**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЯ ОБРАЗОВАНИЯ**

**“ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ”**

Факультет информационных технологий

Кафедра технологий программирования

Выполнил студент 2 курса, группа 21-ИТ-1 Макеёнок Д. И.

Проверила Деканова М. В.

Полоцк, 2023 г.

**Модуль 1.**

**1.1Определение аппликативного функтора**

**Шаг 4**

В модуле Data.Functor определен оператор <$>, являющийся инфиксным аналогом функции fmap:

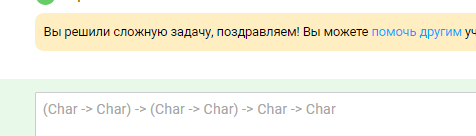
GHCi> :info <$>

(<$>) :: Functor f => (a -> b) -> f a -> f b

-- Defined in `Data.Functor'

infixl 4 <$>

В выражении succ <$> "abc" этот оператор имеет тип (Char -> Char) -> [Char] -> [Char]. Какой тип имеет первое (левое) вхождение этого оператора в выражении succ <$> succ <$> "abc"?



**Шаг 5**

Сделайте типы данных Arr2 e1 e2 и Arr3 e1 e2 e3 представителями класса типов Functor:

newtype Arr2 e1 e2 a = Arr2 { getArr2 :: e1 -> e2 -> a }

newtype Arr3 e1 e2 e3 a = Arr3 { getArr3 :: e1 -> e2 -> e3 -> a }

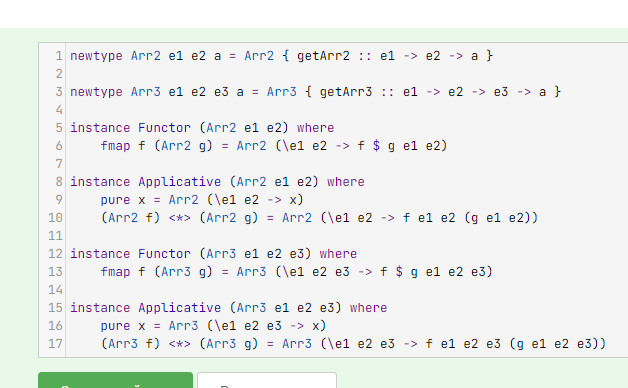
Эти типы инкапсулируют вычисление с двумя и тремя независимыми окружениями соответственно:

GHCi> getArr2 (fmap length (Arr2 take)) 10 "abc"

3

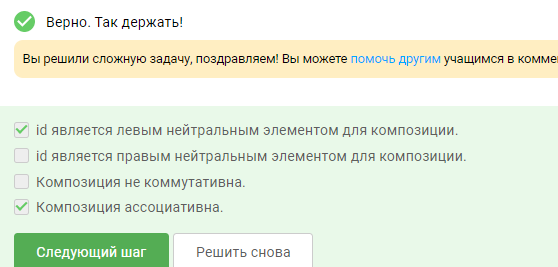
GHCi> getArr3 (tail <$> tail <$> Arr3 zipWith) (+) [1,2,3,4] [10,20,30,40,50]

[33,44]

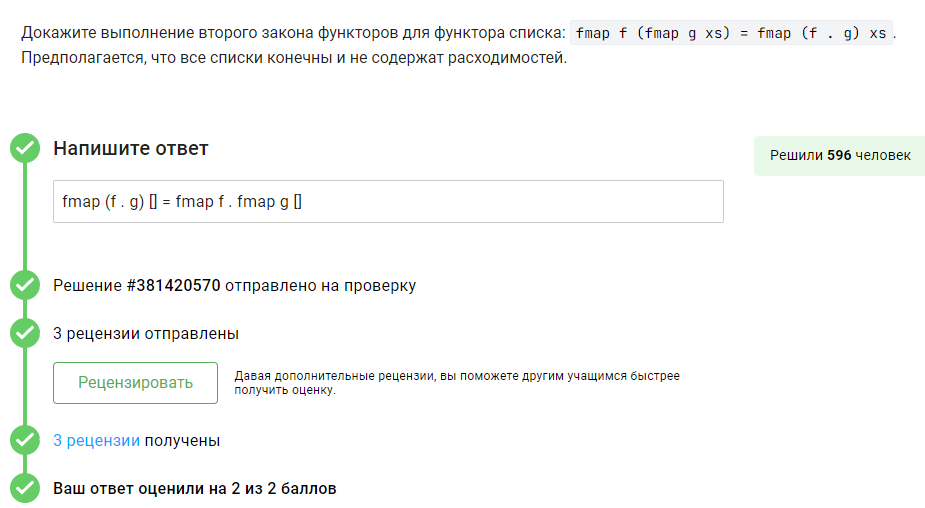


**Шаг 7**

Самостоятельно докажите выполнение первого (fmap id = id) и второго ( fmap f . fmap g = fmap (f . g)) законов функторов для функтора частично примененной функциональной стрелки (->) e. Отметьте те свойства оператора композиции функций, которыми вы воспользовались.



**Шаг 9**



**Шаг 15**

Следующий тип данных задает гомогенную тройку элементов, которую можно рассматривать как трехмерный вектор:

data Triple a = Tr a a a deriving (Eq,Show)

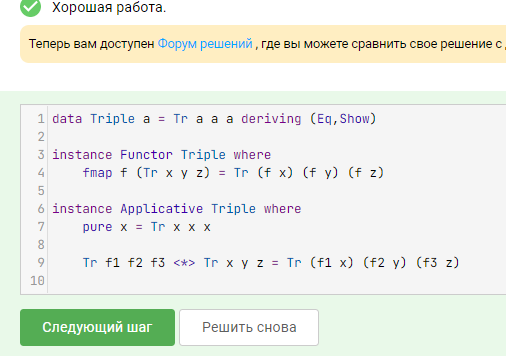
Сделайте этот тип функтором и аппликативным функтором с естественной для векторов семантикой покоординатного применения:

GHCi> (^2) <$> Tr 1 (-2) 3

Tr 1 4 9

GHCi> Tr (^2) (+2) (\*3) <\*> Tr 2 3 4

Tr 4 5 12

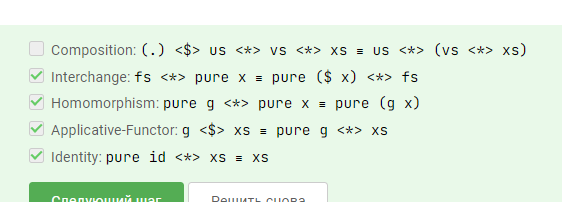


**1.2 Представители класса типов Applicative**

**Шаг 3**

Предположим, что для стандартного функтора списка оператор (<\*>) определен стандартным образом, а метод pure изменен на pure x = [x,x]

К каким законам класса типов Applicative будут в этом случае существовать контрпримеры?



**Шаг 5**

В модуле Data.List имеется семейство функций zipWith, zipWith3, zipWith4,..:

GHCi> let x1s = [1,2,3]

GHCi> let x2s = [4,5,6]

GHCi> let x3s = [7,8,9]

GHCi> let x4s = [10,11,12]

GHCi> zipWith (\a b -> 2\*a+3\*b) x1s x2s

[14,19,24]

GHCi> zipWith3 (\a b c -> 2\*a+3\*b+5\*c) x1s x2s x3s

[49,59,69]

GHCi> zipWith4 (\a b c d -> 2\*a+3\*b+5\*c-4\*d) x1s x2s x3s x4s

[9,15,21]

Аппликативные функторы могут заменить всё это семейство

GHCi> getZipList $ (\a b -> 2\*a+3\*b) <$> ZipList x1s <\*> ZipList x2s

[14,19,24]

GHCi> getZipList $ (\a b c -> 2\*a+3\*b+5\*c) <$> ZipList x1s <\*> ZipList x2s <\*> ZipList x3s

[49,59,69]

GHCi> getZipList $ (\a b c d -> 2\*a+3\*b+5\*c-4\*d) <$> ZipList x1s <\*> ZipList x2s <\*>ZipList x3s <\*> ZipList x4s

[9,15,21]

Реализуйте операторы (>\*<) и (>$<), позволяющие спрятать упаковку ZipList и распаковку getZipList:

GHCi> (\a b -> 2\*a+3\*b) >$< x1s >\*< x2s

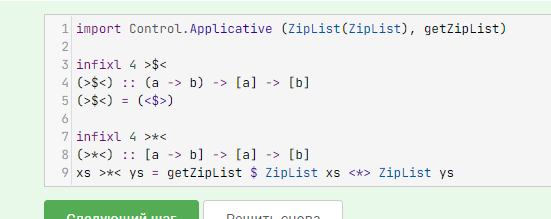
[14,19,24]

GHCi> (\a b c -> 2\*a+3\*b+5\*c) >$< x1s >\*< x2s >\*< x3s

[49,59,69]

GHCi> (\a b c d -> 2\*a+3\*b+5\*c-4\*d) >$< x1s >\*< x2s >\*< x3s >\*< x4s

[9,15,21]



**Шаг 8**

Функция

divideList :: Fractional a => [a] -> a

divideList [] = 1

divideList (x:xs) = (/) x (divideList xs)

сворачивает список посредством деления. Модифицируйте ее, реализовав divideList' :: (Show a, Fractional a) => [a] -> (String,a), такую что последовательность вычислений отражается в логе:

GHCi> divideList [3,4,5]

3.75

GHCi> divideList' [3,4,5]

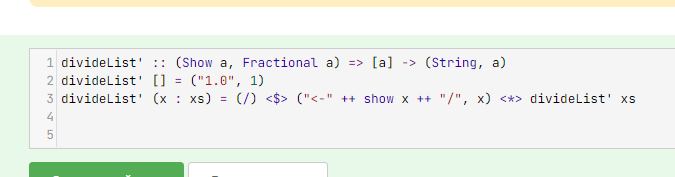
("<-3.0/<-4.0/<-5.0/1.0",3.75)

Используйте аппликативный функтор пары, сохраняя близкую к исходной функции структуру реализации

divideList' :: (Show a, Fractional a) => [a] -> (String,a)

divideList' [] = \_

divideList' (x:xs) = (/) <$> \_ <\*> \_



**Шаг 10**

Сделайте типы данных Arr2 e1 e2 и Arr3 e1 e2 e3 представителями класса типов Applicative

newtype Arr2 e1 e2 a = Arr2 { getArr2 :: e1 -> e2 -> a }

newtype Arr3 e1 e2 e3 a = Arr3 { getArr3 :: e1 -> e2 -> e3 -> a }

с естественной семантикой двух и трех окружений:

GHCi> getArr2 (Arr2 (\x y z -> x+y-z) <\*> Arr2 (\*)) 2 3

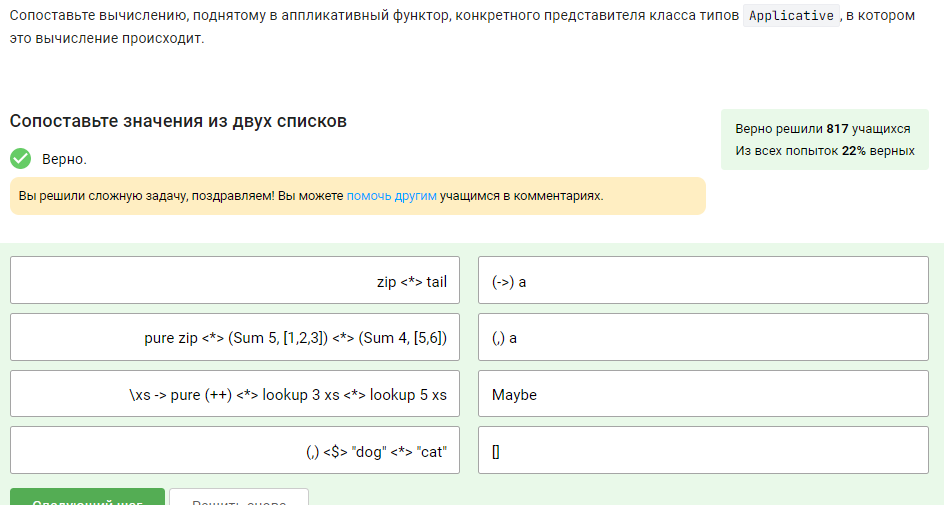
-1

GHCi> getArr3 (Arr3 (\x y z w -> x+y+z-w) <\*> Arr3 (\x y z -> x\*y\*z)) 2 3 4

-15



**Шаг 11**



**Шаг 14**

Двойственный оператор аппликации (<\*\*>) из модуля Control.Applicative изменяет направление вычислений, не меняя порядок эффектов:

infixl 4 <\*\*>

(<\*\*>) :: Applicative f => f a -> f (a -> b) -> f b

(<\*\*>) = liftA2 (flip ($))

Определим оператор (<\*?>) с той же сигнатурой, что и у (<\*\*>), но другой реализацией:

infixl 4 <\*?>

(<\*?>) :: Applicative f => f a -> f (a -> b) -> f b

(<\*?>) = flip (<\*>)

Для каких стандартных представителей класса типов Applicative можно привести цепочку аппликативных вычислений, дающую разный результат в зависимости от того, какой из этих операторов использовался?

В следующих шести примерах вашей задачей будет привести такие контрпримеры для стандартных типов данных, для которых они существуют. Следует заменить аппликативное выражение в предложении in на выражение того же типа, однако дающее разные результаты при вызовах с (<??>) = (<\*\*>) и (<??>) = (<\*?>). Проверки имеют вид exprXXX (<\*\*>) == exprXXX (<\*?>) для различных имеющихся XXX. Если вы считаете, что контрпримера не существует, то менять ничего не надо.

Ответ:

{-# LANGUAGE RankNTypes#-}

import Control.Applicative ((<\*\*>),ZipList(..))

infixl 4 <\*?>

(<\*?>) :: Applicative f => f a -> f (a -> b) -> f b

(<\*?>) = flip (<\*>)

exprMaybe :: (forall a b . Maybe a -> Maybe (a -> b) -> Maybe b) -> Maybe Int

exprMaybe op =

let (<??>) = op

infixl 4 <??>

in Just 5 <??> Just (+2) -- place for counterexample

exprList :: (forall a b . [a] -> [a -> b] -> [b]) -> [Int]

exprList op =

let (<??>) = op

infixl 4 <??>

in [1,2,3] <??> [(+3),(+4)] -- place for counterexample

exprZipList :: (forall a b . ZipList a -> ZipList (a -> b) -> ZipList b) -> ZipList Int

exprZipList op =

let (<??>) = op

infixl 4 <??>

in ZipList [1,2] <??> ZipList [(+3),(+4)] -- place for counterexample

exprEither :: (forall a b . Either String a -> Either String (a -> b) -> Either String b) -> Either String Int

exprEither op =

let (<??>) = op

infixl 4 <??>

in Left "AA" <??> Left "BB" -- place for counterexample

exprPair :: (forall a b . (String,a) -> (String,a -> b) -> (String,b)) -> (String,Int)

exprPair op =

let (<??>) = op

infixl 4 <??>

in ("AA", 3) <??> ("BB",(+1)) -- place for counterexample

exprEnv :: (forall a b . (String -> a) -> (String -> (a -> b)) -> (String -> b)) -> (String -> Int)

exprEnv op =

let (<??>) = op

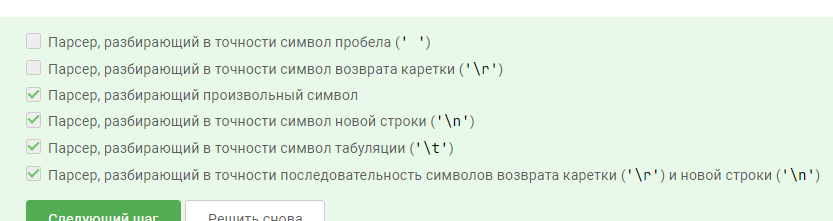
infixl 4 <??>

in length <??> (\\_ -> (+5)) -- place for counterexample

**1.3 Аппликативный парсер Parsec**

**Шаг 3**

Какие из следующих примитивных парсеров имеются в библиотеке Text.Parsec.Char ?



**Шаг 5**

Реализуйте парсер getList, который разбирает строки из чисел, разделенных точкой с запятой, и возвращает список строк, представляющих собой эти числа:

GHCi> parseTest getList "1;234;56"

["1","234","56"]

GHCi> parseTest getList "1;234;56;"

parse error at (line 1, column 10):

unexpected end of input

expecting digit

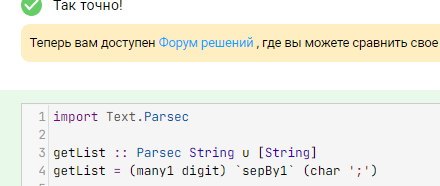
GHCi> parseTest getList "1;;234;56"

parse error at (line 1, column 3):

unexpected ";"

expecting digit

Совет: изучите парсер-комбинаторы, доступные в модуле Text.Parsec, и постарайтесь найти наиболее компактное решение.



