Transformers.lhs

```
> {-# LANGUAGE TupleSections #-}

> module Transformers where

> import Control.Monad.Identity (Identity(..))
> import Control.Monad.Trans.Class (MonadTrans(..))
```

Монадные трансформеры.

Ранее мы уже реализовывали тип, представляющий собой вычисление с состоянием, а именно State:

```
newtype State s a =
  State  { runState :: s -> (a, s) }
```

Но если бы мы захотели реализовать вычисление, которое работало бы с состоянием, но ещё предполагало другой эффект, например ранний выход, как у Maybe, то нам бы пришлось писать что-то не совсем очевидное:

```
calc s = do
  (x, s') <- Just $ runState get s
  if x < 0
    then Nothing  -- early exit
    else Just ()
  ((), _) <- Just $ runState put s'
  pure ()
```

Т.е. мы бы уже не смогли получить пробрасывание состояния, ведь в роли Monad здесь выступает Maybe, а он ничего не знает про какое-то там состояние!

Для того, чтобы мочь завернуть один эффект в другой так, чтобы при композировании оба эффекта сохранялись, позволяют монадные трансформеры.

Трансформер StateT.

Тип StateT выглядит так:

```
> newtype StateT s m a = StateT
>   { runStateT :: s -> m (s, a) }
```

Здесь результат вычисления с состоянием превратился в некое монадическое действие внутри `m`. Которое тоже нужно будет “запустить” каким-то образом.

Кстати, `State` можно получить из `StateT`, если поместить внутрь последнего `Identity` — ту самую “монаду без эффекта”!

```
> type State s = StateT s Identity

> runState :: State s a -> s -> (s, a)
> runState x = runIdentity . runStateT x
```

Реализуем же `Functor`, `Applicative` и `Monad` для `StateT`:

```
> instance Functor m => Functor (StateT s m) where
>   fmap f (StateT g) = StateT $ \s -> fmap (fmap f) (g s)

> instance Monad m => Applicative (StateT s m) where
>   -- тут ^ потребовался констреинт Monad для вложенного
>   -- эффекта для того, чтобы можно было собрать вычисления
>   -- с этим эффектом в цепочку, ведь нам нужно передать
>   -- состояние от одного подвычисления к другому.
>   pure x = StateT $ \s -> pure (s, x)
>   StateT ff <*> StateT fx = StateT $ \s -> do
>     (s', f) <- ff s
>     (s'', x) <- fx s'
>     pure (s'', f x)

> instance Monad m => Monad (StateT s m) where
>   StateT fx >>= f = StateT $ \s -> do
>     (s', x) <- fx s
>     runStateT (f x) s'
```

Вот так будут выглядеть примитивы для StateT:

```
> get :: Applicative m => StateT s m s
> get = StateT $ \s -> pure (s, s)
```

```
> put :: Applicative m => s -> StateT s m ()
> put x = StateT $ \_ -> pure (x, ())
```

Напишем пример вычисления, которое работает с состоянием, но разрешает ранний выход:

```
> comp :: StateT Int Maybe ()
> comp = do
>   x <- get
>   y <- if x < 0
>     then StateT $ \s -> fmap (s,) Nothing
>     else pure (x + 1)
>   put $ x + y
```

(запускать эти вычисления нужно так: `runStateT comp число`)

Обратите внимание на `StateT $ \s -> fmap (s,)` — это так называемый “lift”, т.е. затягивание вычисления `m a` в `StateT s m a`. Для каждого трансформера лифтинг реализуется по-разному, поэтому есть такой класс:

```
class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a
```

Для `StateT` инстанс будет выглядеть так:

```
> instance MonadTrans (StateT s) where
>   lift ma = StateT $ \s -> fmap (s,) ma
```

Теперь можно переписать наш пример так:

```
> compWithLift :: StateT Int Maybe ()
> compWithLift = do
>   x <- get
>   y <- if x < 0
>     then lift Nothing
>     else pure (x + 1)
>   put $ x + y
```

Если вы запустите это вычисление с начальным состоянием меньшим нуля, то в итоге получите `Nothing`. Само состояние не сохранится, потому что на момент выполнения “слоя” `Maybe` слой `StateT` уже не существует!

Чтобы сохранить состояние, нам нужно поменять слои местами. Для этого потребуется трансформер `MaybeT`.

Трансформер MaybeT.

Тип и инстансы:

```
> newtype MaybeT m a = MaybeT
>   { runMaybeT :: m (Maybe a) }

> instance Functor m => Functor (MaybeT m) where
>   fmap f (MaybeT x) = MaybeT $ fmap (fmap f) x

> instance Applicative m => Applicative (MaybeT m) where
>   pure = MaybeT . pure . pure
>   MaybeT ff <*> MaybeT fx = MaybeT $ (<*>) <$> ff <*> fx
>   -- здесь приходится сначала выполнить аппликативное
>   -- вычисление над m, но потом мы получаем Maybe,
>   -- поэтому применяем мы аппликативно функцию (<*>)

> instance Monad m => Monad (MaybeT m) where
>   MaybeT fx >>= f = MaybeT $ fx >>= \x ->
>     case x of
>       Nothing -> pure Nothing
>       Just x' -> runMaybeT (f x')

> instance MonadTrans MaybeT where
>   lift ma = MaybeT $ fmap Just ma
```

А вот так будет выглядеть пример, после того, как мы поменяем эффекты местами:

```
> comp2 :: MaybeT (State Int) ()
> comp2 = do
>   x <- lift get
>   y <- if x < 0
>     then MaybeT $ pure Nothing  -- с Nothing не очень красиво, да :(
>     else pure (x + 1)
>   lift $ put (x + y)
```

Заметьте, lift теперь стоят на stateful-подвычислениях. Если вы запустите это вычисление (`runState (runMaybeT comp2) число`), то увидите, что состояние присутствует в результате вне зависимости от успешности внешнего `MaybeT`-вычисления. Что и

требовалось получить!