

# Trees and Monads

Andres Löh, Jan Rochel

Difficulty: medium

1 (medium). Given the following data type for a binary tree from the previous tutorial

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
            deriving Show
```

write a function that labels a tree, each node differently.

```
labelTree1 :: Tree a -> Tree Int
```

To assign a new number to each visited node you need to maintain a state while traversing the tree, which means you are essentially implementing a stateful computation. First, find a solution that does not rely on monads.

In an imperative language you would probably introduce a global counter variable that is incremented for each visit. In a purely functional language you cannot update the value of variables. Introduce a worker function with additional parameters to maintain the state.

(Hint: multiple output parameters can be achieved by using a tuple as an output type.)

```
labelTree1W :: Tree a -> ... -> ... Tree Int ...
```

Next, implement the same functionality in the State monad.

```
labelTree2W :: Tree a -> State Int (Tree Int)
```

Make use of the do-notation. Remember to import `Control.Monad.State`. It provides the following functions to read the current state and to set a new state.

```
get :: State s s
put :: s -> State s ()
```

To run your stateful computation pass it to `runState` along with an initial state.

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

As a result you obtain the result of the computation along with the final state. Combine your worker function with `runState` to obtain the labelled tree.

```
labelTree2 :: Tree a -> Tree Int
```

**2 (medium).** In the next step, you are going to define your own State monad.

Rewrite `labelTree2W` such that it does not rely on the `do`-notation but solely on `bind (>>=)` and `return`. This is a purely mechanical translation. After convincing yourself that your code still works, remove the import of `Control.Monad.State` and rename all occurrences of `bind` to your own bind operator (`>>-`), and all occurrences of `return` to `returnS`. Add empty function bindings for all the no-longer defined functions and operators:

```
(>>-)    = ⊥  
returnS  = ⊥  
runState = ⊥  
get      = ⊥  
put      = ⊥
```

To make the code compile again, you need to introduce a type synonym for `State`. If you choose it carefully you should not have to change anything else in your code.

```
type State ... = ...
```

If you have trouble coming up with a sensible definition understand that the types of `labelTree2W` and `labelTree1W` have to be essentially the same.

Implement the undefined functions. Remember, that you have already implemented their functionality once before in `labelTree1 / labelTree1W`.