**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЯ ОБРАЗОВАНИЯ**

**“ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ”**

Факультет информационных технологий

Кафедра технологий программирования

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

Проверила Деканова М. В.

Полоцк, 2023 г.

# Модуль 1

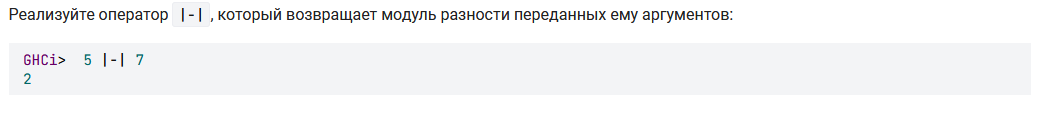
**Задание 1****Решение:**

lenVec3 x y z = sqrt(x\*\*2 + y\*\*2 + z\*\*2)

**Задание 2:** 

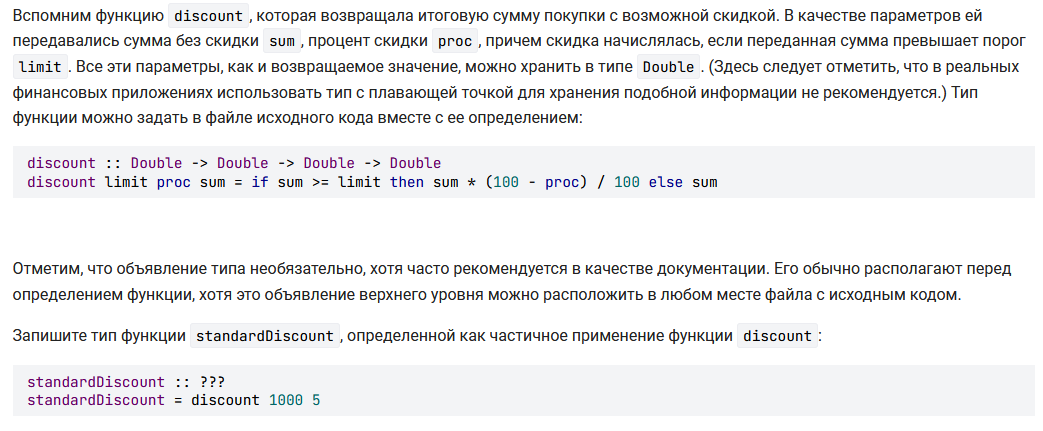
**Решение:**

sign x = if x > 0 then 1 else if x < 0 then (-1) else 0

**Задание 3: **

**Решение:**

x |-| y = abs(x - y)

**Задание 4: **

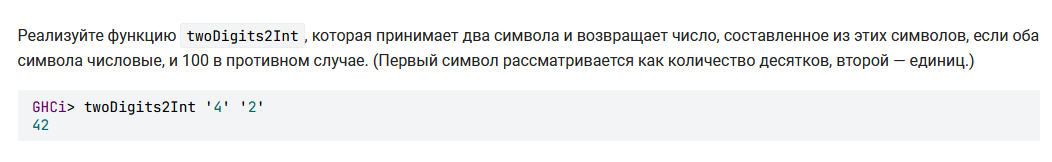
**Решение:**

discount :: Double -> Double -> Double -> Double

discount limit proc sum = if sum >= limit then sum \* (100 - proc) / 100 else sum

standardDiscount :: Double -> Double

standardDiscount = discount 1000 5

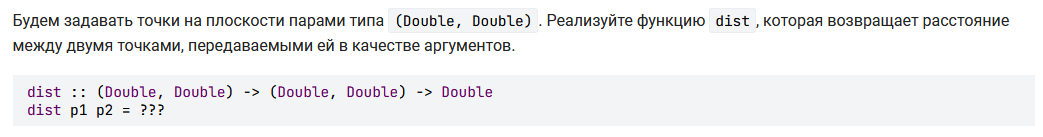
**Задание 5: **

**Решение:**

import Data.Char

twoDigits2Int :: Char -> Char -> Int

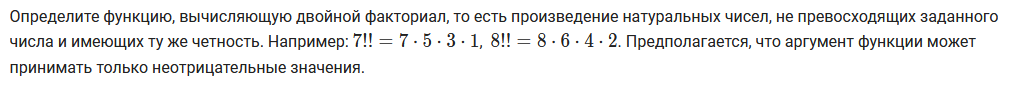
twoDigits2Int x y = if isDigit x && isDigit y then (digitToInt x \* 10 + digitToInt y) else 100

**Задание 6: **

**Решение:**

dist :: (Double, Double) -> (Double, Double) -> Double

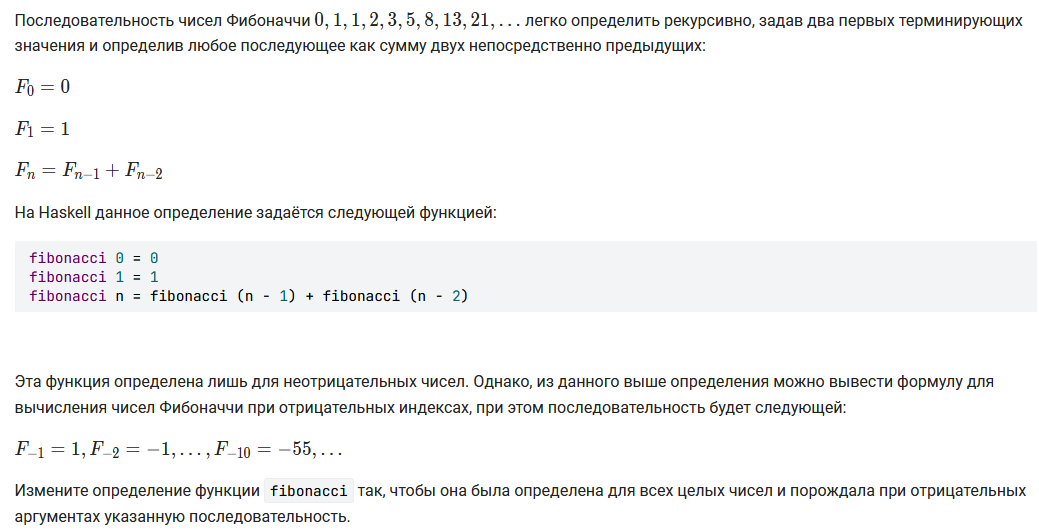
dist p1 p2 = sqrt((fst p2 - fst p1)^2 + (snd p2 - snd p1)^2)

**Задание 7: **

**Решение:**

doubleFact :: Integer -> Integer

doubleFact n = if (n == 1) then 1 else if (n == 2) then 2 else n \* doubleFact (n - 2)

**Задание 8: **

**Решение:**

fibonacci :: Integer -> Integer

fibonacci n | n == 0 = 0

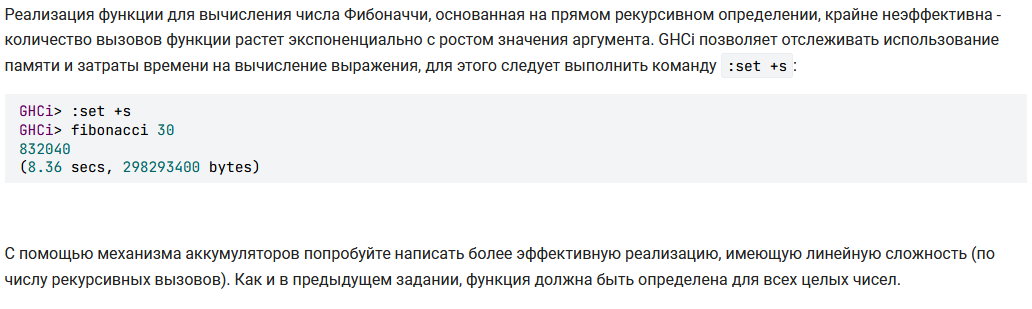
| n == 1 = 1

| n == -1 = 1

| n == -2 = -1

| n > 0 = fibonacci (n - 1) + fibonacci (n - 2)

| n < 0 = (-1) ^ (-n + 1) \* fibonacci(-n)

**Задание 9: **

**Решение:**

fibonacci :: Integer -> Integer

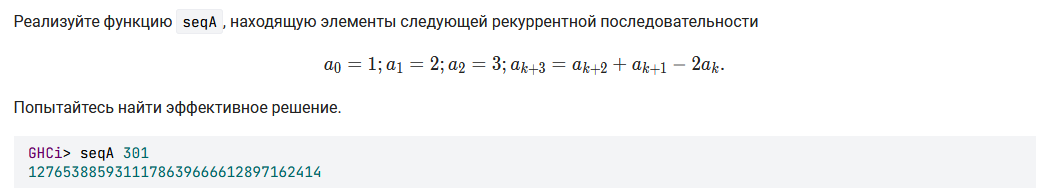
fibonacci n = helper 0 1 n

helper :: Integer -> Integer -> Integer -> Integer

helper acc1 acc2 n | n == 0 = acc1

| n > 0 = helper (acc1 + acc2) acc1 (n - 1)

| n < 0 = helper acc2 (acc1 - acc2) (n + 1)

**Задание 10: **

**Решение:**

seqA :: Integer -> Integer

seqA n = let

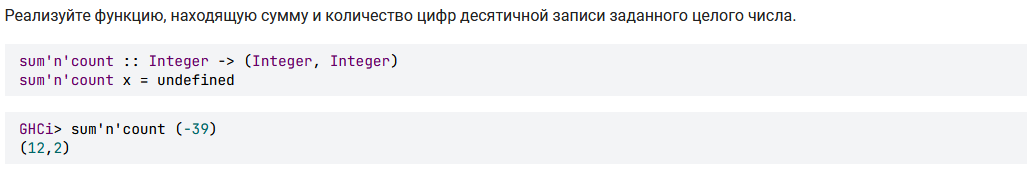
helper a1 a2 a3 0 = a1

helper a1 a2 a3 1 = a2

helper a1 a2 a3 2 = a3

helper a1 a2 a3 n = helper a2 a3 (a3 + a2 - 2 \* a1) (n - 1)

in helper 1 2 3 n

**Задание 11: **

**Решение:**

sum'n'count :: Integer -> (Integer, Integer)

