eg Maybe

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

return :: a -> Maybe a

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

```haskell
instance Monad Maybe where
  (>>=) = andThen
  return = Just
```

```haskell
processDataUnsafe :: [Int] -> String -> String -> Int
processDataUnsafe items s1 s2 =
    let index1 = read s1
        index2 = read s2
        item1 = items !! index1
        item2 = items !! index2
    in
        item1 + item2
```

```haskell
processData :: [Int] -> String -> String -> Maybe Int
processData items s1 s2 =
    case readInt s1 of
        Nothing -> Nothing
        Just index1 ->
            case readInt s2 of
                Nothing -> Nothing
                Just index2 ->
                    case safeIndex items index1 of
                        Nothing -> Nothing
                        Just item1 ->
                            case safeIndex items index2 of
                                Nothing -> Nothing
                                Just item2 ->
                                    Just (item1 + item2)
```

```haskell
processData1 :: [Int] -> String -> String -> Maybe Int
processData1 items s1 s2 =
    readInt s1 `andThen` \index1 -> (
        readInt s2 `andThen` \index2 -> (
            safeIndex items index1 `andThen` \item1 -> (
                safeIndex items index2 `andThen` \item2 -> (
                    Just (item1 + item2)
                )
            )
        )
    )
```

```
processData2 :: [Int] -> String -> String -> Maybe Int
processData2 items s1 s2 =
    readInt s1 >>= \index1 -> (
        readInt s2 >>= \index2 -> (
            safeIndex items index1 >>= \item1 -> (
                safeIndex items index2 >>= \item2 -> (
                    Just (item1 + item2)
                )    return
            )
        )
    )
```

```haskell
processData2 :: [Int] -> String -> String -> Maybe Int
processData2 items s1 s2 =
    readInt s1 >>= \index1 ->
    readInt s2 >>= \index2 ->
    safeIndex items index1 >>= \item1 ->
    safeIndex items index2 >>= \item2 ->
    Just (item1 + item2)
```

return

```haskell
processData3 :: [Int] -> String -> String -> Maybe Int
processData3 items s1 s2 = do
    index1 <- readInt s1
    index2 <- readInt s2
    item1 <- safeIndex items index1
    item2 <- safeIndex items index2
    Just (item1 + item2)
```

return

The power of abstraction

Writing code using only `(>>=)` and `return` (or using do notation) allows that code to work for *any* `Monad` instance.

(Demo with `Either`)

The power of abstraction

Writing code using only `(>>=)` and `return` (or using do notation) allows that code to work for *any* `Monad` instance.

(Demo with `Either`)

Modeling mutation in a pure functional world

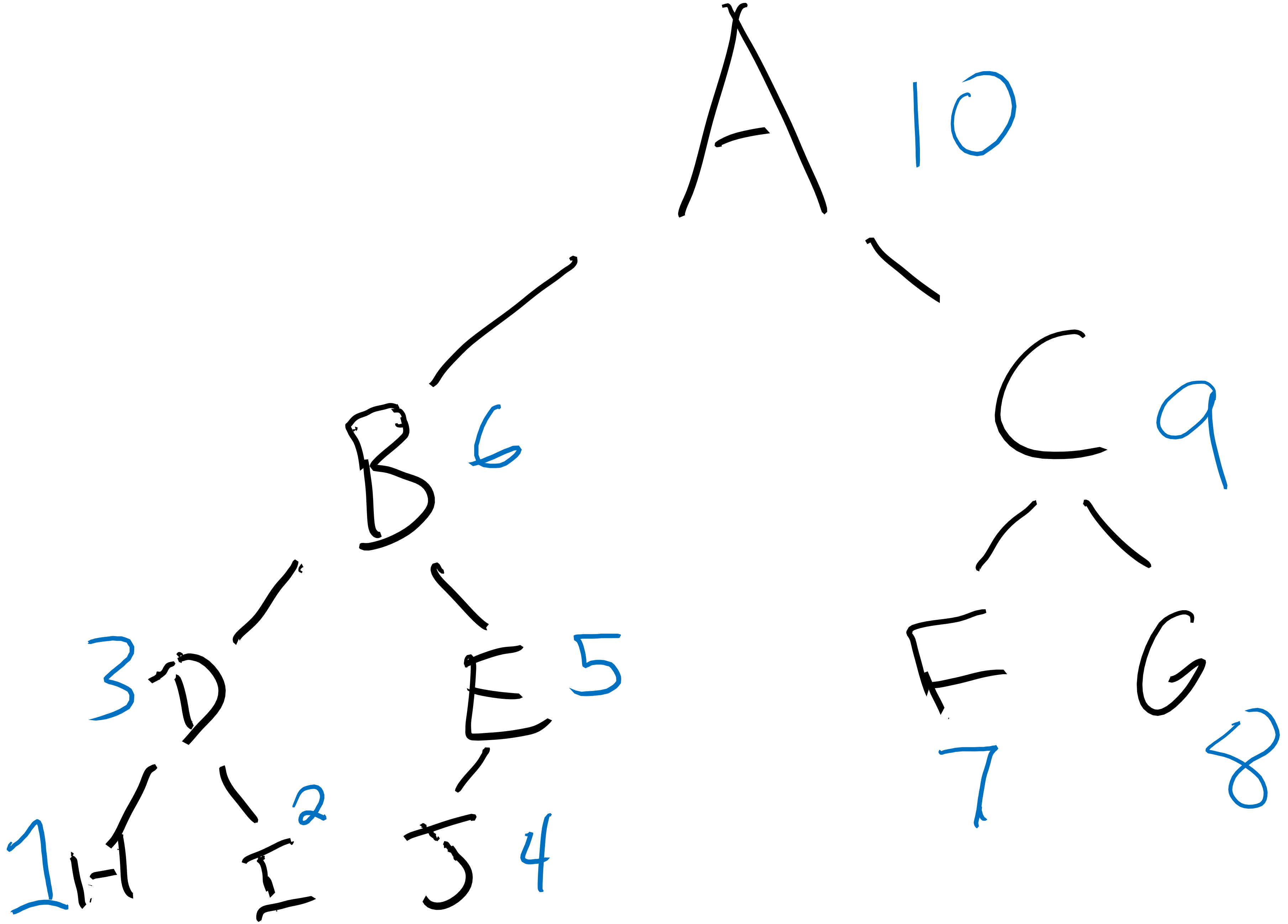
Hopefully your work in this course up to this point has convinced you that explicit mutation is *not* necessary to write substantial programs!