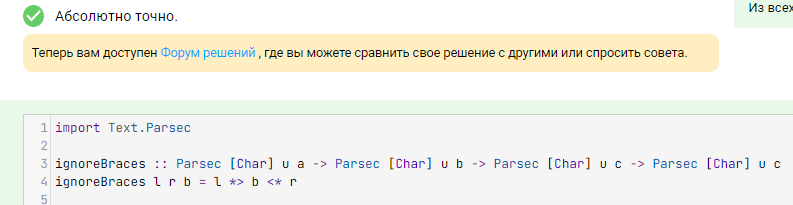
**Шаг 7**

Используя аппликативный интерфейс Parsec, реализуйте функцию ignoreBraces, которая принимает три аргумента-парсера. Первый парсер разбирает текст, интерпретируемый как открывающая скобка, второй — как закрывающая, а третий разбирает весь входной поток, расположенный между этими скобками. Возвращаемый парсер возвращает результат работы третьего парсера, скобки игнорируются.

GHCi> test = ignoreBraces (string «[[«) (string «]]») (many1 letter)

GHCi> parseTest test “[[ABC]]DEF”

«ABC»



**1.4 Аппликативный парсер своими руками**

**Шаг 4**

Предположим, тип парсера определен следующим образом:

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

Сделайте этот парсер представителем класса типов Functor. Реализуйте также парсер anyChr :: Prs Char, удачно разбирающий и возвращающий любой первый символ любой непустой входной строки.

GHCi> runPrs anyChr "ABC"

Just ('A',"BC")

GHCi> runPrs anyChr ""

Nothing

GHCi> runPrs (digitToInt <$> anyChr) "BCD"

Just (11,"CD")



**Шаг 6**

Сделайте парсер

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

из предыдущей задачи аппликативным функтором с естественной для парсера семантикой:

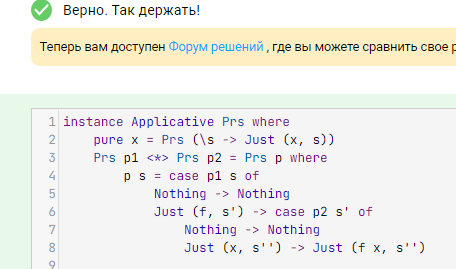
GHCi> runPrs ((,,) <$> anyChr <\*> anyChr <\*> anyChr) "ABCDE"

Just (('A','B','C'),"DE")

GHCi> runPrs (anyChr \*> anyChr) "ABCDE"

Just ('B',"CDE")

Представитель для класса типов Functor уже реализован.



**Шаг 8**

Рассмотрим более продвинутый парсер, позволяющий возвращать пользователю причину неудачи при синтаксическом разборе:

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

Реализуйте функцию satisfyE :: (Char -> Bool) -> PrsE Char таким образом, чтобы функция

charE :: Char -> PrsE Char

charE c = satisfyE (== c)

обладала бы следующим поведением:

GHCi> runPrsE (charE 'A') "ABC"

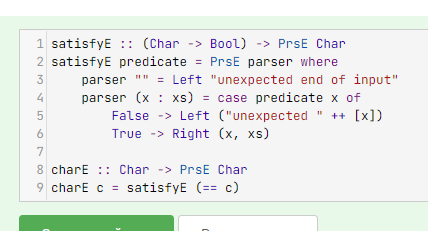
Right ('A',"BC")

GHCi> runPrsE (charE 'A') "BCD"

Left "unexpected B"

GHCi> runPrsE (charE 'A') ""

Left "unexpected end of input"



**Шаг 9**

Сделайте парсер

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

из предыдущей задачи функтором и аппликативным функтором:

GHCi> let anyE = satisfyE (const True)

GHCi> runPrsE ((,) <$> anyE <\* charE 'B' <\*> anyE) "ABCDE"

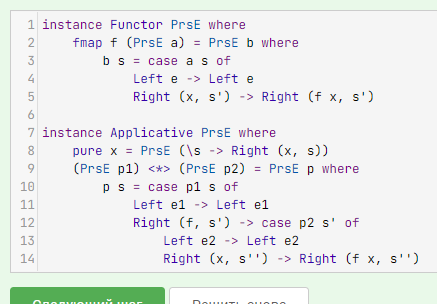
Right (('A','C'),"DE")

GHCi> runPrsE ((,) <$> anyE <\* charE 'C' <\*> anyE) "ABCDE"

Left "unexpected B"

GHCi> runPrsE ((,) <$> anyE <\* charE 'B' <\*> anyE) "AB"

Left "unexpected end of input"



**Шаг 13**

Сделайте парсер

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

представителем класса типов Alternative с естественной для парсера семантикой:

GHCi> runPrs (char 'A' <|> char 'B') "ABC"

Just ('A',"BC")

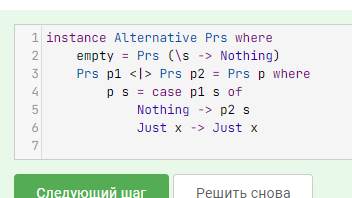
GHCi> runPrs (char 'A' <|> char 'B') "BCD"

Just ('B',"CD")

GHCi> runPrs (char 'A' <|> char 'B') "CDE"

Nothing

Представители для классов типов Functor и Applicative уже реализованы. Функцию char :: Char -> Prs Char включать в решение не нужно, но полезно реализовать для локального тестирования.



**Шаг 14**

Реализуйте для парсера

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

парсер-комбинатор many1 :: Prs a -> Prs [a], который отличается от many только тем, что он терпит неудачу в случае, когда парсер-аргумент неудачен на начале входной строки.

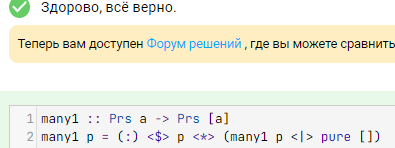
> runPrs (many1 $ char 'A') "AAABCDE"

Just ("AAA","BCDE")

> runPrs (many1 $ char 'A') "BCDE"

Nothing

Функцию char :: Char -> Prs Char включать в решение не нужно, но полезно реализовать для локального тестирования.



**Шаг 15**

Реализуйте парсер nat :: Prs Int для натуральных чисел, так чтобы парсер

mult :: Prs Int

mult = (\*) <$> nat <\* char '\*' <\*> nat

обладал таким поведением

GHCi> runPrs mult "14\*3"

Just (42,"")

GHCi> runPrs mult "64\*32"

Just (2048,"")

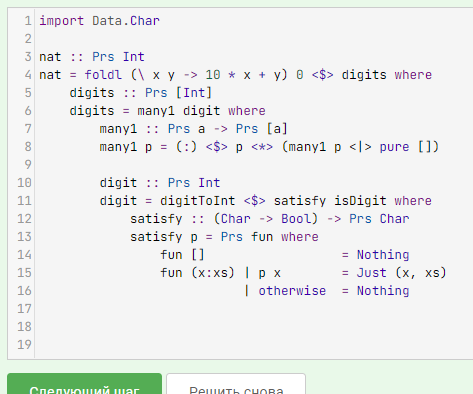
GHCi> runPrs mult "77\*0"

Just (0,"")

GHCi> runPrs mult "2\*77AAA"

Just (154,"AAA")

Реализацию функции char :: Char -> Prs Char следует включить в присылаемое решение, только если она нужна для реализации парсера nat.



**1.5 Композиция на уровне типов(3/4)**

**Шаг 3**

Населите допустимыми нерасходящимися выражениями следующие типы

type A = ((,) Integer |.| (,) Char) Bool

type B t = ((,,) Bool (t -> t) |.| Either String) Int

type C = (|.|) ((->) Bool) ((->) Integer) Integer



**Шаг 5**

Сделайте тип

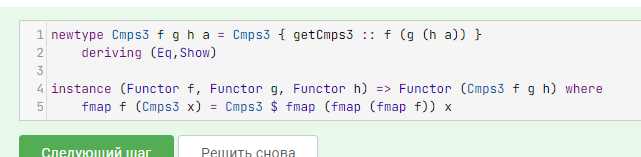
newtype Cmps3 f g h a = Cmps3 { getCmps3 :: f (g (h a)) }

deriving (Eq,Show)

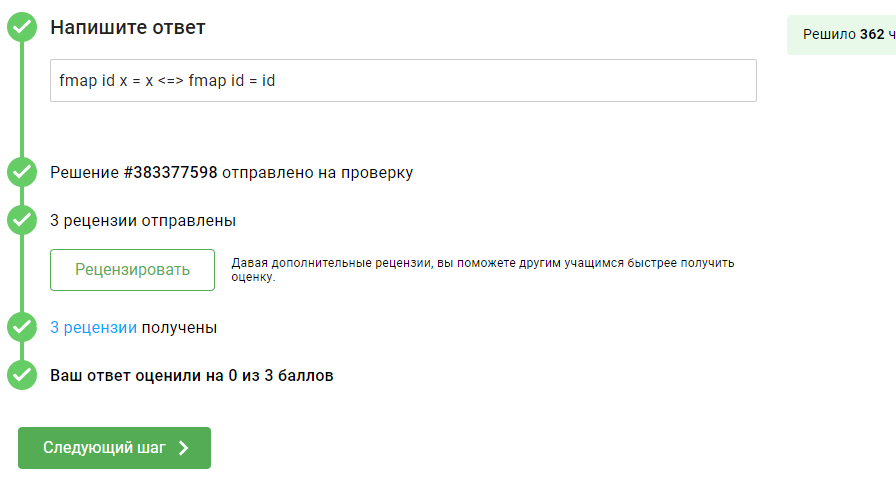
представителем класса типов Functor при условии, что первые его три параметра являются функторами:

GHCi> fmap (^2) $ Cmps3 [[[1],[2,3,4],[5,6]],[],[[7,8],[9,10,11]]]

Cmps3 {getCmps3 = [[[1],[4,9,16],[25,36]],[],[[49,64],[81,100,121]]]}



**Шаг 7**



**Шаг 9**

Напишите универсальные функции

unCmps3 :: Functor f => (f |.| g |.| h) a -> f (g (h a))

unCmps4 :: (Functor f2, Functor f1) => (f2 |.| f1 |.| g |.| h) a -> f2 (f1 (g (h a)))

позволяющие избавляться от синтаксического шума для композиции нескольких функторов:

GHCi> pure 42 :: ([] |.| [] |.| []) Int

Cmps {getCmps = [Cmps {getCmps = [[42]]}]}

GHCi> unCmps3 (pure 42 :: ([] |.| [] |.| []) Int)

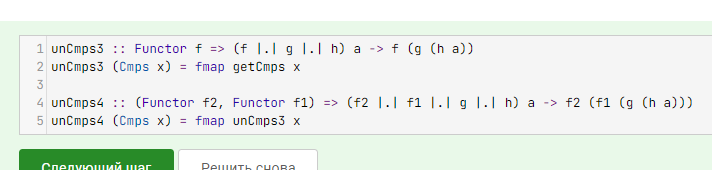
[[[42]]]

GHCi> unCmps3 (pure 42 :: ([] |.| Maybe |.| []) Int)

[Just [42]]

GHCi> unCmps4 (pure 42 :: ([] |.| [] |.| [] |.| []) Int)

[[[[42]]]]



**Модуль 2**

**2.1 Класс типов Foldable**

**Шаг 4**

Сделайте тип

data Triple a = Tr a a a deriving (Eq,Show)

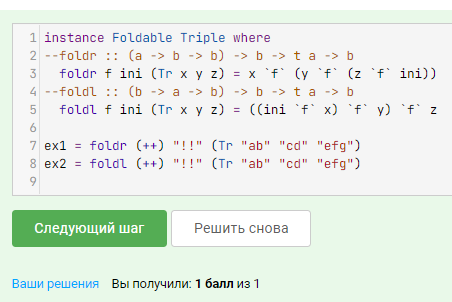
представителем класса типов Foldable:

GHCi> foldr (++) "!!" (Tr "ab" "cd" "efg")

"abcdefg!!"

GHCi> foldl (++) "!!" (Tr "ab" "cd" "efg")

"!!abcdefg"



**Шаг 6**

Для реализации свертки двоичных деревьев нужно выбрать алгоритм обхода узлов дерева (см., например, <http://en.wikipedia.org/wiki/Tree_traversal>).

Сделайте двоичное дерево

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

представителем класса типов Foldable, реализовав симметричную стратегию (in-order traversal). Реализуйте также три другие стандартные стратегии (pre-order traversal, post-order traversal и level-order traversal), сделав типы-обертки

newtype Preorder a = PreO (Tree a) deriving (Eq, Show)

newtype Postorder a = PostO (Tree a) deriving (Eq, Show)

newtype Levelorder a = LevelO (Tree a) deriving (Eq, Show)

представителями класса Foldable.

GHCi> tree = Branch (Branch Nil 1 (Branch Nil 2 Nil)) 3 (Branch Nil 4 Nil)

GHCi> foldr (:) [] tree

[1,2,3,4]

GHCi> foldr (:) [] $ PreO tree

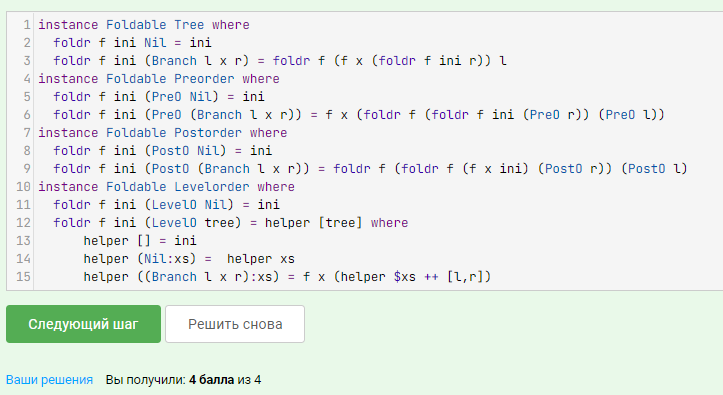
[3,1,2,4]

GHCi> foldr (:) [] $ PostO tree

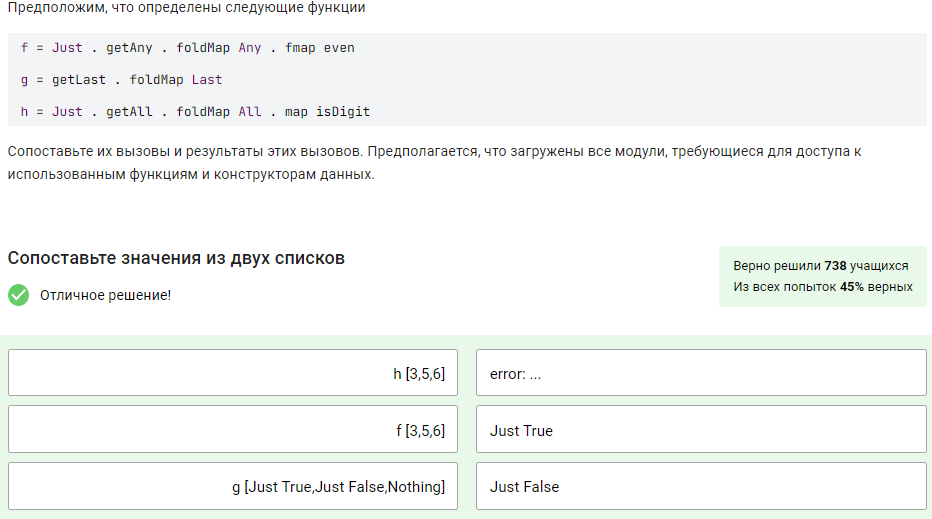
[2,1,4,3]

GHCi> foldr (:) [] $ LevelO tree

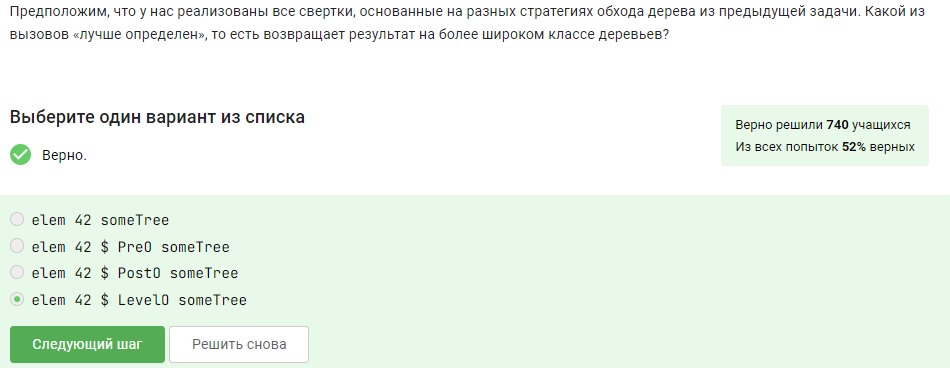
[3,1,4,2]



**Шаг 8**



**Шаг 10**



**Шаг 13**

Реализуйте функцию

mkEndo :: Foldable t => t (a -> a) -> Endo a

принимающую контейнер функций и последовательно сцепляющую элементы этого контейнера с помощью композиции, порождая в итоге эндоморфизм.

