State Monad

We once again consider a data type for trees with values only in leaf positions

```
data Tree a = L a | Tree a :+:  Tree a
```

Using this data type, we want to define a function, which enumerates the leafs of a tree from left to right.

```
numberTree :: Tree a -> Tree (Int,a)
```

With the example usage:

```
ghci> numberTree ((L 'a') :+: (L 'b')) :+: (L 'c')
((L (1,'a')) :+: (L (2,'b'))) :+: (L (3,'c'))
```

The recursive definition of this function will pass through the algebraic data structure and use some kind of accumulator (an `Int` parameter), giving us the next number, we should use, when we reach a leaf. Hence, we could try to define an auxiliary function `numberTreeWithNum`, which is then initially called as follows:

```
numberTree t = numberTreeWithNum t 1
```

With the type `numberTreeWithNum ::Tree a -> Int -> Tree (Int,a)`. Unfortunately, we obtain a problem, when we try to define this function. In the case for the nonempty tree, we do not know, how many numbers, will be produced in the pass through the left tree:

```
numberTreeWithNum (tl :+: tr) n = numberTreeWithNum tl n :+: numberTreeWithNum tr (size c
```

Computing the `size` of the left is an expensive operation and might result into exponential run-time, because this size is computed in every single recursive step. A better solution is, that the computation returns the next number as an additional return value. In other words, the (next) number is some kind of accumulator passed through the tree. We get the type

```
numberTreeWithAcc :: Tree a -> Int -> (Tree (Int,a), Int)
```

Using this function, it is easy to number all leafs. In the initial call, we only have to select the first element of the result tuple.

```
numberTreeAcc t = fst (numberTreeWithAcc t 1)
```

In the auxiliary function a leaf simply consumes the number and increments it for the next leaf:

```
numberTreeWithAcc (L x) n = (L x,n+1)
```

In the case of an inner node, we can then simply pass the numbers through the trees.

```
numberTreeWithAcc (l :+: r) n =
  let (l',n1) = numberTreeWithAcc l n
```

```haskell
      (r',n2) = numberTreeWithAcc r n1
   in (l' :+: r', n2)
```

Defining functions for larger data types (esp. with larger branching) using this approach my easily result in mistakes, since many variables have to be defined and the correct variable has to be used. The manual passing of the state can be confusing.

Inspecting the code, we see, that it has an almost sequential structure. Hence, this code might be a good candidate for using monads and the do-notation as well. A variant might look like this:

```haskell
numberTreeState (L x) = do
  n <- get
  put (n+1)
  return (L (n,x))
numberTreeState (l :+: r) = do
  l' <- numberTreeState l
  r' <- numberTreeState r
  return (l' :+: r')
```

The functions `put` and `get` are supposed to read and modify the existing state. In this implementation the `return` function should have the type `a -> Int -> (a,Int)`. Hence, the type constructor `IntState` for the monad should be defined as:

```haskell
type IntState a = Int -> (a,Int)
```

and bind would be of type `IntState a -> (a -> IntState b) -> IntState b`. Furthermore, `get` and `put` would be of type

```haskell
get :: IntState Int
put :: Int -> IntState ()
```

This type can be further abstracted. The state can have an arbitrary type, which means we should use polymorphism for the type of the state as well. Furthermore, it is better to use a `newtype` instead of a type synonym, because this avoid type clashes with other functions of the same structure and allows a proper instance definition for the monad class.

```haskell
newtype State s a = State (s -> (a,s))
```

For this type we also define the function

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

which allows us to start a computation in our state monad. Obviously, it holds, that `runState (State f) = f` and for all `a :: State s a` also `State (runState a) = a`.

Now we can defined an instance of class `Monad` for the type constructor `State s`. `return` leaves the state unchanged and bind passes the state through the computation. The idea is the same as in the initial definition of `numberTreeWithAcc`

```
instance Monad (State s) where
  return x = State (\s -> (x,s))

  a >>= f = State (\s -> let (x,s') = runState a s
                         in runState (f x) s')
```

Here we exactly once program, how the state is pulled through the computation. Hence, it is not necessary to program this again and again. We simply use the bind-operator instead.

It remains to prove, the monad laws are fulfilled. `return` is a left identity for bind:

```
   return x >>= f
 = State (\s ->
     let (x',s') = runState (return x) s
      in runState (f x') s')
 = State (\s ->
     let (x',s') = runState (State (\s -> (x,s))) s
      in runState (f x') s')
 = State (\s ->
     let (x',s') = (\s -> (x,s)) s
      in runState (f x') s')
 = State (\s ->
     let (x',s') = (x,s)
      in runState (f x') s')
 = State (\s -> runState (f x) s)
 = State (runState (f x))
 = f x
```

