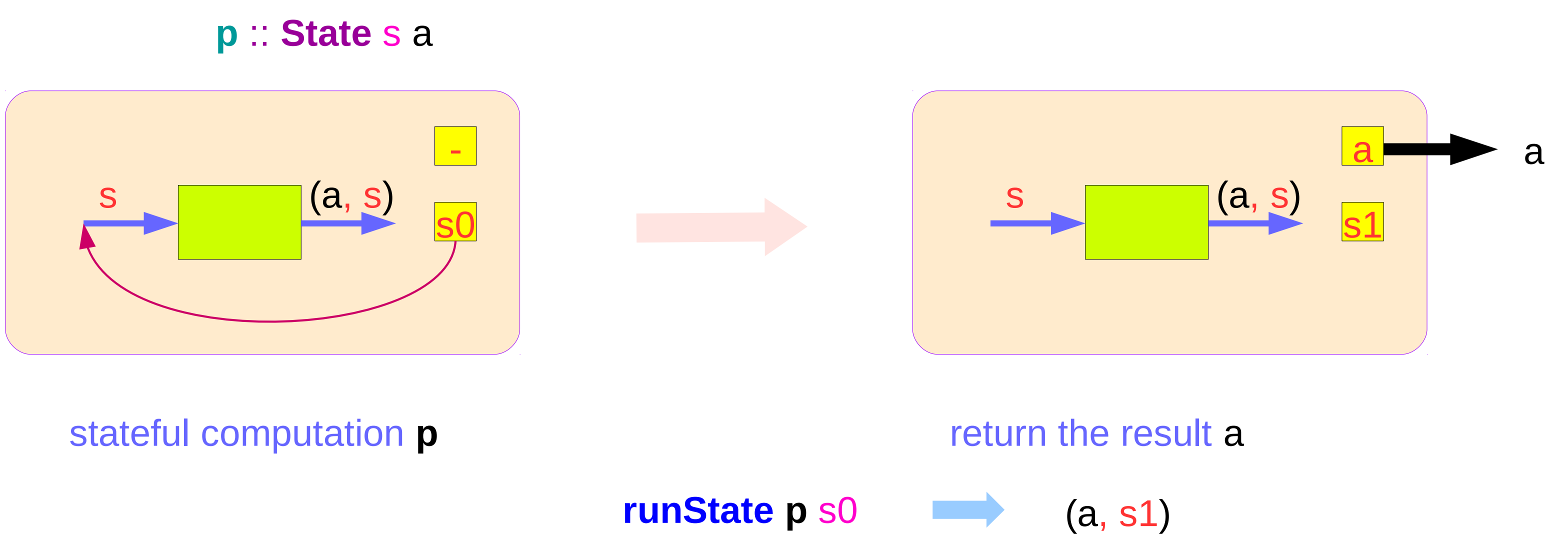
https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

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



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~~

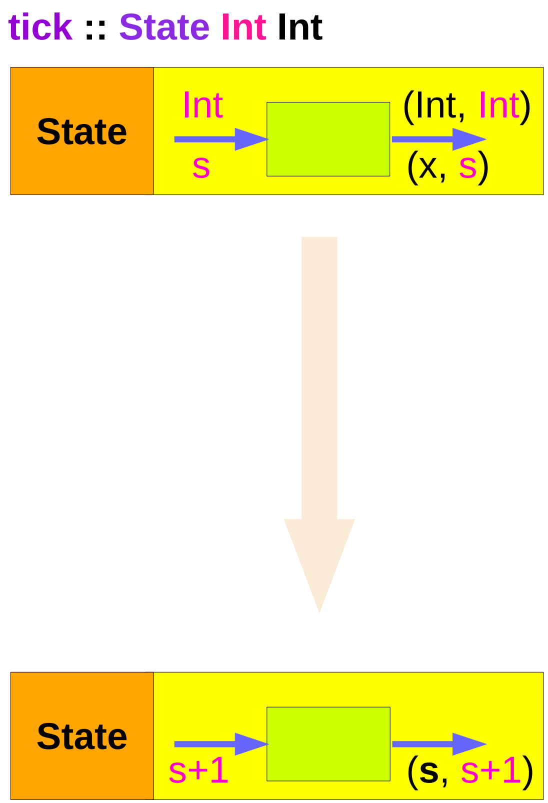
Add one to the given number using the state monad:

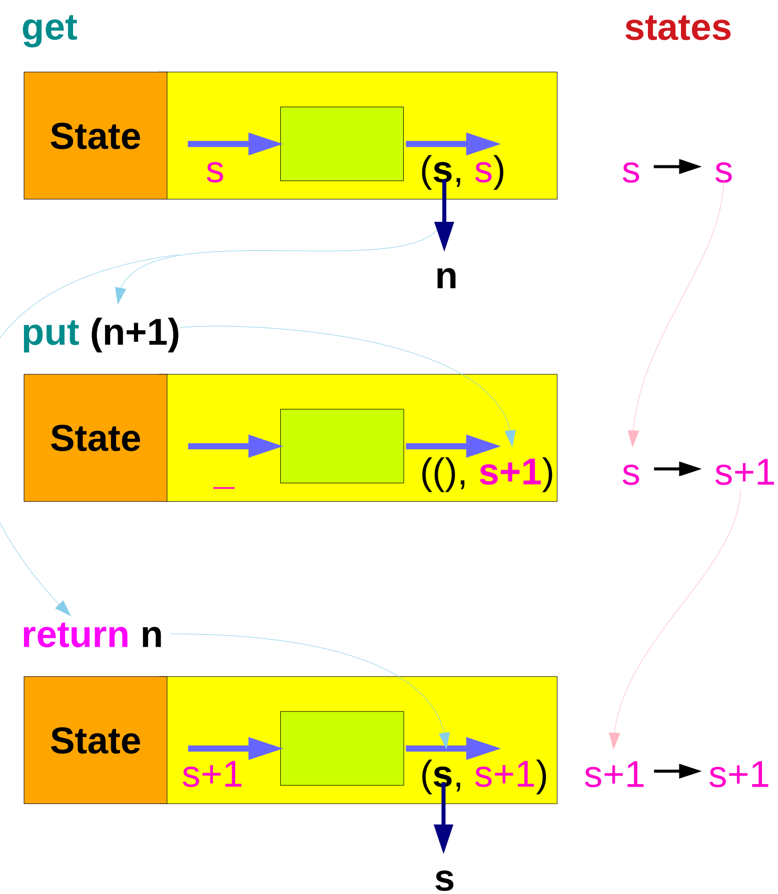
A contrived addition example. Works only with positive numbers:

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)
          execState n
```





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

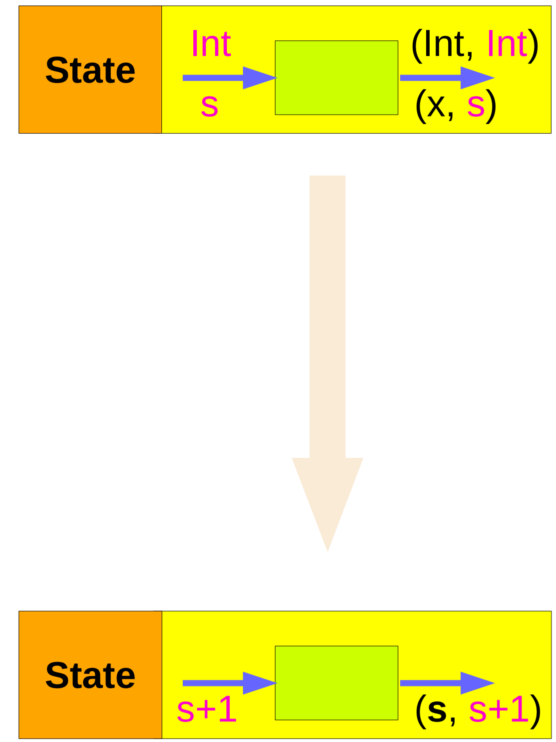
Counter Example – tick without **do**

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

Int cannot receive **()**

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

tick :: State Int Int

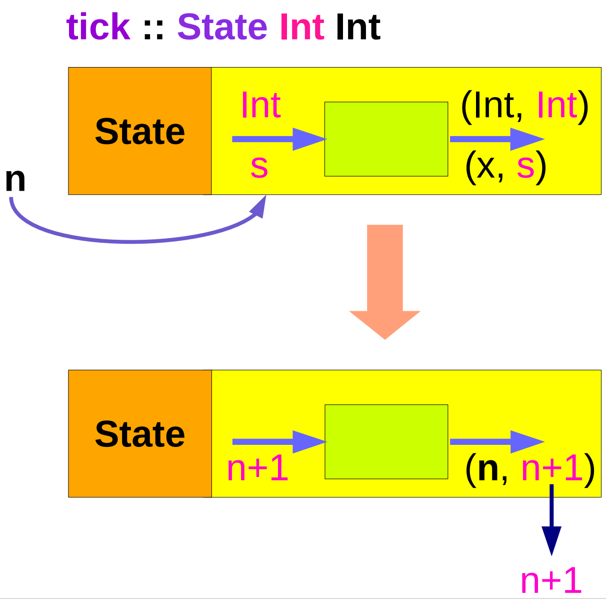


https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html

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
```

```
               1    2       n
sequence $ [tick, tick, … ,tick]
```

```
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]
```

