class Monad m where

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

return :: a -> Maybe a

e.g., Maybe

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

return :: a -> Maybe a
instance Monad Maybe where
    (>>>=) = andThen
    return = Just
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
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)
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)
                     )
                     )
                     )
                     )
                     )
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
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
  return (Just (item1 + item2))
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.

\[
\text{data } \text{BTree } a = \text{Empty} \\
| \text{BTree } a \ (\text{BTree } a) \ (\text{BTree } a)
\]

\[
\text{postOrderLabel} :: \text{BTree } a \rightarrow \text{BTree } \text{Int}
\]
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

\[ \text{foldl} \ f \ \text{init} \ \text{lst} \]

\[
\begin{array}{c}
\text{acc} = \text{init} \\
\text{for } x \text{ in } \text{lst:} \\
\quad \text{acc} = f(x, \text{acc})
\end{array}
\]
Recall `foldl` and generic list iteration

\[(\text{foldl } f \text{ init lst})\]

\[
\text{acc} = \text{init}
\]

for \(x\) in \(lst\):
  \[
  \text{acc} = f(x, \text{acc})
  \]
In most languages, mutable state is *implicit*, managed by the language implementation.

\[ f :: \text{BTree} \ a \rightarrow \text{BTree} \ \text{Int} \quad (\text{with mutable} \ \text{Int}) \]
In most languages, mutable state is implicit, managed by the language implementation.

\[ f :: \text{BTree} \ a \to \text{BTree} \ \text{Int} \quad (\text{with mutable} \ \text{Int}) \]

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

\[ f' :: \text{BTree} \ a \to \text{Int} \to (\text{BTree} \ \text{Int}, \ \text{Int}) \]
In general, if a function $f :: t_1 \to \ldots \to t_n \to a$ uses mutable state of type $s$, we can make this explicit as

$$f' :: t_1 \to \ldots \to t_n \to s \to (a, s)$$
The State type constructor represents an operation that uses “mutable” state.

\[
\text{data State } s \ a = \text{State } (s \to (a, s))
\]

NOTE: not the same “State” as on A2!
Primitive state operations: accessing the state

get :: State s s
get = State ( \state \rightarrow
    ( state , state )
)

"returned value" new state same as old state
Primitive state operations: setting the state

\[ \text{put} :: s \rightarrow \text{State } s \]

\[ \text{put } x = \text{State } (\lambda \text{state } \rightarrow \text{my_var = 3}) \]

\[ ( ( ) , x ) \]

\[ \text{"unit" (like } \text{"void"}) \]
Extracting/Performing a state operation

\[
\text{runState} :: \text{State}\ s\ a \rightarrow (s \rightarrow (a, s)) \\
\text{runState} (\text{State } f) = f
\]

\textit{--- equivalently}

\[
\text{runState} :: \text{State}\ s\ a \rightarrow s \rightarrow (a, s) \\
\text{runState} (\text{State } f) \text{ init } = f \text{ init}
\]
Chaining stateful operations
Running example ("trivial" mutation)

```python
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”
\[ m \ a \rightarrow (a \rightarrow m\ b) \rightarrow m \ b \]

State Int a -> (a -> State Int b) -> State Int b
Back to trees!