Министерство образования Республики Беларусь

«ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. ЕВФРОСИНИ

ПОЛОЦКОЙ»

Факультет информационных технологий

Кафедра технологий программирования

Отчет по курсу «ФП(Haskell)\_часть2\_22ИТ-зд» на stepik.org

Дисциплина «Функциональное программирование» (практическая часть курса)

Выполнил: студент группы 22-ИТзд,

Шастовская М.С.

Проверил: старший преподаватель

Струк Т.C.

Полоцк, 2025 г.

**Модуль 1: Введение**

**1.4 Аппликативный парсер своими руками**

**1.5 Композиция на уровне типов**

**2 Управление эффектами**

**2.1 Класс типов Foldable**

**2.2 Класс типов Traversable**

**2.3 Законы и свойства класса Traversable**

**2.4 Связь классов Monad и Applicative**

**2.5 Классы типов Alternative и MonadPlus**

**3 Монады и эффекты**

**3.1 Монада Except**

**3.2 Монада Cont**

**3.3 Трансформеры монад**

**3.4 Трансформер ReaderT**

**4 Трансформеры монад**

**4.1 Трансформер WriterT**

**4.2 Трансформер StateT**

**4.3 Трансформер ExceptT**

**4.4 Неявный лифтинг**

**4.5 Задачи на трансформер**

**1 Аппликативные функторы**

**1.1 Определение аппликативного функтора**

**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 }

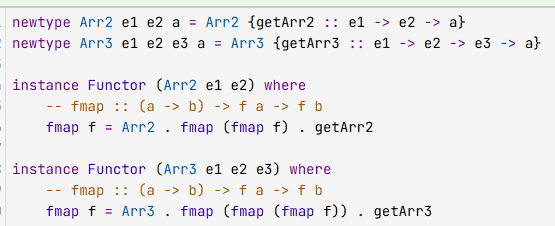
Эти типы инкапсулируют вычисление с двумя и тремя независимыми окружениями соответственно:

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

3

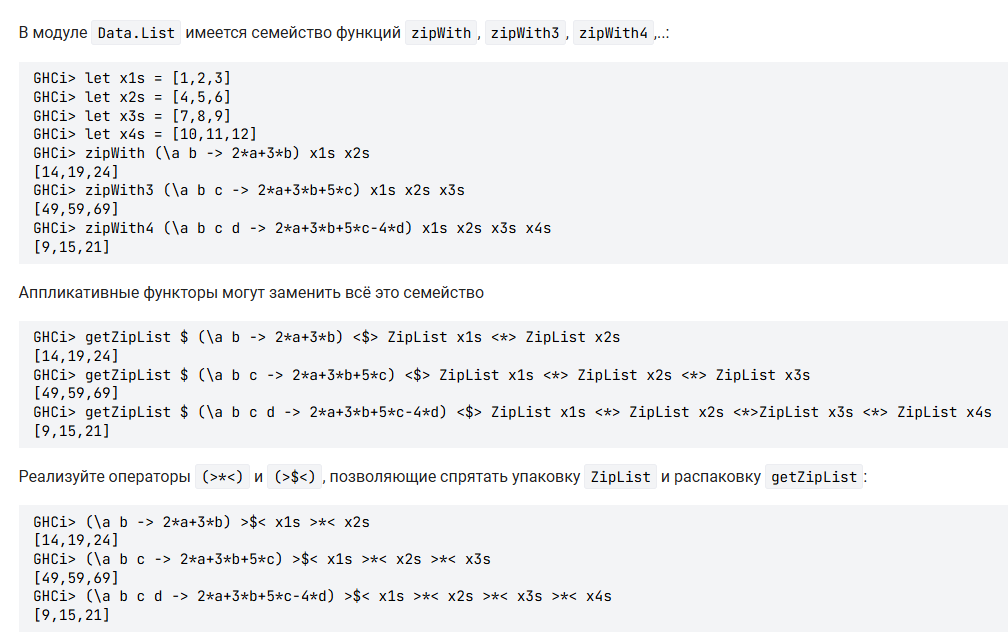
GHCi> getArr3 (tail <$> tail <$> Arr3 zipWith) (+) [1,2,3,4] [10,20,30,40,50]

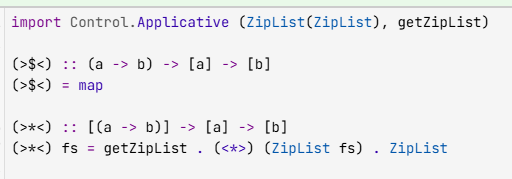
[33,44]



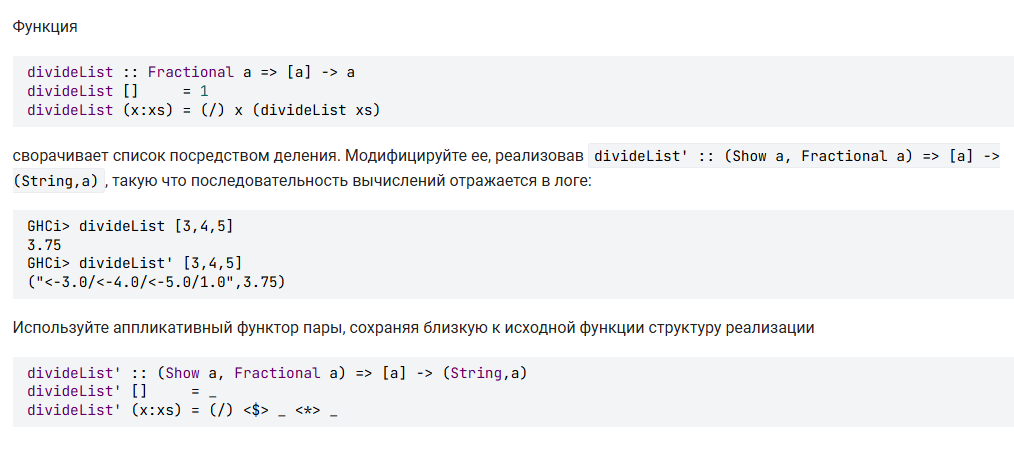
**1.2 Представители класса типов Applicative**

