class Monad m where
    (>>>=) :: m a -> (a -> m b) -> m b
return :: a -> m 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 -> (n
      safeIndex items index1 >>= \item1 -> (n
        safeIndex items index2 >>= \item2 -> (n
          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)
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)
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
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 \texttt{foldl} and generic list iteration

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

\[
\texttt{acc} = \texttt{init} \\
\texttt{for} \ x \ \texttt{in} \ \texttt{lst}: \\
\quad \texttt{acc} = f(x, \ acc)
\]
Recall **foldl** and generic list iteration

\[
(foldl \ f \ init \ lst)
\]

\[
acc = init
\]

\[
\text{for } x \text{ in } lst:\n\quad acc = f(x, \ acc)
\]
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.

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

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

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

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

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

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

get :: State s s
get = State (\state \rightarrow
Primitive state operations: setting the state

\[
\text{put} :: s \to \text{State} s \rightarrow \\
\text{put } x = \text{State } (\text{\textbackslash state } \rightarrow \\
)
\]
Extracting/Performing a state operation

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

-- equivalently

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