GHCi> e1 = mkEndo [(+5),(\*3),(^2)]

GHCi> appEndo e1 2

17

GHCi> e2 = mkEndo (42,(\*3))

GHCi> appEndo e2 2

6



**Шаг 16**

Сделайте тип

infixr 9 |.|

newtype (|.|) f g a = Cmps { getCmps :: f (g a) } deriving (Eq,Show)

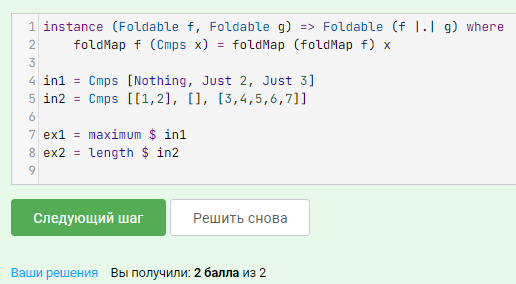
представителем класса типов Foldable при условии, что аргументы композиции являются представителями Foldable.

GHCi> maximum $ Cmps [Nothing, Just 2, Just 3]

3

GHCi> length $ Cmps [[1,2], [], [3,4,5,6,7]]

7



**2.2 Класс типов Traversable**

**Шаг 4**

Предположим для двоичного дерева

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

реализован представитель класса типов Foldable, обеспечивающий стратегию обхода pre-order traversal. Какую строку вернет следующий вызов

GHCi> tree = Branch (Branch Nil 1 Nil) 2 (Branch (Branch Nil 3 Nil) 4 (Branch Nil 5 Nil))

GHCi> fst $ sequenceA\_ $ (\x -> (show x,x)) <$> tree

**Ответ: 21435**

**Шаг 7**

Реализуйте функцию

traverse2list :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f [b]

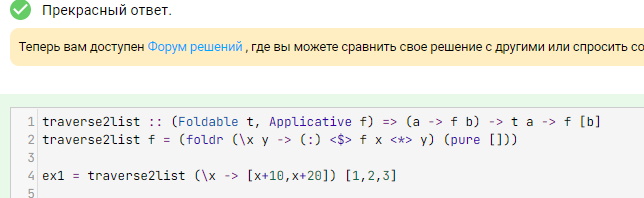
работающую с эффектами как traverse\_, , но параллельно с накоплением эффектов «восстанавливающую» сворачиваемую структуру в виде списка:

GHCi> traverse2list (\x -> [x+10,x+20]) [1,2,3]

[[11,12,13],[11,12,23],[11,22,13],[11,22,23],[21,12,13],[21,12,23],[21,22,13],[21,22,23]]

GHCi> traverse2list (\x -> [x+10,x+20]) $ Branch (Branch Nil 1 Nil) 2 (Branch Nil 3 Nil)

[[12,11,13],[12,11,23],[12,21,13],[12,21,23],[22,11,13],[22,11,23],[22,21,13],[22,21,23]]



**Шаг 11**

Сделайте тип

data Triple a = Tr a a a deriving (Eq,Show)

представителем класса типов Traversable:

GHCi> foldl (++) "!!" (Tr "ab" "cd" "efg")

"!!abcdefg"

GHCi> traverse (\x -> if x>10 then Right x else Left x) (Tr 12 14 16)

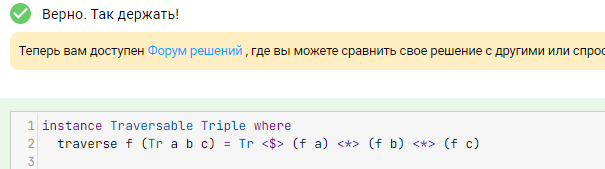
Right (Tr 12 14 16)

GHCi> traverse (\x -> if x>10 then Right x else Left x) (Tr 12 8 4)

Left 8

GHCi> sequenceA (Tr (Tr 1 2 3) (Tr 4 5 6) (Tr 7 8 9))

Tr (Tr 1 4 7) (Tr 2 5 8) (Tr 3 6 9)



**Шаг 12**

Сделайте тип данных

data Result a = Ok a | Error String deriving (Eq,Show)

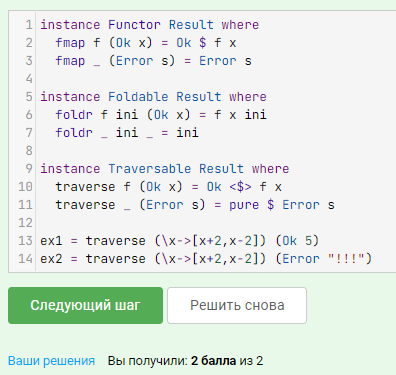
представителем класса типов Traversable (и всех других необходимых классов типов).

GHCi> traverse (\x->[x+2,x-2]) (Ok 5)

[Ok 7,Ok 3]

GHCi> traverse (\x->[x+2,x-2]) (Error "!!!")

[Error "!!!"]



**Шаг 14**

Сделайте двоичное дерево

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

представителем класса типов Traversable (а также всех других необходимых классов типов).

GHCi> traverse (\x -> if odd x then Right x else Left x) (Branch (Branch Nil 1 Nil) 3 Nil)

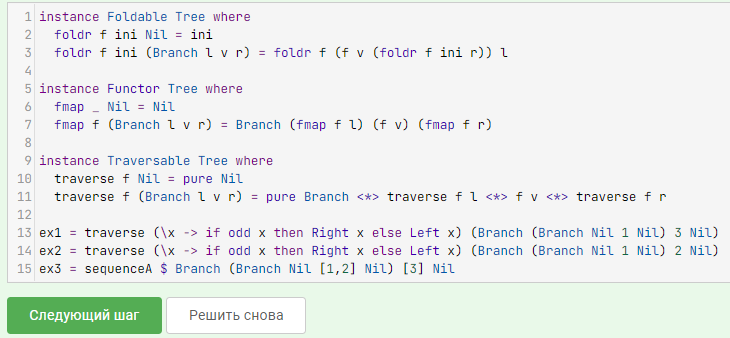
Right (Branch (Branch Nil 1 Nil) 3 Nil)

GHCi> traverse (\x -> if odd x then Right x else Left x) (Branch (Branch Nil 1 Nil) 2 Nil)

Left 2

GHCi> sequenceA $ Branch (Branch Nil [1,2] Nil) [3] Nil

[Branch (Branch Nil 1 Nil) 3 Nil,Branch (Branch Nil 2 Nil) 3 Nil]



**Шаг 15**

Сделайте тип

infixr 9 |.|

newtype (|.|) f g a = Cmps { getCmps :: f (g a) } deriving (Eq,Show)

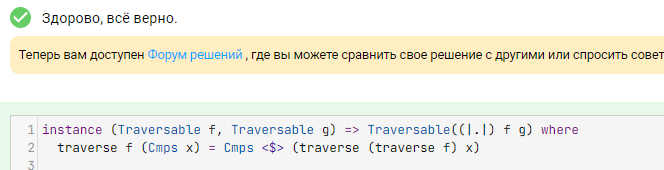
представителем класса типов Traversable при условии, что аргументы композиции являются представителями Traversable.

GHCi> sequenceA (Cmps [Just (Right 2), Nothing])

Right (Cmps {getCmps = [Just 2,Nothing]})

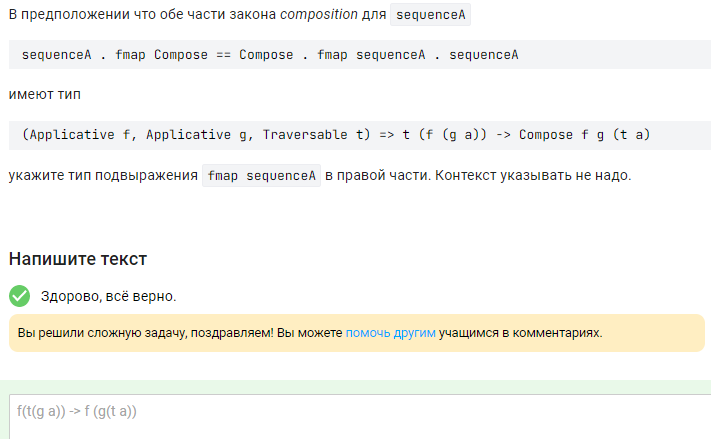
GHCi> sequenceA (Cmps [Just (Left 2), Nothing])

Left 2

****

**2.3 Законы и свойства класса Traversable**

**Шаг 5**



**Шаг 6**

Рассмотрим следующий тип данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

Этот тип представляет собой контейнер-последовательность, который по построению может содержать только нечетное число элементов:

GHCi> cnt1 = Un 42

GHCi> cnt3 = Bi 1 2 cnt1

GHCi> cnt5 = Bi 3 4 cnt3

GHCi> cnt5

Bi 3 4 (Bi 1 2 (Un 42))

GHCi> cntInf = Bi 'A' 'B' cntInf

GHCi> cntInf

Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'Interrupted.

GHCi>

Сделайте этот тип данных представителем классов типов Functor, Foldable и Traversable:

GHCi> (+1) <$> cnt5

Bi 4 5 (Bi 2 3 (Un 43))

GHCi> toList cnt5

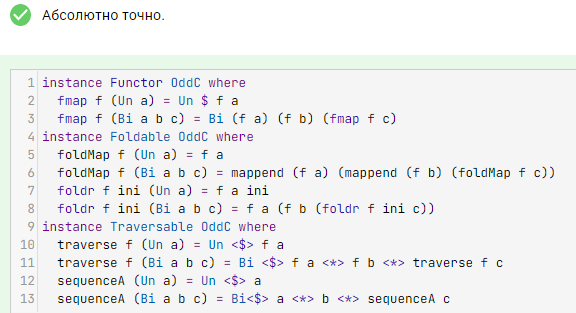
[3,4,1,2,42]

GHCi> sum cnt5

52

GHCi> traverse (\x->[x+2,x-2]) cnt1

[Un 44,Un 40]



**Шаг 8**

Расширьте интерфейс для работы с температурами из предыдущего видео Кельвинами и реализуйте функцию

k2c :: Temperature Kelvin -> Temperature Celsius

обеспечивающую следующее поведение

GHCi> k2c 0

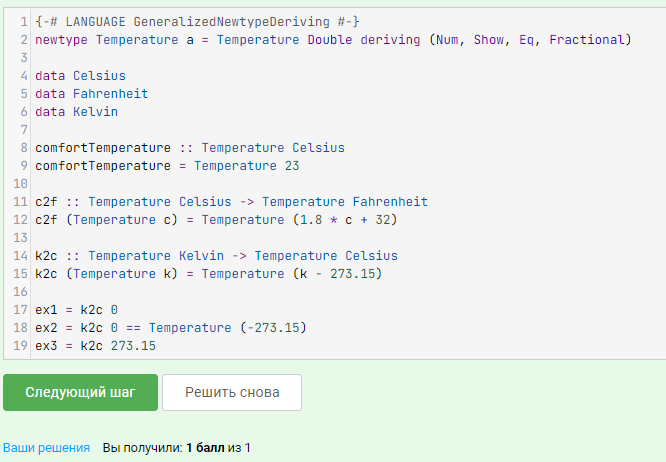
Temperature (-273.15)

GHCi> k2c 0 == Temperature (-273.15)

True

GHCi> k2c 273.15

Temperature 0.0



**Шаг 12**

Сделайте двоичное дерево

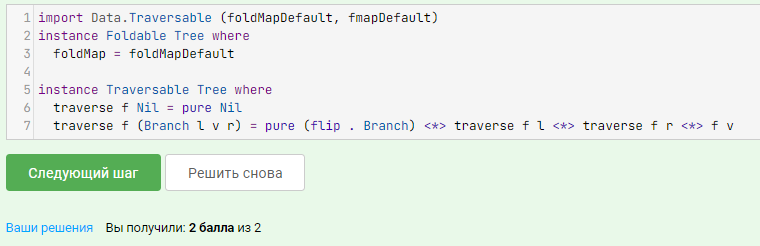
data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

представителем класса типов Traversable таким образом, чтобы обеспечить для foldMapDefault порядок обхода «postorder traversal»:

GHCi> testTree = Branch (Branch (Branch Nil 1 Nil) 2 (Branch Nil 3 Nil)) 4 (Branch Nil 5 Nil)

GHCi> foldMapDefault (\x -> [x]) testTree

[1,3,2,5,4]



**2.4 Связь классов Monad и Appliccative**

**Шаг 7**

Сделайте парсер

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

из [первого модуля курса](https://stepik.org/lesson/%D0%90%D0%BF%D0%BF%D0%BB%D0%B8%D0%BA%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9-%D0%BF%D0%B0%D1%80%D1%81%D0%B5%D1%80-%D1%81%D0%B2%D0%BE%D0%B8%D0%BC%D0%B8-%D1%80%D1%83%D0%BA%D0%B0%D0%BC%D0%B8-30425/step/8) представителем класса типов Monad:

GHCi> runPrsE (do {a <- charE 'A'; b <- charE 'B'; return (a,b)}) "ABC"

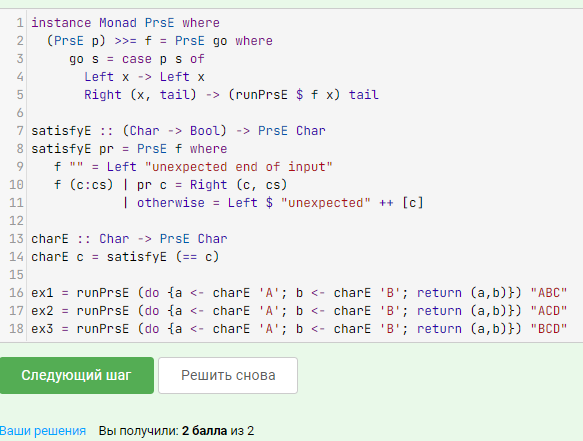
Right (('A','B'),"C")

GHCi> runPrsE (do {a <- charE 'A'; b <- charE 'B'; return (a,b)}) "ACD"

Left "unexpected C"

GHCi> runPrsE (do {a <- charE 'A'; b <- charE 'B'; return (a,b)}) "BCD"

Left "unexpected B"



**Шаг 10**

Для типа данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

(контейнер-последовательность, который по построению может содержать только нечетное число элементов) реализуйте функцию

concat3OC :: OddC a -> OddC a -> OddC a -> OddC a

конкатенирующую три таких контейнера в один:

GHCi> tst1 = Bi 'a' 'b' (Un 'c')

GHCi> tst2 = Bi 'd' 'e' (Bi 'f' 'g' (Un 'h'))

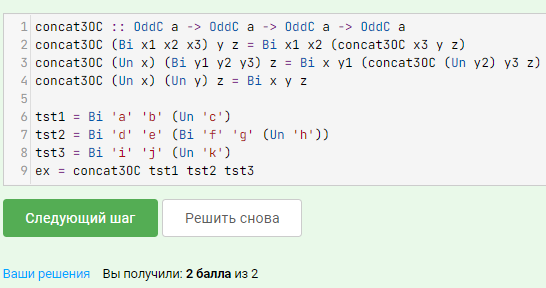
GHCi> tst3 = Bi 'i' 'j' (Un 'k')

GHCi> concat3OC tst1 tst2 tst3

Bi 'a' 'b' (Bi 'c' 'd' (Bi 'e' 'f' (Bi 'g' 'h' (Bi 'i' 'j' (Un 'k')))))

Обратите внимание, что соображения четности запрещают конкатенацию *двух* контейнеров OddC.

Реализуйте всё «честно», не сводя к стандартным спискам.