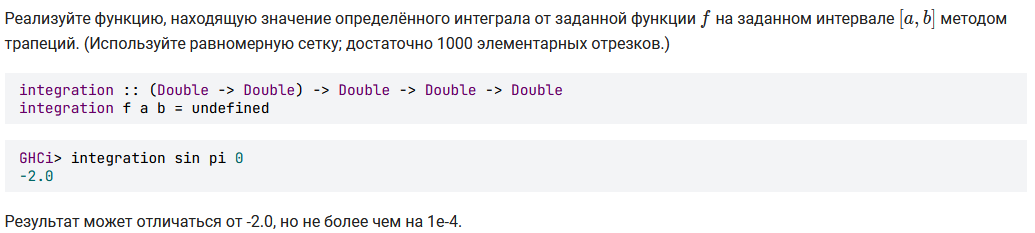
sum'n'count x = if x > 0 then helper x 0 0 else helper (-x) 0 0

where

helper 0 0 0 = (0, 1)

helper 0 sum count = (sum, count)

helper x sum count = helper (x `div` 10) (sum + x `mod` 10) (count + 1)

**Задание 12: **

**Решение:**

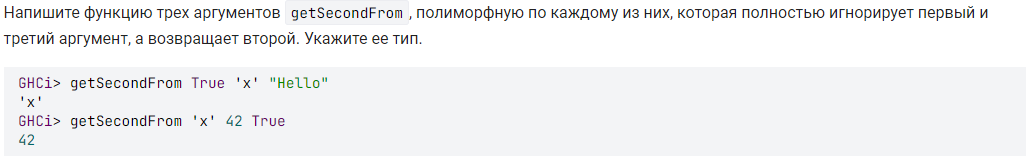
integration :: (Double -> Double) -> Double -> Double -> Double

integration f a b = integ f a b 1000 where

integ f a b 0 = 0

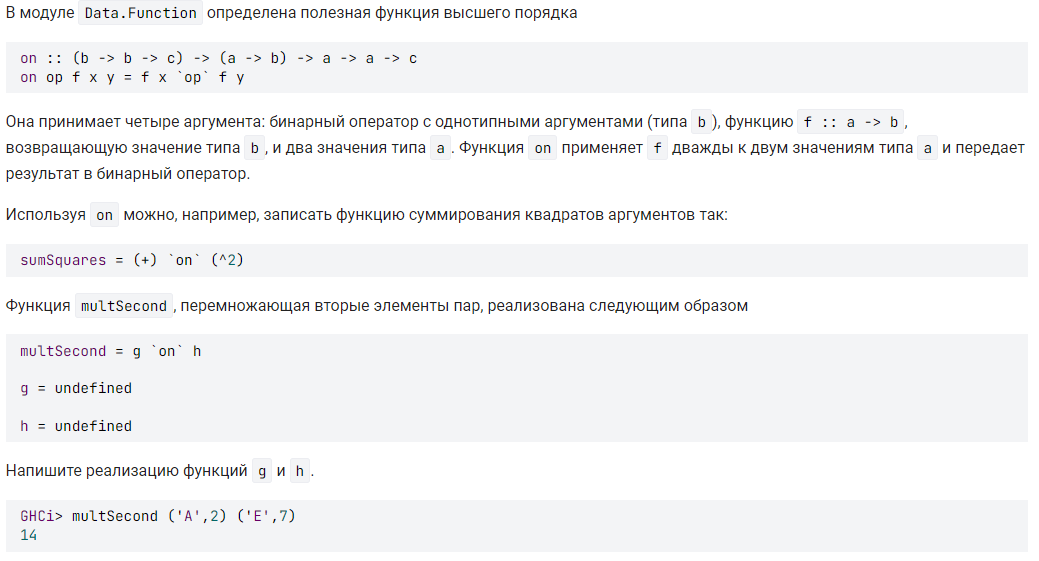
integ f a b n = h \* (f (a) + f (a + h)) / 2 + integ **f (a + h) b (n - 1) where h = (b - a) / n**

# Модуль 2

**Задание 1Решение:**

getSecondFrom :: t1 -> t2 -> t3 -> t2

getSecondFrom x y z = y

**Задание 2: **

**Решение:**

import Data.Function

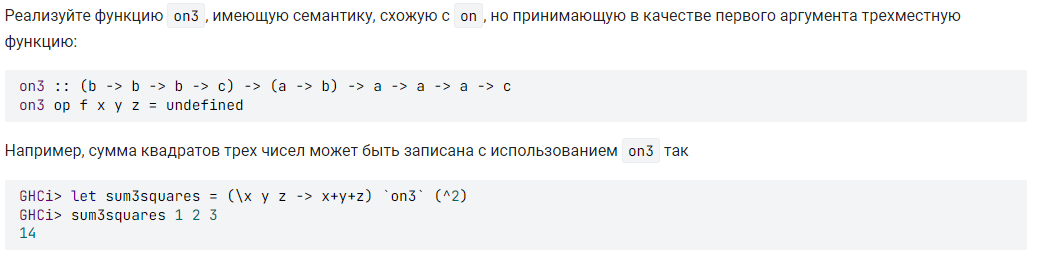
multSecond = g `on` h

g :: Num b => b -> b -> b

g x y = x \* y

h :: (a,b) -> b

h p = snd p

**Задание 3: **

**Решение:**

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

on3 op f x y z = op (f x) (f y) (f z)

**Задание 4:**



**Решение:**

doItYourself = f . g . h

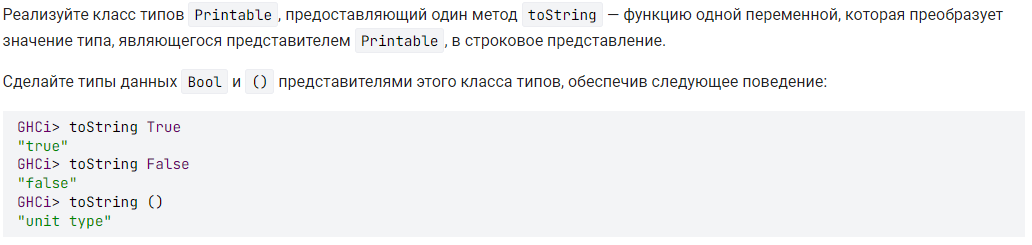
h = (max 42)

g = (^3)

f = (logBase 2)

standardDiscount :: Double -> Double

standardDiscount = discount 1000 5

**Задание 5: **

**Решение:**

class Printable a where

toString :: a -> [Char]

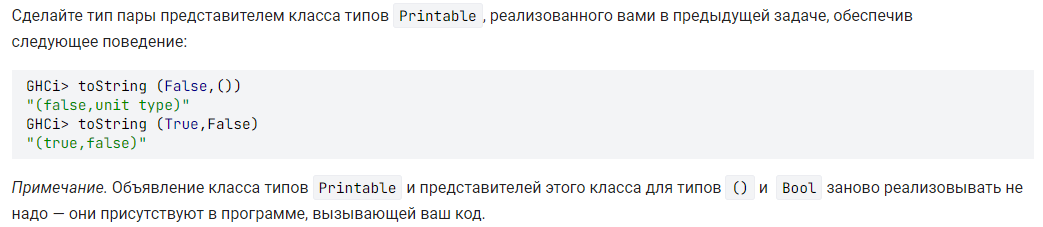
instance Printable Bool where

toString True = "true"

toString False = "false"

instance Printable () where

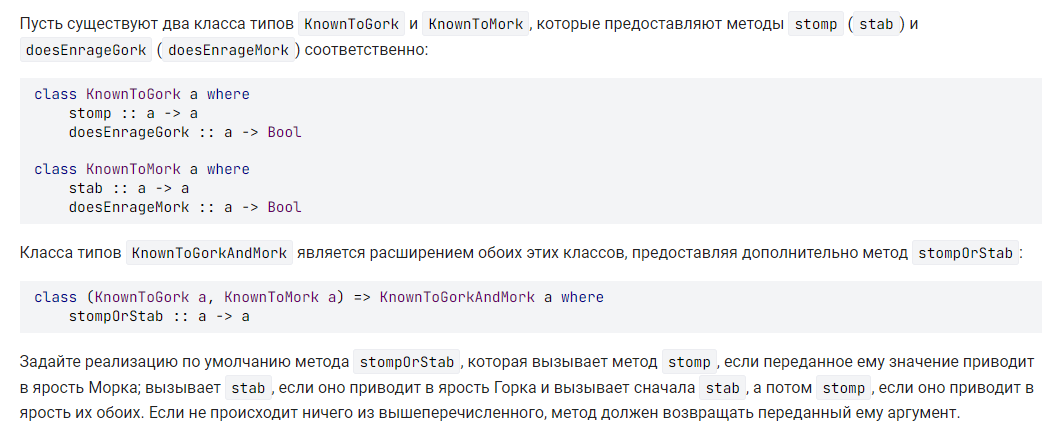
toString () = "unit type"

**Задание 6: **

**Решение:**

instance (Printable a, Printable b) => Printable (a,b) where

toString p = "(" ++ ((toString.fst) p) ++ "," ++ ((toString.snd) p) ++ ")

