State Monad (3D)

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

This document was produced by using OpenOffice.

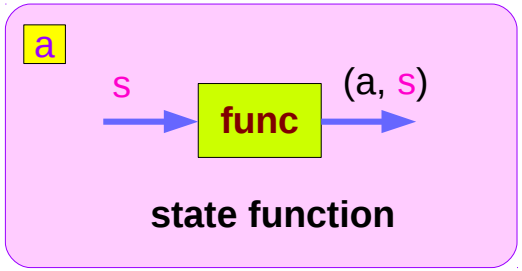
Based on

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

The state function

The Haskell type **State** describes **functions**
that take a state
and return both a result and an updated state,
which are given back in a tuple.

The **state function** is wrapped by a data type definition
which comes along with a **runState accessor**
no need for pattern matching



p :: **State** s a

one-field record

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

accessor function

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

Type State

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

s : the type of the state,

a : the type of the produced result

s -> (a, s) : function type

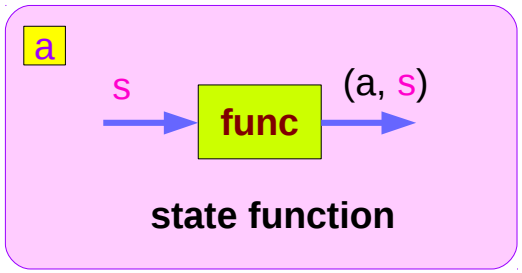
State String,

State Int,

State SomeLargeDataStructure,

and so forth.

Calling the type **State** is arguably a bit of a misnomer because the wrapped value is not the state itself but a **state processor** (**accessor function**: **runState**)



p :: **State** s a

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

State Packages

Control.Monad.**Trans**.**State**, **transformers** package. (focused here)

Control.Monad.**State**, **mtl** package.
Control.Monad.**State.Lazy**, **mtl** package.

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

The "**state**" function

Control.Monad.**Trans**.**State**, **transformers** package. (focused here)

no State constructor

but a "**state" function**

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

Control.Monad.**State**, **mtl** package

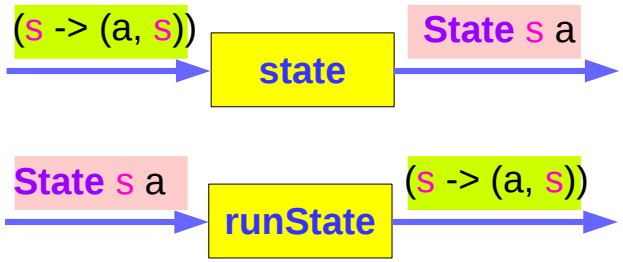
Implements the State in somewhat different way

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

runState function

State is a record with only one element,
whose type is a function (:: s -> (a, s))

runState converts a value of type **State** s a
to a function of this type (:: s -> (a, s))



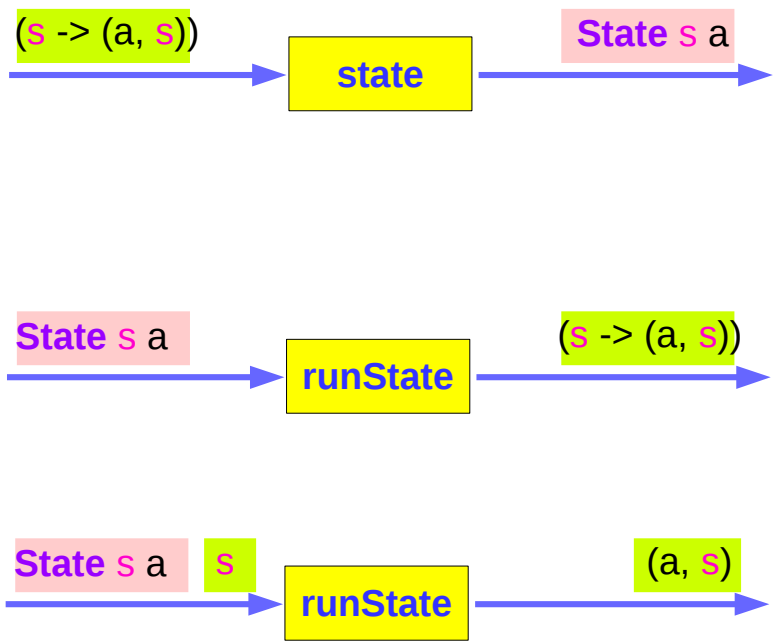

```
ghci> :t runState
runState :: State s a -> s -> (a, s)
```

Every time you apply **runState** to the value of type **State** s a,
the result is a function of type s -> (a, s).

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

https://stackoverflow.com/questions/3240947/understanding-haskell-accessor-functions

state & runState function



runState :: **State** s a -> s -> (a, s)

runState :: **State** s a -> s -> (a, s)

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

https://stackoverflow.com/questions/3240947/understanding-haskell-accessor-functions

Instantiating a State Monad

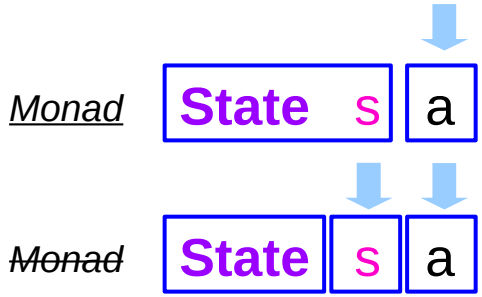
wrap a function type and give it a **name**. s -> (a, s)

State s can be made a *Monad instance*, for every type s

the *Monad instance* is **State** s, and not just **State**

(**State** can't be made an instance of Monad,
as it takes two type parameters, rather than one.)

State String,
State Int,
State SomeLargeDataStructure,
and so forth.




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

instance Monad (State s) where
    return implementation
    (>>=) implementation
```

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

Common implementation of **return** and **>>=**

instance Monad (**State** s) where

many different **State** monads,
one for each possible type of state -

State String,

State Int,

State SomeLargeDataStructure,

and so forth.

one implementation of

return and
(**>>=**);

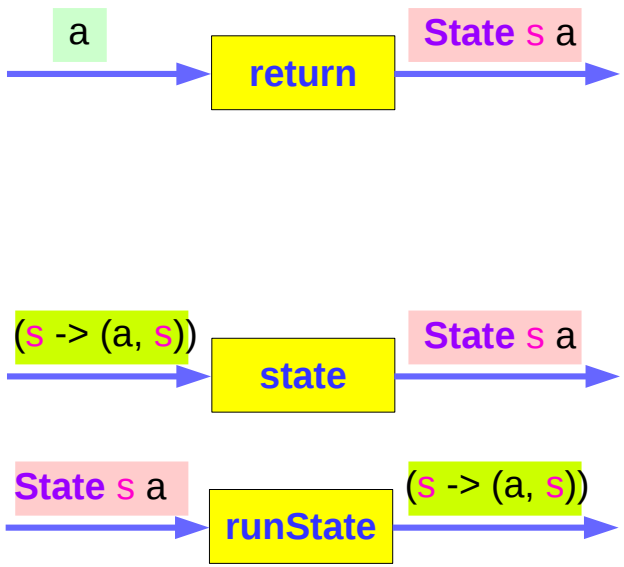
can handle these different (**State** s) monads
according to different choices of s.

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

return method

