Imperative Programming with The State Monad

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

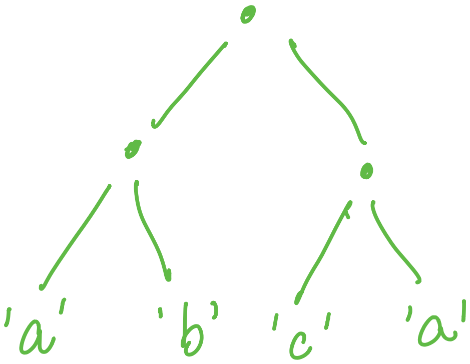
A Tree Datatype

A tree with data at the **leaves**

```
data Tree a
   = Leaf a
   | Node (Tree a) (Tree a)
   deriving (Eq, Show)
```

Here's an example `Tree Char`

```
charT :: Tree Char
charT = Node
            (Node
                (Leaf 'a')
                (Leaf 'b'))
            (Node
                (Leaf 'c')
                (Leaf 'a'))
```



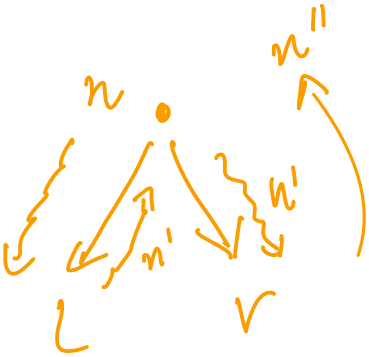
Lets Work it Out!

Write a function to add a *distinct* label to each *leaf*

```
label :: Tree a -> Tree (a, Int)
label = ???
label :: Tree a -> Tree (a, Int)
label t       = t'
  where
      (_, t') = (helper 0 t)
```

old count   new count   output tree

```
helper :: Int -> (Int, Tree (a, Int))
helper n (Leaf x)   = (n+1, Leaf (x, n))
helper n (Node l r) = (n'', Node l' r')
  where
      (n', l')      = helper n l
      (n'', r')     = helper n' r
```



EXERCISE

Now, modify `label` so that you get new numbers for each `letter` so,

```
>>> keyLabel (Node (Node (Leaf 'a') (Leaf 'b')) (Node (Leaf 'c')
(Leaf 'a')))
    (Node
        (Node (Leaf ('a', 0)) (Leaf ('b', 0)))
        (Node (Leaf ('c', 0)) (Leaf ('a', 1))))
```

That is, a *separate* counter for each *key* `a` , `b` , `c` etc.

HINT Use the following `Map k v` type

```haskell
-- | The empty Map
empty :: Map k v

-- | 'insert key val m` returns a new map that extends 'm'
--   by setting `key` to `val`
insert :: k -> v -> Map k v -> Map k v

-- | 'findWithDefault def key m' returns the value of `key`
--   in `m`  or `def` if `key` is not defined
findWithDefault :: v -> k -> Map k v -> v
```

Common Pattern?

Both the functions have a common "shape"

```
helper ::  OldInt -> (NewInt, NewTree)

keyhelp :: OldMap -> (NewMap, NewTree)
```

"global val"

If we generally think of `Int` and `Map Char Int` as **global state**

```
OldState -> (NewState, NewVal)
```

"old-global" → ("new/upd global", Result)

State Transformers

Lets capture the above "pattern" as a type

1. A **State** Type

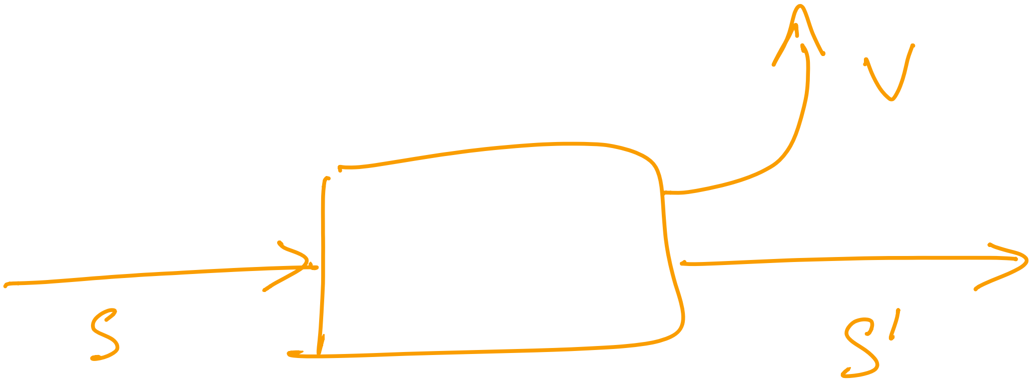
```
type State = ... -- lets "fix" it to Int for now...
```

