state.lhs

```
> module State where

> import Control.Monad (when, mapM_)
```

Монада State

Реализуем работу с состоянием:

```
> newtype State s a = State
>   { runState
>     :: s
>     -- ^ старое состояние
>     -> (a, s)
>     -- ^ результат вычисления и новое состояние
>   }
```

Если мы выпишем Reader, Writer и State рядом, станет видно, что State — наиболее общая из тройки.

```
newtype Reader r a = Reader { runReader :: r ->  a     }
newtype Writer w a = Writer { runWriter ::      (a, w) }
newtype State  s a = Writer { runState  :: s -> (a, s) }
```

При композировании stateful вычислений мы, как и в случае с Reader, будем композировать функцию. Но в отличие от Reader, где на вход каждого подвычисления передавалось одно и то же окружение, здесь состояние будет “протаскиваться” через функции, изменяясь по дороге.

Реализуем же привычную тройку классов

```
> instance Functor (State s) where
>   fmap f fx = State $ \s ->
>     let (x, s') = runState fx s
>     in (f x, s')
>     --  ^ вернули новое состояние и применили функцию к
>     -- результату stateful вычисления fx

> instance Applicative (State s) where
>   pure x = State $ \s -> (x, s)
>   --  ^ чистое значение не влияет на состояние

>   -- (<*>) :: State s (a -> b) -> State s a -> State s b
>   ff <*> fx = State $ \s ->
>     let
>       (f, s')  = runState ff s
>       (x, s'') = runState fx s'
>     in (f x, s'')
>   -- заметьте, состояние передаётся из одного подвычисления
>   -- в другое

> instance Monad (State s) where
>   -- (>>=) :: State s a -> (a -> State s b) -> State s b
>   fx >>= f = State $ \s ->
>     let (x, s')  = runState fx s
>     in runState (f x) s'
```

А это уже примитивы stateful вычислений:

```
> -- | Запрос текущего состояния
> get :: State s s
> get = State $ \s -> (s, s)

> -- | Замена значения состояния на заданное
> put :: s -> State s ()
> put x = State $ \_ -> ((), x)

> -- | Изменение состояния без явного извлечения
> modify :: (s -> s) -> State s ()
> modify f = get >>= put . f

-- или так
modify' f = do
  x <- get
  put (f x)
```

Парочка примеров (немного синтетических):

```haskell
> -- | Вычисление максимальной длины элемента списка списков
> maxLength :: [[a]] -> State Int ()
> maxLength [] = pure ()
> maxLength (x:xs) = do
>   old <- get
>   let current = length x
>   when (old < current) $
>     put current
>   maxLength xs

> -- | Вычисление максимальной длины элемента списка списков
> -- (вариант номер два использующий mapM_)
> maxLength' :: [[a]] -> State Int ()
> maxLength' = mapM_ $ \x -> do
>   old <- get
>   let new = length x
>   when (new > old) $
>     put new
```