```
instance Monad (State s) where

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

State s a



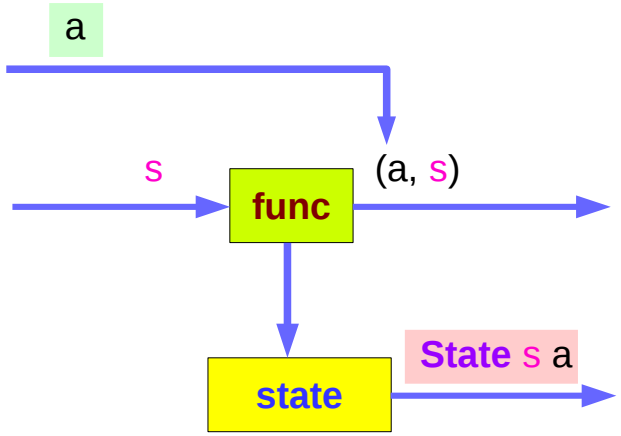
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

return method

```
instance Monad (State s) where

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

State s a

giving a value (x) to **return** results in a **state processor** function
which takes a state (s) and returns it unchanged (s),
together with value x we want to be returned.
Finally, the function is wrapped up by **state.**



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

State Monad Examples – return

```
runState (return 'X') 1

  ('X',1)
```

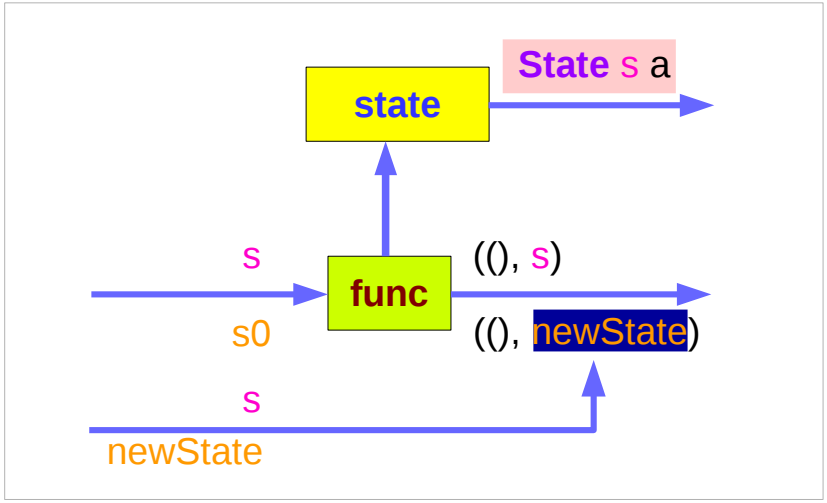
return
set the result value but leave the state unchanged.

```
return 'X' :: State Int Char
runState (return 'X') :: Int -> (Char, Int)
initial state = 1 :: Int
final value = 'X' :: Char
final state = 1 :: Int
result = ('X', 1) :: (Char, Int)
```

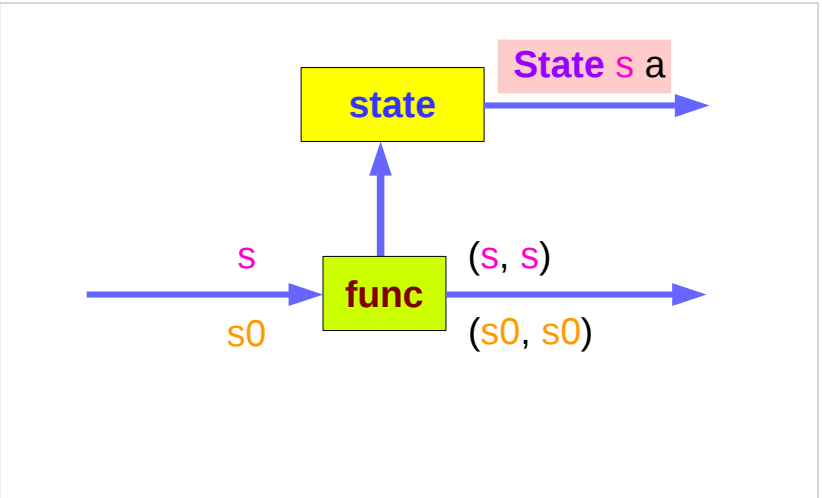
https://wiki.haskell.org/State_Monad

Setting and Getting the State

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



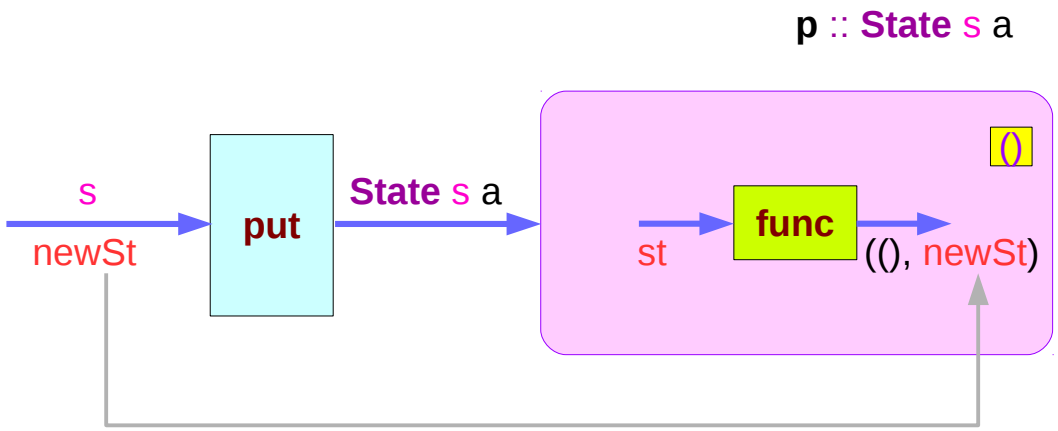

```
get :: State s s
get = state $ \s -> (s, s)

-- getting s
```



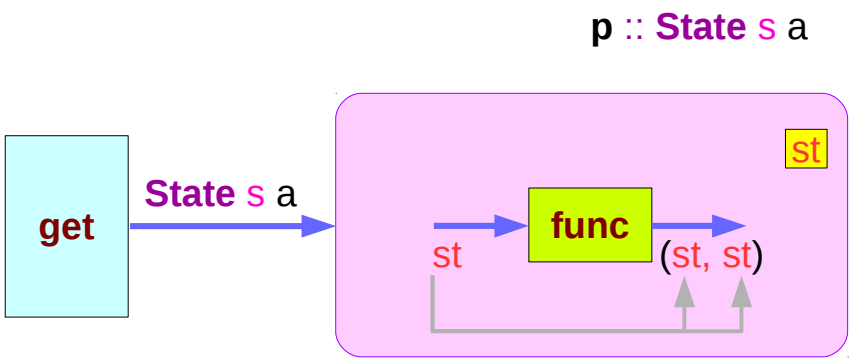
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

put and get

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



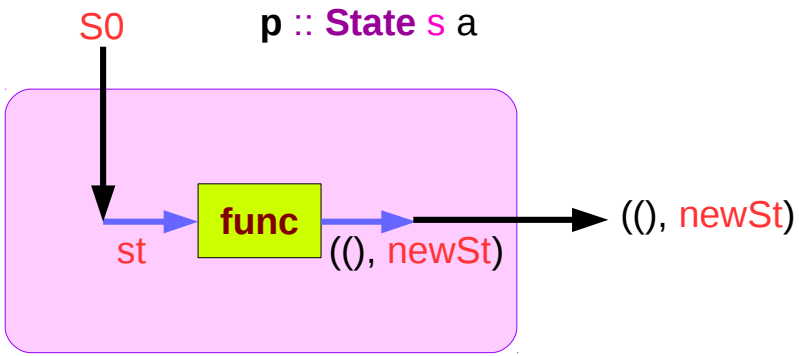