(3,4) → (4, 5)

(3,4) → (4,5) → (5,6)

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 – **Method 1**

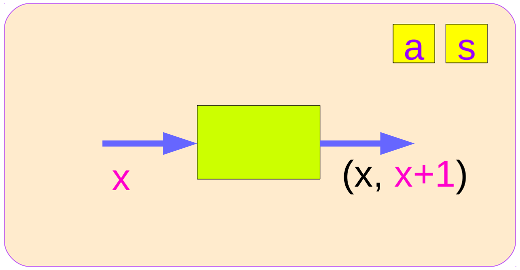
```
collectUntil f comp = do
    st <- get                      -- Get the current state
    if f st  then return [ ]       -- If it satisfies predicate, return
             else do               -- Otherwise...
                x  <- comp         -- Perform the computation s
                xs <- collectUntil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

```
comp :: State s a
st  :: s
f  :: s -> Bool
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

Method 1 and Method 2

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

Method 1

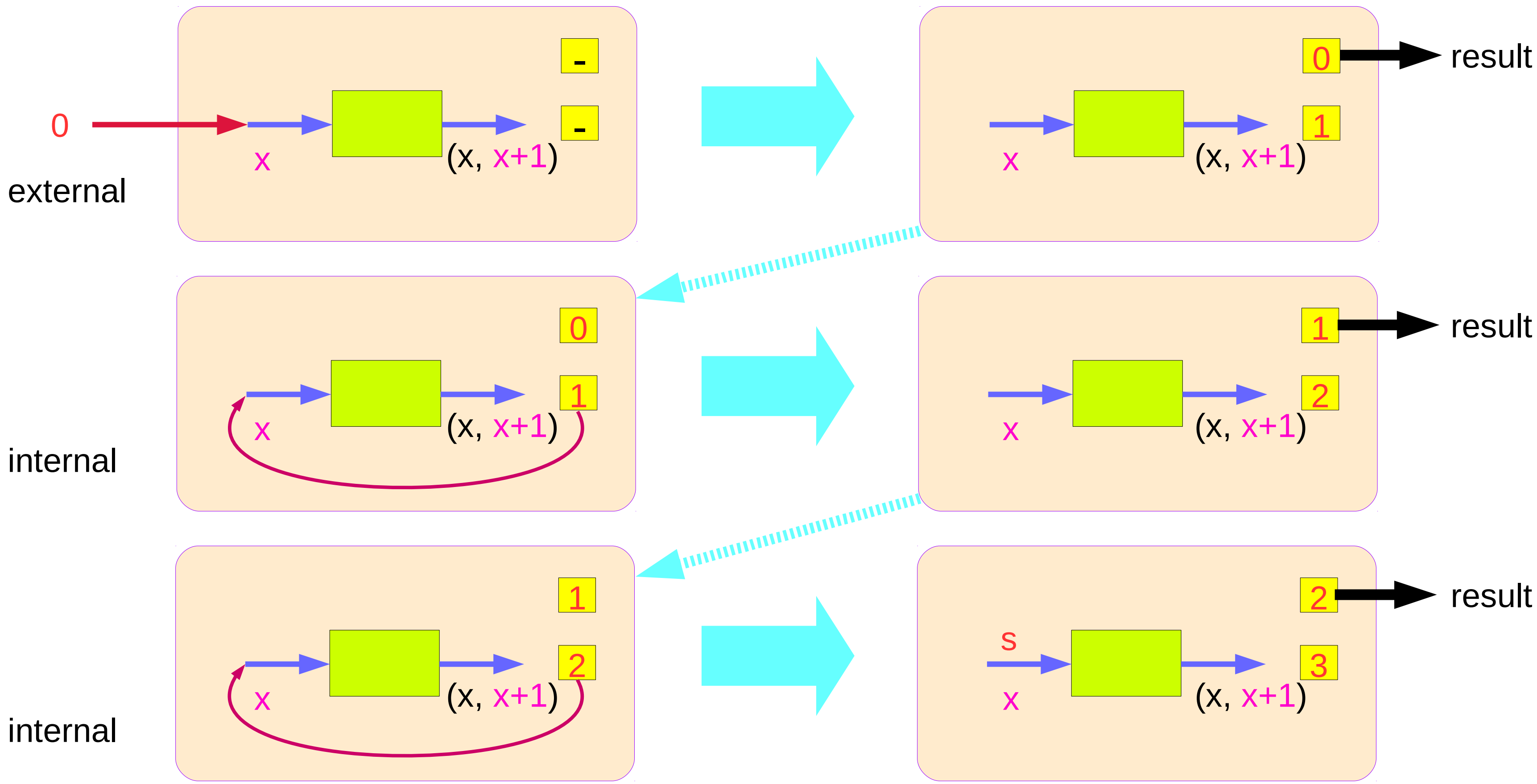
```
collectUntil :: (s -> Bool) -> State s a -> State s [a]
collectUntil f comp = step
  where
    step = do a <- comp
              liftM (a : ) continue
    continue = do b <- get
                            if f b then return []
                            else step
```