**Шаг 11**

Для типа данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

реализуйте функцию

concatOC :: OddC (OddC a) -> OddC a

Она должна обеспечивать для типа OddC поведение, аналогичное поведению функции concat для списков:

GHCi> concatOC $ Un (Un 42)

Un 42

GHCi> tst1 = Bi 'a' 'b' (Un 'c')

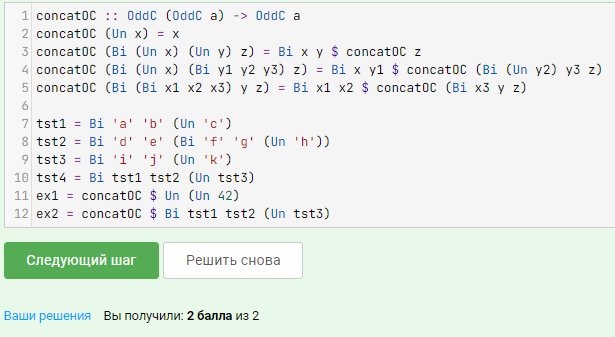
GHCi> tst2 = Bi 'd' 'e' (Bi 'f' 'g' (Un 'h'))

GHCi> tst3 = Bi 'i' 'j' (Un 'k')

GHCi> concatOC $ Bi tst1 tst2 (Un tst3)

Bi 'a' 'b' (Bi 'c' 'd' (Bi 'e' 'f' (Bi 'g' 'h' (Bi 'i' 'j' (Un 'k')))))

Реализуйте всё «честно», не сводя к стандартным спискам.

**Шаг 12**

Сделайте тип данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

представителем классов типов Functor, Applicative и Monad. Семантика должна быть подобной семантике представителей этих классов типов для списков: монада OddC должна иметь эффект вычисления с произвольным нечетным числом результатов:

GHCi> tst1 = Bi 10 20 (Un 30)

GHCi> tst2 = Bi 1 2 (Bi 3 4 (Un 5))

GHCi> do {x <- tst1; y <- tst2; return (x + y)}

Bi 11 12 (Bi 13 14 (Bi 15 21 (Bi 22 23 (Bi 24 25 (Bi 31 32 (Bi 33 34 (Un 35)))))))

GHCi> do {x <- tst2; y <- tst1; return (x + y)}

Bi 11 21 (Bi 31 12 (Bi 22 32 (Bi 13 23 (Bi 33 14 (Bi 24 34 (Bi 15 25 (Un 35)))))))

Функцию fail можно не реализовывать, полагаясь на реализацию по умолчанию.

Реализуйте всё «честно», не сводя к стандартным спискам.

Ответ :

instance Functor OddC where

--fmap :: (a -> b) -> OddC a -> OddC b

fmap f (Un x) = Un $ f x

fmap f (Bi x y z) = Bi (f x) (f y) (f <$> z)

concatOC :: OddC (OddC a) -> OddC a

concatOC (Un x) = x

concatOC (Bi (Un x) (Un y) z) = Bi x y $ concatOC z

concatOC (Bi (Un x) (Bi y1 y2 y3) z) = Bi x y1 $ concatOC (Bi (Un y2) y3 z)

concatOC (Bi (Bi x1 x2 x3) y z) = Bi x1 x2 $ concatOC (Bi x3 y z)

concat3OC :: OddC a -> OddC a -> OddC a -> OddC a

concat3OC (Bi x1 x2 x3) y z = Bi x1 x2 (concat3OC x3 y z)

concat3OC (Un x) (Bi y1 y2 y3) z = Bi x y1 (concat3OC (Un y2) y3 z)

concat3OC (Un x) (Un y) z = Bi x y z

instance Applicative OddC where

pure = Un

(Un f) <\*> (Un v) = Un $ f v

(Un f) <\*> (Bi v1 v2 v3) = Bi (f v1) (f v2) (f <$> v3)

(Bi f1 f2 f3) <\*> (Un v) = Bi (f1 v) (f2 v) (f3 <\*> (Un v))

(Bi f1 f2 f3) <\*> (Bi v1 v2 v3) =

let

frst = (Bi (f1 v1) (f1 v2) (f1 <$> v3))

scnd = (Bi (f2 v1) (f2 v2) (f2 <$> v3))

thrd = concatOC $ (Bi (f3 <\*> (Un v1)) (f3 <\*> (Un v2)) (Un $ f3 <\*> v3))

in concat3OC frst scnd thrd

instance Monad OddC where

(Un l) >>= f = f l

(Bi x y z) >>= f =

let

frst = f x

scnd = f y

thrd = z >>= f

in concat3OC frst scnd thrd

tst1 = Bi 10 20 (Un 30)

tst2 = Bi 1 2 (Bi 3 4 (Un 5))

tst3 = Bi (+1) (+2) $ Un (^2)

tst4 = Bi 1 2 tst4

ex1 = do {x <- tst1; y <- tst2; return (x + y)}

ex2 = do {x <- tst2; y <- tst1; return (x + y)}

ex3 = tst3 <\*> tst4

ex4 = do {x <- tst1; y <- tst4; return (x + y)}

**2.5 Класс типов Alternative и MonadPlus (4/5)**

**Шаг 5**

Выполняются ли для стандартных представителей Applicative, Alternative, Monad и MonadPlus типа данных Maybe следующие законы дистрибутивности:

(u <|> v) <\*> w = u <\*> w <|> v <\*> w

(u `mplus` v) >>= k = (u >>= k) `mplus` (v >>= k)

Если нет, то приведите контрпример, если да, то доказательство.

Предполагается, что расходимости отсутствуют.



**Шаг 6**

Предположим мы сделали парсер

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

представителем классов типов Alternative следующим образом

instance Alternative PrsE where

empty = PrsE f where

f \_ = Left "empty alternative"

p <|> q = PrsE f where

f s = let ps = runPrsE p s

in if null ps

then runPrsE q s

else ps

Эта реализация нарушает закон дистрибутивности для Alternative:

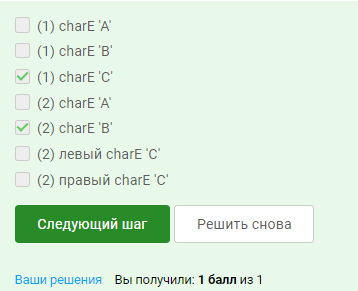
GHCi> runPrsE ((charE 'A' <|> charE 'B') \*> charE 'C') "ABC"

Left "unexpected B"

GHCi> runPrsE (charE 'A' \*> charE 'C' <|> charE 'B' \*> charE 'C') "ABC"

Left "unexpected A"

От какого парсера приходит сообщение об ошибке в первом и втором примерах?



**Шаг 7**

Реализуем улучшенную версию парсера PrsE

newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

parseEP :: PrsEP a -> String -> Either String (a, String)

parseEP p = snd . runPrsEP p 0

Этот парсер получил дополнительный целочисленный параметр в аргументе и в возвращаемом значении. С помощью этого параметра мы сможем отслеживать и передвигать текущую позицию в разбираемой строке и сообщать о ней пользователю в случае ошибки:

GHCi> charEP c = satisfyEP (== c)

GHCi> runPrsEP (charEP 'A') 0 "ABC"

(1,Right ('A',"BC"))

> runPrsEP (charEP 'A') 41 "BCD"

(42,Left "pos 42: unexpected B")

> runPrsEP (charEP 'A') 41 ""

(42,Left "pos 42: unexpected end of input")

Вспомогательная функция parseEP дает возможность вызывать парсер более удобным образом по сравнению с runPrsEP, скрывая технические детали:

GHCi> parseEP (charEP 'A') "ABC"

Right ('A',"BC")

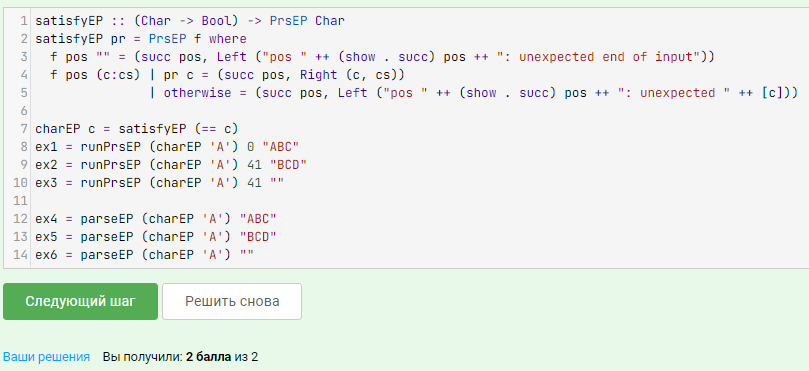
GHCi> parseEP (charEP 'A') "BCD"

Left "pos 1: unexpected B"

GHCi> parseEP (charEP 'A') ""

Left "pos 1: unexpected end of input"

Реализуйте функцию satisfyEP :: (Char -> Bool) -> PrsEP Char, обеспечивающую описанное выше поведение.

**Шаг 8**

Сделайте парсер

newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

parseEP :: PrsEP a -> String -> Either String (a, String)

parseEP p = snd . runPrsEP p 0

представителем классов типов Functor и Applicative, обеспечив следующее поведение:

GHCi> runPrsEP (pure 42) 0 "ABCDEFG"

(0,Right (42,"ABCDEFG"))

GHCi> charEP c = satisfyEP (== c)

GHCi> anyEP = satisfyEP (const True)

GHCi> testP = (,) <$> anyEP <\* charEP 'B' <\*> anyEP

GHCi> runPrsEP testP 0 "ABCDE"

(3,Right (('A','C'),"DE"))

GHCi> parseEP testP "BCDE"

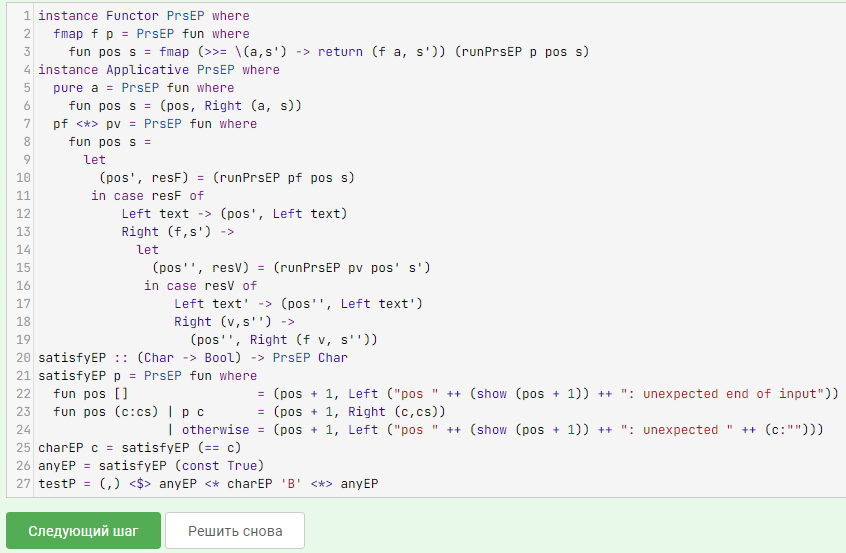
Left "pos 2: unexpected C"

GHCi> parseEP testP ""

Left "pos 1: unexpected end of input"

GHCi> parseEP testP "B"

Left "pos 2: unexpected end of input"

**Шаг 9**

Сделайте парсер

newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

parseEP :: PrsEP a -> String -> Either String (a, String)

parseEP p = snd . runPrsEP p 0

представителем класса типов Alternative, обеспечив следующее поведение для пары неудачных альтернатив: сообщение об ошибке возвращается из той альтернативы, которой удалось распарсить входную строку глубже.

GHCi> runPrsEP empty 0 "ABCDEFG"

(0,Left "pos 0: empty alternative")

GHCi> charEP c = satisfyEP (== c)

GHCi> tripleP [a,b,c] = (\x y z -> [x,y,z]) <$> charEP a <\*> charEP b <\*> charEP c

GHCi> parseEP (tripleP "ABC" <|> tripleP "ADC") "ABE"

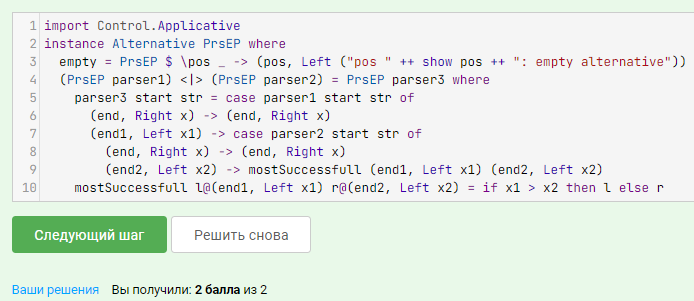
Left "pos 3: unexpected E"

GHCi> parseEP (tripleP "ABC" <|> tripleP "ADC") "ADE"

Left "pos 3: unexpected E"

GHCi> parseEP (tripleP "ABC" <|> tripleP "ADC") "AEF"

Left "pos 2: unexpected E"

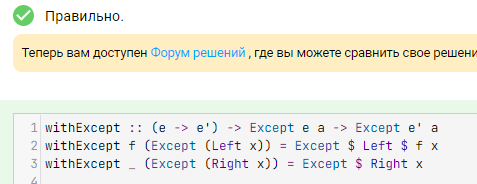


**Модуль 3**

**3.1 Монада Except**

**Шаг 3**

Реализуйте функцию withExcept :: (e -> e') -> Except e a -> Except e' a, позволящую, если произошла ошибка, применить к ней заданное преобразование.



**Шаг 7**

В модуле Control.Monad.Trans.Except библиотеки transformers имеется реализация монады Except с интерфейсом, идентичным представленному в видео-степах, но с более общими типами. Мы изучим эти типы в следующих модулях, однако использовать монаду Except из библиотеки transformers мы можем уже сейчас.

Введём тип данных для представления ошибки обращения к списку по недопустимому индексу:

data ListIndexError = ErrIndexTooLarge Int | ErrNegativeIndex

deriving (Eq, Show)

Реализуйте оператор (!!!) :: [a] -> Int -> Except ListIndexError a доступа к элементам массива по индексу, отличающийся от стандартного (!!) поведением в исключительных ситуациях. В этих ситуациях он должен выбрасывать подходящее исключение типа ListIndexError.

GHCi> runExcept $ [1..100] !!! 5

Right 6

GHCi> (!!!!) xs n = runExcept $ xs !!! n

GHCi> [1,2,3] !!!! 0

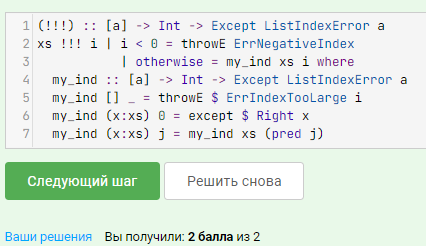
Right 1

GHCi> [1,2,3] !!!! 42

Left (ErrIndexTooLarge 42)

GHCi> [1,2,3] !!!! (-3)

Left ErrNegativeIndex



**Шаг 8**

Реализуйте функцию tryRead, получающую на вход строку и пытающуюся всю эту строку превратить в значение заданного типа. Функция должна возвращать ошибку в одном из двух случаев: если вход был пуст или если прочитать значение не удалось.

Информация об ошибке хранится в специальном типе данных:

data ReadError = EmptyInput | NoParse String

  deriving Show

GHCi> runExcept (tryRead "5" :: Except ReadError Int)

Right 5

GHCi> runExcept (tryRead "5" :: Except ReadError Double)

Right 5.0

GHCi> runExcept (tryRead "5zzz" :: Except ReadError Int)

Left (NoParse "5zzz")

GHCi> runExcept (tryRead "(True, ())" :: Except ReadError (Bool, ()))

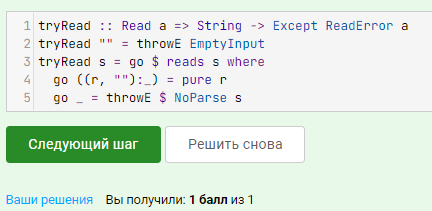
Right (True,())

GHCi> runExcept (tryRead "" :: Except ReadError (Bool, ()))

Left EmptyInput

GHCi> runExcept (tryRead "wrong" :: Except ReadError (Bool, ()))

Left (NoParse "wrong")



**Шаг 9**

Используя tryRead из прошлого задания, реализуйте функцию trySum, которая получает список чисел, записанных в виде строк, и суммирует их. В случае неудачи, функция должна возвращать информацию об ошибке вместе с номером элемента списка (нумерация с единицы), вызвавшим ошибку.