**Задание 7: **

**Решение:**

class KnownToGork a where

stomp :: a -> a

doesEnrageGork :: a -> Bool

class KnownToMork a where

stab :: a -> a

doesEnrageMork :: a -> Bool

class (KnownToGork a, KnownToMork a) => KnownToGorkAndMork a where

stompOrStab :: a -> a

stompOrStab smth = if doesEnrageMork smth && not (doesEnrageGork smth)

then stomp smth

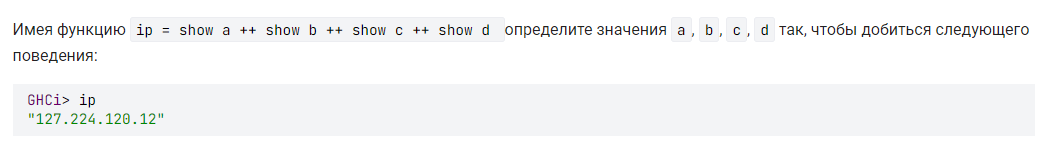
else if doesEnrageGork smth && not (doesEnrageMork smth)

then stab smth

else if doesEnrageMork smth && doesEnrageGork smth

then stomp $ stab smth

else smth

**Задание 8: **

**Решение:**

data MyIp = MyIp Int

instance Show MyIp where

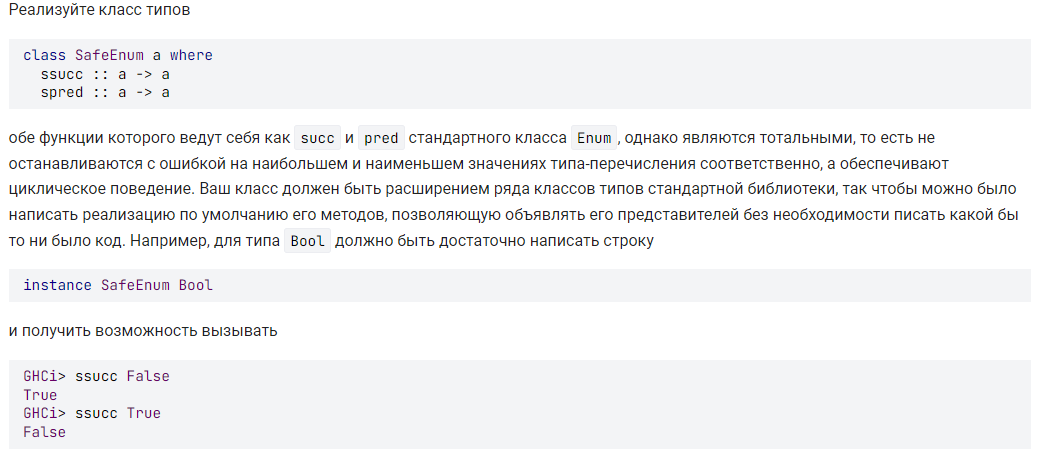
show (MyIp a) = show a ++ "."

a = MyIp 127

b = MyIp 224

c = MyIp 120

d = 12

**Задание 9: **

**Решение:**

class (Ord t, Enum t, Bounded t) => SafeEnum t where

ssucc :: t -> t

ssucc x = if x < maxBound then succ x

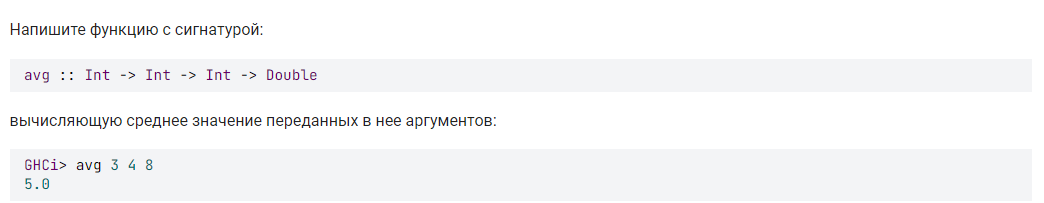
else minBound

spred :: t -> t

spred x = if x > minBound then pred x

else maxBound

instance SafeEnum Bool where

**Задание 10: **

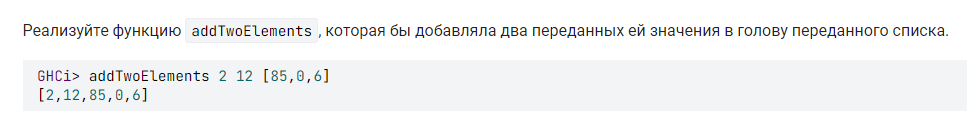
**Решение:**

avg :: Int -> Int -> Int -> Double

avg x y z = fromIntegral (x + y + z) / 3

# Модуль 3

**Задание 1**

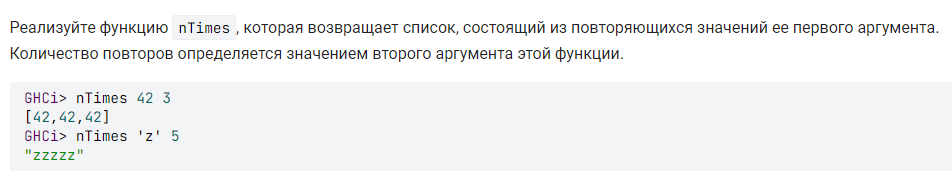


**Решение:**

addTwoElements :: a -> a -> [a] -> [a]

addTwoElements a b l = a : b : l

**Задание 2**



**Решение:**

nTimes:: a -> Int -> [a]

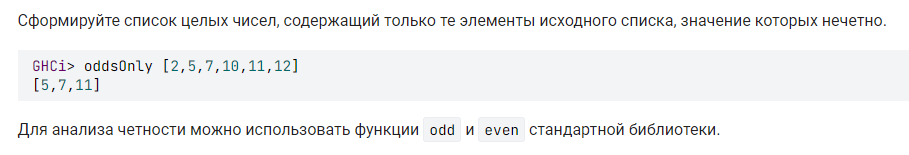
nTimes x n = iter [] x n

where

iter a \_ 0 = a

iter a x n = iter (x : a) x (n - 1)

**Задание 3**



**Решение:**

oddsOnly :: Integral a => [a] -> [a]

oddsOnly = iter []

where

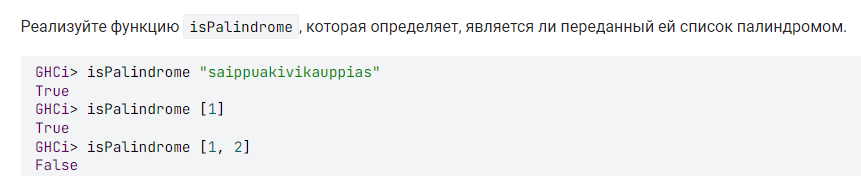
reverse l [] = l

reverse l (x:xs) = reverse (x:l) xs

iter l [] = reverse [] l

iter l (x:xs) = iter (if odd x then (x:l) else l) xs

**Задание 4**

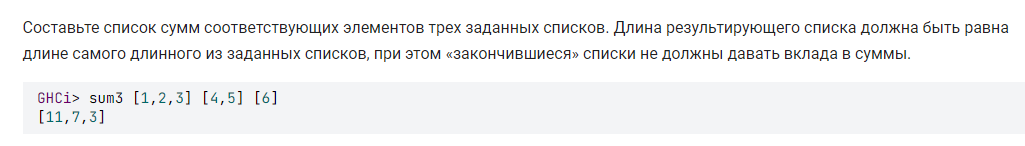


**Решение:**

isPalindrome :: Eq a => [a] -> Bool

isPalindrome x = x == reverse x

**Задание 5**



**Решение:**

sum3 :: Num a => [a] -> [a] -> [a] -> [a]

sum3 = iter []

where

iter a [] [] [] = reverse a

iter a [] a2 a3 = iter a [0] a2 a3

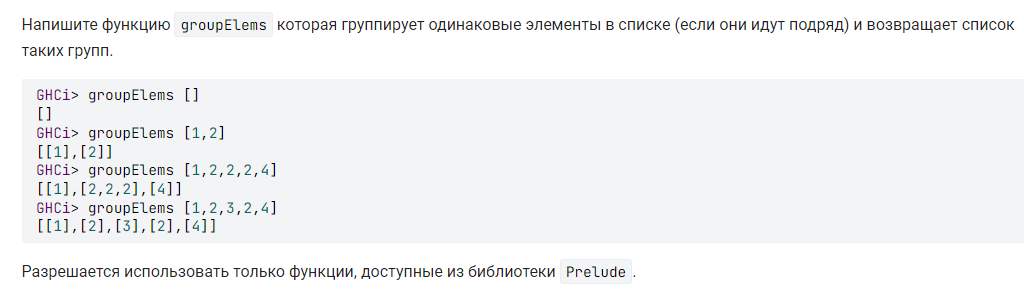
iter a a1 [] a3 = iter a a1 [0] a3

iter a a1 a2 [] = iter a a1 a2 [0]

iter a (x1:xs1) (x2:xs2) (x3:xs3) =

iter (x1 + x2 + x3 : a) xs1 xs2 xs3

**Задание 6**



**Решение:**

groupElems :: Eq a => [a] -> [[a]]

groupElems = iter []

where iter a [] = reverse a

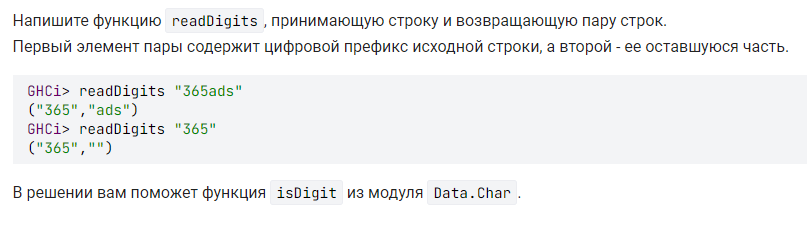
iter [] (x:xs) = iter [[x]] xs

iter ((y:ys):yss) (x:xs)

| x == y = iter ((x:y:ys):yss) xs

| otherwise = iter ([x]:(y:ys):yss) xs

**Задание 7**



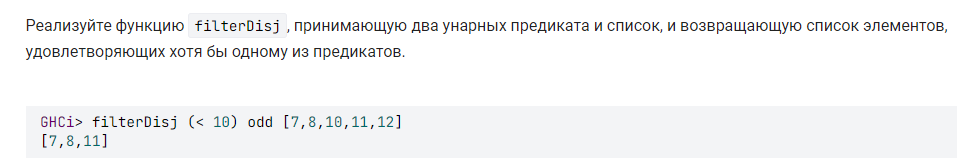
**Решение:**

import Data.Char

readDigits :: String -> (String, String)

readDigits x = (takeWhile isDigit x, dropWhile isDigit x)