Method 2

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

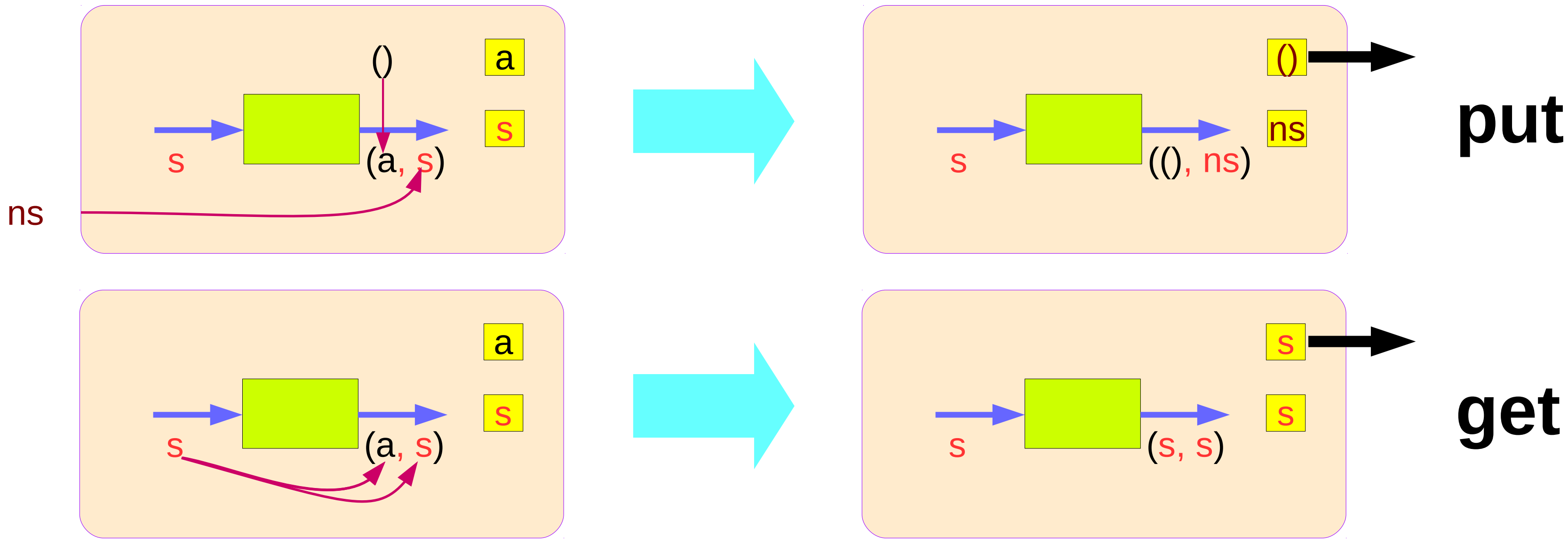
Stateful Computation of **comp**

comp (= simpleState)



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

Method 1: steps of stateful computations

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

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

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

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

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

Method 1: merge steps of collecting

```
collectUntil f comp = do
    10 <- get
    if f 10 then return [ ]
        else do
            10  <- comp
            xs <- collectUntil f comp
            return (10 : xs)        10:[] = [10]
```

```
collectUntil f comp = do
    11 <- get
    if f 11 then return [ ]
        else do
            10  <- comp
            xs <- collectUntil f comp
            return (10 : xs)
```

0: (1: (2: (3: (4: (5: (6: (7: (8: (9: (10: [ ]))))))))))

0: (1: (2: (3: (4: (5: (6: (7: (8: (9: [10])))))))))

0: (1: (2: (3: (4: (5: (6: (7: (8: [9,10]))))))))