```
get :: State s s
get = state $ \s -> (s, s)
```



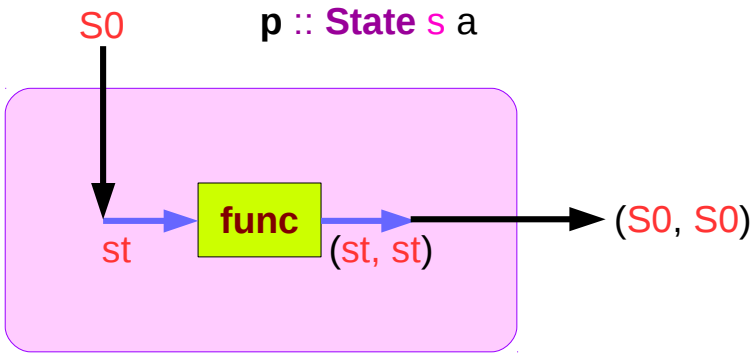
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

runState put and runState get

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

runState (put newSt) S0

((), newSt)
```



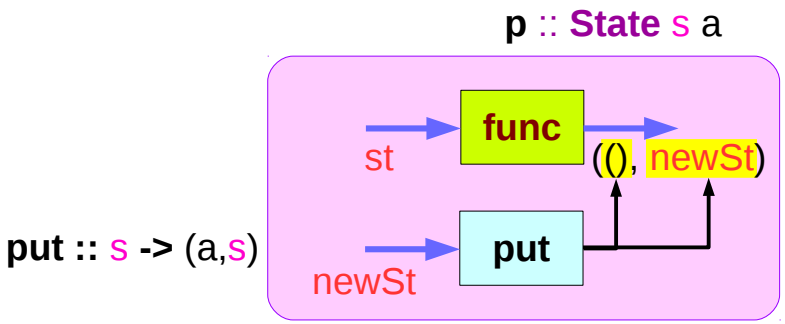

```
get :: State s s
get = state $  \s -> (s, s)

runState (get) S0

(S0, S0)
```



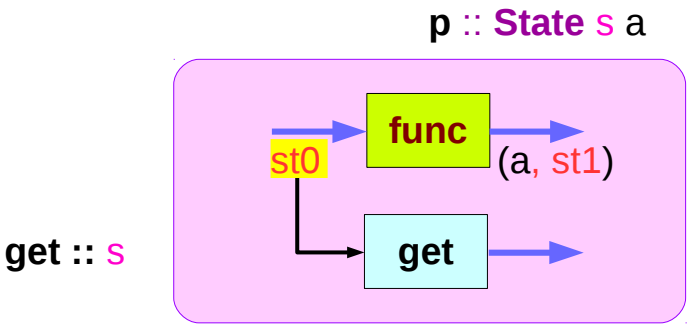
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

put and get viewed as inside functions

```
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

Inside the state monad

Whenever **sc** is a stateful computation
sc can be directly assigned to x, **inside** the state monad,

```
x <- sc
```

the result of the stateful computation **sc** is assigned to x
(like **evalState** is called with an initial state).

In order to check the current state, you can do

```
s <- get
```

and s will have the value of the current state.

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

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

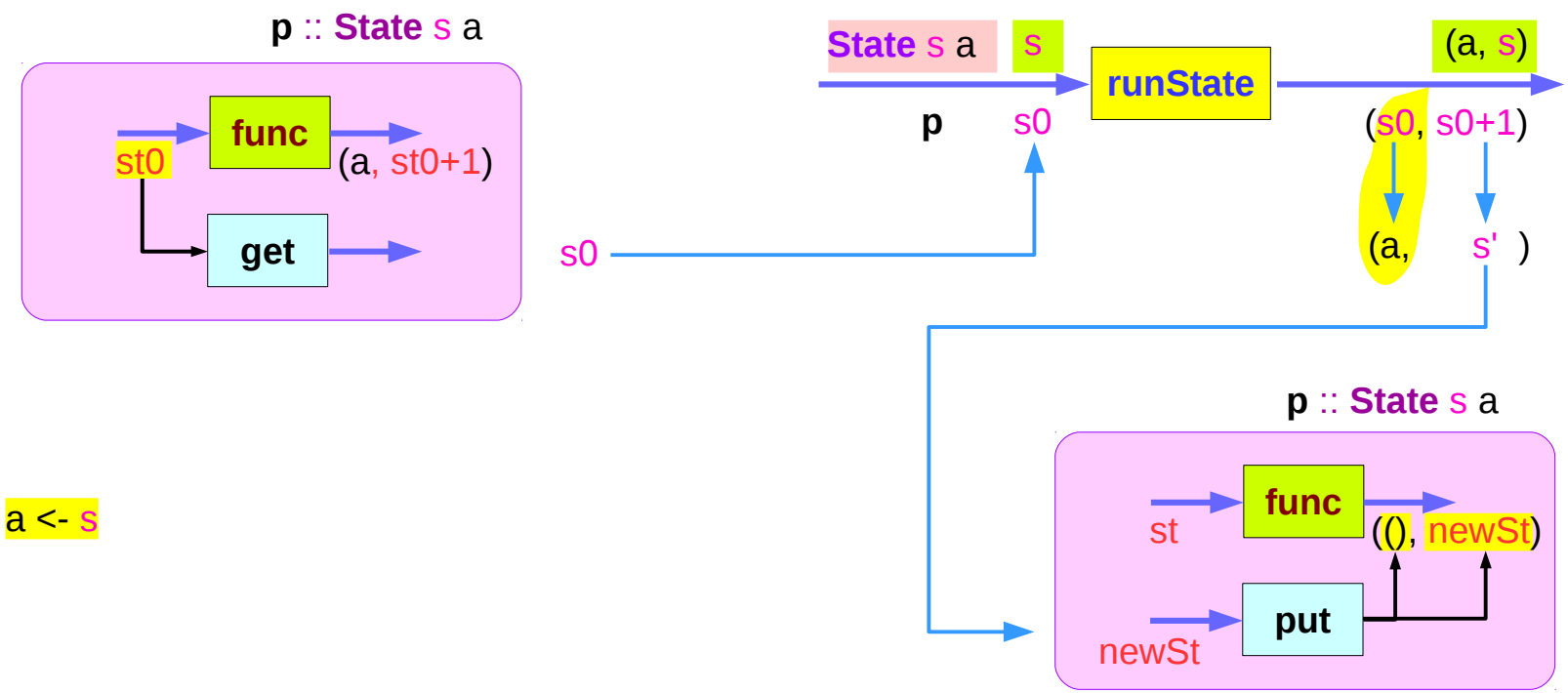
```
s0 <- get                          -- Read state
let (a,s') = runState s s0         -- Pass state to 's', get new state
put s'                             -- Save new state
```

```
a <- s
```

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

Redundant computation examples

```
s0 <- get                    -- Read state
let (a,s') = runState p s0   -- Pass state to p, get new state
put s'                       -- Save new state
```



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

Inside function examples