But sometimes a domain/algorithm is most easily modeled using mutable state.

Problem: given a binary tree, set each node's value to its position in a *left-to-right postorder traversal* of the tree.

```
data BTree a = Empty
             | BTree a (BTree a) (BTree a)

postOrderLabel :: BTree a -> BTree Int
```


A 10
B 6
C 9
3 D
E 5
F 7
G 8
1 H
I 2
J 4

```
i = 0

def post_order_label(tree):
    if tree.is_empty():
        return
    else:
        post_order_label(tree.left)
        post_order_label(tree.right)
        tree.root = i
        i = i + 1
```

The Python code makes use of a *global mutable counter* to keep track of the number of nodes "seen so far."

How do we do this without using mutation?

Recall **`foldl`** and generic list iteration

b [a]

```
(foldl f init lst)
```

```
acc = init
for x in lst:
  acc = f(x, acc)
```

$f :: a \to b \to b$

x

old acc

new acc

Recall `foldl` and generic list iteration

```
(foldl f init lst)
```

```
acc = init
for x in lst:
   acc = f(x, acc)
```

In most languages, mutable state is implicit, managed by the language implementation.

`f :: BTree a -> BTree Int` (*with mutable* ***`Int`***)

In most languages, mutable state is implicit, managed by the language implementation.

```
f :: BTree a -> BTree Int
```
(*with mutable* `Int`)

To turn this into a pure function, we make the state an explicit input and output.

```
f’ :: BTree a -> Int -> (BTree Int, Int)
```

In general, if a function `f :: t1 -> … -> tn -> a` uses mutable state of type `s`, we can make this explicit as

```
f’ :: t1 -> … -> tn -> s -> (a, s)
```

The `State` type constructor represents an operation that using "mutable" state.

type of state   old state   new state

```
data State s a = State (s -> (a, s))
```

type of "return value"   "State Operation"

NOTE: not the same "State" as on A2!

Primitive state operations: accessing the state

print(x)

```
get :: State s s
get = State (\state ->
```

(state , state)

"returned value"

new state same as old state

```
    )
```

Primitive state operations: setting the state

my-var = 3

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

"unit" (like "void")

Extracting/Performing a state operation

```
runState :: State s a -> (s -> (a, s))
runState (State f) = f

-- equivalently
runState :: State s a -> s -> (a, s)
runState (State f) init = f init
```

initial state

Chaining stateful operations

Running example (“trivial” mutation)

```
x = 10
x = x * 2
return “Final result: ” + str(x)
```

```
(_, s1)  = runState (put 10) s0
(x, s2)  = runState get s1
(_, s3)  = runState (put (x * 2)) s2
(x', s4) = runState get s3

("Final Result" ++ show x', s4)
```

```
(_, s1)  = runState (put 10) s0
(x, s2)  = runState get s1
(_, s3)  = runState (put (x * 2)) s2
(x', s4) = runState get s3

("Final Result" ++ show x', s4)
```

Need to *sequence* stateful operations, using values from previous operations in future ones.

“lookup x, use it to perform the next operation” m a -> (a -> m b) -> m b

```
State Int a -> (a -> State Int b) -> State Int b
```

Back to trees!