0: (1: (2: (3: (4: (5: (6: (7: [8,9,10])))))))

0: (1: (2: (3: (4: (5: (6: [7,8,9,10]))))))

0: (1: (2: (3: (4: (5: [6,7,8,9,10])))))

0: (1: (2: (3: (4: [5,6,7,8,9,10]))))

0: (1: (2: (3: [4,5,6,7,8,9,10])))

0: (1: (2: [3,4,5,6,7,8,9,10]))

0: (1: [2,3,4,5,6,7,8,9,10])

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

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

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

Method 1: return steps of collecting

0: (1: (2: (3: (4: (5: (6: (7: (8: (9: (10: [ ]))))))))))	return []	([], 11)
0: (1: (2: (3: (4: (5: (6: (7: (8: (9: [10])))))))))	return [10]	([10], 11)
0: (1: (2: (3: (4: (5: (6: (7: (8: [9,10]))))))))	return [9,10]	([9,10], 11)
0: (1: (2: (3: (4: (5: (6: (7: [8,9,10])))))))	return [8,9,10]	([8,9,10], 11)
0: (1: (2: (3: (4: (5: (6: [7,8,9,10]))))))	return [7,8,9,10]	([7,8,9,10], 11)
0: (1: (2: (3: (4: (5: [6,7,8,9,10])))))	return [6,7,8,9,10]	([6,7,8,9,10], 11)
0: (1: (2: (3: (4: [5,6,7,8,9,10]))))	return [5,6,7,8,9,10]	([5,6,7,8,9,10], 11)
0: (1: (2: (3: [4,5,6,7,8,9,10])))	return [4,5,6,7,8,9,10]	([4,5,6,7,8,9,10], 11)
0: (1: (2: [3,4,5,6,7,8,9,10]))	return [3,4,5,6,7,8,9,10]	([3,4,5,6,7,8,9,10], 11)
0: (1: [2,3,4,5,6,7,8,9,10])	return [2,3,4,5,6,7,8,9,10]	([2,3,4,5,6,7,8,9,10], 11)
0: [1,2,3,4,5,6,7,8,9,10]	return [1,2,3,4,5,6,7,8,9,10]	([1,2,3,4,5,6,7,8,9,10], 11)
[0, 1,2,3,4,5,6,7,8,9,10]	return [0, 1,2,3,4,5,6,7,8,9,10]	([0, 1,2,3,4,5,6,7,8,9,10], 11)

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

Method 1: branch within a **do** block

```
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
```

the same return monadic type value

x :: a

xs :: [a]

(x : xs) :: [a]

0: (1: (2: (3: (4: (5: (6: (7: (8: (9: (10: [ ]))))))))))

nesting do statement

- is possible if they are within the same monad

- enables **branching** within one do block,

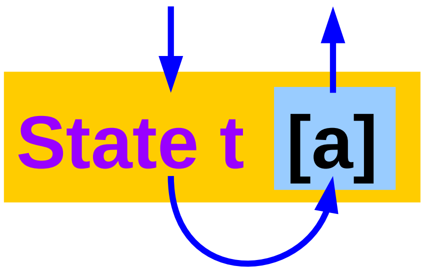
as long as both branches of the **if statement**

results in the same monadic type.

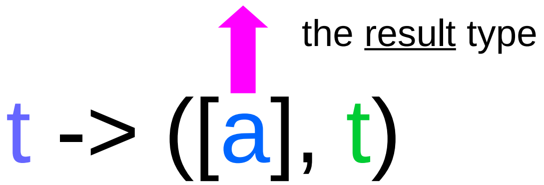
https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

Method 1: return stateful compuation

```
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]
```



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

Method 1: the inferred 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)

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

simpleState :: State t a

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

collectUntil (>10) simpleState :: State t [a]

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

Another implementation 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 b <- get                               -- current state getting stateful computation
                  if f b then return []
                  else step
```

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

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

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

Method 2: other representation

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

```
step = do a <- comp
          liftM (a : ) do b <- get
                          if f b then return []
                          else step
```

```
step = do a <- comp
          liftM (a : ) do b <- get
                          if f b then return []  else step
```

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

Method 2: the return type

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

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

returns only once at the last iteration

```
continue :: State s [a]
liftM (a : ) continue :: State s [a]
```

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

Method 2: liftM 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 b <- get
                  if f b then return []  else step
```

return :: State t [a] type

collectUntil f comp :: State t [a] type

continue :: State t [a] type

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

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

a :: a

continue :: State s [a]

liftM (a :) continue :: State s [a]