**Задание 8**

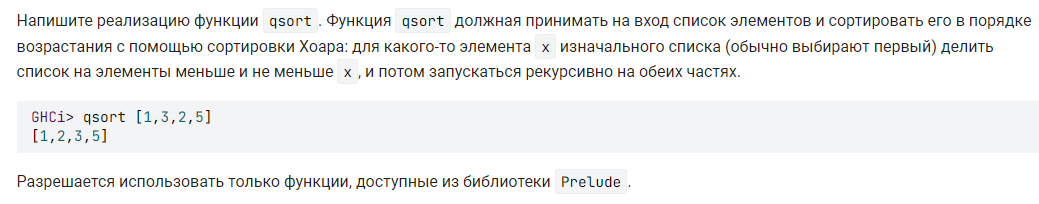


**Решение:**

filterDisj :: (a -> Bool) -> (a -> Bool) -> [a] -> [a]

filterDisj a b = filter (\x -> a x || b x)

**Задание 9**



**Решение:**

qsort :: Ord a => [a] -> [a]

qsort [] = []

qsort [x] = [x]

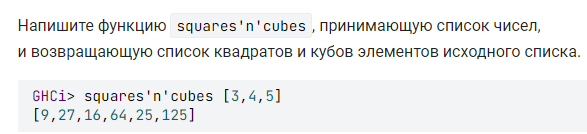
qsort (x:xs) = let

left = filter (\y -> y <= x) xs

right = filter (\y -> y > x) xs

in (qsort left) ++ [x] ++ (qsort right)

**Задание 10**

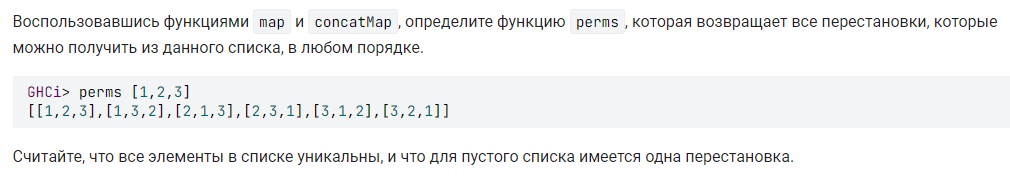


**Решение:**

squares'n'cubes :: Num a => [a] -> [a]

squares'n'cubes = concatMap (\x -> [x\*x, x\*x\*x])

**Задание 11**



**Решение:**

perms :: [a] -> [[a]]

perms [] = [[]]

perms (x:xs) = let

len = length xs

xperm p n = let

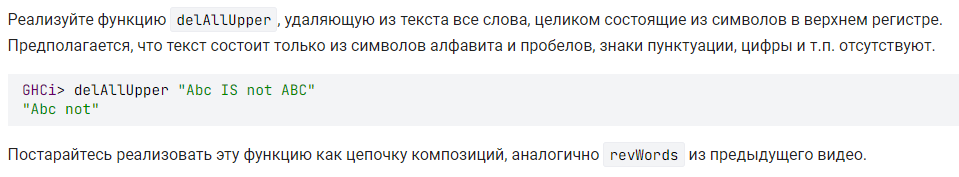
(l, r) = splitAt n p

in l ++ [x] ++ r

xperms p = map (xperm p) [0..len]

in concatMap xperms $ perms xs

**Задание 12**



**Решение:**

import Data.Char

delAllUpper :: String -> String

delAllUpper = unwords . (filter $ any isLower) . words

**Задание 13**

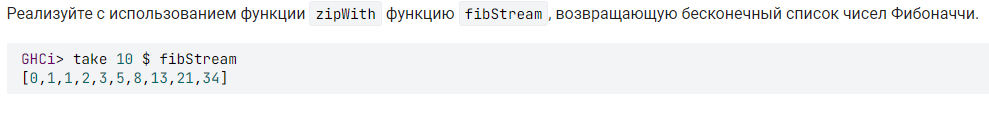


**Решение:**

max3 :: Ord a => [a] -> [a] -> [a] -> [a]

max3 = zipWith3 (\a b c -> max c $ max b a)

**Задание 14**

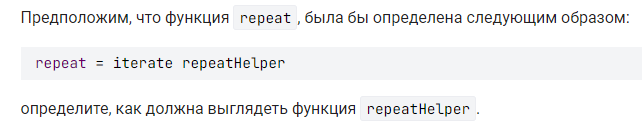


**Решение:**

fibStream :: [Integer]

fibStream = [0, 1] ++ zipWith (+) fibStream (tail fibStream)

**Задание 15**



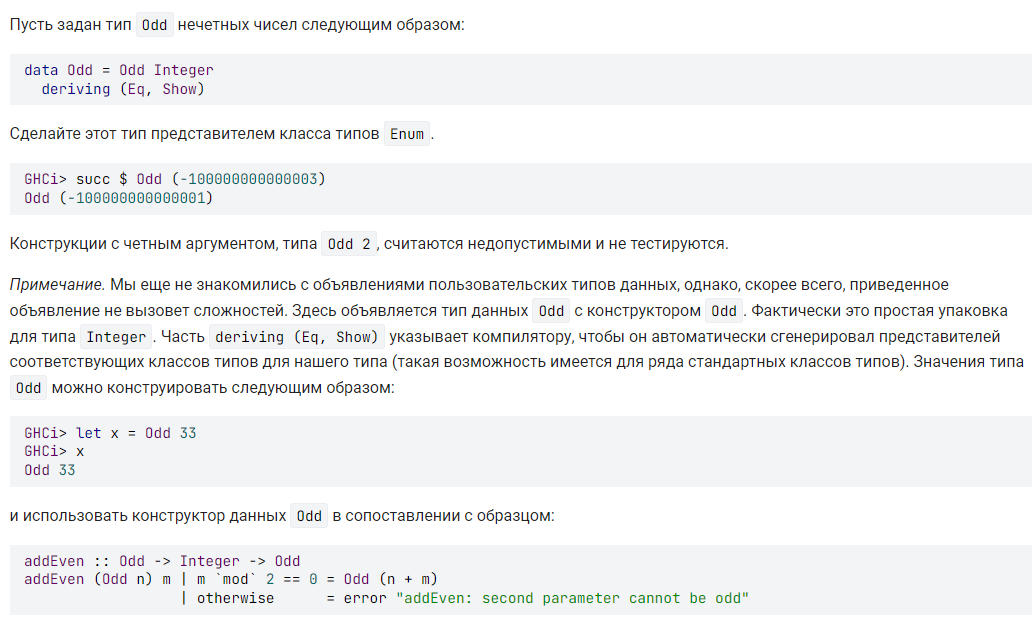
**Решение:**

import Prelude hiding (repeat)

repeat = iterate repeatHelper

repeatHelper = id

**Задание 16**



**Решение:**

instance Enum Odd where

succ (Odd x) = Odd $ x + 2

pred (Odd x) = Odd $ x - 2

toEnum x = Odd $ toInteger x \* 2 + 1

fromEnum (Odd x) = quot (fromInteger x - 1) 2

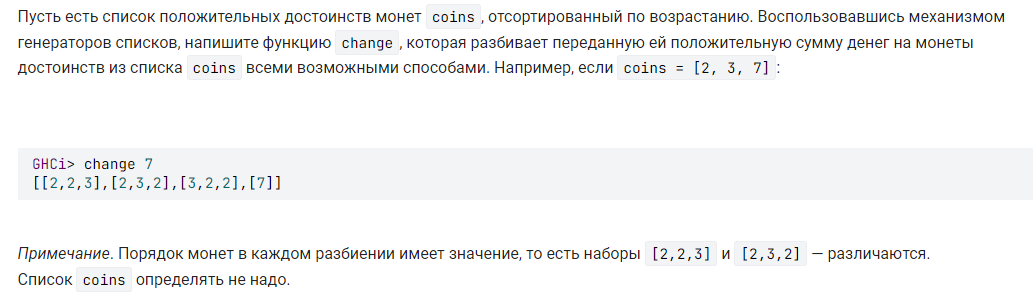
enumFrom = iterate succ

enumFromThen (Odd x) (Odd y) = map Odd [x, y ..]

enumFromTo (Odd x) (Odd y) = map Odd [x, x + 2 .. y]

enumFromThenTo (Odd x) (Odd y) (Odd z) = map Odd [x , y .. z]

**Задание 17**



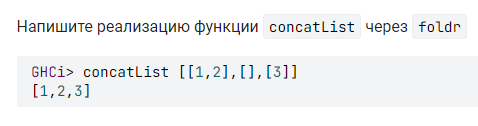
**Решение:**

change :: (Ord a, Num a) => a -> [[a]]

change 0 = [[]]

change s = [coin:ch | coin <- coins, coin <= s, ch <- (change $ s - coin)]