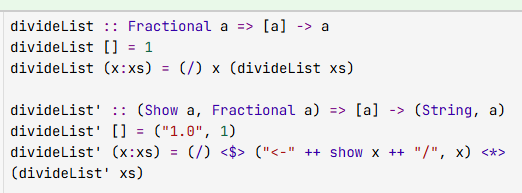
**Шаг 5**





**Шаг 8**





**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 }

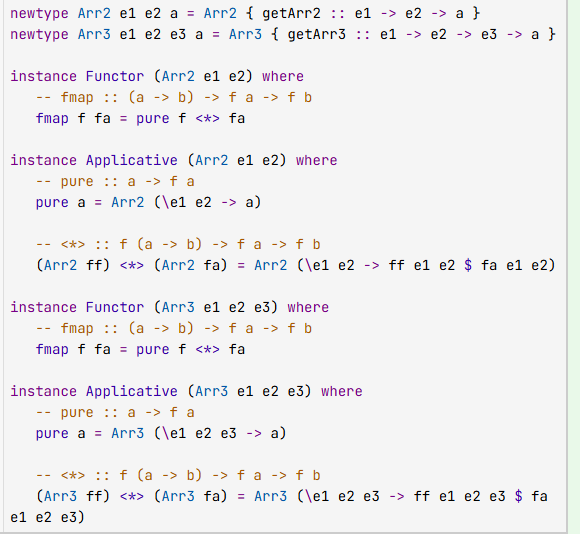
с естественной семантикой двух и трех окружений:

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

-1

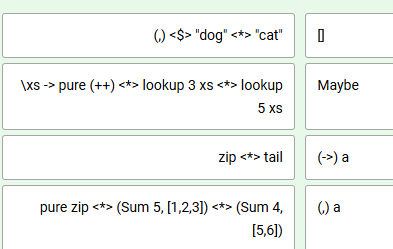
GHCi> getArr3 (Arr3 (\x y z w -> x+y+z-w) <\*> Arr3 (\x y z -> x\*y\*z)) 2 3 4

-15



**Шаг 11**

Сопоставьте вычислению, поднятому в аппликативный функтор, конкретного представителя класса типов Applicative, в котором это вычисление происходит.



**1.3 Аппликативный парсер Parsec**

**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

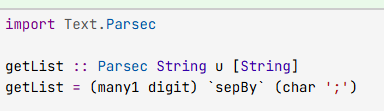
GHCi> parseTest getList "1;;234;56"

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.



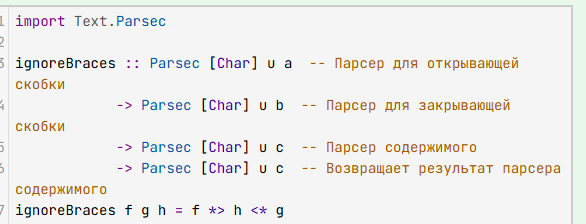
**Шаг 7**

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"

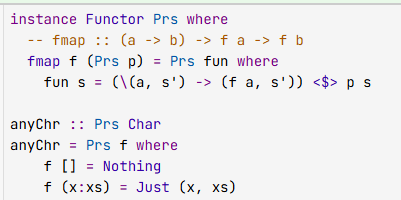
Just ('A',"BC")

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:

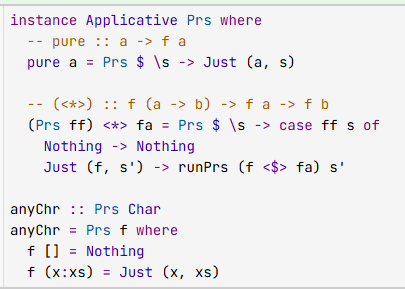
GHCi> runPrs ((,,) <$> anyChr <\*> anyChr <\*> anyChr) "ABCDE"

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"

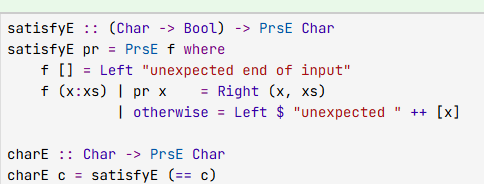
Right ('A',"BC")

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")

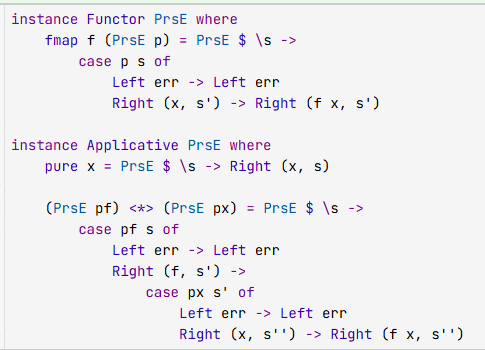
GHCi> runPrsE ((,) <$> anyE <\* charE 'C' <\*> anyE) "ABCDE"