Для хранения информации об ошибке и номере проблемного элемента используем новый тип данных:

data SumError = SumError Int ReadError

  deriving Show

GHCi> runExcept $ trySum ["10", "20", "30"]

Right 60

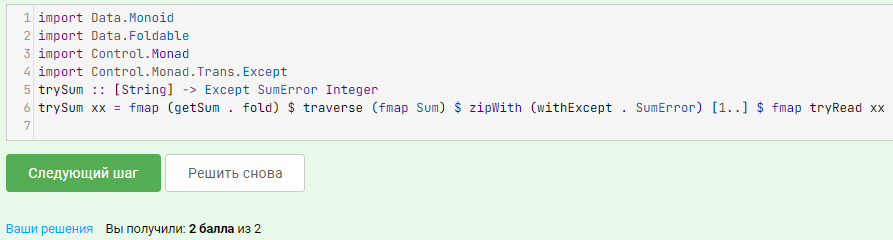
GHCi> runExcept $ trySum ["10", "20", ""]

Left (SumError 3 EmptyInput)

GHCi> runExcept $ trySum ["10", "two", "30"]

Left (SumError 2 (NoParse "two"))

Подсказка: функция [withExcept](https://hackage.haskell.org/package/transformers-0.5.4.0/docs/Control-Monad-Trans-Except.html" \l "v:withExcept" \t "_blank) в этом задании может быть чрезвычайно полезна. Постарайтесь максимально эффективно применить знания, полученные на прошлой неделе.



**Шаг 13**

Тип данных для представления ошибки обращения к списку по недопустимому индексу

data ListIndexError = ErrIndexTooLarge Int | ErrNegativeIndex

deriving (Eq, Show)

не очень естественно делать представителем класса типов Monoid. Однако, если мы хотим обеспечить накопление информации об ошибках, моноид необходим. К счастью, уже знакомая нам функция [withExcept](http://hackage.haskell.org/package/transformers-0.5.4.0/docs/Control-Monad-Trans-Except.html" \l "v:withExcept" \t "_blank) :: (e -> e') -> Except e a -> Except e' a позволяет изменять тип ошибки при вычислении в монаде Except.

Сделайте тип данных

newtype SimpleError = Simple { getSimple :: String }

deriving (Eq, Show)

представителем необходимых классов типов и реализуйте преобразователь для типа данных ошибки lie2se :: ListIndexError -> SimpleError так, чтобы обеспечить следующее поведение

GHCi> toSimple = runExcept . withExcept lie2se

GHCi> xs = [1,2,3]

GHCi> toSimple $ xs !!! 42

Left (Simple {getSimple = "[index (42) is too large]"})

GHCi> toSimple $ xs !!! (-2)

Left (Simple {getSimple = "[negative index]"})

GHCi> toSimple $ xs !!! 2

Right 3

GHCi> import Data.Foldable (msum)

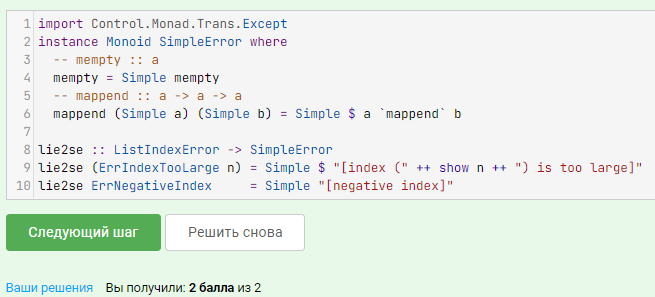
GHCi> toSimpleFromList = runExcept . msum . map (withExcept lie2se)

GHCi> toSimpleFromList [xs !!! (-2), xs !!! 42]

Left (Simple {getSimple = "[negative index][index (42) is too large]"})

GHCi> toSimpleFromList [xs !!! (-2), xs !!! 2]

Right 3



**Шаг 14**

Стандартная семантика Except как аппликативного функтора и монады: выполнять цепочку вычислений до первой ошибки. Реализация представителей классов Alternative и MonadPlus наделяет эту монаду альтернативной☺ семантикой: попробовать несколько вычислений, вернуть результат первого успешного, а в случае неудачи — все возникшие ошибки.

Довольно часто возникает необходимость сделать нечто среднее. К примеру, при проверке корректности заполнения анкеты или при компиляции программы для общего успеха необходимо, чтобы ошибок совсем не было, но в то же время, нам хотелось бы не останавливаться после первой же ошибки, а продолжить проверку, чтобы отобразить сразу все проблемы. Except такой семантикой не обладает, но никто не может помешать нам сделать свой тип данных (назовем его Validate), представители которого будут обеспечивать требую семантику, позволяющую сохранить список всех произошедших ошибок:

newtype Validate e a = Validate { getValidate :: Either [e] a }

Реализуйте функцию validateSum :: [String] -> Validate SumError Integer:

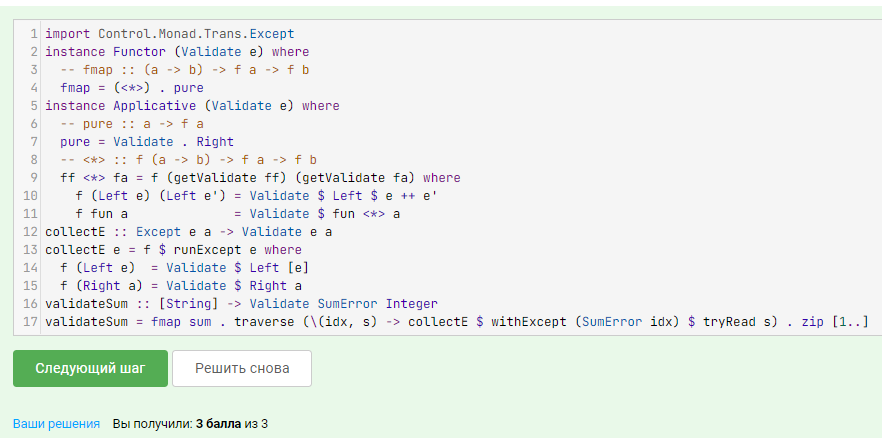
GHCi> getValidate $ validateSum ["10", "20", "30"]

Right 60

GHCi> getValidate $ validateSum ["10", "", "30", "oops"]

Left [SumError 2 EmptyInput,SumError 4 (NoParse "oops")]

Эта функция практически ничем не отличается от уже реализованной ранее trySum, если использовать функцию-адаптер collectE :: Except e a -> Validate e a и представителей каких-нибудь классов типов для Validate.



**3.2 Монада Cont**

**Шаг 3**

CPS-преобразование часто применяют для создания [предметно-ориентированных языков](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%BC%D0%B5%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ([DSL](https://en.wikipedia.org/wiki/Domain-specific_language)).

Реализуйте комбинаторы, которые позволят записывать числа вот в таком забавном формате:

GHCi> decode one hundred twenty three as a number

123

GHCi> decode one hundred twenty one as a number

121

GHCi> decode one hundred twenty as a number

120

GHCi> decode one hundred as a number

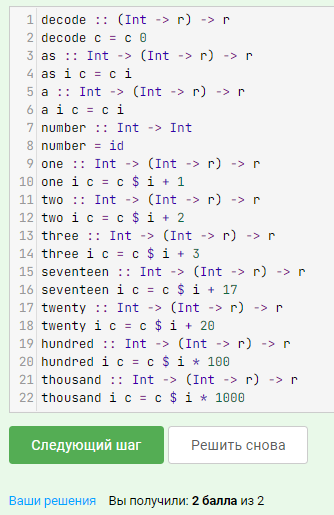
100

GHCi> decode three hundred as a number

300

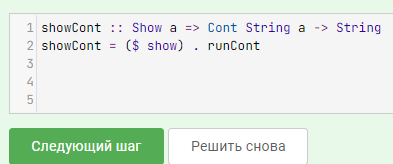
GHCi> decode two thousand seventeen as a number

2017



**Шаг 6**

Реализуйте функцию showCont, запускающую вычисление и возвращающую его результат в виде строки.



**Шаг 9**

Возможность явно работать с продолжением обеспечивает доступ к очень гибкому управлению исполнением. В этом задании вам предстоит реализовать вычисление, которое анализирует и модифицирует значение, возвращаемое кодом, написанным *после* него.

В качестве примера рассмотрим следующую функцию:

addTens :: Int -> Checkpointed Int

addTens x1 = \checkpoint -> do

  checkpoint x1

  let x2 = x1 + 10

  checkpoint x2 {- x2 = x1 + 10 -}

  let x3 = x2 + 10

  checkpoint x3 {- x3 = x1 + 20 -}

  let x4 = x3 + 10

  return x4 {- x4 = x1 + 30 -}

Эта функция принимает значение x1, совершает с ним последовательность операций (несколько раз прибавляет 10) и после каждой операции «сохраняет» промежуточный результат. При запуске такой функции используется дополнительный предикат, который является критерием «корректности» результата, и в случае, если возвращенное функцией значение этому критерию не удовлетворяет, вернется последнее удовлетворяющее ему значение из «сохраненных»:

GHCi> runCheckpointed (< 100) $ addTens 1

31

GHCi> runCheckpointed (< 30) $ addTens 1

21

GHCi> runCheckpointed (< 20) $ addTens 1

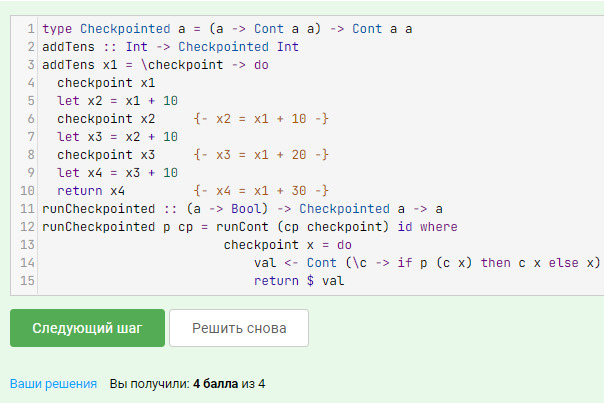
11

GHCi> runCheckpointed (< 10) $ addTens 1

1

(Если ни возвращенное, ни сохраненные значения не подходят, результатом должно быть первое из сохраненных значений; если не было сохранено ни одного значения, то результатом должно быть возвращенное значение.)

Обратите внимание на то, что функция checkpoint передается в Checkpointed вычисление как параметр, поскольку её поведение зависит от предиката, который будет известен только непосредственно при запуске.



**Шаг 10**

Вычисление в монаде Cont передает результат своей работы в функцию-продолжение. А что, если наши вычисления могут завершиться с ошибкой? В этом случае мы могли бы явно возвращать значение типа Either и каждый раз обрабатывать два возможных исхода, что не слишком удобно. Более разумный способ решения этой проблемы предоставляют трансформеры монад, но с ними мы познакомимся немного позже.

Определите тип данных FailCont для вычислений, которые получают *два* продолжения и вызывают одно из них в случае успеха, а другое — при неудаче. Сделайте его представителем класса типов Monad и реализуйте вспомогательные функции toFailCont и evalFailCont, используемые в следующем коде:

add :: Int -> Int -> FailCont r e Int

add x y = FailCont $ \ok \_ -> ok $ x + y

addInts :: String -> String -> FailCont r ReadError Int

addInts s1 s2 = do

  i1 <- toFailCont $ tryRead s1

  i2 <- toFailCont $ tryRead s2

  return $ i1 + i2

(Здесь используется функция tryRead из предыдущего урока; определять её заново не надо.)

GHCi> evalFailCont $ addInts "15" "12"

Right 27

GHCi> runFailCont (addInts "15" "") print (putStrLn . ("Oops: " ++) . show)

Oops: EmptyInput

Ответ:

import Control.Monad.Trans.Except

import Control.Monad.Trans.Cont

import Control.Monad(ap)

import Control.Applicative(liftA)

newtype FailCont r e a = FailCont { runFailCont :: (a -> r) -> (e -> r) -> r }

-- deriving Functor

instance Functor (FailCont r e) where

fmap = liftA

instance Applicative (FailCont r e) where

pure = return

(<\*>) = ap

bindCont :: ((a -> r) -> r)

-> (a -> (b -> r) -> r)

-> (b -> r) -> r

bindCont f g = \h -> f (\a -> g a h)

bindFailCont :: ((a -> r) -> (e -> r) -> r)

-> (a -> (b -> r) -> (e -> r) -> r)

-> (b -> r) -> (e -> r) -> r

bindFailCont f g = \h -> \k -> f (\a -> g a h k) k

instance Monad (FailCont r e) where

return x = FailCont $ \c \_ -> c x

FailCont f >>= g =

FailCont $ \h -> \k ->

f (\a -> runFailCont (g a) h k) k

toFailCont :: Except e a -> FailCont r e a

toFailCont e = FailCont $ \f -> \g ->

case (runExcept e) of

Right x -> f x

Left x -> g x

evalFailCont :: FailCont (Either e a) e a -> Either e a

evalFailCont (FailCont f) = f Right Left

add :: Int -> Int -> FailCont r e Int

add x y = FailCont $ \ok \_ -> ok $ x + y

addInts :: String -> String -> FailCont r ReadError Int

addInts s1 s2 = do

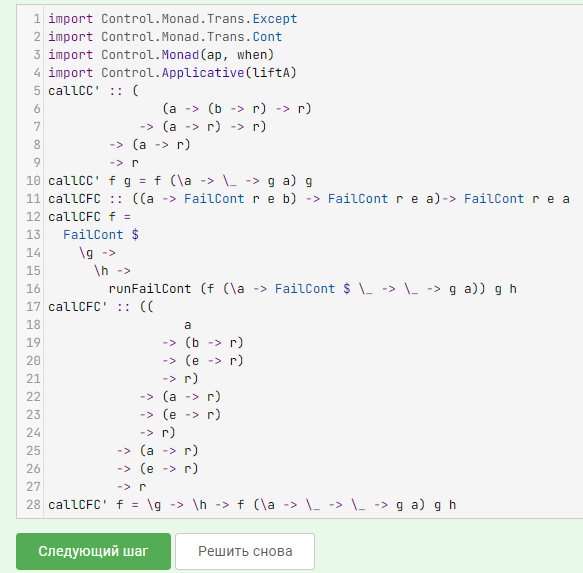
i1 <- toFailCont $ tryRead s1

i2 <- toFailCont $ tryRead s2

return $ i1 + i2

**Шаг 13**

Реализуйте функцию callCFC для монады FailCont по аналогии с callCC.



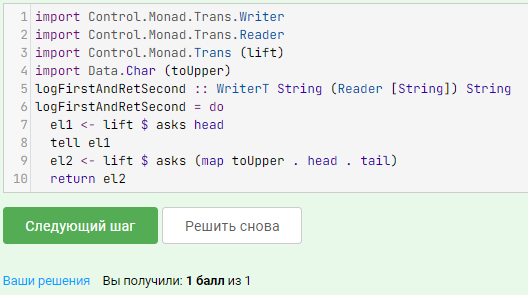
**3.3 Трансформеры монад**

**Шаг 4**

Перепишите функцию logFirstAndRetSecond из предыдущего видео, используя трансформер WriterT из модуля Control.Monad.Trans.Writer библиотеки transformers, и монаду Reader в качестве базовой.

GHCi> runReader (runWriterT logFirstAndRetSecond) strings

("DEFG","abc")



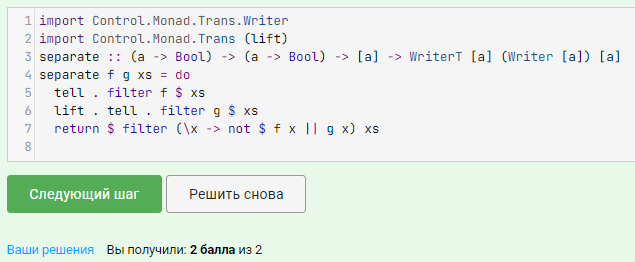
**Шаг 5**

Реализуйте функцию separate :: (a -> Bool) -> (a -> Bool) -> [a] -> WriterT [a] (Writer [a]) [a].

Эта функция принимает два предиката и список и записывает в один лог элементы списка, удовлетворяющие первому предикату, в другой лог — второму предикату, а возвращающает список элементов, ни одному из них не удовлетворяющих.

GHCi> (runWriter . runWriterT) $ separate (<3) (>7) [0..10]

(([3,4,5,6,7],[0,1,2]),[8,9,10])



**Шаг 7**

Наша абстракция пока что недостаточно хороша, поскольку пользователь всё ещё должен помнить такие детали, как, например, то, что asks нужно вызывать напрямую, а tell — только с помощью lift.