**Задание 18**

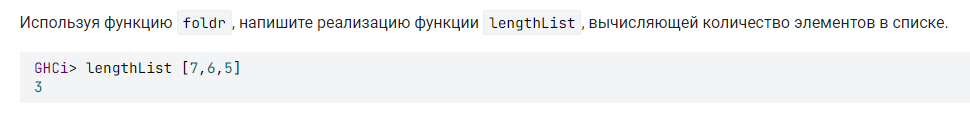


**Решение:**

concatList :: [[a]] -> [a]

concatList = foldr (++) []

**Задание 19**



**Решение:**

lengthList :: [a] -> Int

lengthList = foldr (\x s -> s + 1) 0

**Задание 20**

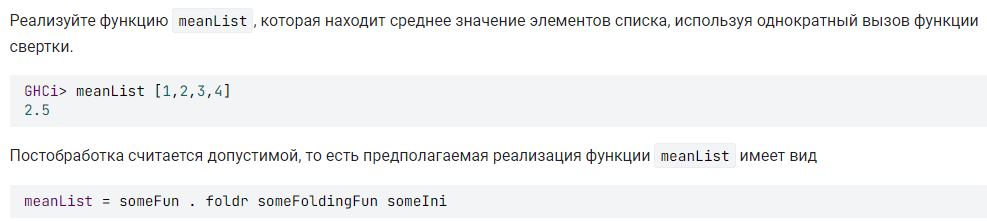


**Решение:**

sumOdd :: [Integer] -> Integer

sumOdd = (foldr (+) 0) . (filter odd)

**Задание 21**

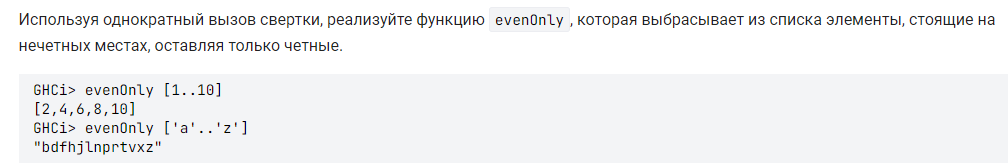


**Решение:**

meanList :: [Double] -> Double

meanList = (uncurry (/)) . foldr (\x (s,c) -> (s+x,c+1)) (0,0)

**Задание 22**

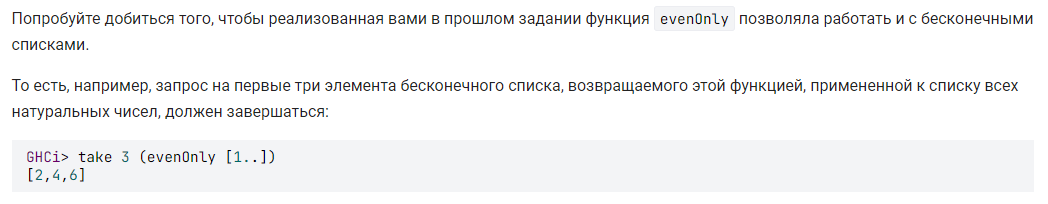


**Решение:**

evenOnly :: [a] -> [a]

evenOnly = (foldr (\(n, x) xs -> if even n then x:xs else xs) []) . (zip [1..])

**Задание 23**

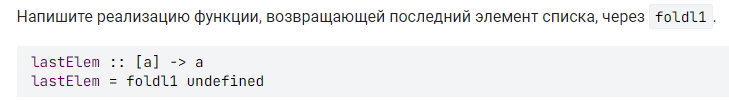


**Решение:**

evenOnly :: [a] -> [a]

evenOnly = (foldr (\(n, x) xs -> if even n then x:xs else xs) []) . (zip [1..])

**Задание 24**

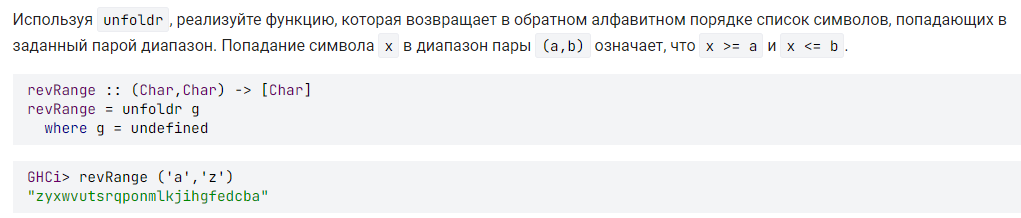


**Решение:**

lastElem :: [a] -> a

lastElem = foldl1 $ flip const

**Задание 25**



**Решение:**

import Data.List

revRange :: (Char,Char) -> [Char]

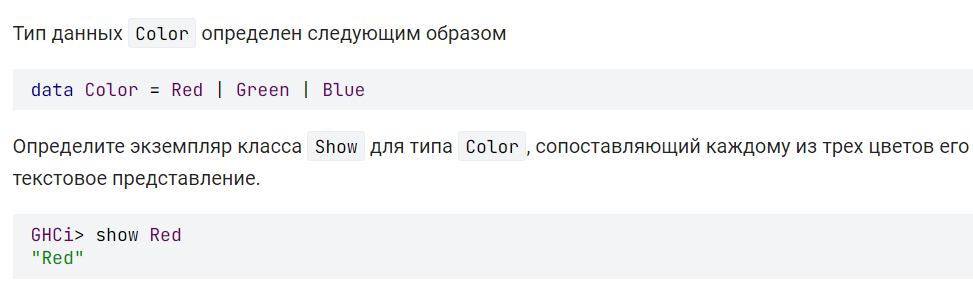
revRange = unfoldr g

where g (a, b) | a > b = Nothing

| otherwise = Just (b, (a, pred b))

# Модуль 4

**Задание 1**

**Решение:**

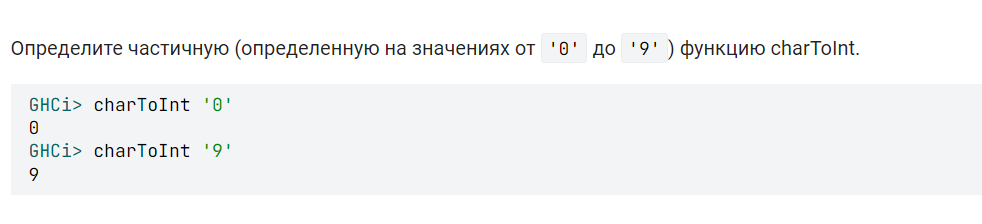
instance Show Color where

show Red = "Red"

show Green = "Green"

show Blue = "Blue"