Left "unexpected B"

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

Left "unexpected end of input"

42



**Шаг 12**

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")

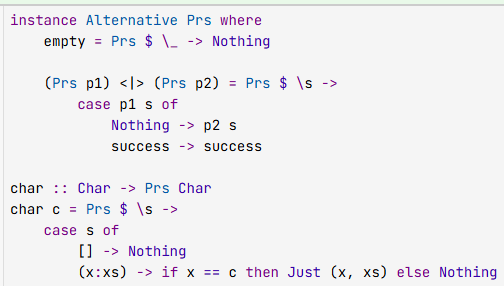
GHCi> runPrs (char 'A' <|> char 'B') "BCD"

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 13**

Реализуйте для парсера

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

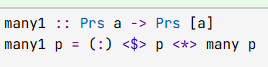
парсер-комбинатор many1 :: Prs a -> Prs [a], который отличается от many только тем, что он терпит неудачу в случае, когда парсер-аргумент неудачен на начале входной строки.

> 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. 

**1.5 Композиция на уровне типов**

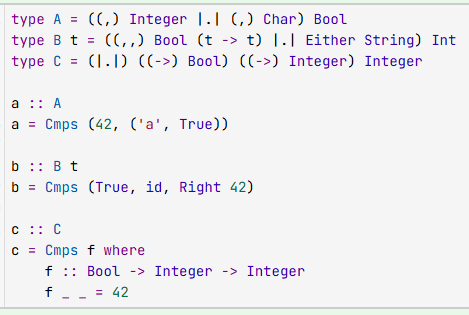
**Шаг 3**

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 7**

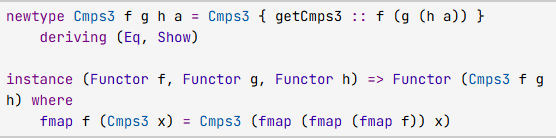
Сделайте тип

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 11**

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)

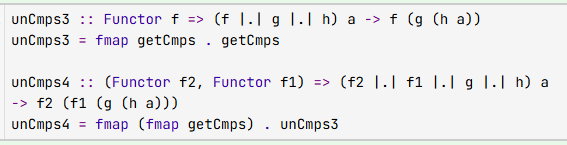
[[[42]]]

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

[Just [42]]

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

[[[[42]]]]



**2 Управление эффектами**

**2.1 Класс типов Foldable**

**Шаг 4**

Сделайте тип

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

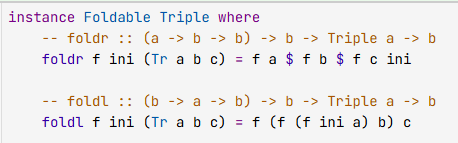
представителем класса типов Foldable:

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

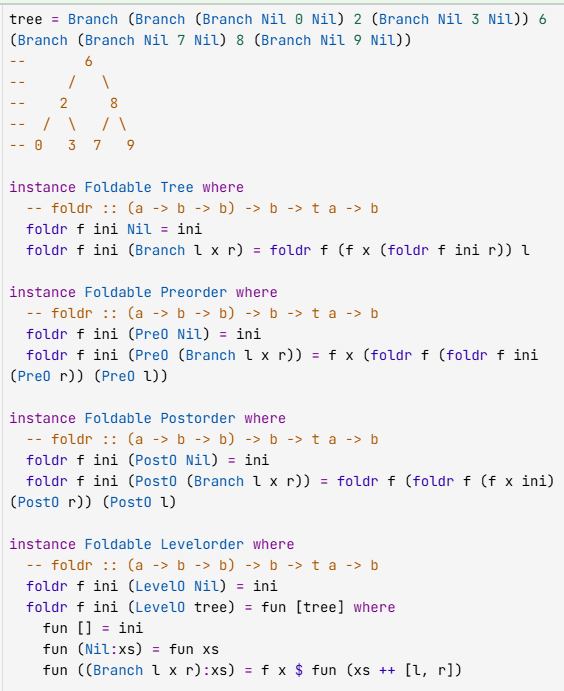
[3,1,2,4]

GHCi> foldr (:) [] $ PostO tree

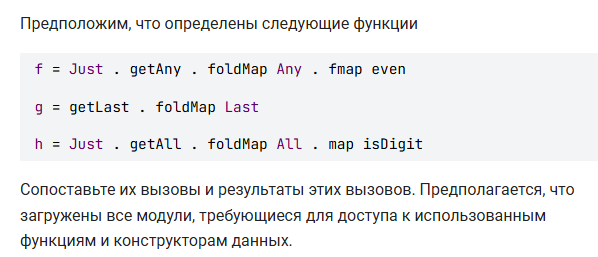
[2,1,4,3]

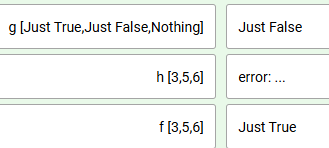
GHCi> foldr (:) [] $ LevelO tree

[3,1,4,2]



