Imperative Programming with The State Monad

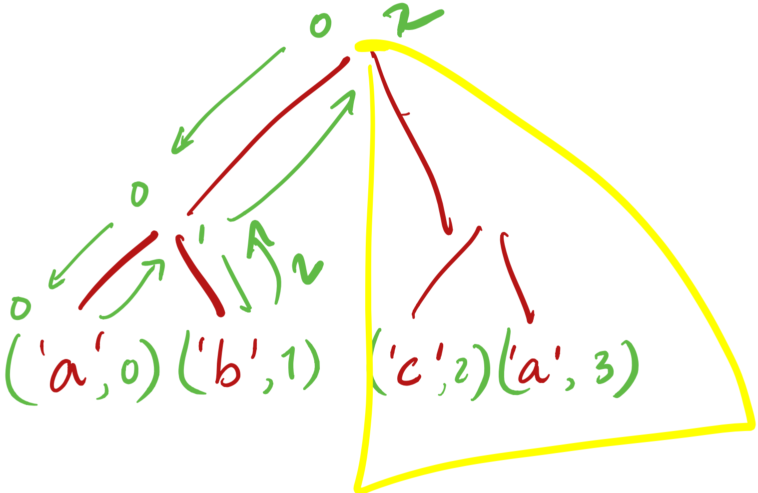
A Tree Datatype

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 = ???
```

such that

```
>>> label charT
Node
    (Node
        (Leaf ('a', 0))
        (Leaf ('b', 1)))
    (Node
        (Leaf ('c', 2))
        (Leaf ('a', 3)))
```

Labeling a Tree

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

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

key val

```
-- | 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
```

key val map

```
-- | '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
```

default key map

Common Pattern?

Both the functions have a common “shape”

label

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

key-label

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

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

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

out-val

OLD NEW

State Transformers

Lets capture the above "pattern" as a type

1. A **State** Type

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

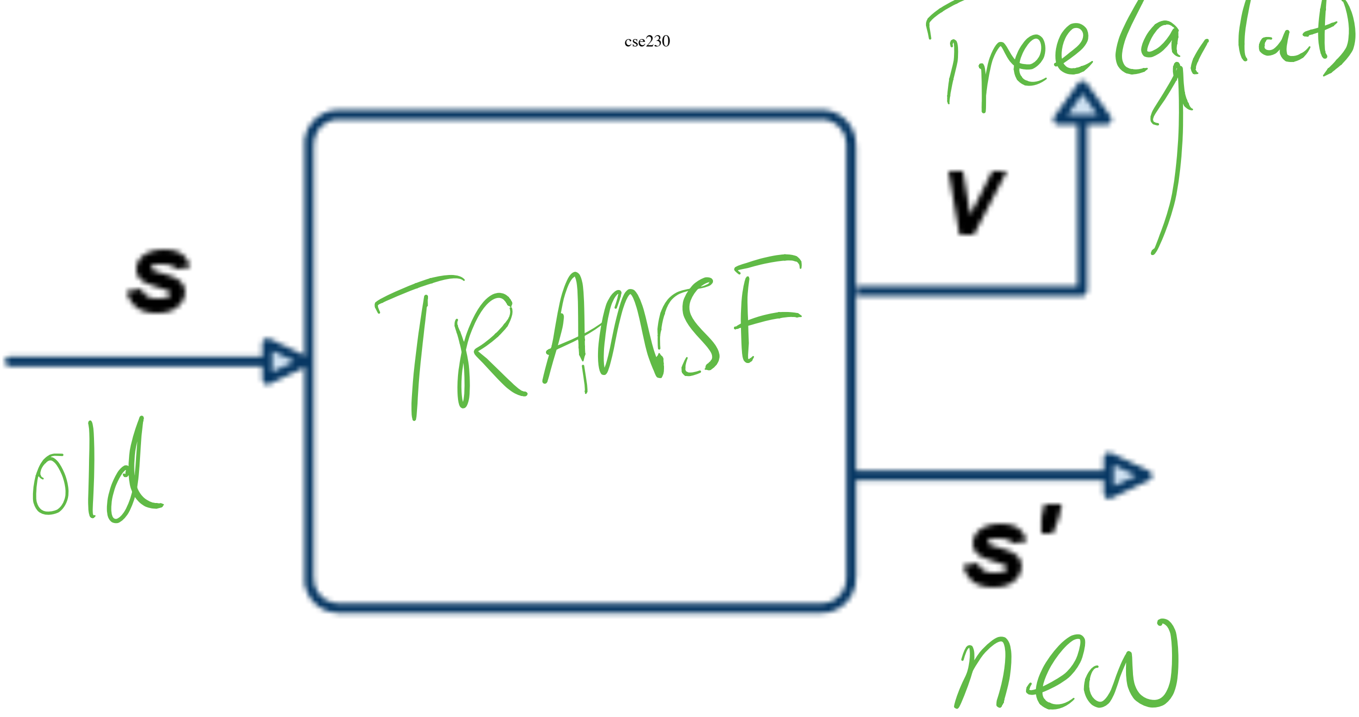
2. A **State Transformer** Type

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`?

```
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 … ???

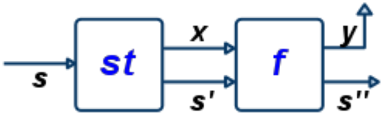
(Can someone suggest an explanation in English?)

`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`



5/15 STOP