	(:)	:: a ->	[a] ->	[a]
liftM	(:)	:: a -> State s	[a] -> State s	[a]

	(a :)	::	[a] ->	[a]
liftM	(a :)	:: State s	[a] -> State s	[a]

liftM (a :) continue :: State s [a]

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

Method 2: steps of stateful computations

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

a <- comp

b <- get

return []

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

stateful computation

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

Method 2: merge computation steps

```
collectUntil :: (s -> Bool) -> State s a -> State s [a]
collectUntil f comp = step
  where
    step = do 10 <- comp
              liftM (10 : ) continue
    continue = do 11 <- get
                  if f 11 then return []  else step
```

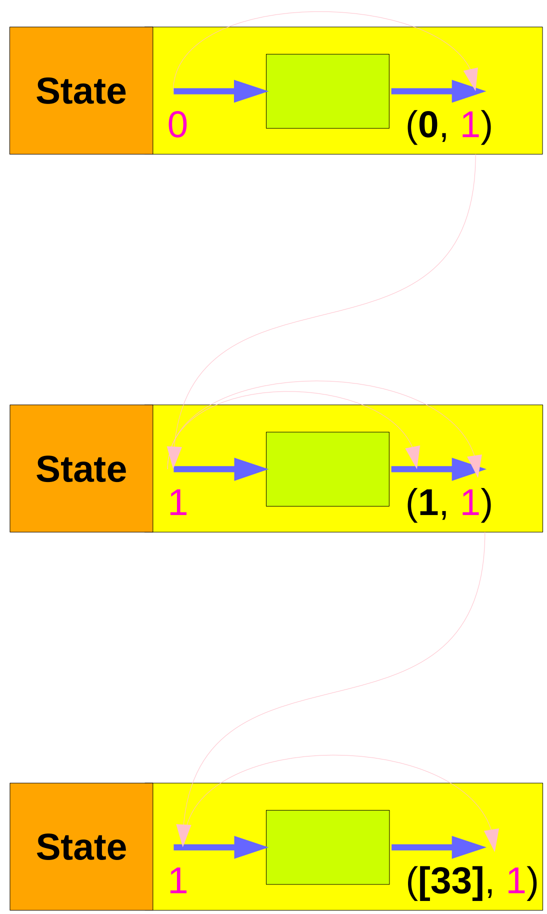
(>10)

(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) (liftM (4:) (liftM (5:) (liftM (6:) (liftM (7:) (liftM (8:) (liftM (9:) (liftM (10:) ([],11))))))))))))
(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) (liftM (4:) (liftM (5:) (liftM (6:) (liftM (7:) (liftM (8:) (liftM (9:) ([10],11)))))))))))
(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) (liftM (4:) (liftM (5:) (liftM (6:) (liftM (7:) (liftM (8:) ([9,10],11))))))))))
(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) (liftM (4:) (liftM (5:) (liftM (6:) (liftM (7:) ([8,9,10],11)))))))))
(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) (liftM (4:) (liftM (5:) (liftM (6:) ([7,8,9,10],11))))))))
(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) (liftM (4:) (liftM (5:) ([6,7,8,9,10],11)))))))
(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) (liftM (4:) ([5,6,7,8,9,10],11))))))
(liftM (0:) (liftM (1:) (liftM (2:) (liftM (3:) ([4,5,6,7,8,9,10],11)))))
(liftM (0:) (liftM (1:) (liftM (2:) ([3,4,5,6,7,8,9,10],11))))
(liftM (0:) (liftM (1:) ([2,3,4,5,6,7,8,9,10],11)))
(liftM (0:) ([1,2,3,4,5,6,7,8,9,10],11))
[0,1,2,3,4,5,6,7,8,9,10]

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

Method 2: comp, get, return

a <- comp



a=0

state (\x -> (x,x+1))