**Шаг 7**





**Шаг 9**

Реализуйте функцию on3, имеющую семантику, схожую с on, но принимающую в качестве первого аргумента трехместную функцию:

on3 :: (b -> b -> b -> c) -> (a -> b) -> a -> a -> a -> c

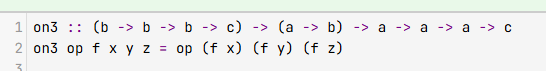
on3 op f x y z = undefined

Например, сумма квадратов трех чисел может быть записана с использованием on3 так

GHCi> let sum3squares = (\x y z -> x+y+z) `on3` (^2)

GHCi> sum3squares 1 2 3

14



**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)]

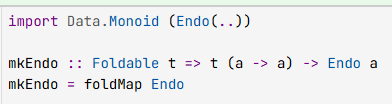
GHCi> appEndo e1 2

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)

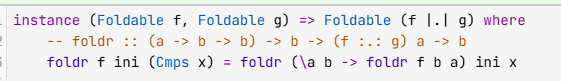
представителем класса типов Foldable при условии, что аргументы композиции являются представителями Foldable.

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 12**

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)

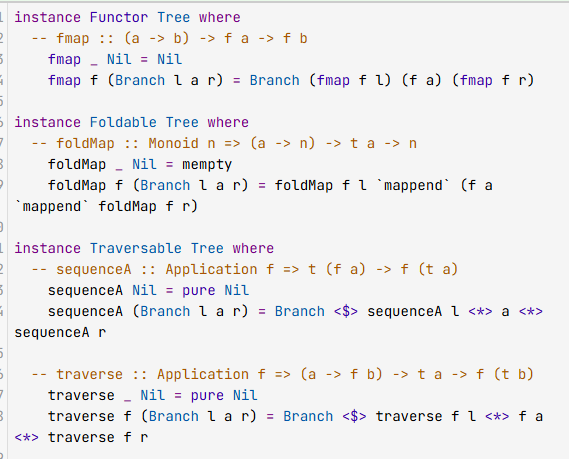
Right (Branch (Branch Nil 1 Nil) 3 Nil)

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]



**2.3 Законы и свойства класса Traversable**

**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

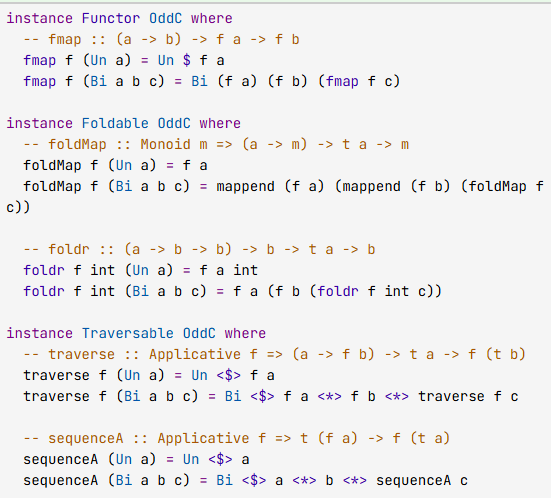
[3,4,1,2,42]

GHCi> sum cnt5

52

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

[Un 44,Un 40]



**Шаг 5**

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

k2c :: Temperature Kelvin -> Temperature Celsius

обеспечивающую следующее поведение

GHCi> k2c 0

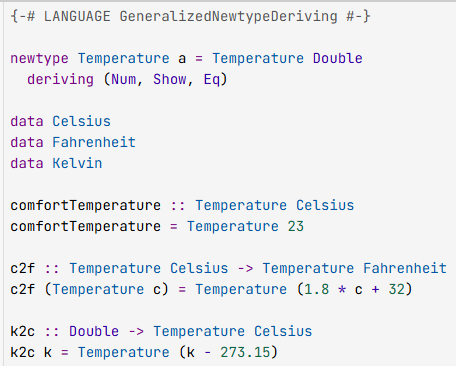
Temperature (-273.15)

GHCi> k2c 0 == Temperature (-273.15)

True

GHCi> k2c 273.15

Temperature 0.0



**2.4 Связь классов Monad и Applicative**

**Шаг 7**

Сделайте парсер

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

из первого модуля курса представителем класса типов Monad:

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

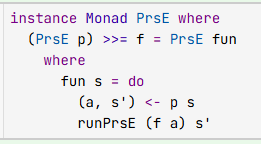
Right (('A','B'),"C")

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'))

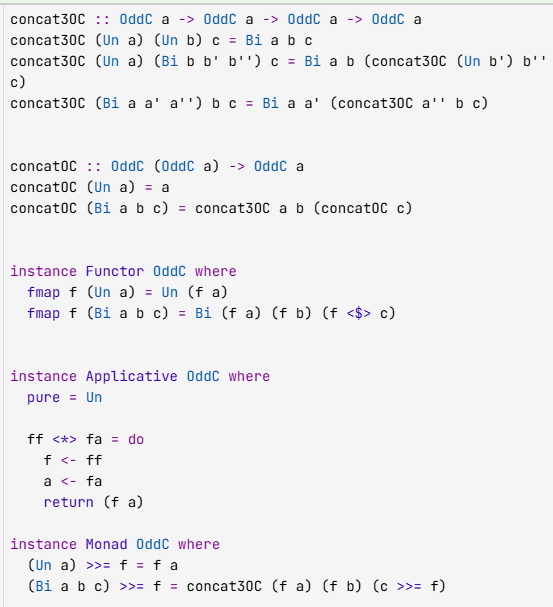
GHCi> tst3 = Bi 'i' 'j' (Un 'k')

