State Transformers

Lets capture the above “pattern” as a type

1. A **State** Type

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

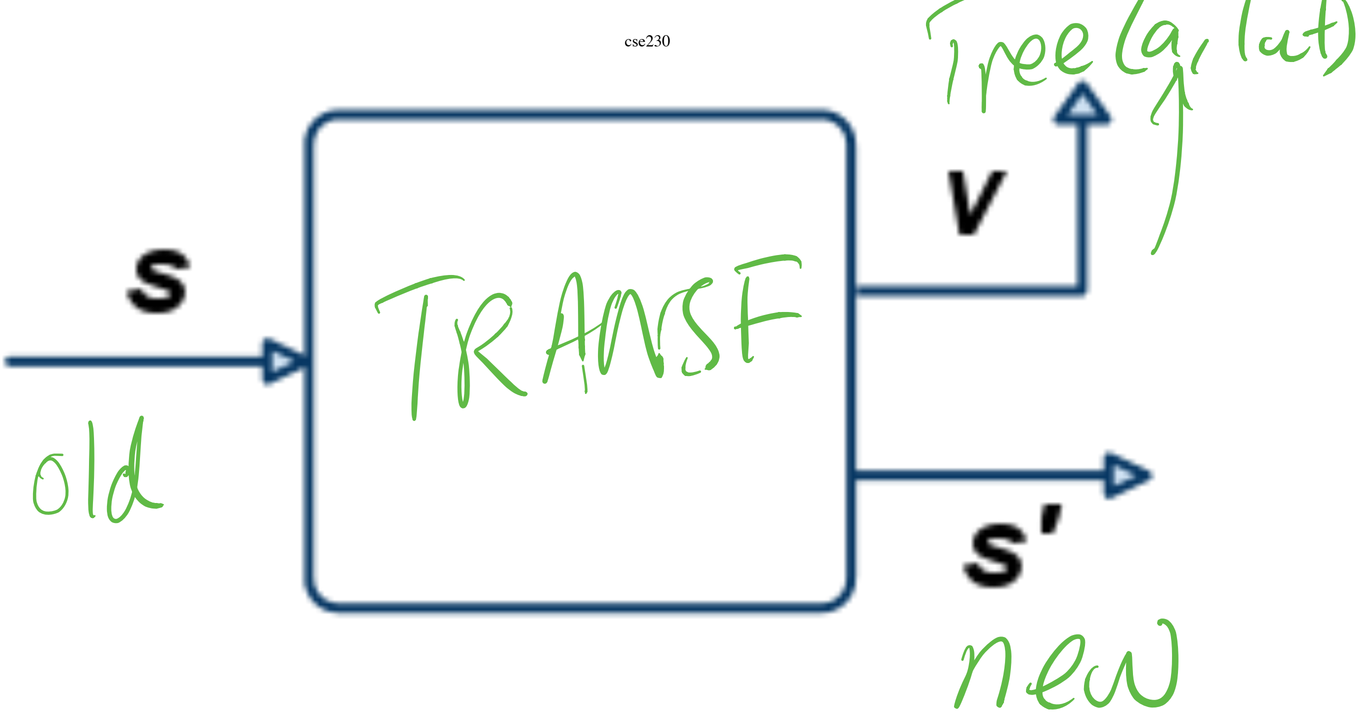
2. A **State Transformer** Type

OLD NEW VAL

```
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`


Tree (a, Int)
v
s
old
TRANSF
s'
new

Executing Transformers

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

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

QUIZ

What is the value of `quiz` ?

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

quiz = evalState100 st
```

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
```

EXERCISE: Implement `returnST`!

What is a valid implementation of `returnST` ?

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

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

What is `returnST` *doing ?*

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

(Can someone suggest an explanation in English?)

HELP

Now, lets implement `bindST`!

```
type State = Int

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

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

What is `returnST` doing ?

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

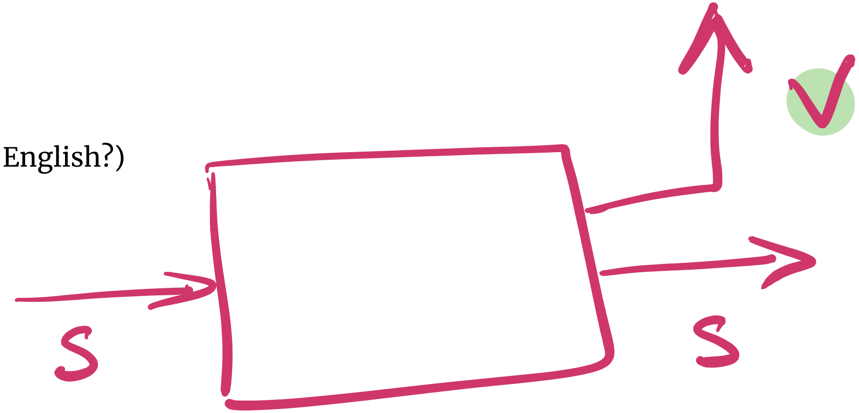
(Can someone suggest an explanation in English?)