2. A **State Transformer** Type

```
data ST a = STC (State -> (State, a))
```

A *state transformer* is a function that

- takes as input an **old** `s :: State`
- returns as output a **new** `s' :: State` and **value** `v :: a`



Executing Transformers

Lets write a function to *evaluate* an `ST a`

```
evalState :: State -> ST a -> a
evalState = ???
```

:: (State, a)

evalState s (STC f) = snd (f s)

result

State

"key"

State → (State, a)

"portal" "key"

QUIZ

What is the value of `quiz` ?

100 101 102

```
st :: ST [Int]
st = STC (\n -> (n+3, [n, n+1, n+2]))
```

f

```
quiz = evalState 100 st
```

s

State → ST [Int] → [Int]

evalState :: State → ST a → a

evalState s (STC f) = snd (f s)

100 (\n→...)

A. ~~103~~

? **B.** [100, 101, 102]

C. ~~(103, [100, 101, 102])~~

? **D.** [0, 1, 2]

E. Type error

Lets Make State Transformer a Monad!

```
instance Monad ST where
  return :: a -> ST a
  return = returnST

  (>>=)  :: ST a -> (a -> ST b) -> ST b
  (>>=) = bindST
```

Monad m

return :: a → m a

(>>=) :: m a → (a → m b) → m b

EXERCISE: Implement `returnST`!

What is a valid implementation of `returnST`?

```
type State = Int
data ST a  = STC (State -> (State, a))

returnST :: a -> ST a
returnST = ???
```

returnST v = STC (\s → (s, v))

old

new

What is `returnST` doing ?

`returnST v` is a *state transformer* that … ???

(Can someone suggest an explanation in English?)

HELP

Now, lets implement `bindST` !

```haskell
type State = Int

data ST a  = STC (State -> (State, a))

bindST :: ST a -> (a -> ST b) -> ST b
bindST = ???
```

What is `bindST` *doing ?*

`bindST v` is a *state transformer* that … ???

(Can someone suggest an explanation in English?)

`bindST` *lets us **sequence** state transformers*

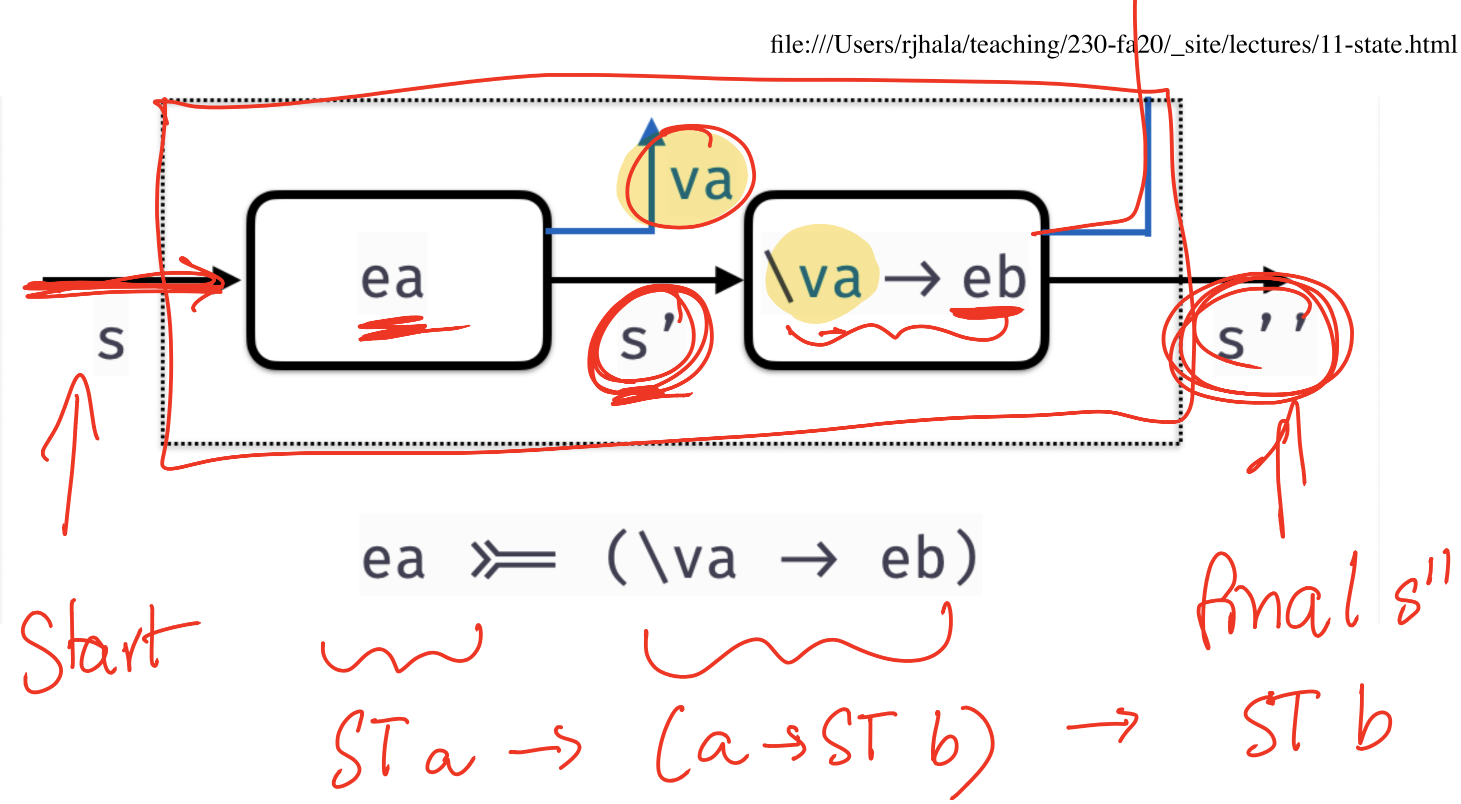
```
(>>=) :: ST0 a -> (a -> ST0 b) -> ST0 b
sta >>= f = STC (\s ->
                  let (s', va)  = runState sta s
                      stb       = f va
                      (s'', vb) = runState stb s'
                  in
                      (s'', vb)
               )
```