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')

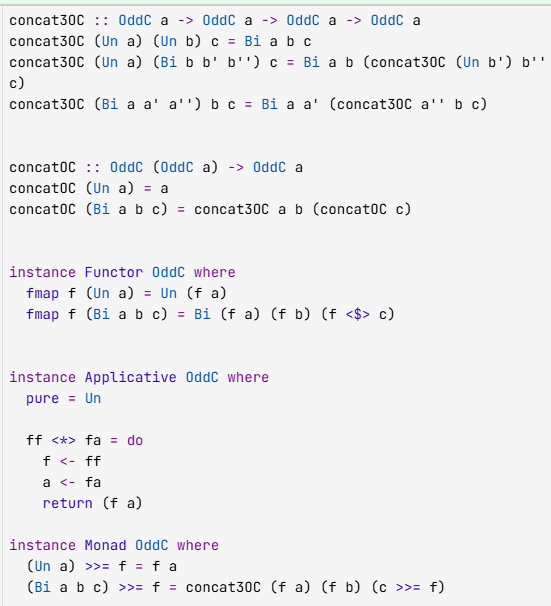
GHCi> tst2 = Bi 'd' 'e' (Bi 'f' 'g' (Un 'h'))

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)}

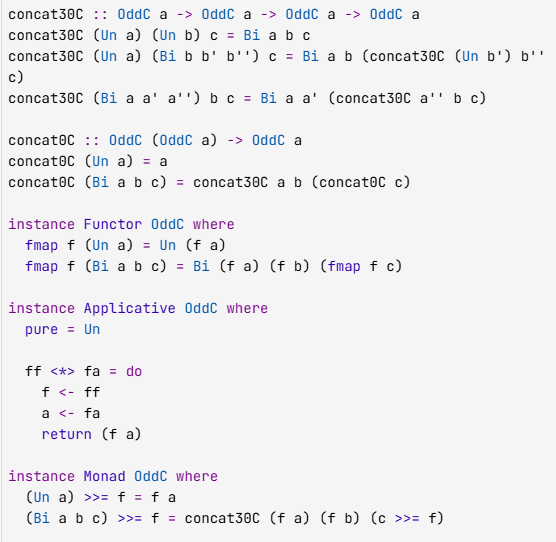
Bi 11 12 (Bi 13 14 (Bi 15 21 (Bi 22 23 (Bi 24 25 (Bi 31 32 (Bi 33 34 (Un 35)))))))

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.



**2.5 Классы типов Alternative и MonadPlus**

**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:

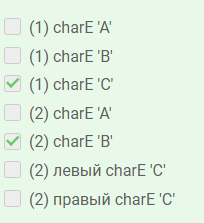
GHCi> runPrsE ((charE 'A' <|> charE 'B') \*> charE 'C') "ABC"

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")

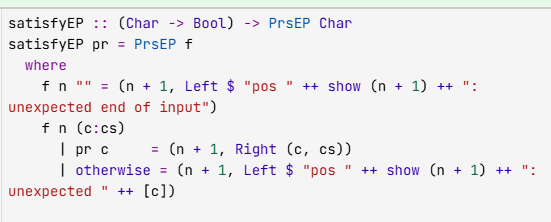
GHCi> parseEP (charEP 'A') "BCD"

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"

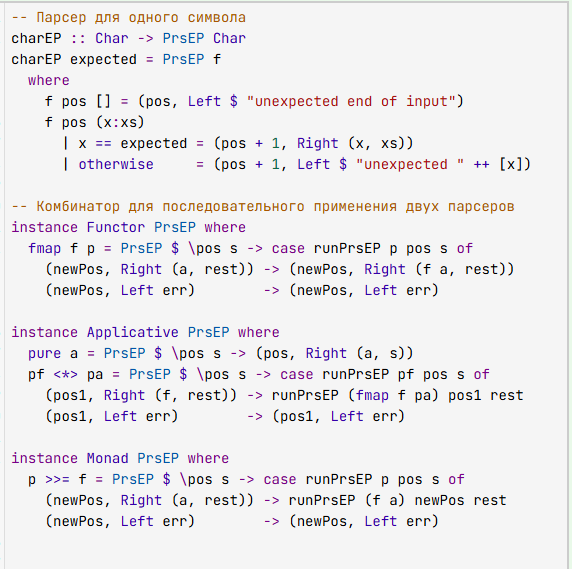
Left "pos 2: unexpected C"

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

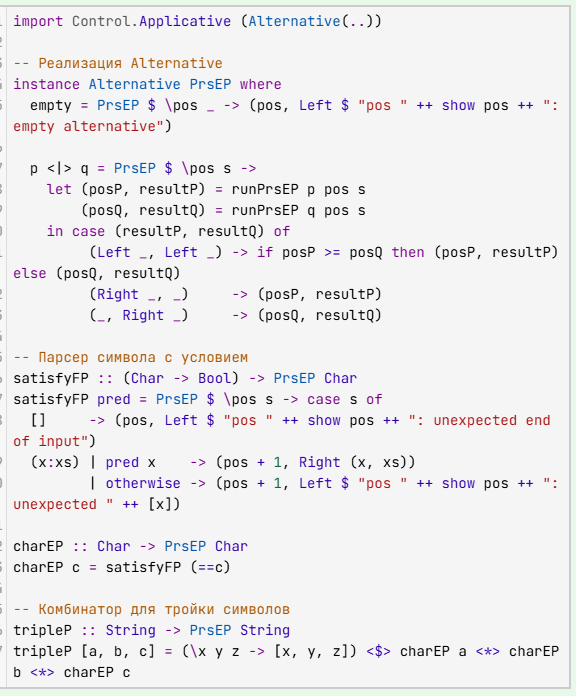
GHCi> parseEP (tripleP "ABC" <|> tripleP "ADC") "ABE"