b <- get

b=1

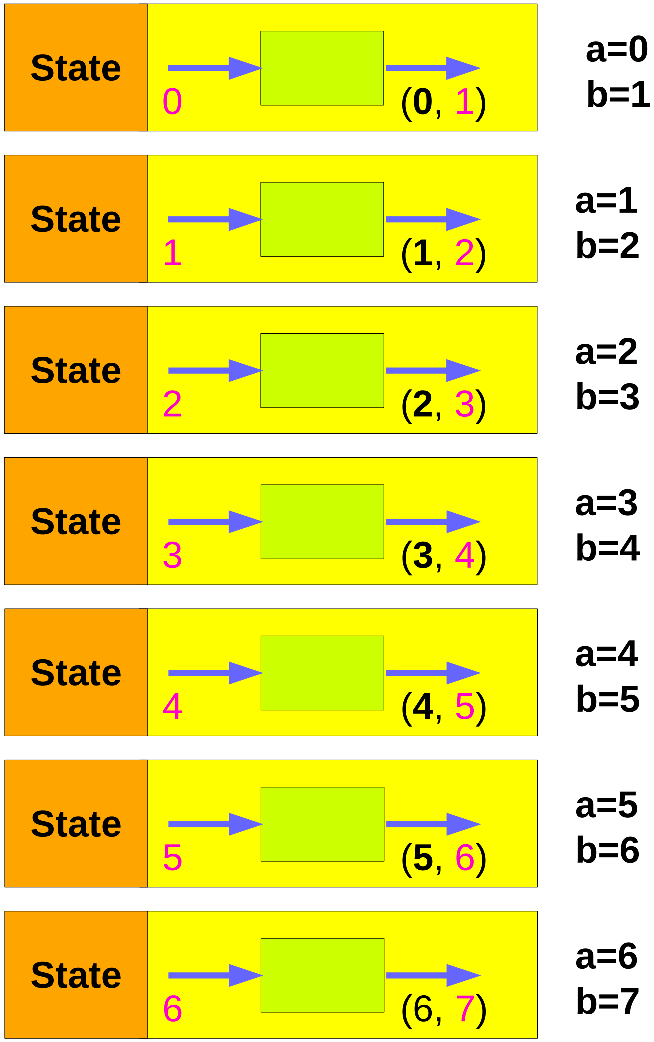
return [33]

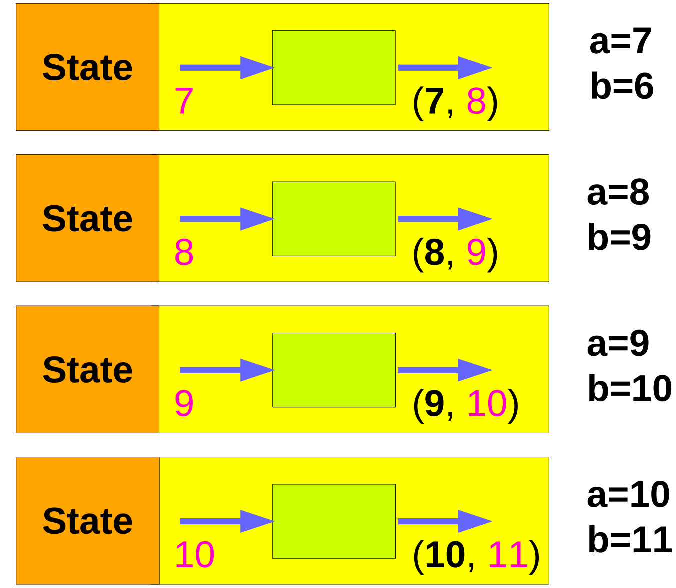
[33]

```
collectUntil f comp = step
  where
    step = do a <- comp
              liftM (a : ) continue
    continue = do b <- get
                  if f b then return []
                  else step
```