```
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

liftM

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

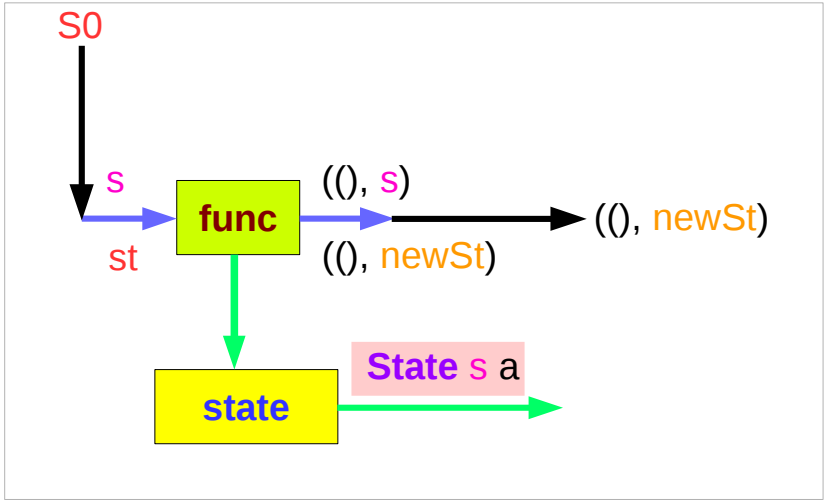
mapM

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


> map (x -> [x+1]) [1,2,3]
[[2],[3],[4]]


> mapM (x -> [x+1]) [1,2,3]
[[2,3,4]]
```

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

Setting the State

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



Given a wanted state newState,
put generates a **state processor**
which ignores whatever the state it receives,
and gives back the state we originally provided to put.
the same state

Since we don't care about the result (a) of this processor
(all we want to do is to change the state),
the first element of the tuple will be (),
the **universal placeholder value**.

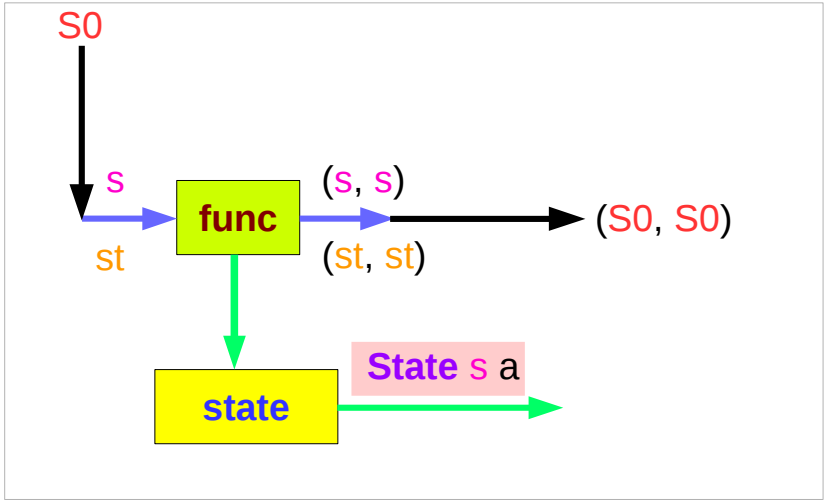
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Getting the State

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

The resulting **state processor** gives back the state st
it is given in both as a result and as a state.

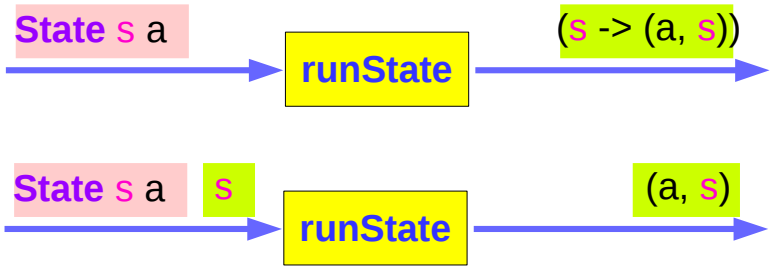
That means the state will remain unchanged,
and that a copy of it will be made available
for us to manipulate.



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

evalState and execState

runState
unwrap the **State** s a value
to get the actual **state processing function**
which is then applied to some initial state.

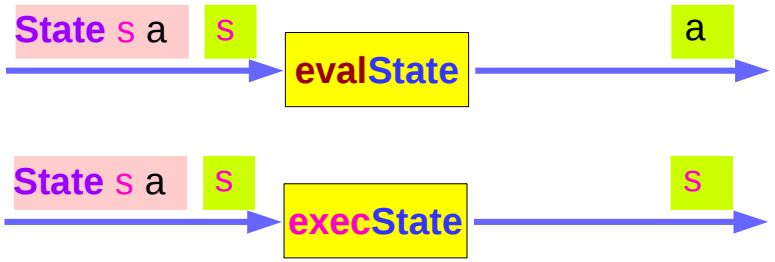


Given a **State** s a and an initial state s,

evalState only the result value

execState just the new state.

```
evalState :: State s a -> s -> a
evalState p s = fst (runState p s)

execState :: State s a  -> s -> s
execState p s = snd (runState p s)
```



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

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

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

Put and get in mtl packages

Return leaves the state unchanged and sets the result:

-- ie: (**return** 5) 1 -> (5,1)

```
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)

```
get :: State s s
get s = (s,s)
```

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

-- ie: (**put** 5) 1 -> ((),5)

```
put :: s -> State s ()
put x s = ((),x)
```

https://wiki.haskell.org/State_Monad

Unwrapped Implementation Examples (1)

Return leaves the state unchanged and sets the result:

```
-- ie: (return 5)   1 -> (5,1)

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)

get :: State s s
get s = (s,s)
```

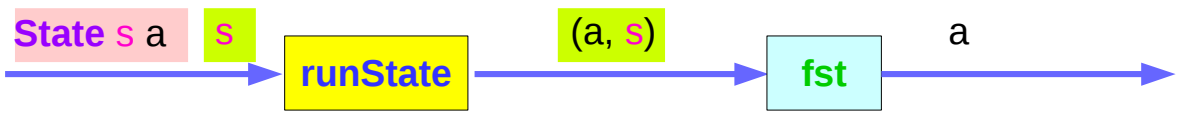
Put sets the result to () and sets the state:

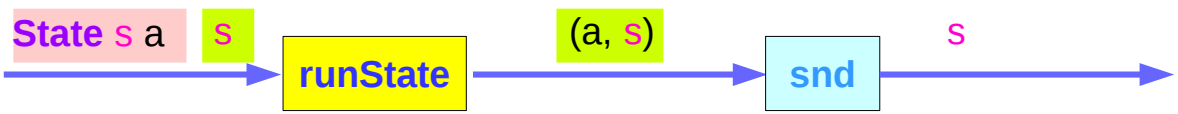
```
-- ie: (put 5)   1 -> ((),5)

put :: s -> State s ()
put x s = ((),x)
```

https://wiki.haskell.org/State_Monad

Unwrapped Implementation Examples (1)

```
evalState :: State s a -> s -> a
evalState act = fst . runState act

execState :: State s a -> s -> s
execState act = snd . runState act
```





