Extra office hour: Tuesday 4-5pm in BA4260
Today: two last type concepts

1. Combining monads
2. Return-type (ad-hoc) polymorphism
Recap

monads as abstraction of *sequenced computations*
Computations that can fail

Maybe a
Computations over “mutable” state

State s a
How do we combine monads?
Problem: label each node with its position in the tree’s postorder traversal, **but fail if see “David”**

```
labelMaybe :: BTree String -> Maybe (BTree String)
```
State Int (Maybe (BTree String))

(>>>=) will extract a Maybe (BTree String), not the BTree String!
Demo 1 (manually making a new Monad)
Demo 2 (using a monad transformer)
Return-type polymorphism
Recap: ad hoc polymorphism is when a single entity (e.g., function identifier) behaves differently for different “contained” types
A function identifier can refer to *different implementations*, depending on how it is used.

E.g., (>>>=)
Consider method *overloading* in Java

class Point {
    void   move(int dx, int dy) { ... }
    int    move(float dx, float dy) { ... }
    String move() { ... }
}


Now consider `return`

```
return :: a -> m a
```
return is ad-hoc polymorphic in its return type
class Point {
    void move(int dx, int dy) { ... }
    int move(int dx, int dy) { ... }
    String move(int dx, int dy) { ... }
}
How does Haskell know which `return` to use?
Type inference!
f :: Maybe Int -> Either String Int -> [Int] -> ___

f (return 1) (return 1) (return 1)
Limitation of type inference

show (return 1)