`st >>= f`

1. Applies transformer `st` to an initial state `s`
   - to get output `s'` and value `va`
2. Then applies function `f` to the resulting value `va`
   - to get a *second* transformer
3. The *second* transformer is applied to `s'`
   - to get final `s''` and value `vb`

OVERALL: Transform `s` to `s''` and produce value `vb`

vb ::b



Lets Implement a Global Counter

The (counter) `State` is an `Int`

```
type State = Int
```

A function that *increments* the counter to *return* the `next Int`.

```
next :: ST String
next = STC (\s -> (s+1, show s))
```

`next` is a *state transformer* that that returns `String` values

QUIZ

Recall that

```
evalState :: State -> ST a -> a
evalState s (STC st) = snd (st s)
```

```
next :: ST String
next = STC (\s -> (s+1, show s))
```

STRING

What does `quiz` evaluate to?

```
quiz = evalState 100 next
```

[✓ **A.** "100"]

B. "101"

C. "0"

D. "1"

~~**E.** (101, "100")~~

QUIZ

Recall the definitions

```
evalState :: State -> ST a -> a
evalState s (STC st) = snd (st s)

next :: ST String
next = STC (\s -> (s+1, show s))
```

Now suppose we have

string

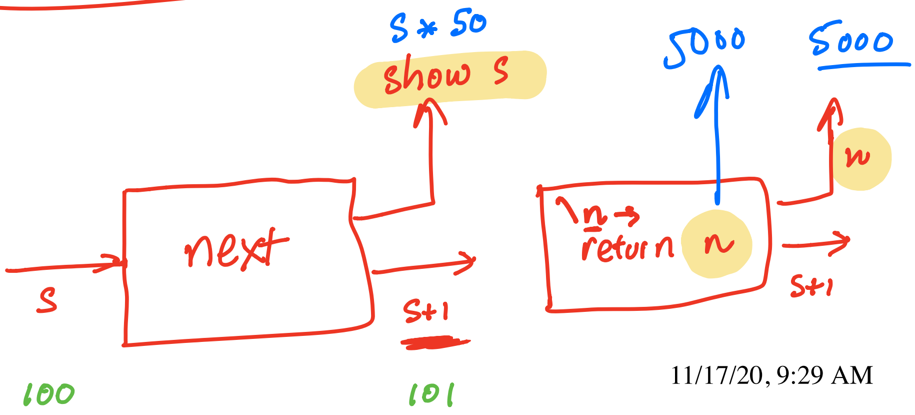
```
wtf1 = ST Int
wtf1 = next >>= \n ->
         return n
```

next >>= (\n -> return n)