Нам хотелось бы скрыть такие детали реализации, обеспечив унифицированный интерфейс доступа к возможностям нашей монады, связанным с чтением окружения, и к возможностям, связанным с записью в лог. Для этого реализуйте функции myAsks и myTell, позволяющие записать logFirstAndRetSecond следующим образом:

logFirstAndRetSecond :: MyRW String

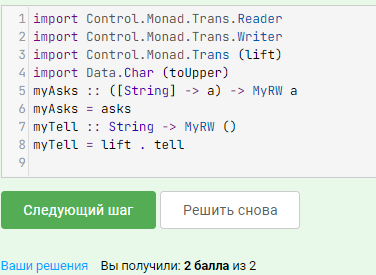
logFirstAndRetSecond = do

  el1 <- myAsks head

  el2 <- myAsks (map toUpper . head . tail)

  myTell el1

  return el2



**Шаг 9**

Превратите монаду MyRW в трансформер монад MyRWT:

logFirstAndRetSecond :: MyRWT IO String

logFirstAndRetSecond = do

  el1 <- myAsks head

  myLift $ putStrLn $ "First is " ++ show el1

  el2 <- myAsks (map toUpper . head . tail)

  myLift $ putStrLn $ "Second is " ++ show el2

  myTell el1

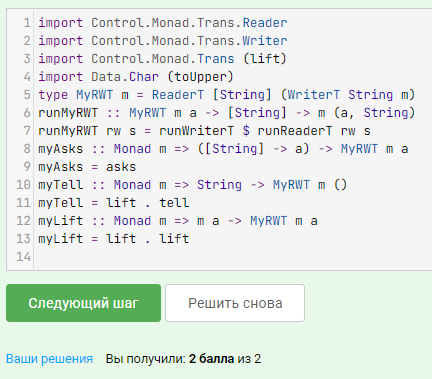
  return el2

GHCi> runMyRWT logFirstAndRetSecond ["abc","defg","hij"]

First is "abc"

Second is "DEFG"

("DEFG","abc")



**Шаг 10**

С помощью трансформера монад MyRWT мы можем написать *безопасную* версию logFirstAndRetSecond:

logFirstAndRetSecond :: MyRWT Maybe String

logFirstAndRetSecond = do

  xs <- myAsk

  case xs of

    (el1 : el2 : \_) -> myTell el1 >> return (map toUpper el2)

    \_ -> myLift Nothing

GHCi> runMyRWT logFirstAndRetSecond ["abc","defg","hij"]

Just ("DEFG","abc")

GHCi> runMyRWT logFirstAndRetSecond ["abc"]

Nothing

Реализуйте *безопасную* функцию veryComplexComputation, записывающую в лог через запятую первую строку четной длины и первую строку нечетной длины, а возвращающую пару из второй строки четной и второй строки нечетной длины, приведенных к верхнему регистру:

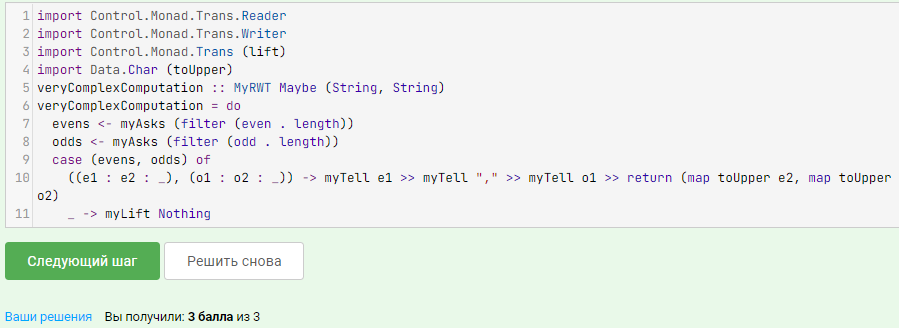
GHCi> runMyRWT veryComplexComputation ["abc","defg","hij"]

Nothing

GHCi> runMyRWT veryComplexComputation ["abc","defg","hij","kl"]

Just (("KL","HIJ"),"defg,abc")

Подсказка: возможно, полезно будет реализовать функцию [myWithReader](https://hackage.haskell.org/package/transformers-0.5.4.0/docs/Control-Monad-Trans-Reader.html" \l "v:withReader" \t "_blank).



**Шаг 11**

Предположим мы хотим исследовать свойства рекуррентных последовательностей. Рекуррентные отношения будем задавать вычислениями типа State Integer Integer, которые, будучи инициализированы текущим значением элемента последовательности, возвращают следующее значение в качестве состояния и текущее в качестве возвращаемого значения, например:

tickCollatz :: State Integer Integer

tickCollatz = do

n <- get

let res = if odd n then 3 \* n + 1 else n `div` 2

put res

return n

Используя монаду State из модуля Control.Monad.Trans.State и трансформер ExceptT из модуля Control.Monad.Trans.Except библиотеки transformers, реализуйте для монады

type EsSi = ExceptT String (State Integer)

функцию runEsSi :: EsSi a -> Integer -> (Either String a, Integer), запускающую вычисление в этой монаде, а также функцию go :: Integer -> Integer -> State Integer Integer -> EsSi (), принимающую шаг рекуррентного вычисления и два целых параметра, задающие нижнюю и верхнюю границы допустимых значений вычислений. Если значение больше или равно верхнему или меньше или равно нижнему, то оно прерывается исключением с соответствующим сообщением об ошибке

GHCi> runEsSi (go 1 85 tickCollatz) 27

(Right (),82)

GHCi> runEsSi (go 1 80 tickCollatz) 27

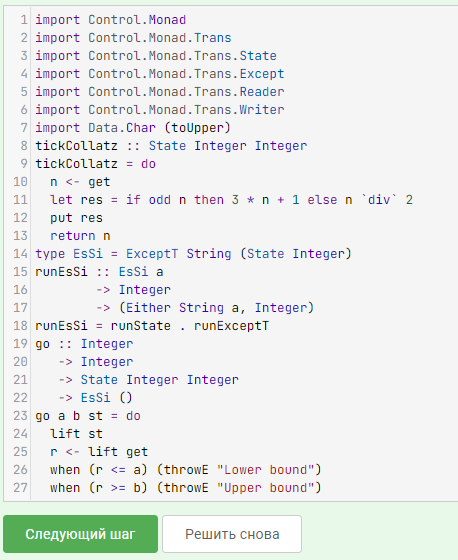
(Left "Upper bound",82)

GHCi> runEsSi (forever $ go 1 1000 tickCollatz) 27

(Left "Upper bound",1186)

GHCi> runEsSi (forever $ go 1 10000 tickCollatz) 27

(Left "Lower bound",1)



**Шаг 12**

Модифицируйте монаду EsSi из предыдущей задачи, обернув ее в трансформер ReaderT с окружением, представляющим собой пару целых чисел, задающих нижнюю и верхнюю границы для вычислений. Функции go теперь не надо будет передавать эти параметры, они будут браться из окружения. Сделайте получившуюся составную монаду трансформером:

type RiiEsSiT m = ReaderT (Integer,Integer) (ExceptT String (StateT Integer m))

Реализуйте также функцию для запуска этого трансформера

runRiiEsSiT :: ReaderT (Integer,Integer) (ExceptT String (StateT Integer m)) a

-> (Integer,Integer)

-> Integer

-> m (Either String a, Integer)

и модифицируйте код функции go, изменив её тип на

go :: Monad m => StateT Integer m Integer -> RiiEsSiT m ()

так, чтобы для шага вычисления последовательности с отладочным выводом текущего элемента последовательности на экран

tickCollatz' :: StateT Integer IO Integer

tickCollatz' = do

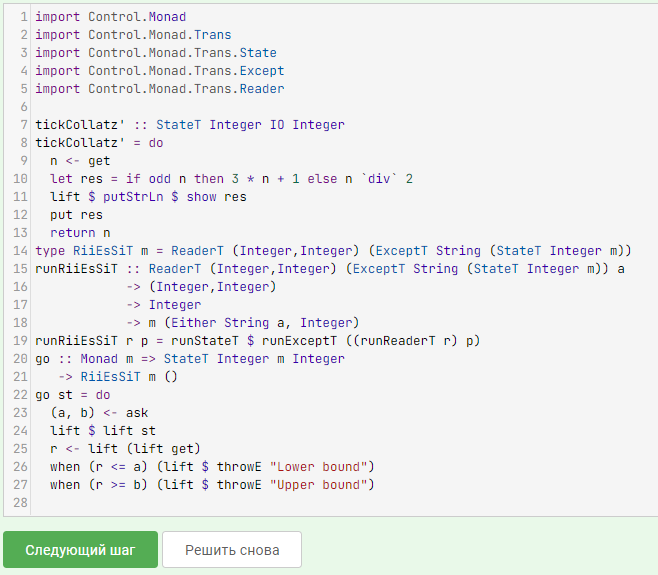
n <- get

let res = if odd n then 3 \* n + 1 else n `div` 2

lift $ putStrLn $ show res

put res

return n



**3.4 Трансформер ReaderT**

**Шаг 3**

В задачах из предыдущих модулей мы сталкивались с типами данных

newtype Arr2 e1 e2 a = Arr2 { getArr2 :: e1 -> e2 -> a }

newtype Arr3 e1 e2 e3 a = Arr3 { getArr3 :: e1 -> e2 -> e3 -> a }

задающих вычисления с двумя и тремя окружениями соответственно. Можно расширить их до трансформеров:

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

Напишите «конструирующие» функции

arr2 :: Monad m => (e1 -> e2 -> a) -> Arr2T e1 e2 m a

arr3 :: Monad m => (e1 -> e2 -> e3 -> a) -> Arr3T e1 e2 e3 m a

обеспечивающие следующее поведение

GHCi> (getArr2T $ arr2 (+)) 33 9 :: [Integer]

[42]

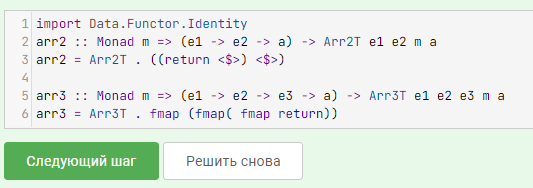
GHCi> (getArr3T $ arr3 foldr) (\*) 1 [1..5] :: Either String Integer

Right 120

GHCi> import Data.Functor.Identity

GHCi> runIdentity $ (getArr2T $ arr2 (+)) 33 9

42



**Шаг 6**

Сделайте трансформеры

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

представителями класса типов Functor в предположении, что m является функтором:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2,e1+e2]

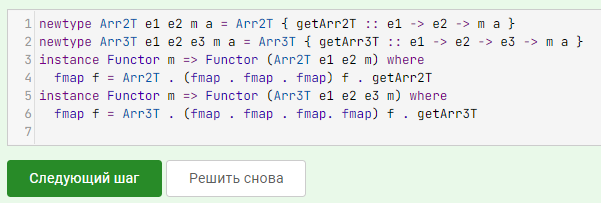
GHCi> (getArr2T $ succ <$> a2l) 10 100

[11,101,111]

GHCi> a3e = Arr3T $ \e1 e2 e3 -> Right (e1+e2+e3)

GHCi> (getArr3T $ sqrt <$> a3e) 2 3 4

Right 3.0



**Шаг 10**

Сделайте трансформеры

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

представителями класса типов Applicative в предположении, что m является аппликативным функтором:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2]

GHCi> a2fl = Arr2T $ \e1 e2 -> [(e1\*e2+),const 7]

GHCi> getArr2T (a2fl <\*> a2l) 2 10

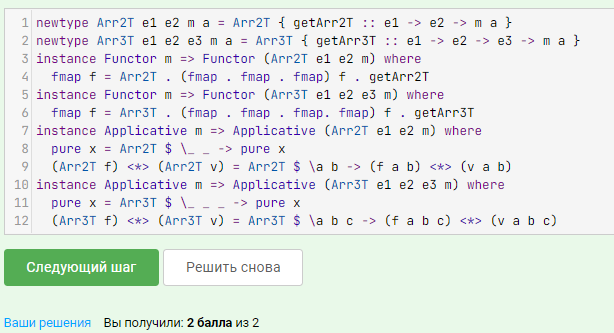
[22,30,7,7]

GHCi> a3fl = Arr3T $ \e1 e2 e3 -> [(e2+),(e3+)]

GHCi> a3l = Arr3T $ \e1 e2 e3 -> [e1,e2]

GHCi> getArr3T (a3fl <\*> a3l) 3 5 7

[8,10,10,12]



**Шаг 12**

Сделайте трансформеры

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

представителями класса типов Monad в предположении, что m является монадой:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2]

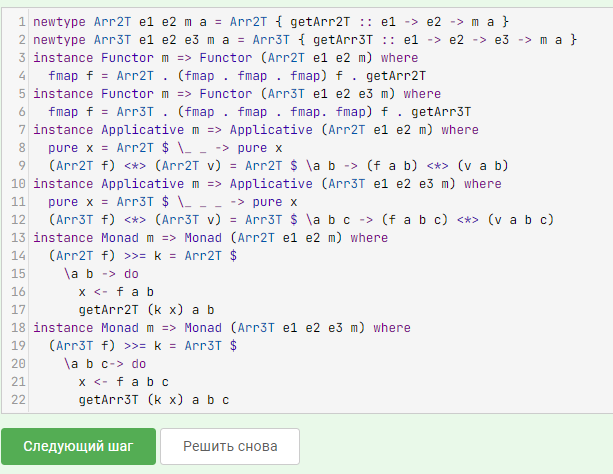
GHCi> getArr2T (do {x <- a2l; y <- a2l; return (x + y)}) 3 5

[6,8,8,10]

GHCi> a3m = Arr3T $ \e1 e2 e3 -> Just (e1 + e2 + e3)

GHCi> getArr3T (do {x <- a3m; y <- a3m; return (x \* y)}) 2 3 4

Just 81



**Шаг 13**

Разработанная нами реализация интерфейса монады для трансформера Arr3T (как и для Arr2T и ReaderT) имеет не очень хорошую особенность. При неудачном сопоставлении с образцом вычисления в этой монаде завершаются аварийно, с выводом сообщения об ошибке в диагностический поток:

GHCi> a3m = Arr3T $ \e1 e2 e3 -> Just (e1 + e2 + e3)

GHCi> getArr3T (do {9 <- a3m; y <- a3m; return y}) 2 3 4

Just 9

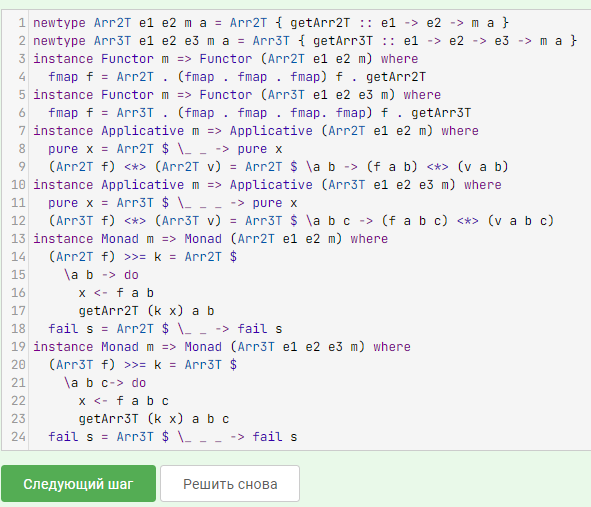
GHCi> getArr3T (do {10 <- a3m; y <- a3m; return y}) 2 3 4

\*\*\* Exception: Pattern match failure in do expression at :12:15-16

Для обычного ридера такое поведение нормально, однако у трансформера внутренняя монада может уметь обрабатывать ошибки более щадащим образом. Переопределите функцию fail класса типов Monad для Arr3T так, чтобы обработка неудачного сопоставления с образцом осуществлялась бы во внутренней монаде:

GHCi> getArr3T (do {10 <- a3m; y <- a3m; return y}) 2 3 4

Nothing



**Шаг 15**

Сделайте трансформер

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

представителями класса типов MonadTrans:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2]

GHCi> getArr2T (do {x <- a2l; y <- lift [10,20,30]; return (x+y)}) 3 4

[13,23,33,14,24,34]

Реализуйте также «стандартный интерфейс» для этой монады — функцию

asks2 :: Monad m => (e1 -> e2 -> a) -> Arr2T e1 e2 m a

работающую как asks для ReaderT, но принимающую при этом функцию от обоих наличных окружений:

GHCi> getArr2T (do {x <- asks2 const; y <- asks2 (flip const); z <- asks2 (,); return (x,y,z)}) 'A' 'B'

('A','B',('A','B'))



**Модуль 4**

***4.1 Трансформер WriterT***

**Шаг 5**

Предположим, что мы дополнительно реализовали строгую версию аппликативного функтора Writer:

newtype StrictWriter w a = StrictWriter { runStrictWriter :: (a, w) }

instance Functor (StrictWriter w) where

fmap f = StrictWriter . updater . runStrictWriter

where updater (x, log) = (f x, log)

instance Monoid w => Applicative (StrictWriter w) where

  pure x = StrictWriter (x, mempty)

  f <\*> v = StrictWriter $ updater (runStrictWriter f) (runStrictWriter v)

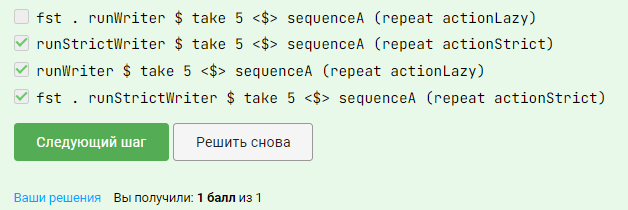
    where updater (g, w) (x, w') = (g x, w `mappend` w')

и определили такие вычисления:

actionLazy = Writer (42,"Hello!")

actionStrict = StrictWriter (42,"Hello!")