https://wiki.haskell.org/State_Monad

Unwrapped Implementation Examples (2)

```
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
   ((),2)

runState (gets (+1)) 1
   (2,1)
```

```
evalState (gets (+1)) 1
   2

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

https://wiki.haskell.org/State_Monad

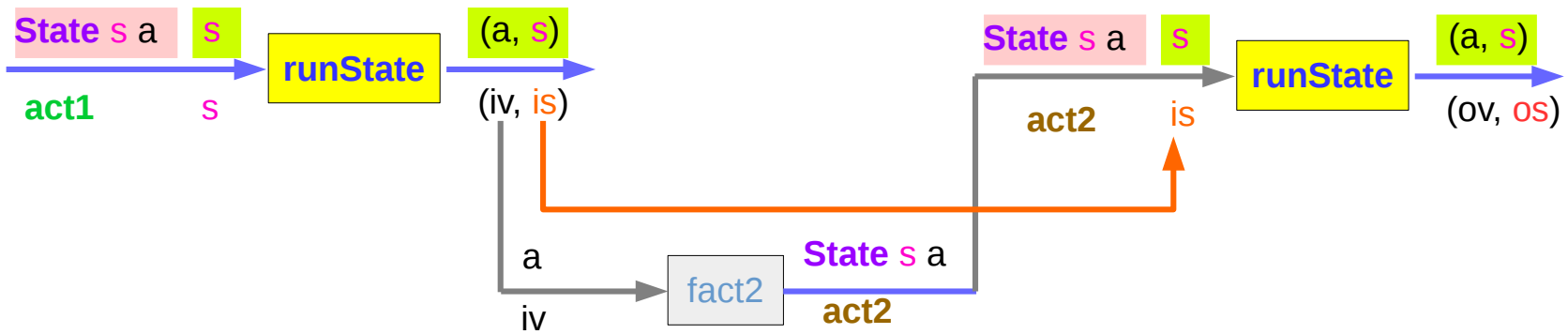
get & **put** :
functions inside the State monad

get :: s

put :: s -> (a, s)

Unwrapped Implementation Examples (3)

```
(>>=) :: State s a -> (a -> State s b) -> State s b
(act1 >>= fact2) s = runState act2 is
   where (iv, is) = runState act1 s
         act2 = fact2 iv
```



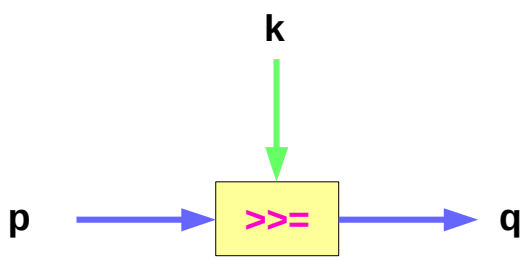
https://wiki.haskell.org/State_Monad

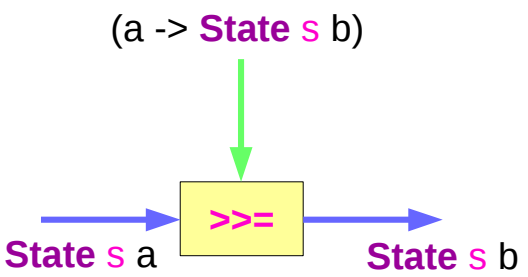
Function type of >>=

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```

p :: **State** s a
k :: (a -> **State** s b)

State s a ->	(a -> **State** s b) ->	**State** s b
p	**k**	**p >>= k**

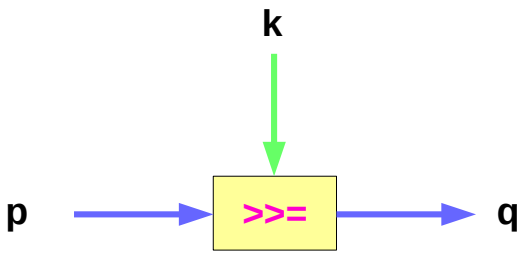




https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

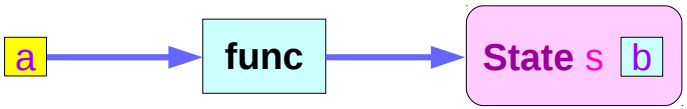
1st and 2nd arguments of >>= :

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```



p :: **State** s a

k :: (a -> **State** s b)



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

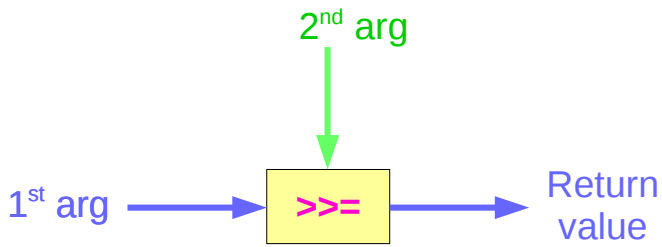
Binding operator >>=

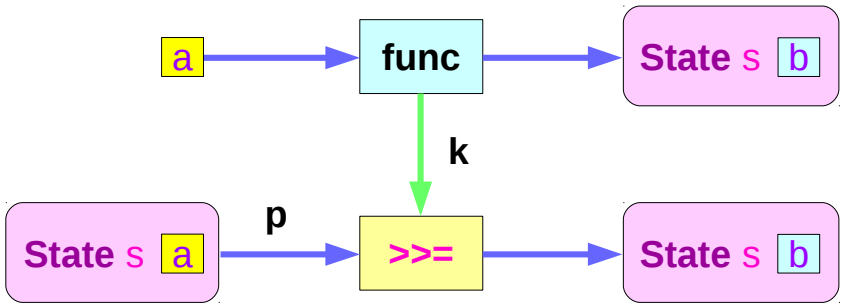
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```

p :: **State** s a — State Monad value

k :: (a -> **State** s b) — State Monad returning function

p >>= k = q

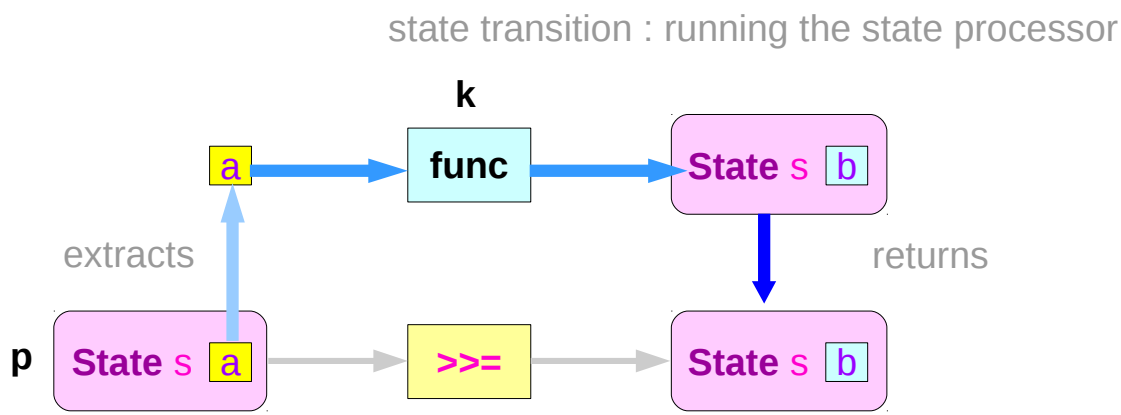




https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Conceptual computation flow of >>=

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```



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

Three Orthogonal Functions

Thinking of extraction : a slightly misleading intuition.

Nothing is being "extracted" from a monad.

The more *fundamental* definition of a monad
can be stated by <u>three</u> <u>orthogonal</u> <u>functions</u>:

fmap :: (a -> b) -> (m a -> m b)
return :: a -> m a
join :: m (m a) -> m a

m is a monad.

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

Three Orthogonal Functions and >>=

fmap :: (a -> b) -> (m a -> m b)
return :: a -> m a
join :: m (m a) -> m a

(a -> b) -> (m a -> m b)
(a -> m b) -> (m a -> m (m b))
(a -> m b) -> (m a -> m b)

how to implement (>>=) with these:

starting with arguments of type m a and a -> m b,

your only option is using **fmap** to get something of type m (m b),

(a -> m b) -> (m a -> m (m b))

after which you can use **join** to *flatten* the nested "layers" to get just m b.

(a -> m b) -> (m a ->m b)

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

Monad Law

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

join :: m (m a) -> m a
nothing is being taken "out" of the monad
as the computation going *deeper* into the monad,
with successive steps being *collapsed* into a single layer of the monad.

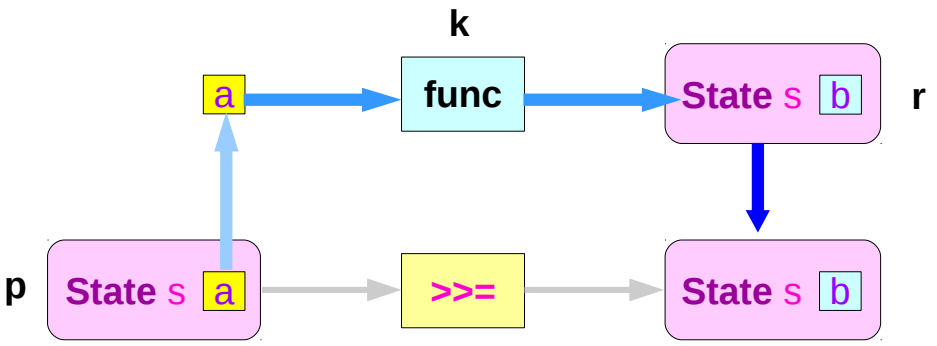
when **join** (m (m a) -> m a) is applied, it doesn't matter
as long as *the nesting order is preserved* (a form of *associativity*) and
that the *monadic* *layer* introduced by **return** does *nothing* (an *identity* value for **join**).

Left identity	**return** a **>>=** f	f a
Right identity	m **>>= return**	m
Associativity	(m **>>=** f) **>>=** g	m **>>=** (\x -> f x **>>** g)

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

Applying the state function to **p** and **r**

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p              -- p' :: s -> (a, s)
    k' = runState . k            -- k' :: a -> s -> (b, s)
```