Left "pos 3: unexpected E"

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

Left "pos 3: unexpected E"

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

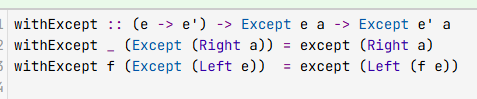
Left "pos 2: unexpected E"

**3 Монады и эффекты**

**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

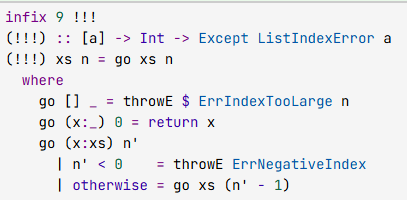
Right 1

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, ()))

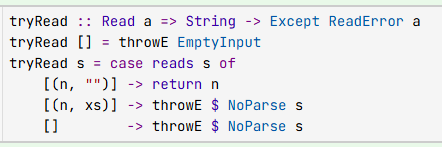
Right (True,())

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

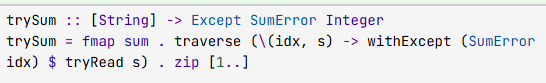
GHCi> runExcept $ trySum ["10", "20", ""]

Left (SumError 3 EmptyInput)

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

Left (SumError 2 (NoParse "two"))

Подсказка: функция withExcept в этом задании может быть чрезвычайно полезна. Постарайтесь максимально эффективно применить знания, полученные на прошлой неделе.



**3.2 Монада Cont**

**Шаг 3**

CPS-преобразование часто применяют для создания предметно-ориентированных языков (DSL).

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

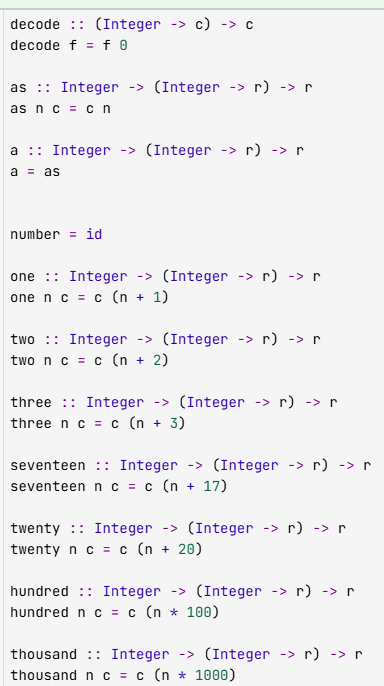
100

GHCi> decode three hundred as a number

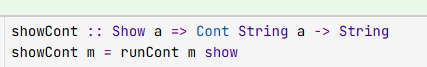
300

GHCi> decode two thousand seventeen as a number

2017



**Шаг 4**

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

**Шаг 9**

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

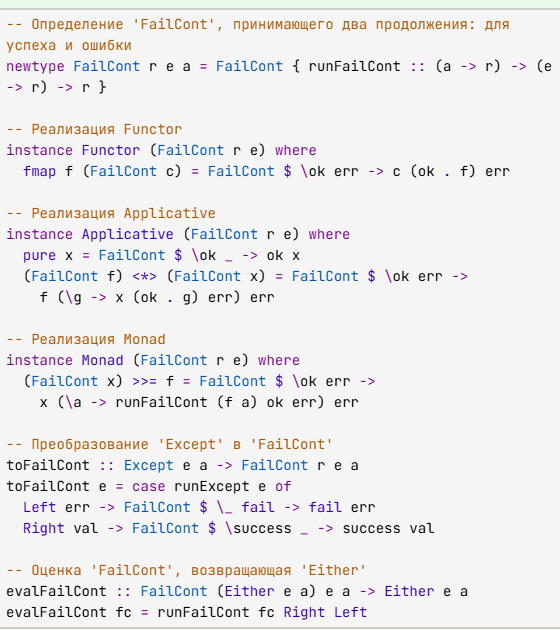
(Здесь используется функция tryRead из предыдущего урока; определять её заново не надо.)

GHCi> evalFailCont $ addInts "15" "12"

Right 27

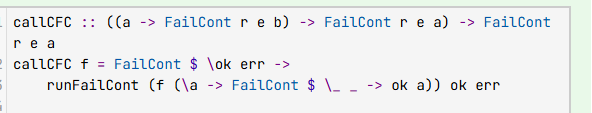
GHCi> runFailCont (addInts "15" "") print (putStrLn . ("Oops: " ++) . show)

Oops: EmptyInput



**Шаг 12**

Реализуйте функцию callCFC для монады FailCont по аналогии с callCC.



