functor.lhs

```
> {-# LANGUAGE DeriveFunctor #-}
> {-# LANGUAGE KindSignatures #-}
> {-# LANGUAGE InstanceSigs #-}
```

(тут я как обычно включаю расширения. Зачем нужно каждое, я скажу ниже)

HKT или Higher Kinded Types

Мы уже знаем, что типы могут быть параметризованы. Тип

```
data T a
```

имеет один параметр — типовую переменную `a`.

У типов есть свои типы. По привычке их называют “сорта” или “кайнды” (kinds). Любой полностью применённый (в том числе и к переменным) конструктор типа имеет кайнд "*". Если пойти в REPL и посмотреть кайнды у разных искоробочных типов мы увидим следующее:

```
λ> :set -XExistentialQuantification
λ> :k Maybe
Maybe :: * -> *
λ> :k Maybe Int
Maybe Int :: *
λ> :k forall a.Maybe a
forall a.Maybe a :: *
```

(первой строчкой я включаю расширение `ExistentialQuantification`, разрешающее явно писать квантор общности `forall` — иначе я не смок бы написать просто `:k Maybe a`, ведь переменная нигде не была введена `a`).

Как вы видите, `Maybe Int` и `Maybe a` имеют кайнд `*`, а вот `Maybe` ещё не получил своего параметра, поэтому имеет кайнд `* -> *`.

Функторы

```
> module Functor where

> import Data.Bifunctor  -- (понадобится позже)
```

Функтор в Haskell, это свойство, позволяющее взять значение типа `f a` и с помощью функции `(a -> b)` превратить (говорят “отобразить”) в значение `f b`. В коде это свойство выглядит как класс следующего вида:

```
class Functor (f :: * -> *) where
  -- Здесь я указал ^ явно кайнд типа. Сделать это мне позволило расширение
  -- KindSignatures. В данном конкретном случае кайнд был бы выведен, но
  -- я для наглядности его указал.
  fmap :: (a -> b) -> f a -> f b
```

Обратите внимание, здесь я первый раз в классе указываю, класс инстанциируется для не полностью построенного типа. Это позволяет классу в методах самостоятельно допременять тип. Посмотрите на сигнатуру `fmap` и вы увидите, что в ней тип `f` уже применён. Эта сигнатура явно кучу вещей:

- у типа `f` может быть изменено “содержимое”;
- это содержимое может быть любым, ведь переменная `a` не вводится в заголовке класса, поэтому никаких ограничений на `a` наложить нельзя;
- изменить содержимое можно только с помощью функции `(a -> b)`, ведь ни `a` ни `b` будут известны только при вызове `fmap` и значения `b` взять неоткуда.

Так же любой тип, выдающий себя за Функтор, должен соблюдать следующие законы:

1. `fmap id x == x`, который гарантирует, что структура функтора не меняется при изменении значений,
2. `fmap f . fmap g == fmap (f . g)`, который позволяет превратить два отображения в одно отображение с помощью композиции применяемых функций.

Эти законы компилятор, увы, не проверяет, поэтому для своих инстансов их нужно проверять самостоятельно (писать тесты, например).

Вот так выглядит инстанс Functor для Maybe:

```haskell
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  -- Расширение InstanceSigs позволяет написать сигнатуру в инстансе:
  -- иногда это удобно с точки зрения наглядности.
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

А для списка инстанс выглядит так:

```haskell
instance Functor [] where
  fmap :: (a -> b) -> [] a -> [] b
  -- "[] a" это рассахаренный "[a]"
  fmap = map
  -- да, для списка "fmap" это "map"
```

Рассмотрим инстанс для пары. Отметьте, что первый параметр пары подставлен, ведь у `(,)` кайнд `* -> * -> *`. Код:

```haskell
instance Functor ((,) a) where
  fmap :: (b -> c) -> (a, b) -> (a, c)
  fmap f (x, y) = (x, f y)
  -- первый элемент не будет затронут, поскольку тип "a" в заголовке инстанса
  -- в контексте fmap недоступен и про него ничего не известно. Помните, что
  -- а) сигнатуру для fmap я здесь указал для наглядности,
  -- б) "a" в заголовке инстанса и "a" в сигнатуре fmap — разные переменные!
```

Стоит отметить, что не нужно думать, что Функтор всегда подразумевает какие-то контейнеры и изменение значений. Не всегда! Вот пример:

```haskell
> newtype Const b a = Const { getConst :: b }

> instance Functor (Const b) where
>   fmap = const (Const . getConst)
>   -- Как вам такое? Разберите сами, как это работает!
```

Здесь заменяемое с помощью fmap значение нигде не хранится! По сути fmap для этого типа меняет только параметр типа!

Что приятно, для простых типов, которые не несут дополнительного смысла, а вложенные значения просто хранят, можно инстансы Functor получать бесплатно:

```haskell
> data Pair a = Pair a a deriving (Functor)

> data Tree a
>   = Tree a (Tree a) (Tree a)
>   | Leaf
>   deriving (Functor)

> p :: Pair String
> p = fmap (++ "!") $ Pair "Hello" "World"

> t :: Tree Int
> t = fmap (+ 1) $ Tree 42 Leaf (Tree 100 Leaf Leaf)
```

Bifunctor

Functor отображает один тип в другой по одному параметру. А Bifunctor — по двум. Выглядит класс так:

```haskell
class Bifunctor (f :: * -> * -> *) where
  first  :: (a -> b)             -> f a c -> f b c
  second ::             (c -> d) -> f a c -> f a d
  bimap  :: (a -> b) -> (c -> d) -> f a c -> f b d
  {-# MINIMAL bimap | first, second #-}
  --  ^ минимальная реализация требует либо first+second, либо bimap
```

Пример инстанса Bifunctor для нашего самодельного типа These

```haskell
> data These a b
>   = This a
>   | That b
>   | These a b

> instance Bifunctor These where
>   bimap f _ (This  x)   = This  (f x)
>   bimap _ g (That    y) = That        (g y)
>   bimap f g (These x y) = These (f x) (g y)
```

Foldable

Есть ещё такой класс — Foldable. Это класс, который говорит, что входящие в него типы можно сворачивать в одно значение.

Этот класс — сугубо утилитарный. Именно с помощью инстанциирования этого класса для множества контейнерных типов мы имеем возможность находить для всех этих типов длину, сумму элементов, находить максимум и прочее. Вот так выглядит сам класс:

```
class Foldable (t :: * -> *) where
  fold    :: Monoid m =>             t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr   :: (a -> b -> b) -> b -> t a -> b
  foldr'  :: (a -> b -> b) -> b -> t a -> b
  foldl   :: (b -> a -> b) -> b -> t a -> b
  foldl'  :: (b -> a -> b) -> b -> t a -> b
  foldr1  :: (a -> a -> a) ->      t a -> a
  foldl1  :: (a -> a -> a) ->      t a -> a
  toList  :: t a -> [a]
  null    :: t a -> Bool
  length  :: t a -> Int
  elem    :: Eq a => a -> t a -> Bool
  maximum :: Ord a => t a -> a
  minimum :: Ord a => t a -> a
  sum     :: Num a => t a -> a
  product :: Num a => t a -> a
  {-# MINIMAL foldMap | foldr #-}
  -- Defined in 'Data.Foldable'
```

Важно здесь увидеть, что для реализации всего этого богатства достаточно реализовать один из двух методов — `foldr` или `foldMap`. Причём, интересен именно последний: это та же свёртка, вот только начальное значение аккумулятра и операцию предоставляют `Monoid` и `Semigroup`.

Задание для смелых: возьмите самодельный список

```
> data List a = Cons a (List a) | Nil
```

и реализуйте инстанс `Foldable` для этого типа через один лишь метод `fold`. Для этого вам придётся изобрести (или подглядеть в `Data.Monoid`) несколько разных обёрток-моноидов.