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

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

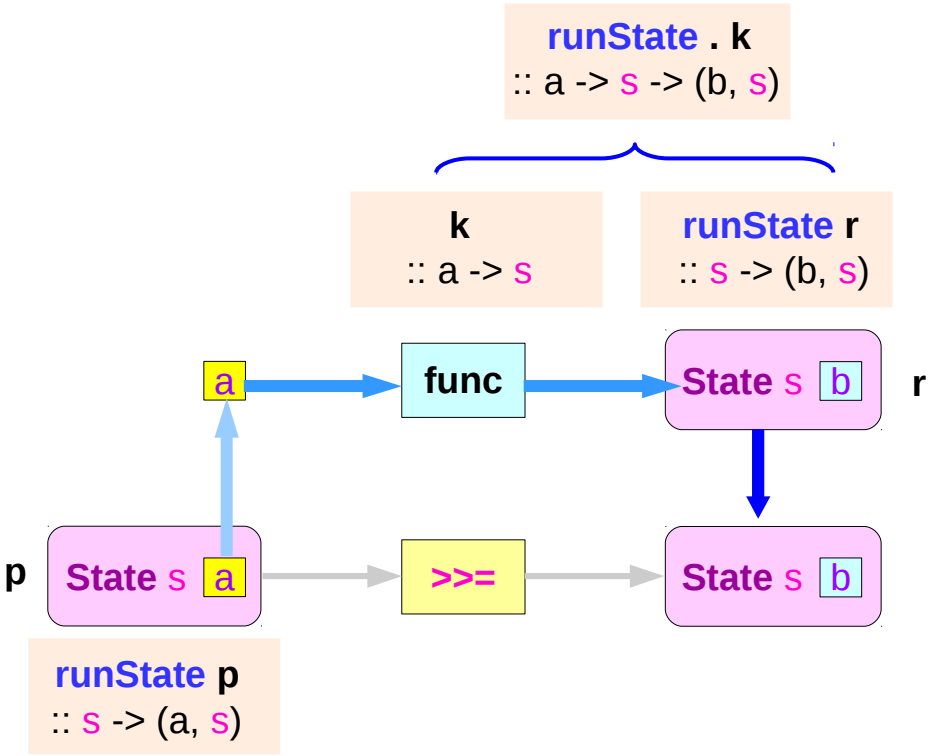
p'	:: s -> (a, s)	p' = runState p	State s a	p
r'	:: s -> (b, s)	r' = runState r	State s b	r

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

Applying **k** and the state function

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p            -- p' :: s -> (a, s)
    k' = runState . k          -- k' :: a -> s -> (b, s)
```

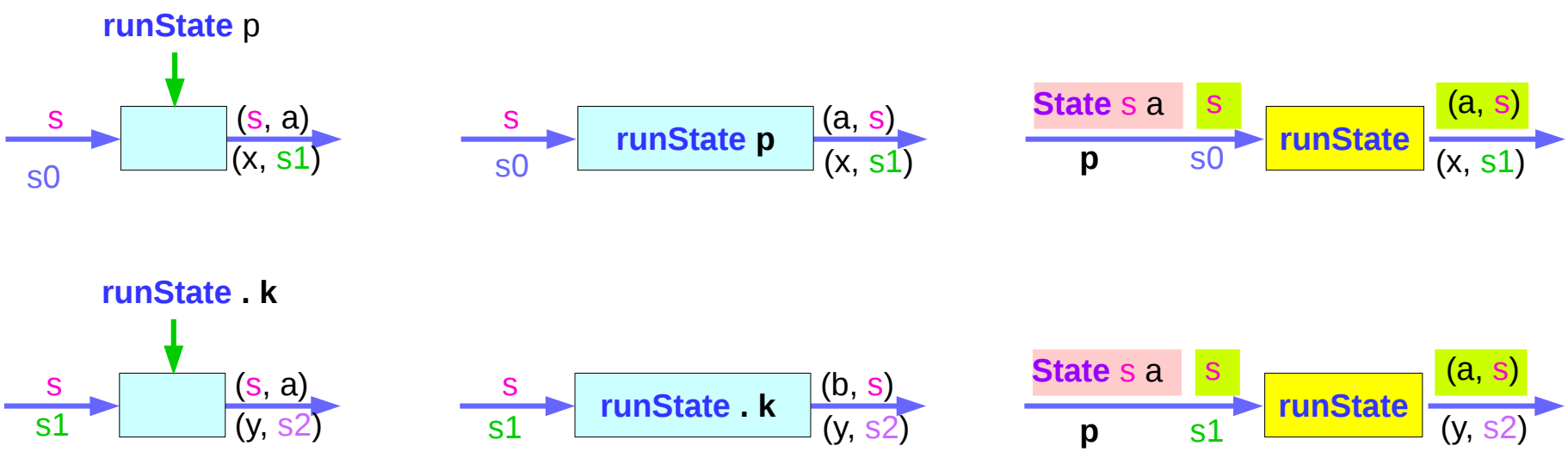
```
newtype State s a = State { runState :: s -> (a, s) }