What does `quiz` evaluate to?

```
quiz = evalState 100 wtf1
```

A. `100`



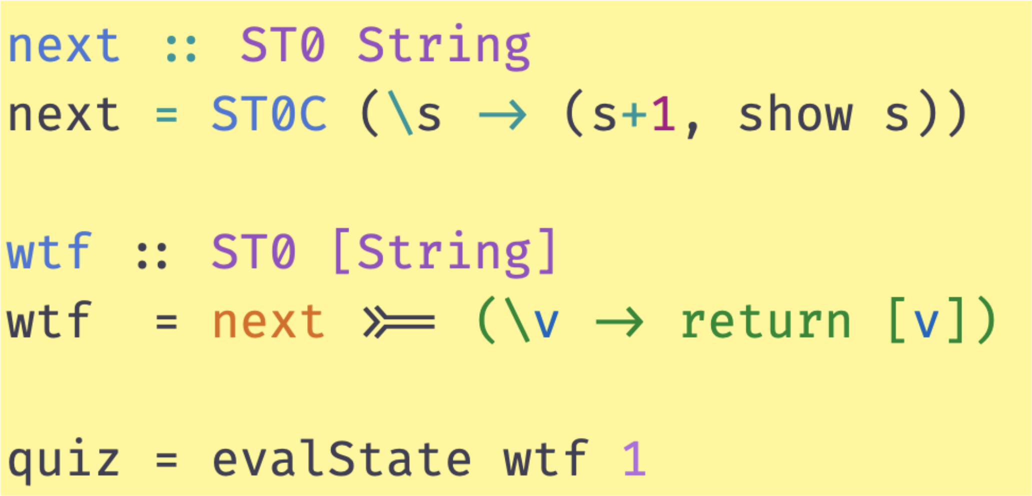
B. `101`

C. `0`

D. `1`

~~**E.** `(101, 100)`~~

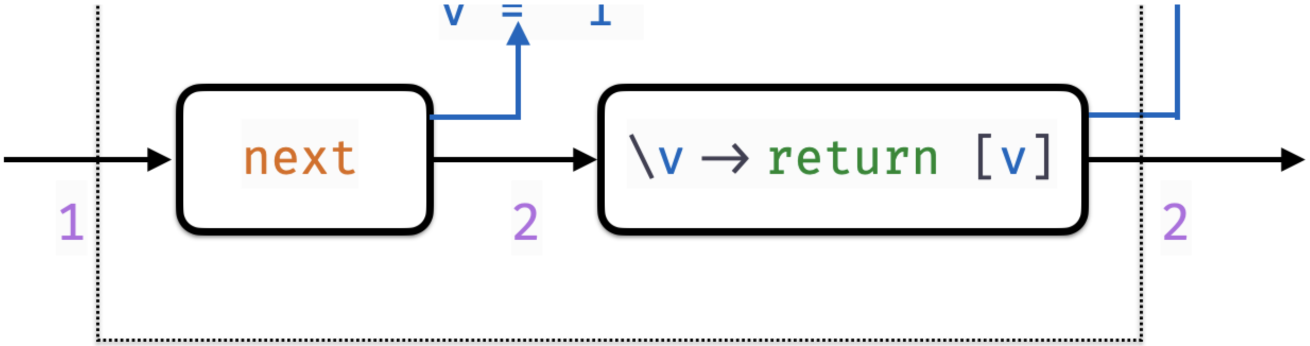
Example



```
next :: ST0 String
next = ST0C (\s → (s+1, show s))

wtf :: ST0 [String]
wtf  = next ≫= (\v → return [v])

quiz = evalState wtf 1
```