Какие из следующих вызовов приведут к расходимостям?



**Шаг 7**

Сделайте на основе типа данных

data Logged a = Logged String a deriving (Eq,Show)

трансформер монад LoggT :: (\* -> \*) -> \* -> \* с одноименным конструктором данных и меткой поля runLoggT:

newtype LoggT m a = LoggT { runLoggT :: m (Logged a) }

Для этого реализуйте для произвольной монады m представителя класса типов Monad для LoggT m :: \* -> \*:

instance Monad m => Monad (LoggT m) where

return x = undefined

m >>= k = undefined

fail msg = undefined

Для проверки используйте функции:

logTst :: LoggT Identity Integer

logTst = do

x <- LoggT $ Identity $ Logged "AAA" 30

y <- return 10

z <- LoggT $ Identity $ Logged "BBB" 2

return $ x + y + z

failTst :: [Integer] -> LoggT [] Integer

failTst xs = do

5 <- LoggT $ fmap (Logged "") xs

LoggT [Logged "A" ()]

return 42

которые при правильной реализации монады должны вести себя так:

GHCi> runIdentity (runLoggT logTst)

Logged "AAABBB" 42

GHCi> runLoggT $ failTst [5,5]

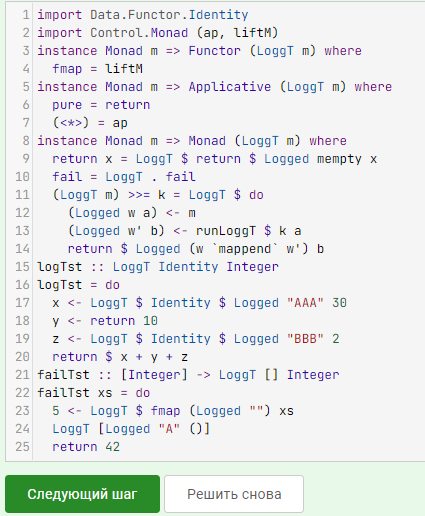
[Logged "A" 42,Logged "A" 42]

GHCi> runLoggT $ failTst [5,6]

[Logged "A" 42]

GHCi> runLoggT $ failTst [7,6]

[]



**Шаг 10**

Напишите функцию write2log обеспечивающую трансформер LoggT стандартным логгирующим интерфейсом:

write2log :: Monad m => String -> LoggT m ()

write2log = undefined

Эта функция позволяет пользователю осуществлять запись в лог в процессе вычисления в монаде LoggT m для любой монады m. Введите для удобства упаковку для LoggT Identity и напишите функцию запускающую вычисления в этой монаде

type Logg = LoggT Identity

runLogg :: Logg a -> Logged a

runLogg = undefined

Тест

logTst' :: Logg Integer

logTst' = do

write2log "AAA"

write2log "BBB"

return 42

должен дать такой результат:

GHCi> runLogg logTst'

Logged "AAABBB" 42

А тест (подразумевающий импорт Control.Monad.Trans.State и Control.Monad.Trans.Class)

stLog :: StateT Integer Logg Integer

stLog = do

modify (+1)

a <- get

lift $ write2log $ show $ a \* 10

put 42

return $ a \* 100

— такой:

GHCi> runLogg $ runStateT stLog 2

Logged "30" (300,42)



**Шаг 12**

В последнем примере предыдущей задачи функция lift :: (MonadTrans t, Monad m) => m a -> t m a позволяла поднять вычисление из внутренней монады (в примере это был Logg) во внешний трансформер (StateT Integer). Это возможно, поскольку для трансформера StateT s реализован представитель класса типов MonadTrans из Control.Monad.Trans.Class.

Сделайте трансформер LoggT представителем этого класса MonadTrans, так чтобы можно было поднимать вычисления из произвольной внутренней монады в наш трансформер:

instance MonadTrans LoggT where

lift = undefined

logSt :: LoggT (State Integer) Integer

logSt = do

lift $ modify (+1)

a <- lift get

write2log $ show $ a \* 10

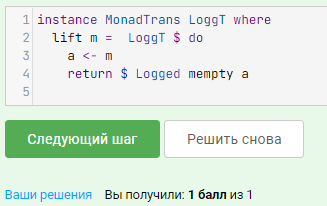
lift $ put 42

return $ a \* 100

Проверка:

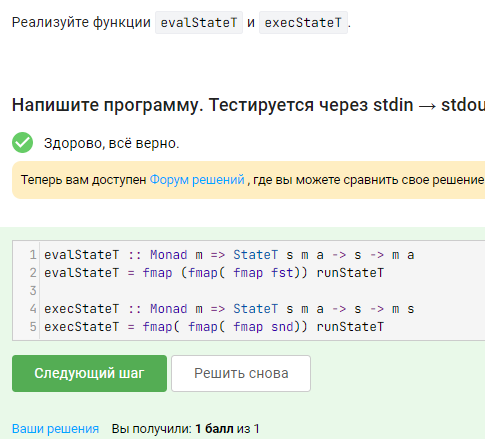
GHCi> runState (runLoggT logSt) 2

(Logged "30" 300,42)



**4.2 Трансформер StateT**

**Шаг 4**



**Шаг 5**

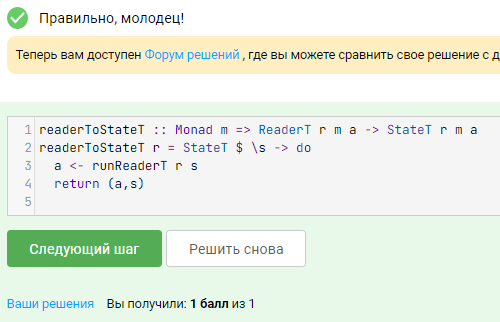
Нетрудно понять, что монада State более «сильна», чем монада Reader: вторая тоже, в некотором смысле, предоставляет доступ к глобальному состоянию, но только, в отличие от первой, не позволяет его менять. Покажите, как с помощью StateT можно эмулировать ReaderT:

GHCi> evalStateT (readerToStateT $ asks (+2)) 4

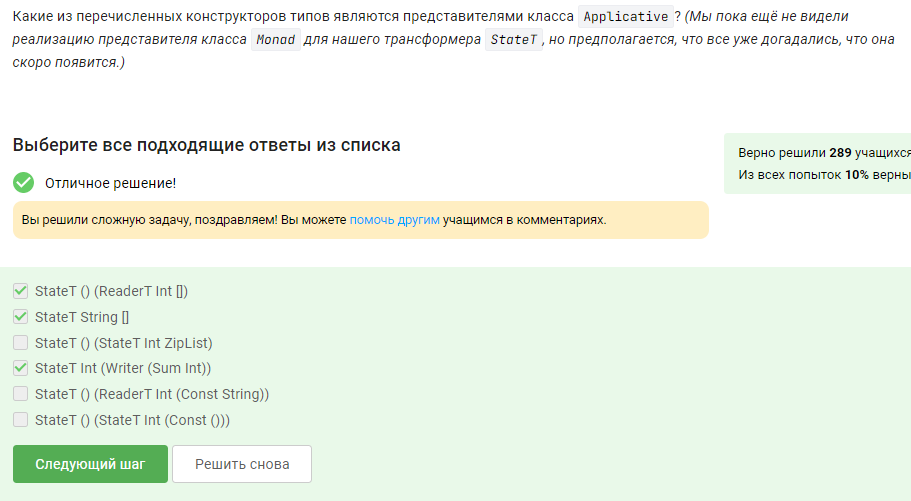
6

GHCi> runStateT (readerToStateT $ asks (+2)) 4

(6,4)



**Шаг 9**



**Шаг 11**

Неудачное сопоставление с образцом для реализованного на предыдущих видео-степах трансформера StateT аварийно прерывает вычисление:

GHCi> sl2 = StateT $ \st -> [(st,st),(st+1,st-1)]

GHCi> runStateT (do {6 <- sl2; return ()}) 5

\*\*\* Exception: Pattern match failure in do expression ...

Исправьте реализацию таким образом, чтобы обработка такой ситуации переадресовывалась бы внутренней монаде:

GHCi> sl2 = StateT $ \st -> [(st,st),(st+1,st-1)]

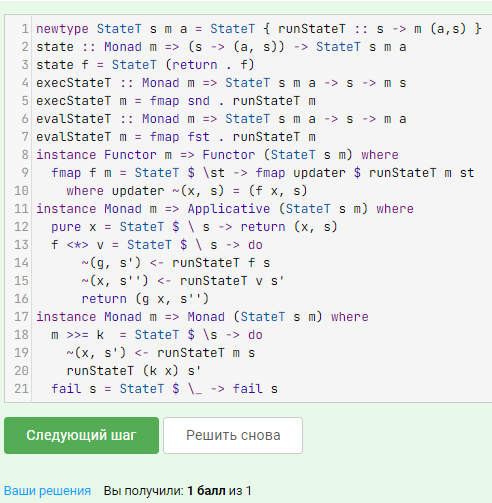
GHCi> runStateT (do {6 <- sl2; return ()}) 5

[((),4)]

GHCi> sm = StateT $ \st -> Just (st+1,st-1)

GHCi> runStateT (do {42 <- sm; return ()}) 5

Nothing



**Шаг 14**

Те из вас, кто проходил первую часть нашего курса, конечно же помнят, последнюю задачу из него. В тот раз всё закончилось монадой State, но сейчас с неё все только начинается!

data Tree a = Leaf a | Fork (Tree a) a (Tree a)

Вам дано значение типа Tree (), иными словами, вам задана *форма* дерева. От вас требуется сделать две вещи: во-первых, пронумеровать вершины дерева, обойдя их *in-order* обходом (левое поддерево, вершина, правое поддерево); во-вторых, подсчитать количество листьев в дереве.

GHCi> numberAndCount (Leaf ())

(Leaf 1,1)

GHCi> numberAndCount (Fork (Leaf ()) () (Leaf ()))

(Fork (Leaf 1) 2 (Leaf 3),2)

Конечно, можно решить две подзадачи по-отдельности, но мы сделаем это всё за один проход. Если бы вы писали решение на императивном языке, вы бы обошли дерево, поддерживая в одной переменной следующий доступный номер для очередной вершины, а в другой — количество встреченных листьев, причем само значение второй переменной, по сути, в процессе обхода не требуется. Значит, вполне естественным решением будет завести состояние для первой переменной, а количество листьев накапливать в «логе»-моноиде.

Вот так выглядит код, запускающий наше вычисление и извлекающий результат:

numberAndCount :: Tree () -> (Tree Integer, Integer)

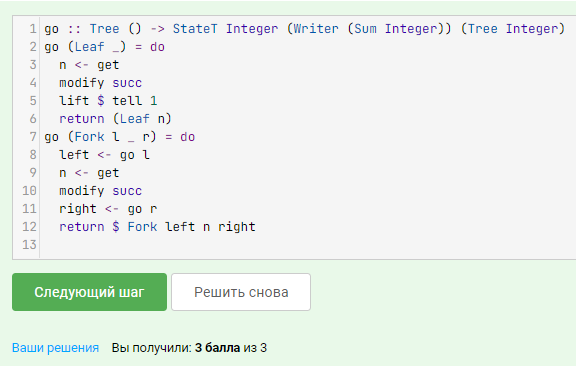
numberAndCount t = getSum <$> runWriter (evalStateT (go t) 1)

  where

    go :: Tree () -> StateT Integer (Writer (Sum Integer)) (Tree Integer)

    go = undefined

Вам осталось только описать само вычисление — функцию go.



**4.3 Трансформер ReaderT**

**Шаг 6**

Представьте, что друг принес вам игру. В этой игре герой ходит по полю. За один ход он может переместиться на одну клетку вверх, вниз, влево и вправо (стоять на месте нельзя). На поле его поджидают различные опасности, такие как пропасти (chasm) и ядовитые змеи (snake). Если игрок наступает на клетку с пропастью или со змеёй, он умирает.

data Tile = Floor | Chasm | Snake

  deriving Show

data DeathReason = Fallen | Poisoned

  deriving (Eq, Show)

Карта задается функцией, отображающей координаты клетки в тип этой самой клетки:

type Point = (Integer, Integer)

type GameMap = Point -> Tile

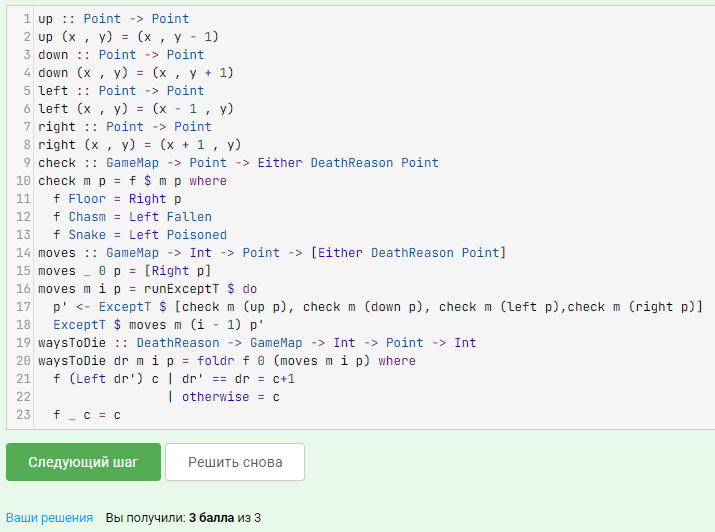
Ваша задача состоит в том, чтобы реализовать функцию

moves :: GameMap -> Int -> Point -> [Either DeathReason Point]

принимающую карту, количество шагов и начальную точку, а возвращающую список всех возможных исходов (с повторениями), если игрок сделает заданное число шагов из заданной точки. Заодно реализуйте функцию

waysToDie :: DeathReason -> GameMap -> Int -> Point -> Int

показывающую, сколькими способами игрок может умереть данным способом, сделав заданное число шагов из заданной точки.



**Шаг 8**

Следующий код

import Control.Monad.Trans.Maybe

import Data.Char (isNumber, isPunctuation)

askPassword0 :: MaybeT IO ()

askPassword0 = do

liftIO $ putStrLn "Enter your new password:"

value <- msum $ repeat getValidPassword0

liftIO $ putStrLn "Storing in database..."

getValidPassword0 :: MaybeT IO String

getValidPassword0 = do

s <- liftIO getLine

guard (isValid0 s)

return s

isValid0 :: String -> Bool

isValid0 s = length s >= 8

&& any isNumber s

&& any isPunctuation s

используя трансформер MaybeT и свойства функции msum, отвергает ввод пользовательского пароля, до тех пор пока он не станет удовлетворять заданным критериям. Это можно проверить, вызывая его в интерпретаторе

GHCi> runMaybeT askPassword0

Используя пользовательский тип ошибки и трансформер ExceptT вместо MaybeT, модифицируйте приведенный выше код так, чтобы он выдавал пользователю сообщение о причине, по которой пароль отвергнут.

data PwdError = PwdError String

type PwdErrorIOMonad = ExceptT PwdError IO

askPassword :: PwdErrorIOMonad ()

askPassword = do

liftIO $ putStrLn "Enter your new password:"

value <- msum $ repeat getValidPassword

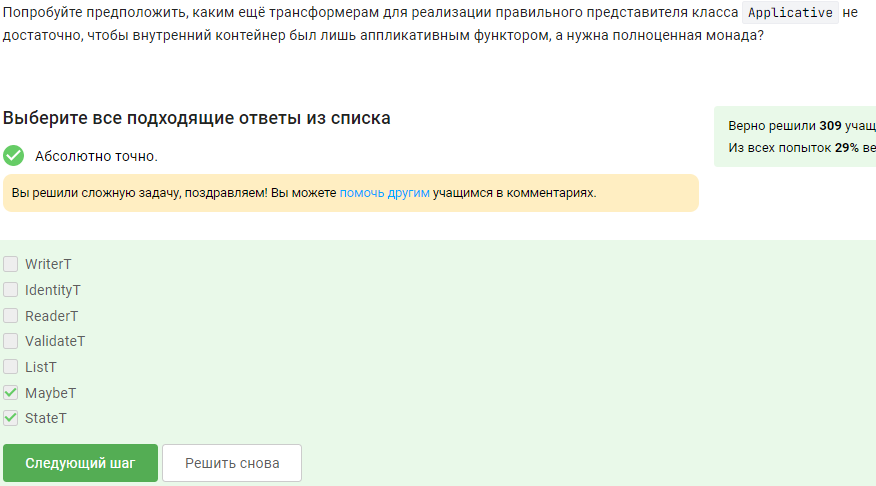
liftIO $ putStrLn "Storing in database..."

getValidPassword :: PwdErrorIOMonad String

getValidPassword = undefined



**Шаг 11**



**Шаг 12**

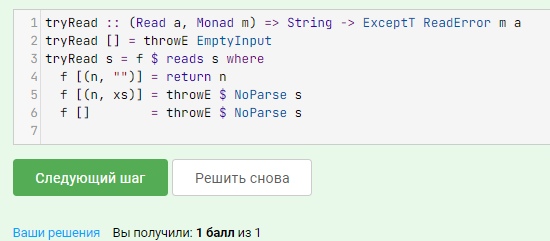
Вспомним функцию tryRead:

data ReadError = EmptyInput | NoParse String

  deriving Show

tryRead :: Read a => String -> Except ReadError a

Измените её так, чтобы она работала в трансформере ExceptT.