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



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

Running the state processor

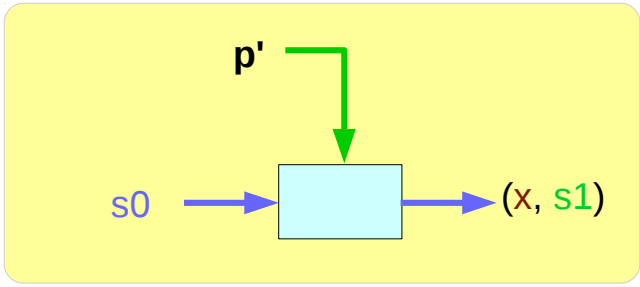
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p              -- p' :: s -> (a, s)
    k' = runState . k            -- k' :: a -> s -> (b, s)
```

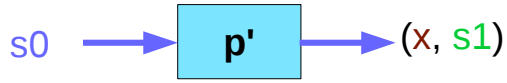


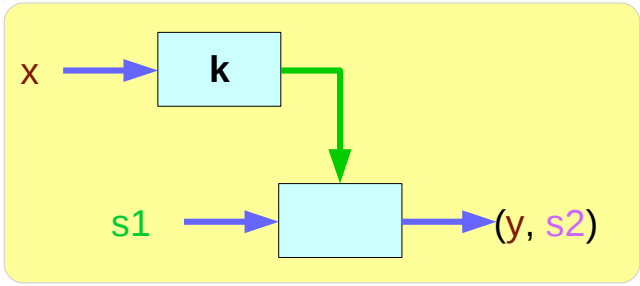
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

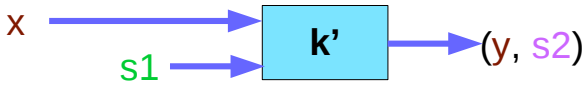
State Transition

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p              -- p' :: s -> (a, s)
    k' = runState . k            -- k' :: a -> s -> (b, s)
    q' s0 = (y, s2) where        -- q' :: s -> (b, s)
        (x, s1) = p' s0          -- (x, s1) :: (a, s)
        (y, s2) = k' x s1        -- (y, s2) :: (b, s)
    q = state q'
```







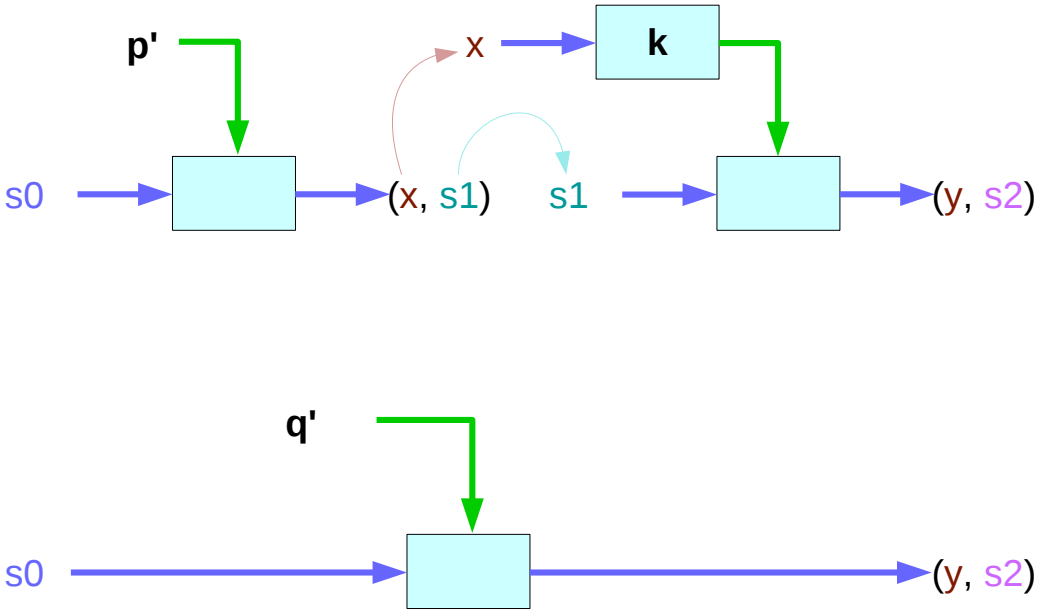


((), s0) ⟶ (x, S1) ⟶ (y, s2)

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

State Transition from s0 to s2

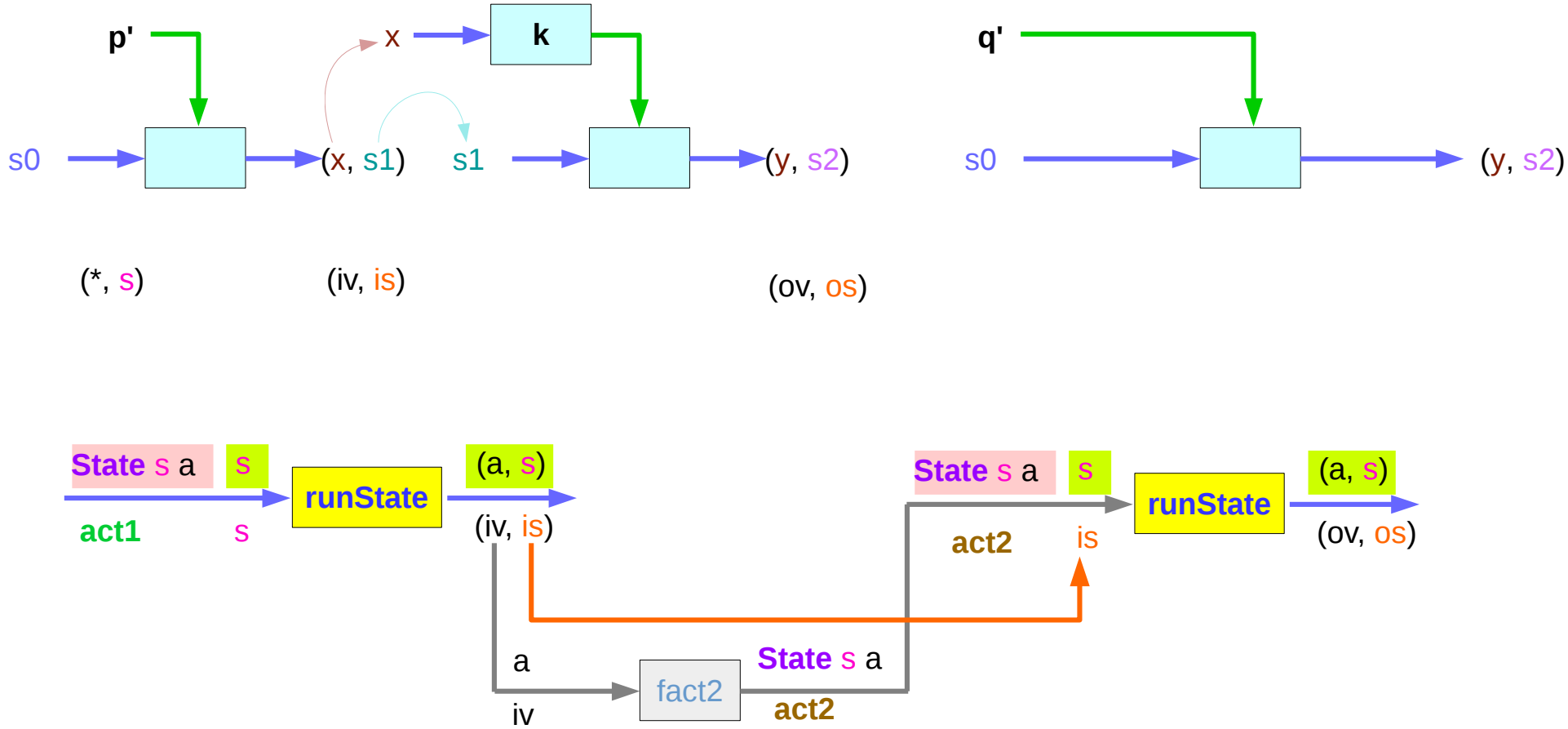
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p              -- p' :: s -> (a, s)
    k' = runState . k            -- k' :: a -> s -> (b, s)
    q' s0 = (y, s2) where        -- q' :: s -> (b, s)
        (x, s1) = p' s0          -- (x, s1) :: (a, s)
        (y, s2) = k' x s1        -- (y, s2) :: (b, s)
    q = state q'
```




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

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

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

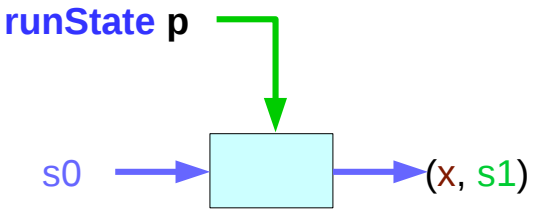
State Transition from s0 to s2



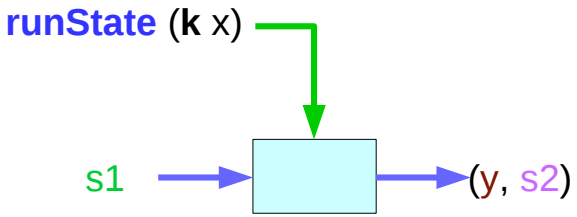
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Another implementation of >>=

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p  >>= k = state $ \ s0 ->
   let (x, s1) = runState p s0
   in runState (k x) s1
```