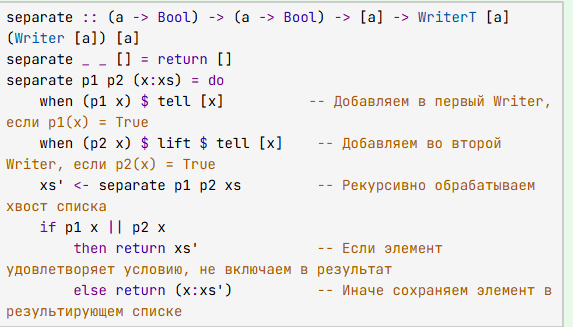
**3.3 Трансформеры монад**

**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 5**

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

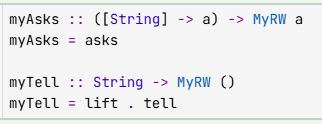
logFirstAndRetSecond = do

el1 <- myAsks head

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

myTell el1

return el2



**Шаг 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

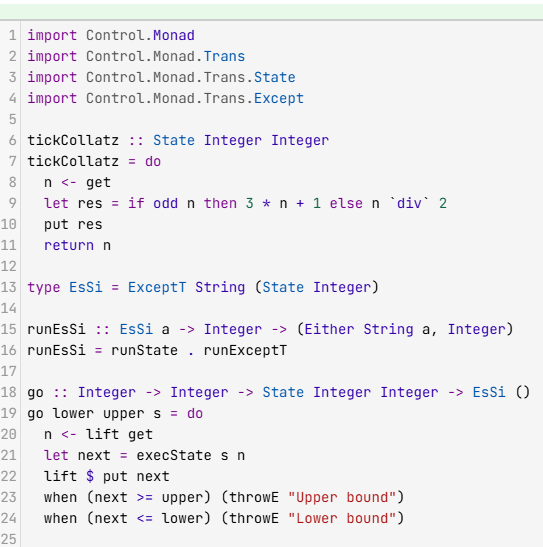
(Left "Upper bound",82)

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

мы получили бы

GHCi> runRiiEsSiT (forever $ go tickCollatz') (1,200) 27

82

41

124

62

31

94

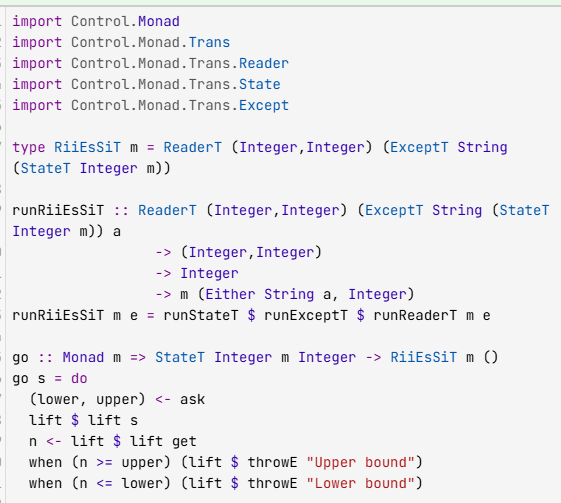
47

142

71

214

(Left "Upper bound",214)



**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]

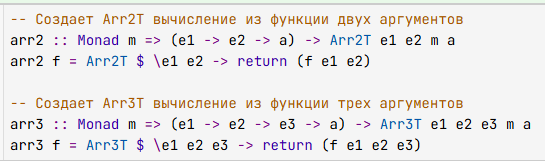
GHCi> (getArr3T $ arr3 foldr) (\*) 1 [1..5] :: Either String Integer

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]

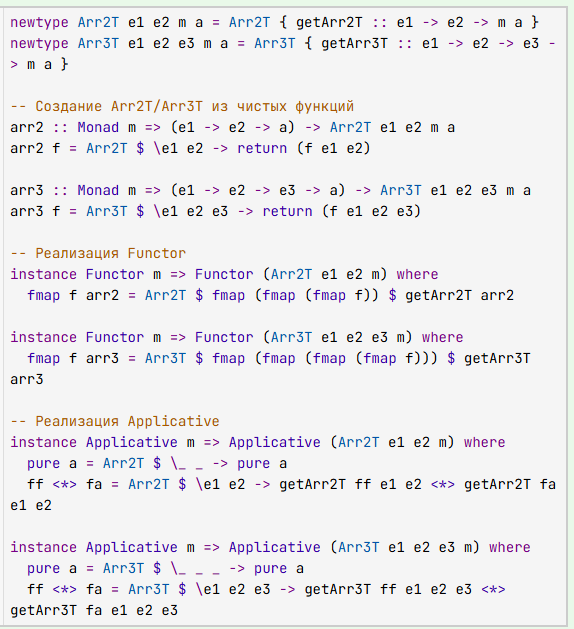
GHCi> (getArr2T $ succ <$> a2l) 10 100

[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

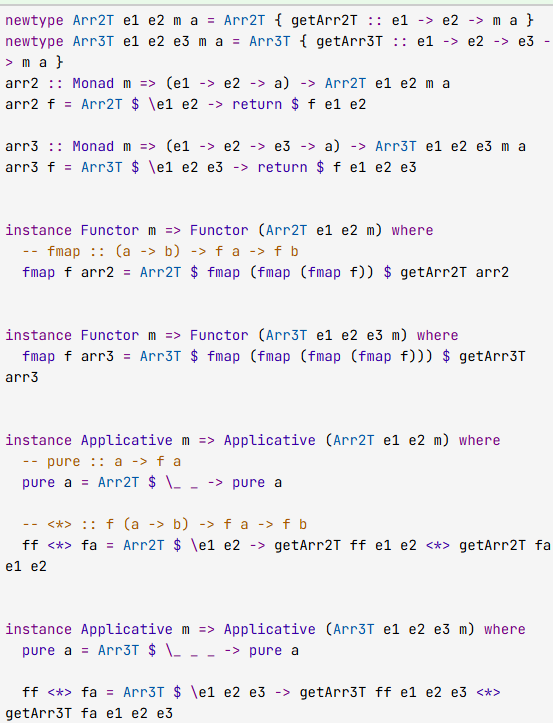
[22,30,7,7]

