Если вы видите что-то необычное, просто сообщите мне.

# State Монада

В прошлой части, мы изучили монады `Reader` и `Writer`. Они пакаказил, что на самом деле имеем алтернативу глобальным переменным. Нам просто нужно каким-то образом заключить их в определенный тип, это то для чего они нужны. В этой части изучим `State` монаду, которая объединяет некоторую функциональность для обоих идей.

# Мотивации пост: Крестики-нолики

Для этой части мы воспользуемся простой моделью для игры Крестки-нолики. Главный объект это тип данных `GameState` содержащий несколько важных кусочков информации. Первое и важное, он содержит "доску", и двумерный массив индексов состояния полей(X/0 или пусто). Так же знает чей ход и имеет случайный генератор.

```
data GameState = GameState
  { board :: A.Array TileIndex TileState
  , currentPlayer :: Player
  , generator :: StdGen
  }

data Player = XPlayer | OPlayer

data TileState = Empty | HasX | HasO
  deriving Eq

type TileIndex = (Int, Int)
```

Давай взглянем на то, как некоторые из функций нашей игры будут работать. Например нужно придумать функцию для случайного выбора хода. Она долна выводить `TileIndex` и изменять генератор нашей игры. Затем основываясь на нем делаем шаг и передаем ход другому игроку. Другими словами, у нас есть операции которые зависят от текущего состояния игры, но так же обновляет это состояние.

# THE STATE MONAD

This is exactly the situation the State monad deals with. The State monad wraps computations in the context of reading and modifying a global state object. This context chains two operations together in an intuitive way. First, it determines what the state should be after the first operation. Then, it resolves the second operation with the new state.

It is parameterized by a single type parameter s, the state type in use. So just like the Reader has a single type we read from, the State has a single type we can both read from and write to. There are two primary actions we can take within the State monad: get and put. The first retrieves the state, the second modifies it by replacing it with a new object. Typically though, this new object will be similar to the original:

```
-- Retrieves the state, like Reader.ask
get :: State s s

-- Overwrites the existing state
put :: s -> State s ()
```

There is also a runState function, similar to runReader and runWriter. Like the Reader monad, we must provide an initial state, in addition to the computation to run. But then like the writer, it produces two outputs: the result of our computation AND the final state:

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

If we wish to discard either the final state or the computation's result, we can use evalState and execState, respectively:

```
evalState :: State s a -> s -> a


execState :: State s a -> s -> s
```

So for our Tic Tac Toe game, many of our functions will have a signature like State GameState a.

# OUR STATEFUL FUNCTIONS

Now we can examine some of the different functions mentioned above and determine their types. We have for instance, picking a random move:

```
chooseRandomMove :: State GameState TileIndex
chooseRandomMove = do
  game <- get
  let openSpots = [ fst pair | pair <- A.assocs (board game), snd pair == Empty]
  let gen = generator game
  let (i, gen') = randomR (0, length openSpots - 1) gen
  put $ game { generator = gen' }
  return $ openSpots !! i
```

This outputs a TileIndex to us, and modifies the random number generator stored in our state! Now we also have the function applying a move:

```
applyMove :: TileIndex -> State GameState ()
applyMove i = do
  game <- get
  let p = currentPlayer game
  let newBoard = board game A.// [(i, tileForPlayer p)]
  put $ game { currentPlayer = nextPlayer p, board = newBoard }

nextPlayer :: Player -> Player
nextPlayer XPlayer = OPlayer
nextPlayer OPlayer = XPlayer

tileForPlayer :: Player -> TileState
tileForPlayer XPlayer = HasX
tileForPlayer OPlayer = HasO
```

This updates the board with the new tile, and then changes the current player, providing no output.

So finally, we can combine these functions together with do-syntax, and it actually looks quite clean! We don't need to worry about the side effects. The different monadic functions handle them. Here's a sample of what your function might look like to play one turn of the game. At the end, it returns a boolean determining if we've filled all the spaces:

```
resolveTurn :: State GameState Bool
resolveTurn = do
  i <- chooseRandomMove
  applyMove i
  isGameDone

isGameDone :: State GameState Bool
isGameDone = do
  game <- get
  let openSpots = [ fst pair | pair <- A.assocs (board game), snd pair == Empty]
  return $ length openSpots == 0
```

Obviously, there are some more complications for how the game would work in full, but the general idea should be clear. Any additional functions could live within the State monad.

# STATE, IO, AND OTHER LANGUAGES

When thinking about Haskell, it is often seen as a restriction that we can't have global variables like you could with Java class variables. However, we see now this isn't true. We could have a data type with exactly the same functionality as a Java class. We would just have many functions that can modify the global state of the class object using the State monad.

The difference is in Haskell we simply put a label on these types of functions. We don't allow it to happen for free. We want to know when side effects can potentially happen, because knowing when they can happen makes our code easier to reason about. In a Java class, many of the methods won't actually need to modify the state. But they could, which makes it harder to debug

them. In Haskell we can simply make these pure functions, and our code will be simpler.

IO is the same way. It's not like we can't perform IO in Haskell. Instead, we want to label the areas where we can, to increase our certainty about the areas where we don't need to. When we know part of our code cannot communicate with the outside world, we can be far more certain of its behavior.

# SUMMARY

That wraps it up for the State monad! Now that we know all these different monad constructs, you might be wondering how we can combine them. What if there was some part of our state that we wanted to be able to modify (using the State monad), but then there was another part that was read-only. How can we get multiple monadic capabilities at the same time? To learn to answer, head to part 6! In the penultimate section of this series, we'll discuss monad transformers. This concept will allow us to compose several monads together into a single monad!

Now that you're starting to understand monads, you can really pick up some steam on learning some useful libraries for important tasks. Download our Production Checklist for some examples of libraries that you can learn!

---

Revision #2
Created 11 March 2022 05:41:58 by gasick
Updated 28 September 2022 05:33:07 by gasick