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

Method 2: steps of a<-comp, b<-get

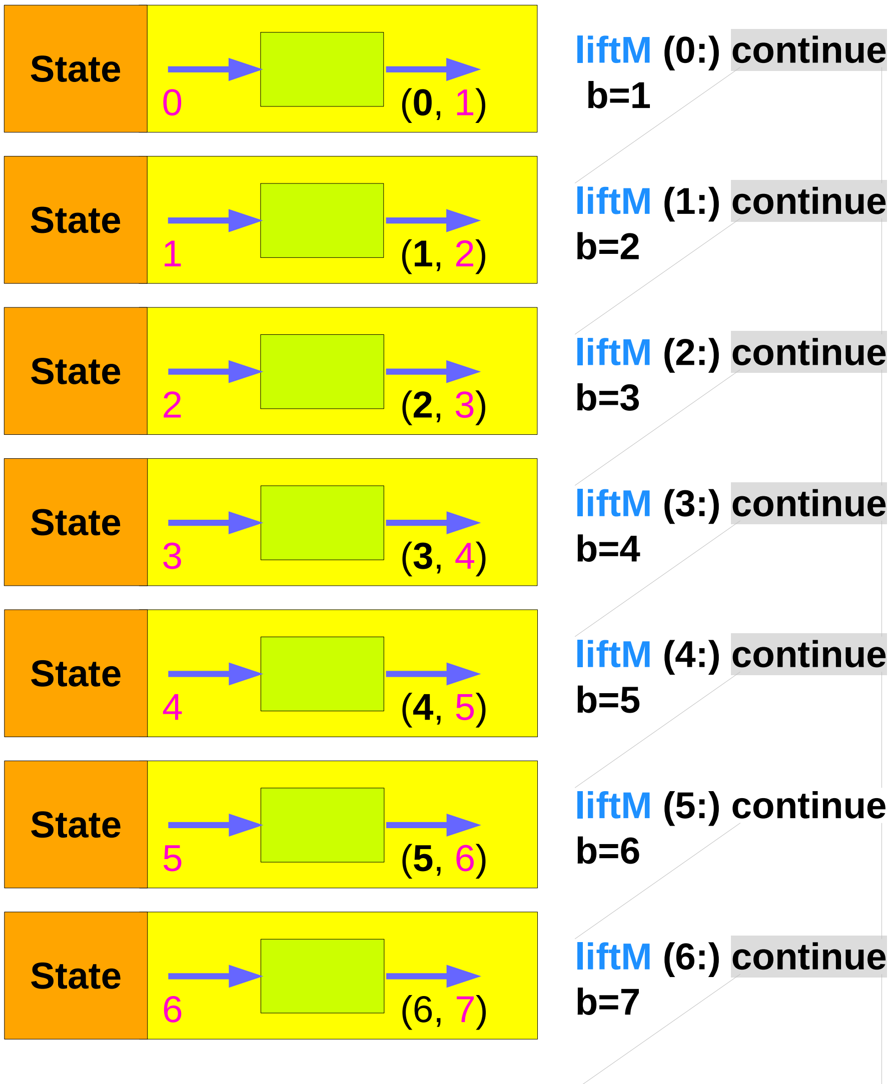


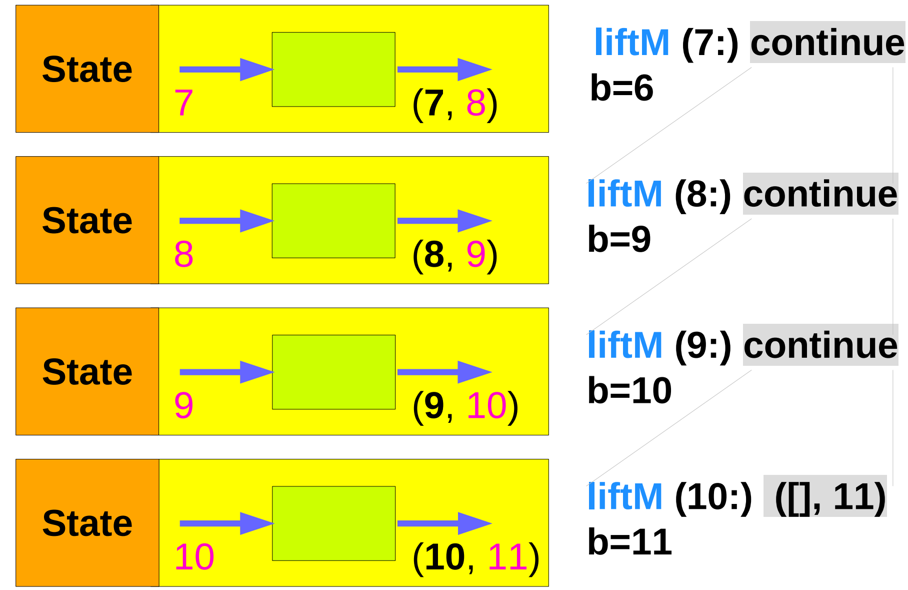



```
collectUntil f comp = step
  where
    step = do a <- comp
               liftM (a : ) continue
    continue = do b <- get
                if f b then return []
                      else step
```

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

Method 2: steps of **continue**






```
collectUntil f comp = step
  where
    step = do a <- comp
              liftM (a : ) continue
    continue = do b <- get
                  if f b then return []
                  else step
```

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

Sequence comparison

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

Method 2

update the current state

then **get** and then **merge**

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

Method 1

get the current state

then **update** and **merge**

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

Merge comparison

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

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

Method 2

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

continue :: State s [a]

liftM (a :) continue :: State s [a]

Method 1

xs :: [a]

x : xs :: [a]

retuns the list of results

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

Example of collecting – source codes

```
import Control.Monad.Trans.State

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

simpleState :: State Int Int
simpleState = state $ \x -> (x,x+1)

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

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

simpleState :: State Int Int
simpleState = state $ \x -> (x,x+1)

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

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
```

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