**Шаг 13**

С деревом мы недавно встречались:

data Tree a = Leaf a | Fork (Tree a) a (Tree a)

Вам на вход дано дерево, содержащее целые числа, записанные в виде строк. Ваша задача обойти дерево *in-order* (левое поддерево, вершина, правое поддерево) и просуммировать числа до первой строки, которую не удаётся разобрать функцией tryRead из прошлого задания (или до конца дерева, если ошибок нет). Если ошибка произошла, её тоже надо вернуть.

Обходить деревья мы уже умеем, так что от вас требуется только функция go, подходящая для такого вызова:

treeSum t = let (err, s) = runWriter . runExceptT $ traverse\_ go t

            in (maybeErr err, getSum s)

  where

    maybeErr :: Either ReadError () -> Maybe ReadError

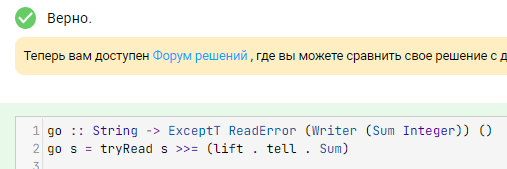
    maybeErr = either Just (const Nothing)

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "oops")) "15" (Leaf "16")

(Just (NoParse "oops"),3)

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "0")) "15" (Leaf "16")

(Nothing,34)



**4.4 Неявный лифтинг**

**Шаг 5**

Предположим мы хотим реализовать следующую облегченную версию функтора, используя многопараметрические классы типов:

class Functor' c e where

fmap' :: (e -> e) -> c -> c

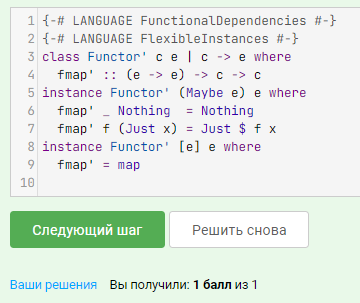
Добавьте в определение этого класса типов необходимые функциональные зависимости и реализуйте его представителей для списка и Maybe так, чтобы обеспечить работоспособность следующих вызовов

GHCi> fmap' succ "ABC"

"BCD"

GHCi> fmap' (^2) (Just 42)

Just 1764



**Шаг 10**

В этой и следующих задачах мы продолжаем работу с трансформером LoggT [разработанным на первом уроке этой недели](https://stepik.org/lesson/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D1%84%D0%BE%D1%80%D0%BC%D0%B5%D1%80-WriterT-38578/step/12):

data Logged a = Logged String a deriving (Eq,Show)

newtype LoggT m a = LoggT { runLoggT :: m (Logged a) }

write2log :: Monad m => String -> LoggT m ()

type Logg = LoggT Identity

runLogg :: Logg a -> Logged a

Теперь мы хотим сделать этот трансформер mtl-совместимым.

Избавьтесь от необходимости ручного подъема операций вложенной монады State, сделав трансформер LoggT, примененный к монаде с интерфейсом MonadState, представителем этого (MonadState) класса типов:

instance MonadState s m => MonadState s (LoggT m) where

get = undefined

put = undefined

state = undefined

logSt' :: LoggT (State Integer) Integer

logSt' = do

modify (+1) -- no lift!

a <- get -- no lift!

write2log $ show $ a \* 10

put 42 -- no lift!

return $ a \* 100



**Шаг 11**

Избавьтесь от необходимости ручного подъема операций вложенной монады Reader, сделав трансформер LoggT, примененный к монаде с интерфейсом MonadReader, представителем этого (MonadReader) класса типов:

instance MonadReader r m => MonadReader r (LoggT m) where

ask = undefined

local = undefined

reader = undefined

Для упрощения реализации функции local имеет смысл использовать вспомогательную функцию, поднимающую стрелку между двумя «внутренними представлениями» трансформера LoggT в стрелку между двумя LoggT:

mapLoggT :: (m (Logged a) -> n (Logged b)) -> LoggT m a -> LoggT n b

mapLoggT f = undefined

Тест:

logRdr :: LoggT (Reader [(Int,String)]) ()

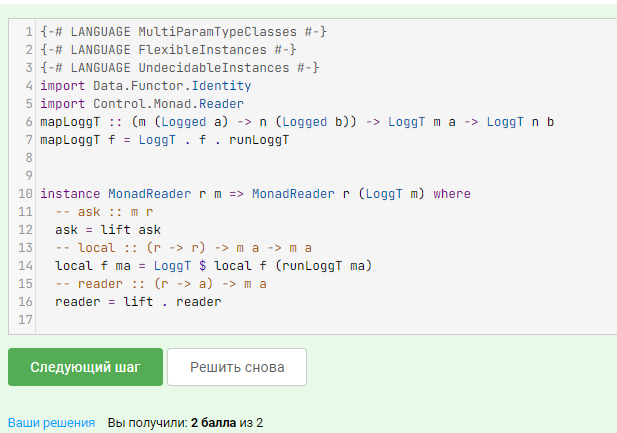
logRdr = do

Just x <- asks $ lookup 2 -- no lift!

write2log x

Just y <- local ((3,"Jim"):) $ asks $ lookup 3 -- no lift!

write2log y



**Шаг 12**

Чтобы избавится от необходимости ручного подъема операции write2log, обеспечивающей стандартный интерфейс вложенного трансформера LoggT, можно поступить по аналогии с другими трансформерами библиотеки mtl. А именно, разработать класс типов MonadLogg, выставляющий этот стандартный интерфейс

class Monad m => MonadLogg m where

w2log :: String -> m ()

logg :: Logged a -> m a

(Замечание: Мы переименовываем функцию write2log в w2log, поскольку хотим держать всю реализацию в одном файле исходного кода. При следовании принятой в библиотеках transformers/mtl идеологии они имели бы одно и то же имя, но были бы определены в разных модулях. При работе с transformers мы импортировали бы свободную функцию c квалифицированным именем Control.Monad.Trans.Logg.write2log, а при использовании mtl работали бы с методом класса типов MonadLogg с полным именем Control.Monad.Logg.write2log. )

Этот интерфейс, во-первых, должен выставлять сам трансформер LoggT, обернутый вокруг произвольной монады:

instance Monad m => MonadLogg (LoggT m) where

w2log = undefined

logg = undefined

Реализуйте этого представителя, для проверки используйте:

logSt'' :: LoggT (State Integer) Integer

logSt'' = do

x <- logg $ Logged "BEGIN " 1

modify (+x)

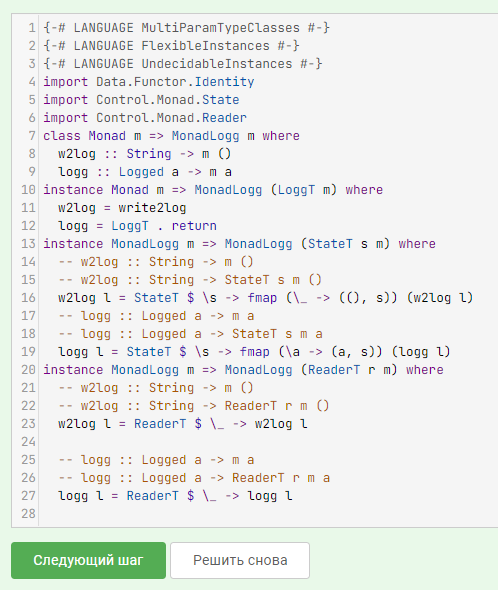
a <- get

w2log $ show $ a \* 10

put 42

w2log " END"

return $ a \* 100



**4.5 Задачи на трансформеры**

**Шаг 2**

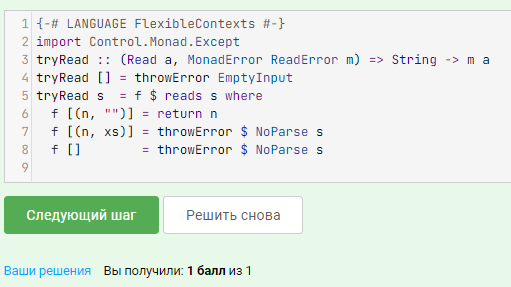
Функция tryRead обладает единственным эффектом: в случае ошибки она должна прерывать вычисление. Это значит, что её можно использовать в любой монаде, предоставляющей возможность завершать вычисление с ошибкой, но сейчас это не так, поскольку её тип это делать не позволяет:

data ReadError = EmptyInput | NoParse String

  deriving Show

tryRead :: (Read a, Monad m) => String -> ExceptT ReadError m a

Измените её так, чтобы она работала в любой монаде, позволяющей сообщать об исключительных ситуациях типа ReadError. Для этого к трансформеру ExceptT в библиотеке mtl прилагается класс типов MonadError (обратите внимание на название класса — это так сделали специально, чтобы всех запутать), находящийся в модуле Control.Monad.Except.



**Шаг 3**

В очередной раз у вас есть дерево строковых представлений чисел:

data Tree a = Leaf a | Fork (Tree a) a (Tree a)

и функция tryRead:

data ReadError = EmptyInput | NoParse String

  deriving Show

tryRead :: (Read a, MonadError ReadError m) => String -> m a

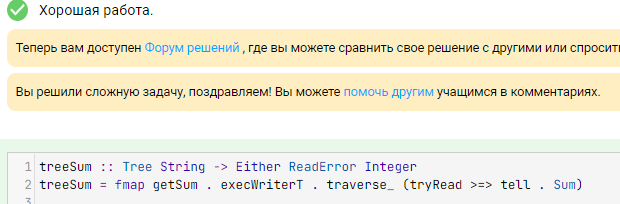
Просуммируйте числа в дереве, а если хотя бы одно прочитать не удалось, верните ошибку:

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "oops")) "15" (Leaf "16")

Left (NoParse "oops")

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "0")) "15" (Leaf "16")

Right 34



**Шаг 4**

Вам дан список вычислений с состоянием (State s a) и начальное состояние. Требуется выполнить все эти вычисления по очереди (очередное вычисление получает на вход состояние, оставшееся от предыдущего) и вернуть список результатов. Но это ещё не всё. Ещё вам дан предикат, определяющий, разрешено некоторое состояние или нет; после выполнения очередного вычисления вы должны с помощью этого предиката проверить текущее состояние, и, если оно не разрешено, завершить вычисление, указав номер вычисления, которое его испортило.

При этом, завершаясь с ошибкой, мы можем как сохранить накопленное до текущего момента состояние, так и выкинуть его. В первом случае наша функция будет иметь такой тип:

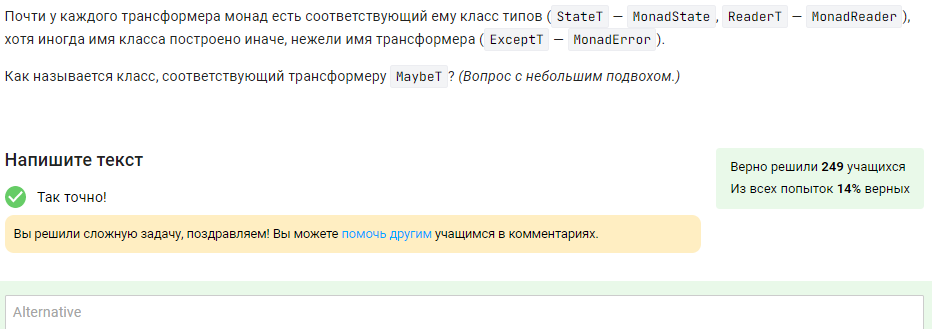
runLimited1 :: (s -> Bool) -> [State s a] -> s -> (Either Int [a], s)

Во втором — такой:

runLimited2 :: (s -> Bool) -> [State s a] -> s -> Either Int ([a], s)



**Шаг 5**



**Шаг 6**

Чтобы закончить наш курс ярко, предлагаем вам с помощью этой задачи в полной мере почувствовать на себе всю мощь continuation-passing style. Чтобы успешно решить эту задачу, вам нужно хорошо понимать, как работает CPS и монада ContT (а этого, как известно, *никто* не понимает). *Кстати, это была подсказка.*

[*Сопрограмма*](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) (корутина, [coroutine](https://en.wikipedia.org/wiki/Coroutine" \t "_blank)) это обобщение понятия *подпрограммы* (по-простому говоря, функции). У функции, в отличие от сопрограммы, есть одна точка входа (то, откуда она начинает работать), а точек выхода может быть несколько, но выйти через них функция может только один раз за время работы; у сопрограммы же точек входа и выхода может быть несколько. Проще всего объяснить на примере:

coroutine1 = do

  tell "1"

  yield

  tell "2"

coroutine2 = do

  tell "a"

  yield

  tell "b"

GHCi> execWriter (runCoroutines coroutine1 coroutine2)

"1a2b"

Здесь используется специальное действие yield, которое передает управление другой сопрограмме. Когда другая сопрограмма возвращает управление (с помощью того же yield или завершившись), первая сопрограмма продолжает работу с того места, на котором остановилась в прошлый раз.

В общем случае, одновременно могут исполняться несколько сопрограмм, причем при передаче управления, они могут обмениваться значениями. В этой задаче достаточно реализовать сопрограммы в упрощенном виде: одновременно работают ровно две сопрограммы и значениями обмениваться они не могут.

Реализуйте трансформер CoroutineT, функцию yield для передачи управления и функцию runCoroutines для запуска. Учтите, что одна сопрограмма может завершиться раньше другой; другая должна при этом продолжить работу:

coroutine3, coroutine4 :: CoroutineT (Writer String) ()

coroutine3 = do

  tell "1"

  yield

yield

  tell "2"

coroutine4 = do

  tell "a"

  yield

  tell "b"

  yield

  tell "c"

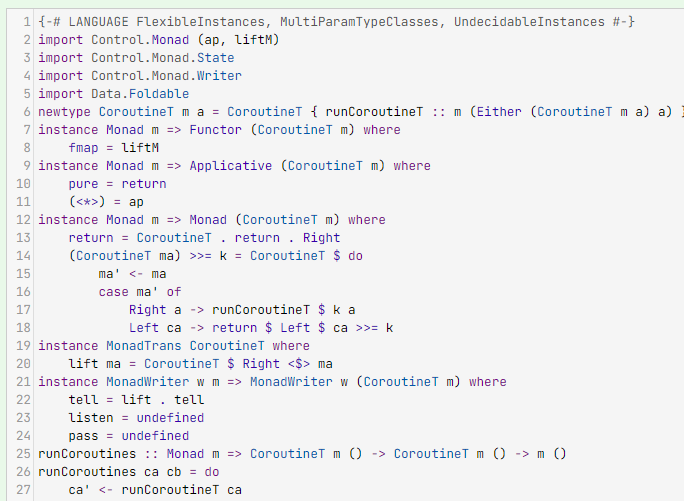
  yield

  tell "d"

  yield

GHCi> execWriter (runCoroutines coroutine3 coroutine4)

"1ab2cd"



**Сертификат:**