**Задание 2**

**Решение:**

charToInt :: Char -> Int

charToInt '0' = 0

charToInt '1' = 1

charToInt '2' = 2

charToInt '3' = 3

charToInt '4' = 4

charToInt '5' = 5

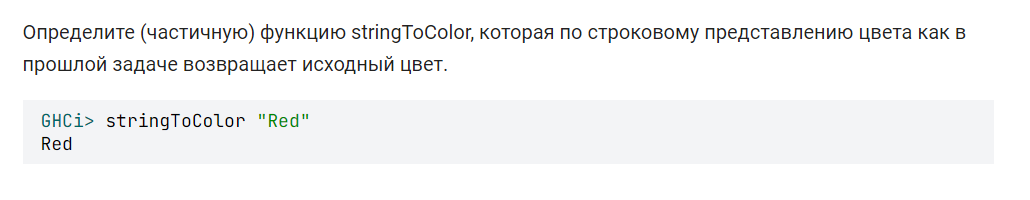
charToInt '6' = 6

charToInt '7' = 7

charToInt '8' = 8

charToInt '9' = 9

**Задание 3**

**Решение:**

data Color = Red | Green | Blue

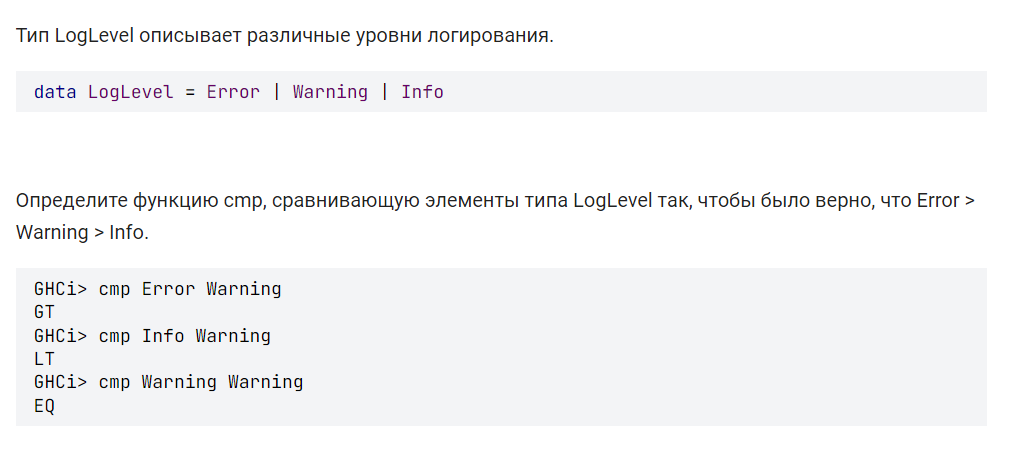
stringToColor :: String -> Color

stringToColor "Red" = Red

stringToColor "Green" = Green

stringToColor "Blue" = Blue

**Задание 4**

**Решение:**

cmp :: LogLevel -> LogLevel -> Ordering

cmp Error Error = EQ

cmp Warning Warning = EQ

cmp Info Info = EQ

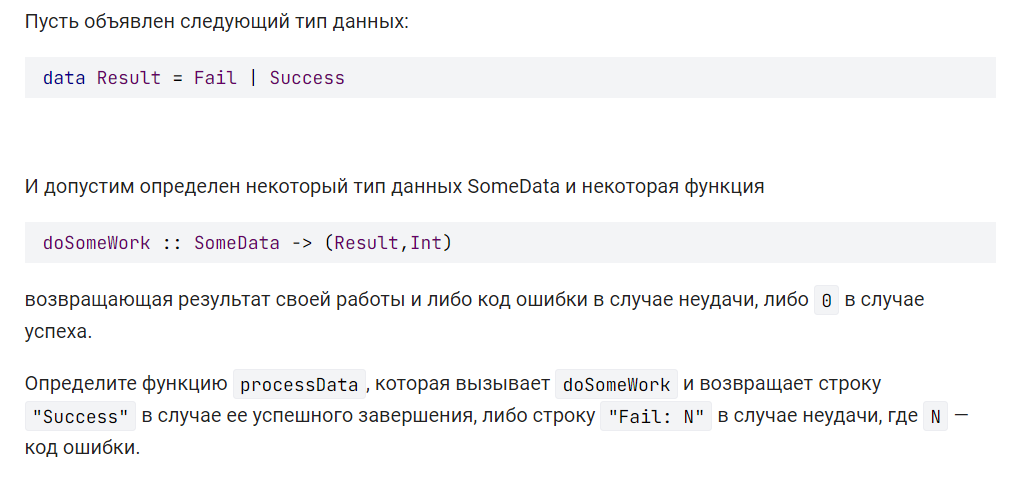
cmp Error \_ = GT

cmp Info \_ = LT

cmp Warning Error = LT

cmp Warning Info = GT

**Задание 5**

**Решение:**

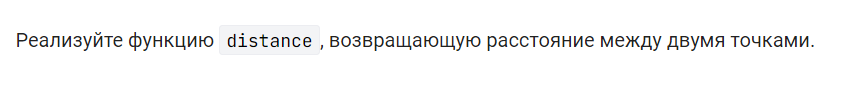
processData :: SomeData -> String

processData x = case doSomeWork x of

(Success, \_) -> "Success"

(\_, n) -> "Fail: " ++ show n

**Задание 6**

**Решение:**

data Point = Point Double Double

origin :: Point

origin = Point 0.0 0.0

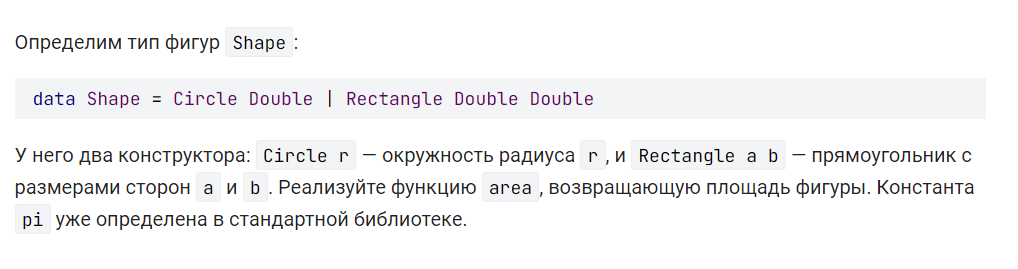
distanceToOrigin :: Point -> Double

distanceToOrigin (Point x y) = sqrt (x ^ 2 + y ^ 2)

distance :: Point -> Point -> Double

distance (Point x1 y1) (Point x2 y2) = sqrt ((x2 - x1) ^ 2 + (y2 - y1) ^ 2)

**Задание 7**

**Решение:**

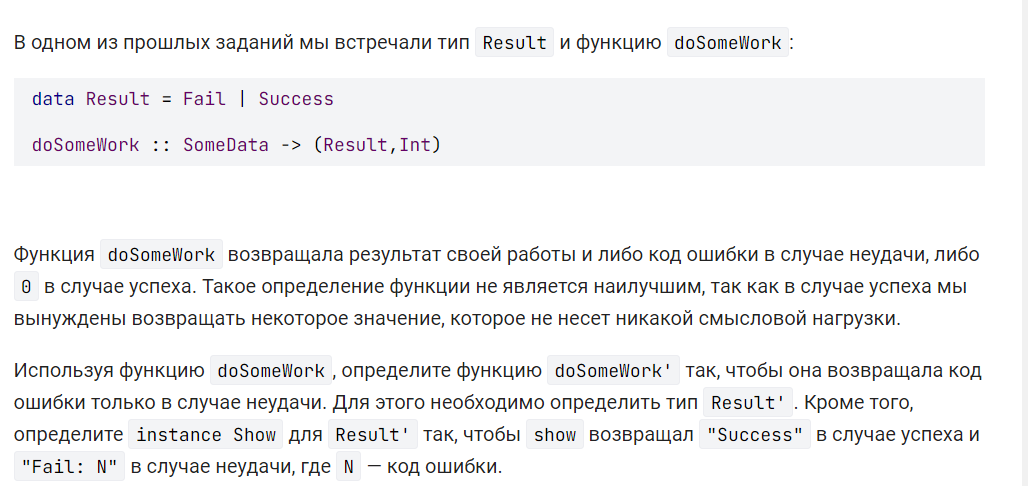
data Shape = Circle Double | Rectangle Double Double

area :: Shape -> Double

area (Circle r) = pi \* (r ^ 2)

area (Rectangle a b) = a \* b

**Задание 8**

**Решение:**

data Result' = Ok | Result' Int

instance Show Result' where

show Ok = "Success"

show (Result' n) = "Fail: " ++ show n

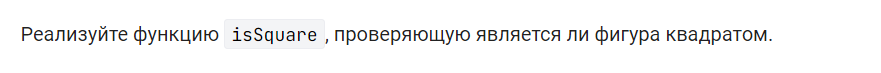
doSomeWork' :: SomeData -> Result'

doSomeWork' x = case snd $ doSomeWork x of

0 -> Ok

n -> Result' n

**Задание 9**

**Решение:**

data Shape = Circle Double | Rectangle Double Double

square :: Double -> Shape

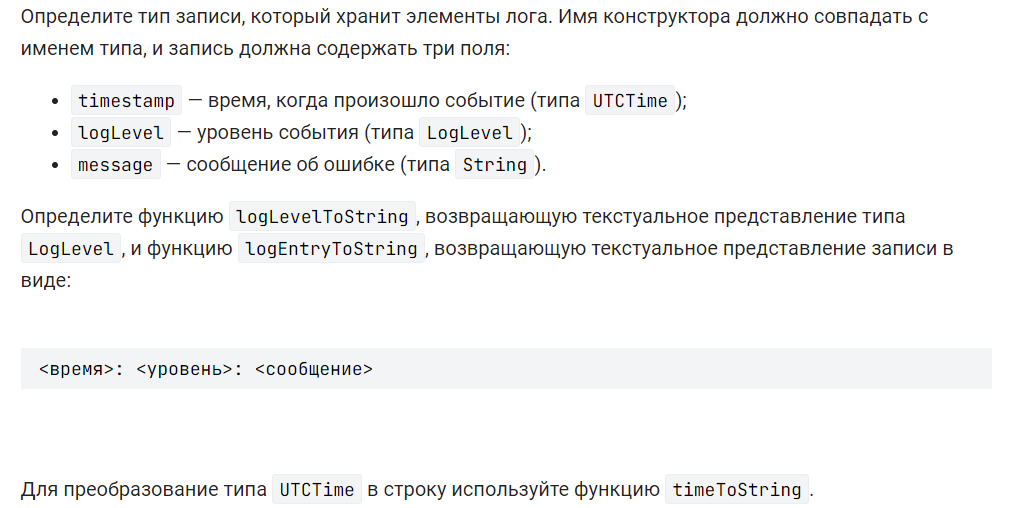
square a = Rectangle a a

isSquare :: Shape -> Bool

isSquare (Rectangle a b) = a == b

isSquare \_ = False

**Задание 10**

**Решение:**

import Data.Time.Clock

import Data.Time.Format

import System.Locale

timeToString :: UTCTime -> String

timeToString = formatTime defaultTimeLocale "%a %d %T"

data LogLevel = Error | Warning | Info

data LogEntry = LogEntry { timestamp :: UTCTime, logLevel :: LogLevel, message :: String}

logLevelToString :: LogLevel -> String

logLevelToString Error = "Error"

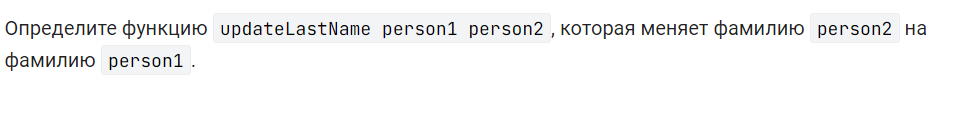
logLevelToString Warning = "Warning"

logLevelToString Info = "Info"

logEntryToString :: LogEntry -> String

logEntryToString a = timeToString (timestamp a) ++ ": " ++ logLevelToString (logLevel a) ++ ": " ++ (message a)