What is `returnST` *doing?*

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

(Can someone suggest an explanation in English?)

return v ≡

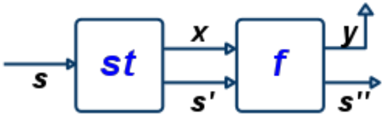


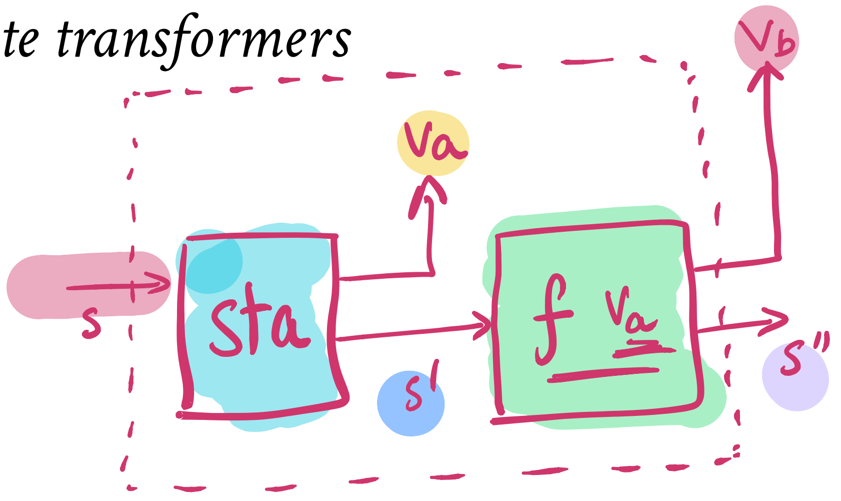
`bindST` *lets us* ***sequence*** *state transformers*

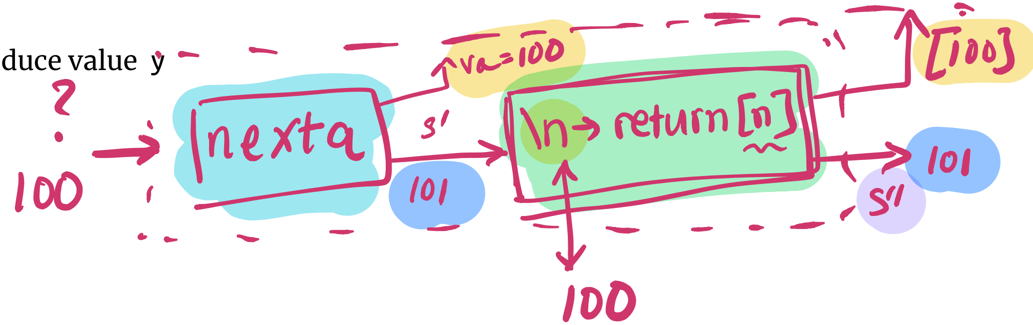
```
st >>= f
```

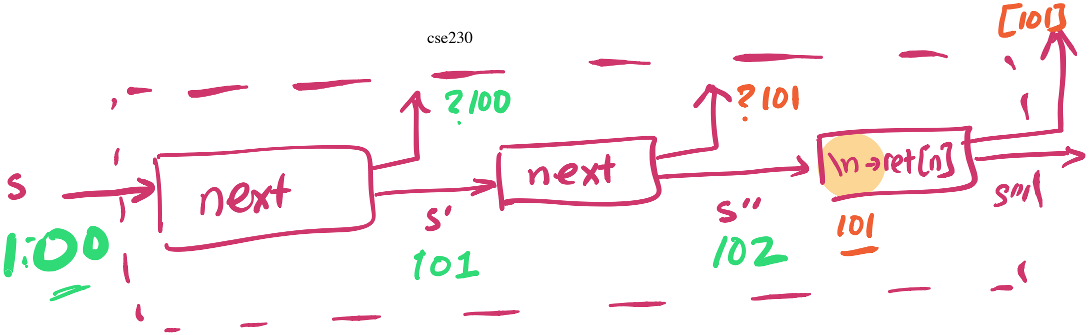
1. Applies transformer `st` to an initial state `s`
   - to get output `s'` and value `x`
2. Then applies function `f` to the resulting value `x`
   - to get a *second* transformer
3. The *second* transformer is applied to `s'`
   - to get final `s''` and value `y`