`return` is also a right identity for bind:

```
   a >>= return
 = State (\s -> let (x,s') = runState a s
                 in runState (return x) s')
 = State (\s -> let (x,s') = runState a s
                 in runState (State (\s -> (x,s))) s')
 = State (\s -> let (x,s') = runState a s
                 in (\s -> (x,s)) s')
 = State (\s -> let (x,s') = runState a s in (x,s'))
 = State (\s -> runState a s)
 = State (runState a)
 = a
```

Finally, we prove the associativity for bind:

```
  (a >>= f) >>= g
```

```
= State (\s -> let (x,s') = runState (a >>= f) s
                in runState (g x) s')
= State (\s ->
    let (x,s') = runState (State (\t ->
                   let (y,t') = runState a t
                    in runState (f y) t')) s
     in runState (g x) s')
= State (\s ->
    let (x,s') = (\t -> let (y,t') = runState a t
                         in runState (f y) t') s
     in runState (g x) s')
= State (\s ->
    let (x,s') = let (y,t') = runState a s
                  in runState (f y) t'
     in runState (g x) s')
= State (\s ->
    let (y,t') = runState a s
        (x,s') = runState (f y) t'
     in runState (g x) s')
= State (\s ->
    let (y,t') = runState a s
     in let (x,s') = runState (f y) t'
         in runState (g x) s')
= State (\s ->
    let (y,t') = runState a s
     in (\t -> let (x,s') = runState (f y) t
                in runState (g x) s') t')
= State (\s ->
    let (y,t') = runState a s
     in runState (State (\t ->
          let (x,s') = runState (f y) t
           in runState (g x) s')) t')
= State (\s -> let (y,t') = runState a s
                in runState (f y >>= g) t')
= a >>= \x -> f x >>= g
```

It remains to define the functions **get** and **put**. The function **get** leaves the state unchanged and additionally yields it as first component of the result tuple.

```
get :: State s s
get = State (\s -> (s,s))
```

The function **put** ignores the passed state and replaces it be the argument parameter

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

Using these definitions, it is now very easy to define `numberTree` by means of a monadic, auxiliary function `numberTreeState`:

```
numberTree :: Tree a -> Tree (Int,a)
numberTree t = fst (runState (numberTreeState t) 1)

numberTreeState (L x) = do
  n <- get
  put (n+1)
  return (L (n,x))
numberTreeState (l :+: r) = do
  l' <- numberTreeState l
  r' <- numberTreeState r
  return (l' :+: r')
```

The presented implementation is not the only possible Implementation of a state monad. Hence, it makes sense to generalize this approach to a class as well. State monads provide, beside the operation of class `Monad` two functions `get` and `put`, which can be abstracted as follows:

```
class Monad m => MonadState s m where
  get :: m s
  put :: s -> m ()
```

`MonadState` is a so called Multi Parameter typeclass. Both, the type of the state and the type constructor for the monad are type parameters of `MonadState`. Multi-Parameter Classes do not belong to Haskell'98 Standard. However, they can be used in GHC oder GHCi by enabling the language extension `MultiParamTypeClasses`. To declare corresponding instances, the extension `FlexibleInstances` must be enabled as well.

The partially applied type constructor `State s` can be made an instance of `MonadState s` by copying the definitions for `get` and `put` into the instance declaration.

```
instance MonadState s (State s) where
  get   = State (\s -> (s,s))
  put s = State (\_ -> ((),s))
```

Similar to `MonadPlus` there are also some reasonable laws for state monads

```
get >>= put  =  return ()
```

which means, that setting the state to the actual state has no effect and

```
put s >> get  =  put s >> return s
```

which means, that get has no effect on the state but yields the state set by `put` before.

Our implementation fulfills these laws:

```
  get >>= put
```

```
= State (\s ->
    let (x,s') = runState get s
     in runState (put x) s'
= State (\s ->
    let (x,s') = runState (State (\s -> (s,s))) s
     in runState (put x) s')
= State (\s ->
    let (x,s') = (s,s)
     in runState (put x) s')
= State (\s -> runState (put s) s)
= State (\s -> runState (State (\_ -> ((),s))) s)
= State (\s -> ((),s))
= return ()
```

And the second law:

```
  put s >> get
= State (\t ->
    let (x,t') = runState (put s) t
     in runState get t')
= State (\t ->
    let (x,t') = runState (State (\_ -> ((),s))) t
     in runState get t')
= State (\t ->
    let (x,t') = ((),s)
     in runState (State (\s -> (s,s))) t')
= State (\t -> (s,s))
= State (\t -> let (x,t') = ((),s) in (s,t'))
= State (\t ->
    let (x,t') = runState (State (\_ -> ((),s))) t
     in runState (State (\s' -> (s,s'))) t'
= State (\t ->
    let (x,t') = runState (put s) t
     in runState (return s) t'
= put s >> return s
```

Using the class `MonadState` it would also be possible to execute the code in an arbitrary state monad. This can be seen from it most general type:

```
numberTreeState :: MonadState Int m => Tree a -> m (Tree (Int,a))
```

So far, we got to know no further instance of `MonadState`. However, we will see an alternative approach later.