**Задание 11**

**Решение:**

data Person = Person { firstName :: String, lastName :: String, age :: Int }

updateLastName :: Person -> Person -> Person

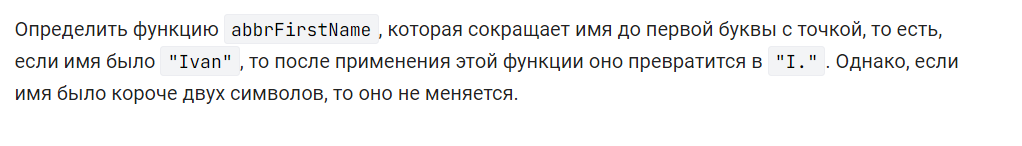
updateLastName (Person \_ lastName1 \_) person2 =

person2

{ lastName = lastName1

}

**Задание 12**

**Решение:**

data Person = Person { firstName :: String, lastName :: String, age :: Int }

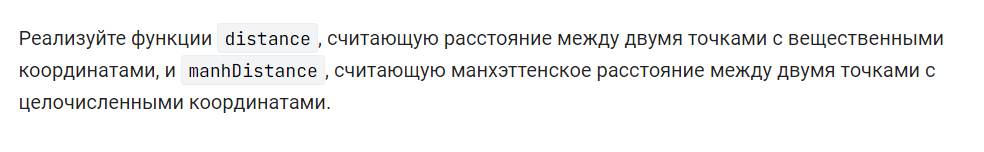
abbrFirstName :: Person -> Person

abbrFirstName person@Person {firstName = fn}

| length fn > 2 = person {firstName = head fn : "."}

| otherwise = person

**Задание 13**

**Решение:**

data Coord a = Coord a a

distance :: Coord Double -> Coord Double -> Double

distance (Coord x1 y1) (Coord x2 y2) =

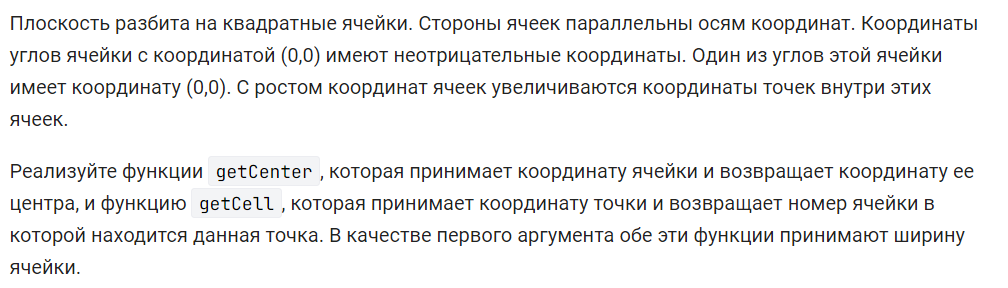
sqrt ((x2 - x1) ^ 2 + (y2 - y1) ^ 2)

manhDistance :: Coord Int -> Coord Int -> Int

manhDistance (Coord x1 y1) (Coord x2 y2) =

abs (x2 - x1) + abs (y2 - y1)

**Задание 14**

**Решение:**

data Coord a = Coord a a

getCenter :: Double -> Coord Int -> Coord Double

getCenter width (Coord x y) = Coord x' y'

where

toCenterCoordinate c = (fromIntegral c + 0.5) \* width

x' = toCenterCoordinate x

y' = toCenterCoordinate y

getCell :: Double -> Coord Double -> Coord Int

getCell width (Coord x y) = Coord x' y'

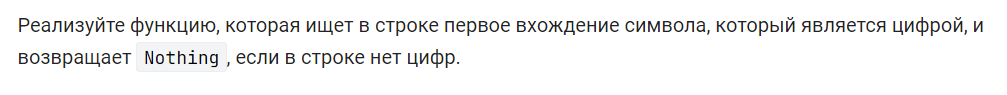
where

toCellCoordinate c = floor (c / width)

x' = toCellCoordinate x

y' = toCellCoordinate y

**Задание 15**

**Решение:**

import Data.Char (isDigit)

import Data.List (find)

findDigit :: [Char] -> Maybe Char

findDigit = find isDigit

**Задание 16**

**Решение:**

import Data.Char(isDigit)

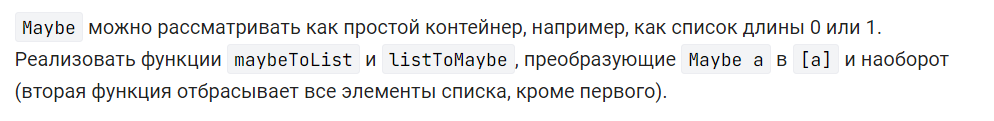
import Data.Maybe (fromMaybe)

findDigit :: [Char] -> Maybe Char

findDigitOrX :: [Char] -> Char

findDigitOrX xs = fromMaybe 'X' (findDigit xs)

**Задание 17**

**Решение:**

maybeToList :: Maybe a -> [a]

maybeToList (Just x) = [x]

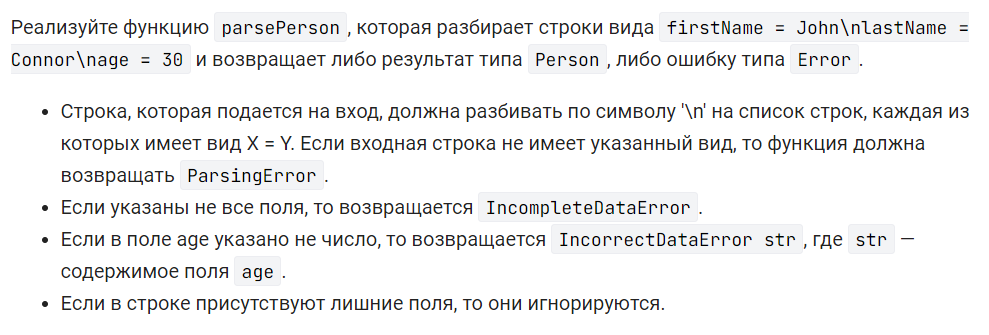
maybeToList \_ = []

listToMaybe :: [a] -> Maybe a

listToMaybe (x : xs) = Just x

listToMaybe \_ = Nothing

**Задание 18**

**Решение:**

import Data.Map (Map, fromList, lookup)

import Data.Text (Text, pack, splitOn, unpack)

import Text.Read (readMaybe)

import Prelude hiding (lookup)

data Error = ParsingError | IncompleteDataError | IncorrectDataError String

data Person = Person { firstName :: String, lastName :: String, age :: Int }

parsePerson :: String -> Either Error Person

parsePerson input = do

map <- toMap input

toPerson map

where

toMap :: String -> Either Error (Map String String)

toMap x =

fmap

fromList

( mapM

(toPair . splitOn (pack " = "))

(splitOn (pack "\n") (pack x))

)

where

toPair :: [Text] -> Either Error (String, String)

toPair xs = case xs of

(a : b : \_) -> Right (unpack a, unpack b)

\_ -> Left ParsingError

toPerson :: Map String String -> Either Error Person

toPerson map = do

firstName <- tryGet "firstName" map

lastName <- tryGet "lastName" map

age <- tryGetInt "age" map

return (Person firstName lastName age)

where

tryGet :: String -> Map String String -> Either Error String

tryGet key map = case lookup key map of

Just value -> Right value

Nothing -> Left IncompleteDataError

tryGetInt :: String -> Map String String -> Either Error Int

tryGetInt key map = do

value <- tryGet key map

tryParseToInt value

where

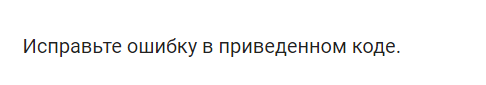
tryParseToInt :: String -> Either Error Int

tryParseToInt value = case readMaybe value of

Just result -> Right result

Nothing -> Left $ IncorrectDataError value

**Задание 19**



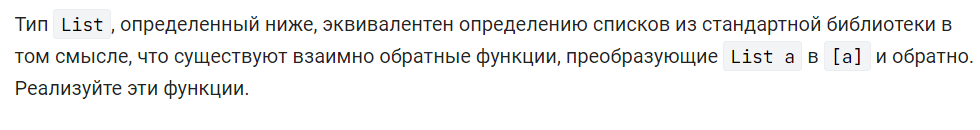
**Решение:**

eitherToMaybe :: Either a b -> Maybe a

eitherToMaybe (Left a) = Just a

eitherToMaybe (Right \_) = Nothing

**Задание 20**

**Решение:**

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

fromList :: List a -> [a]

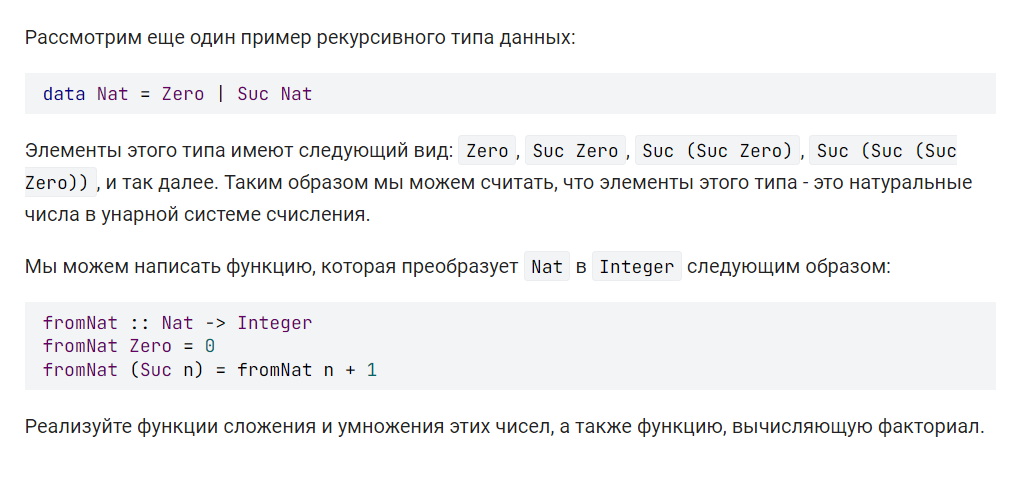
fromList Nil = []

fromList (Cons x xs) = x : fromList xs

toList :: [a] -> List a

toList = foldr Cons Nil

**Задание 21**

**Решение:**

data Nat = Zero | Suc Nat

fromNat :: Nat -> Integer

fromNat Zero = 0

fromNat (Suc n) = fromNat n + 1

toNat :: Integer -> Nat

toNat 0 = Zero

toNat n = Suc (toNat (n - 1))