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









Lets Implement a Global Counter

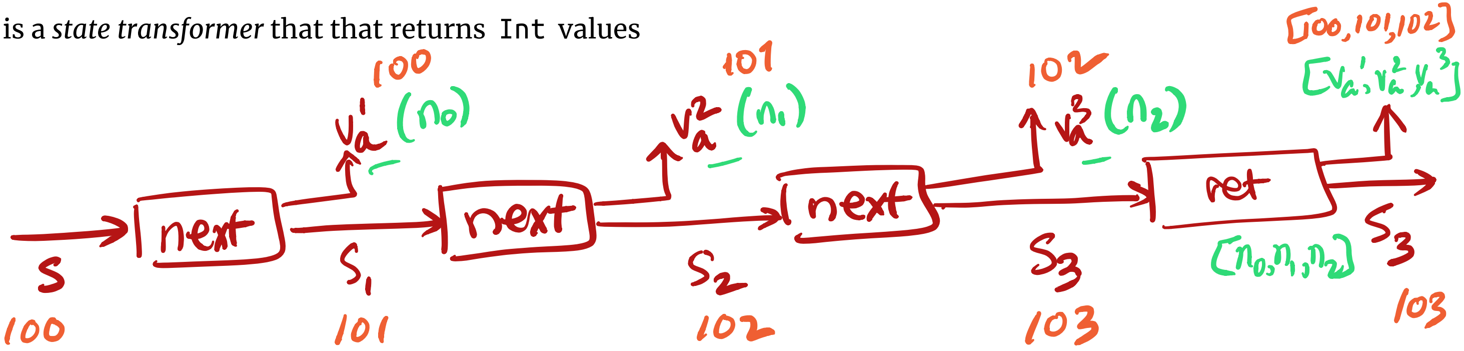
The (counter) `State` is an `Int`

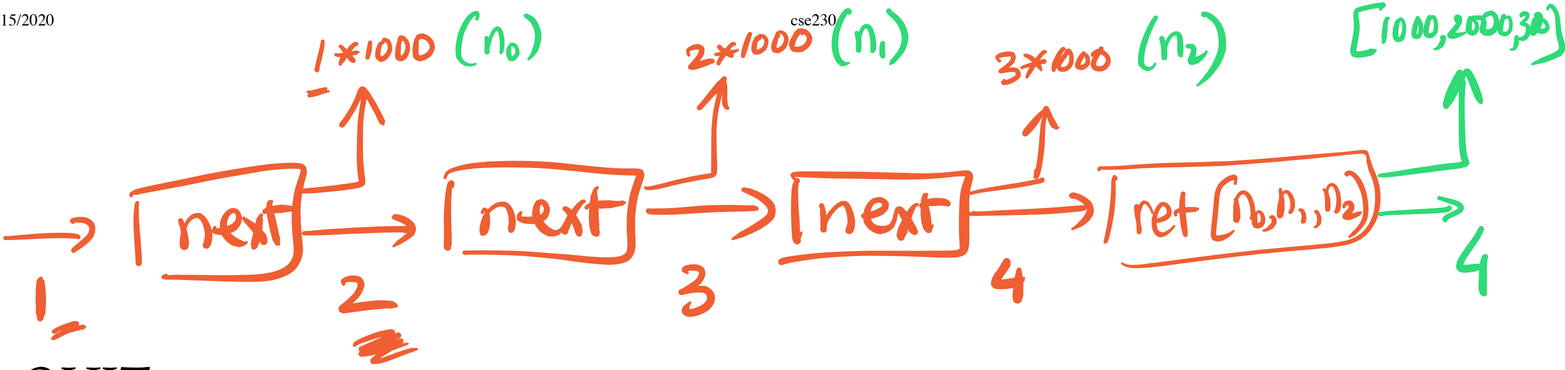
```
type State = Int
```

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

```
next :: ST Int
next = STC (\old -> let new = old + 1 in (new, old))
```

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





QUIZ

Recall that

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

```
next :: ST Int
next = STC (\n -> (n+1, n))
```

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 Int
next = STC (\n -> (n+1, n))
```

Now suppose we have

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

What does `quiz` evaluate to?

```
quiz = evalState 100 wtf1
```

A. `100`

B. `101`

C. `0`

D. `1`

E. `(101, 100)`

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!

B. [100, 100, 100]

C. [0, 0, 0]

D. [100, 101, 102]

E. [102, 102, 102]

Chaining Transformers

`>>=` lets us *chain* transformers into *one* big transformer!

So we can define a function to *increment the counter by 3*

```
-- Increment the counter by 3
next3 :: ST [Int, Int]
next3 = next >>= \n1 ->
          next >>= \n2 ->
            next >>= \n3 ->
                return [n1,n2,n3]
```

And then sequence it *twice* to get