-- running the first processor on s0.



-- running the second processor on s1.

```
state ( \ s0 ->   (y, s2) )
```

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

Examples (1)

```
type ST = State -> State

type ST a = State -> (a, State)

Char -> State -> (Int, State)
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Examples (1)

```
instance Monad ST where
  -- return :: a -> ST a
  return x  =  \s -> (x,s)

  -- (>>=)  :: ST a -> (a -> ST b) -> ST b
  st >>= f  =  \s -> let (x,s') = st s in f x s'
```

>>= provides a means of sequencing state transformers:
st >>= **f** applies the state transformer **st** to an initial state s,
then applies the function **f** to the resulting value x
to give a second state transformer (**f** x),
which is then applied to the modified state s' to give the final result:

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Maybe Monad

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b



instance Monad Maybe where
   -- return      :: a -> Maybe a
   return x       =  Just x

   -- (>>=)       :: Maybe a -> (a -> Maybe b) -> Maybe b
   Nothing  >>= _ =  Nothing
   (Just x) >>= f =  f x
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Maybe Monad

a monad is a parameterised type m

that supports return and >>= functions of the specified types

m must be a parameterised type, rather than just a type

It is because of this declaration

that the do notation can be used to sequence Maybe values.

More generally, Haskell supports the use of this notation with any monadic type.

examples of types that are monadic,

the benefits that result from recognising and exploiting this fact.

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

List Monad

The maybe monad provides a simple model of computations
that can fail,
a value of type Maybe a is either Nothing (failure)
the form Just x for some x of type a (success)

The list monad generalises this notion,
by permitting multiple results in the case of success.

More precisely, a value of [a] is
either the empty list [] (failure)
or the form of a non-empty list [x1,x2,...,xn] (success)
for some xi of type a

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

List Monad

```
instance Monad [] where
  -- return :: a -> [a]
  return x  =  [x]

  -- (>>=)  :: [a] -> (a -> [b]) -> [b]
  xs >>= f  =  concat (map f xs)
```

return converts a value into a *successful* result containing that value

>>= provides a means of *sequencing* computations
that may produce *multiple results*:

xs >>= **f** applies the function **f** to each of the *results* in the list **xs**
to give a *nested list* of *results*,
which is then concatenated to give a *single list* of *results*.

(Aside: in this context, [] denotes the list type [a] without its parameter.)

```
instance Monad ST where
  -- return :: a -> ST a
  return x  =  \s -> (x,s)

  -- (>>=)  :: ST a -> (a -> ST b) -> ST b
  st >>= f  =  \s -> let (x,s') = st s in f x s'
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Examples (1)

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys =  do x <- xs
                  y <- ys
                  return (x, y)
```

do

this function returns all possible ways of pairing elements from two lists

each possible value x from the list xs, and
each value y from the list ys, and
return the pair (x,y).

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Examples (1)

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys =  do x <- xs
                  y <- ys
                  return (x, y)
```

do

```
pairs xs ys = [(x,y) | x <- xs, y <- ys]
```

comprehension

In fact, there is a formal connection
between the do notation and the comprehension notation.
Both are simply different shorthands
for repeated use of the >>= operator for lists.

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Simple Examples (1)

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

 a1 >> a2 takes the actions a1 and a2 and
returns the mega action which is
a1-then-a2-returning-the-value-returned-by-a2.



> type State = Int

> fresh :: ST0 Int
> fresh =  S0 (\n -> (n, n+1))

> wtf1 = fresh >>
>        fresh >>
>        fresh >>
>        fresh

ghci> apply0 wtf1 0
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Simple Examples (2)

```
return :: a -> ST0 a

> wtf2 = fresh >>= \n1 ->
>        fresh >>= \n2 ->
>        fresh >>
>        fresh >>
>        return [n1, n2]

> wtf2' = do { n1 <- fresh;
>              n2 <- fresh;
>              fresh ;
>              fresh ;
>              return [n1, n2];
>            }
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Simple Examples (3)

```
ghci> apply0 wtf2 0



> wtf3 = do n1 <- fresh
>         fresh
>         fresh
>         fresh
>         return n1
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Dice Examples

to generate Int dice - result : a number between 1 and 6
throw results from a pseudo-random generator of type StdGen.

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

State StdGen Int

StdGen -> (Int, StdGen)

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

randomR

the StdGen type : an instance of **RandomGen**

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

assume a is Int and g is StdGen

the type of **randomR**

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

already have a **state processing function**

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

randomR

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

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

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

Some Examples (1)

```
module StateGame where

import Control.Monad.State

-- Example use of State monad
-- Passes a string of dictionary {a,b,c}
-- Game is to produce a number from the string.
-- By default the game is off, a C toggles the
-- game on and off. A 'a' gives +1 and a b gives -1.
-- E.g
-- 'ab'    = 0
-- 'ca'    = 1
-- 'cabca' = 0
-- State = game is on or off & current score
--       = (Bool, Int)
```

https://wiki.haskell.org/State_Monad

Some Examples (2)

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

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

https://wiki.haskell.org/State_Monad

Some Examples (3)

```
playGame (x:xs) = do
    (on, score) <- get
    case x of
        'a' | on -> put (on, score + 1)
        'b' | on -> put (on, score - 1)
        'c'      -> put (not on, score)
        _        -> put (on, score)
    playGame xs

startState = (False, 0)

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

https://wiki.haskell.org/State_Monad

References

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