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]

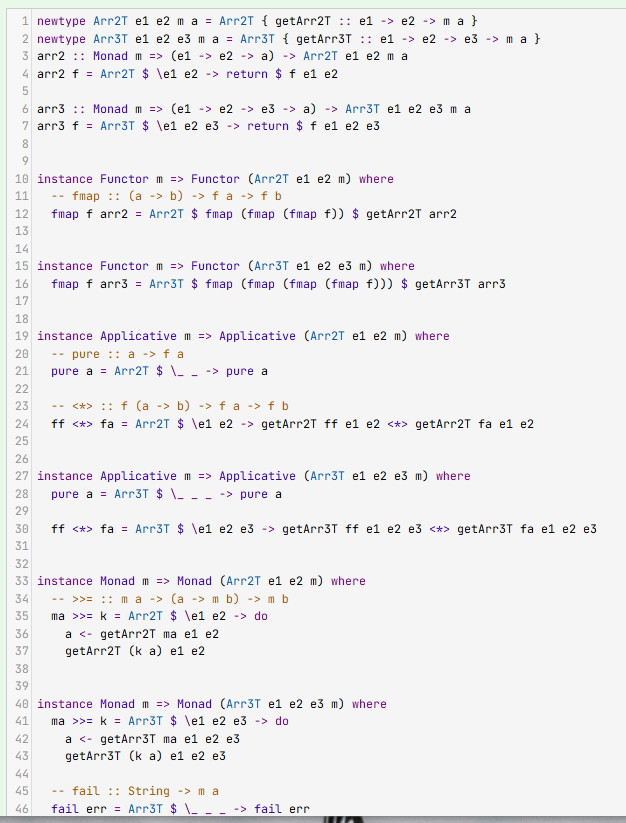
GHCi> getArr2T (do {x <- a2l; y <- a2l; return (x + y)}) 3 5

[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

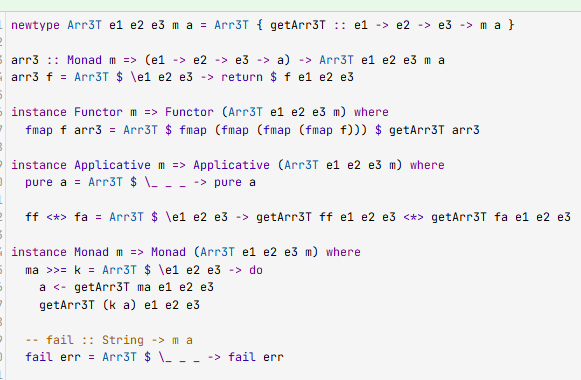
GHCi> getArr3T (do {10 <- a3m; y <- a3m; return y}) 2 3 4

\*\*\* 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 16**

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]

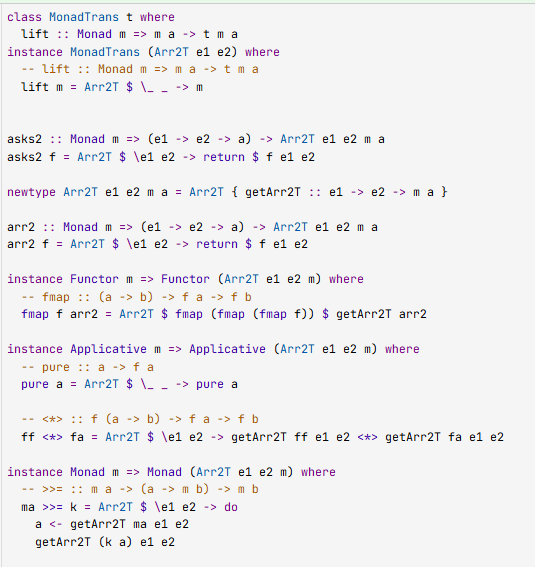
Реализуйте также «стандартный интерфейс» для этой монады — функцию

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'))



**4 Трансформеры монад**

**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)

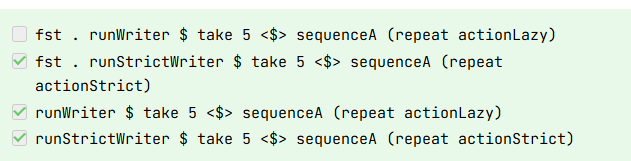
f <\*> v = StrictWriter $ updater (runStrictWriter f) (runStrictWriter v)

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

и определили такие вычисления:

actionLazy = Writer (42,"Hello!")

actionStrict = StrictWriter (42,"Hello!")



**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

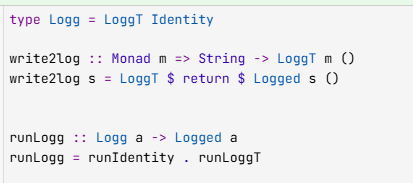
put 42

return $ a \* 100

— такой:

GHCi> runLogg $ runStateT stLog 2

Logged "30" (300,42)



**Шаг 7**

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)