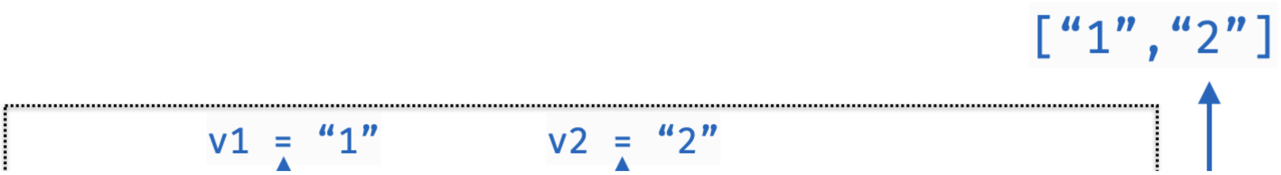


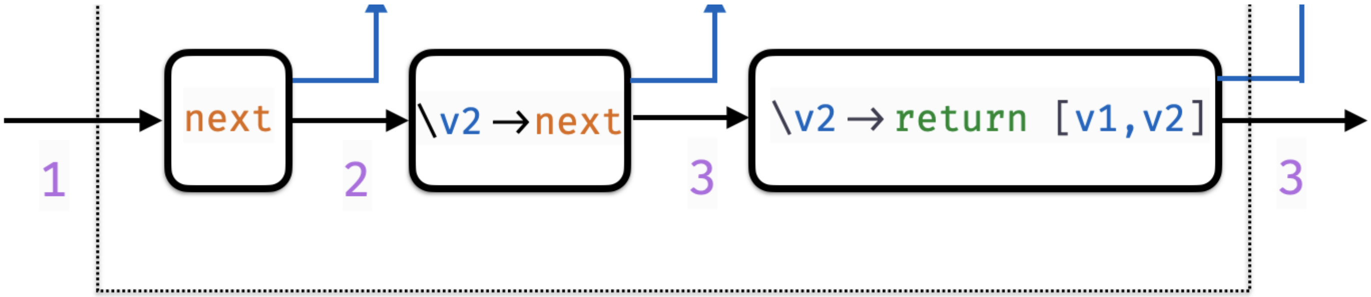
Example

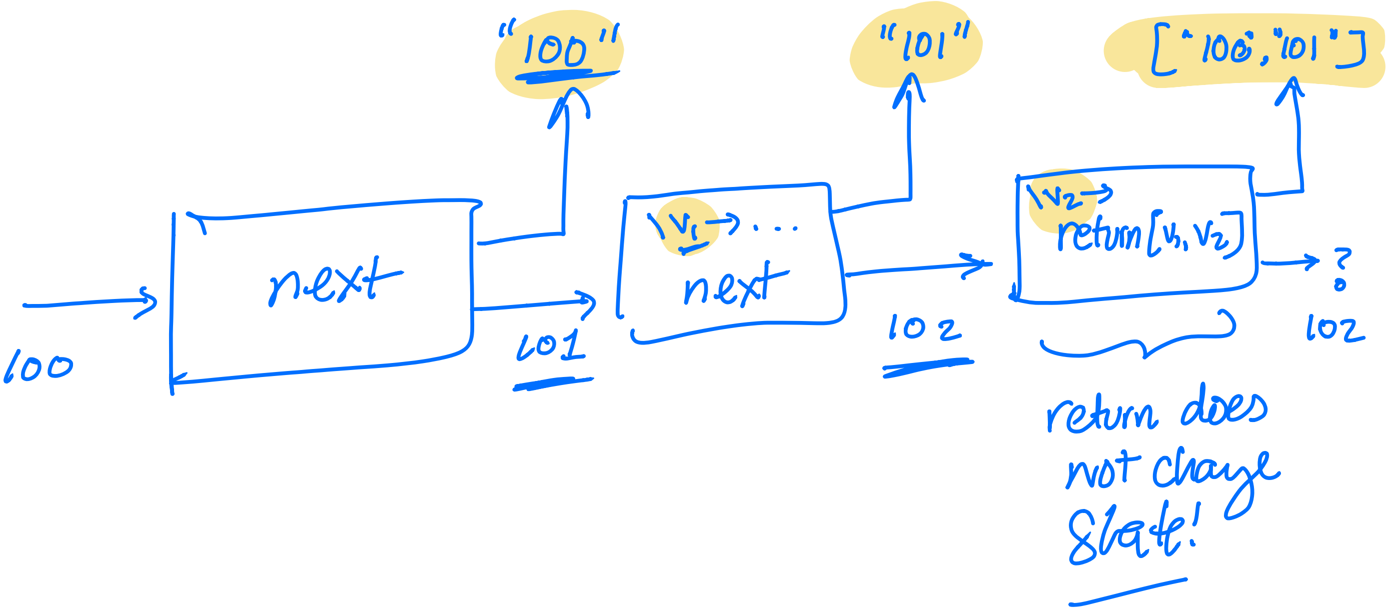
```
next :: ST0 String
next = ST0C (\s -> (s+1, show s))

wtf  :: ST0 [String]
wtf   = next >>= (\v1 -> next >>= (\v2 -> return [v1, v2]))

quiz = evalState wtf 1
```







QUIZ

Consider a function `wtf2` defined as

```
wtf2 = next >>= \n1 ->
         next >>= \n2 ->
           next >>= \n3 ->
             return [n1, n2, n3]
```

What does `quiz` evaluate to?

```
quiz = evalState 100 wtf
```

A. Type Error!