add :: Nat -> Nat -> Nat

add a b = toNat $ fromNat a + fromNat b

mul :: Nat -> Nat -> Nat

mul a b = toNat $ fromNat a \* fromNat b

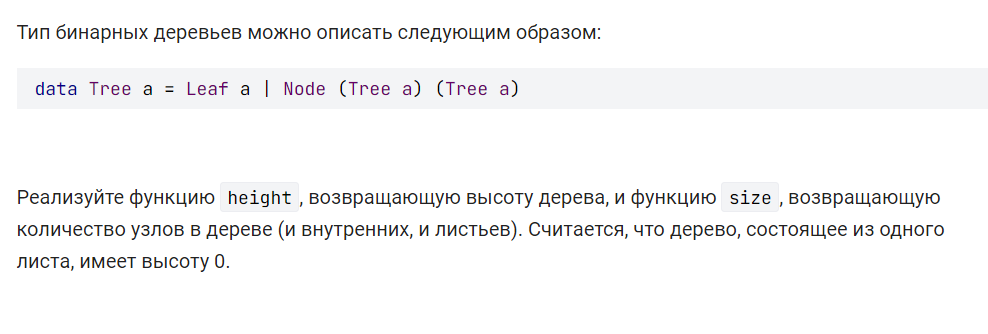
factorial :: Integer -> Integer

factorial n = product [1 .. n]

fac :: Nat -> Nat

fac = toNat . factorial . fromNat

**Задание 22**

**Решение:**

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

height :: Tree a -> Int

height (Leaf \_) = 0

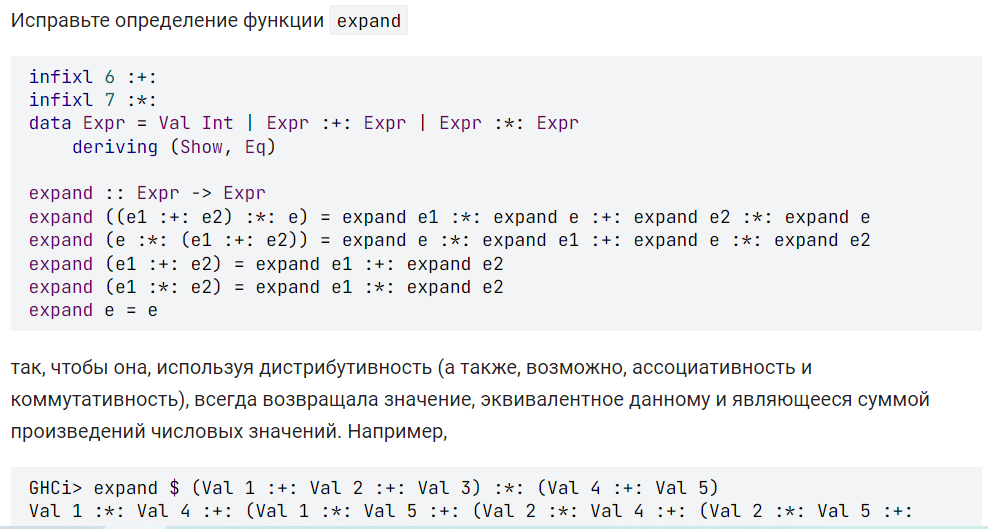
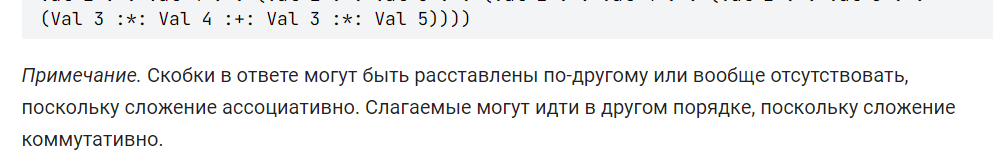
height (Node a b) = 1 + max (height a) (height b)

size :: Tree a -> Int

size (Leaf \_) = 1

size (Node a b) = 1 + size a + size b

**Задание 23**

 **Решение:**

infixl 6 :+:

infixl 7 :\*:

data Expr = Val Int | Expr :+: Expr | Expr :\*: Expr

deriving (Show, Eq)

expand :: Expr -> Expr

expand e

| e /= e' = expand e'

| otherwise = e

where

e' = go e

go :: Expr -> Expr

go ((e1 :+: e2) :\*: e) = go e1 :\*: go e :+: go e2 :\*: go e

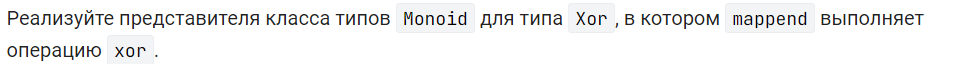
go (e :\*: (e1 :+: e2)) = go e :\*: go e1 :+: go e :\*: go e2

go (e1 :+: e2) = go e1 :+: go e2

go (e1 :\*: e2) = go e1 :\*: go e2

go e = e

**Задание 24**

**Решение:**

newtype Xor = Xor {getXor :: Bool}

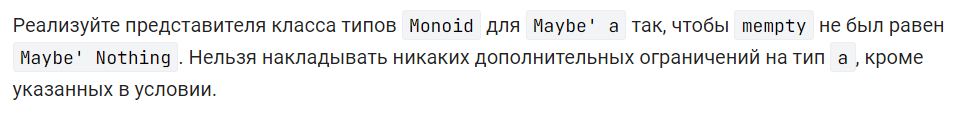
deriving (Eq, Show)

instance Monoid Xor where

mempty = Xor False

a `mappend` b = Xor $ a /= b

**Задание 25**

**Решение:**

newtype Maybe' a = Maybe' {getMaybe :: Maybe a}

deriving (Eq, Show)

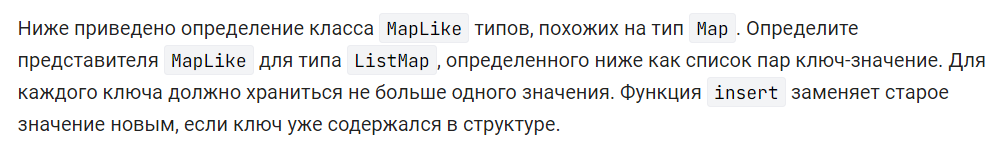
instance (Monoid a) => Monoid (Maybe' a) where

mempty = Maybe' $ Just mempty

(Maybe' (Just a)) `mappend` (Maybe' (Just b)) = Maybe' (Just (a `mappend` b))

\_ `mappend` \_ = Maybe' Nothing

**Задание 26**

**Решение:**

import Prelude hiding (lookup)

import qualified Data.List as L

import Data.Maybe (isNothing)

class MapLike m where

empty :: m k v

lookup :: Ord k => k -> m k v -> Maybe v

insert :: Ord k => k -> v -> m k v -> m k v

delete :: Ord k => k -> m k v -> m k v

fromList :: Ord k => [(k, v)] -> m k v

fromList [] = empty

fromList ((k, v) : xs) = insert k v (fromList xs)

newtype ListMap k v = ListMap {getListMap :: [(k, v)]}

deriving (Eq, Show)

instance MapLike ListMap where

empty = ListMap []

lookup k (ListMap m) = L.lookup k m

insert k v m'@(ListMap m)

| hasNotValue = (ListMap . insert') m

| otherwise = (ListMap . insert' . getListMap) (delete k m')

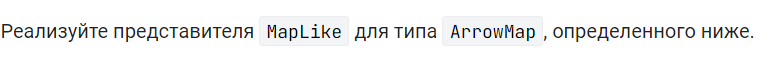
where

hasNotValue = isNothing $ lookup k m'

insert' = (:) (k, v)

delete k (ListMap m) = (ListMap . filter ((/= k) . fst)) m

**Задание 27**



**Решение:**

import Prelude hiding (lookup)

class MapLike m where

empty :: m k v

lookup :: Ord k => k -> m k v -> Maybe v

insert :: Ord k => k -> v -> m k v -> m k v

delete :: Ord k => k -> m k v -> m k v

fromList :: Ord k => [(k, v)] -> m k v

fromList [] = empty

fromList ((k, v) : xs) = insert k v (fromList xs)

newtype ArrowMap k v = ArrowMap {getArrowMap :: k -> Maybe v}

instance MapLike ArrowMap where

empty = ArrowMap $ const Nothing

lookup k (ArrowMap m) = m k

insert k v (ArrowMap m) = ArrowMap f

where

f x =

if x /= k

then m x

else Just v

delete k (ArrowMap m) = ArrowMap f

where

f x =

if x /= k

then m x

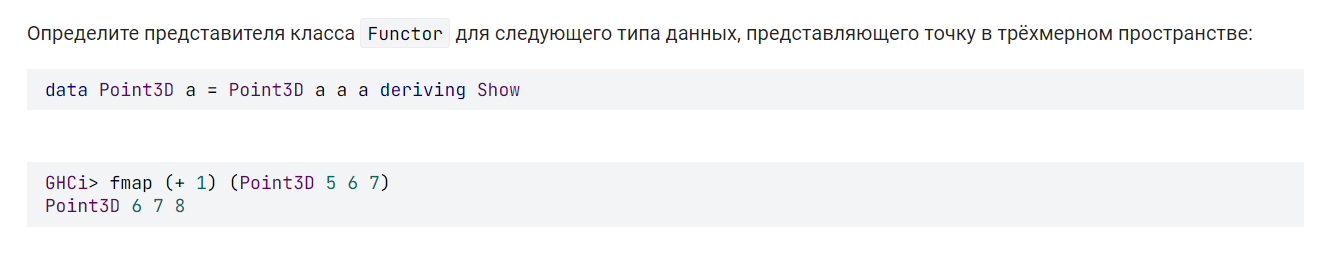
else Nothing

fromList [] = empty

fromList ((k, v) : xs) = insert k v (fromList xs)

# Модуль 5

**Задание 1**

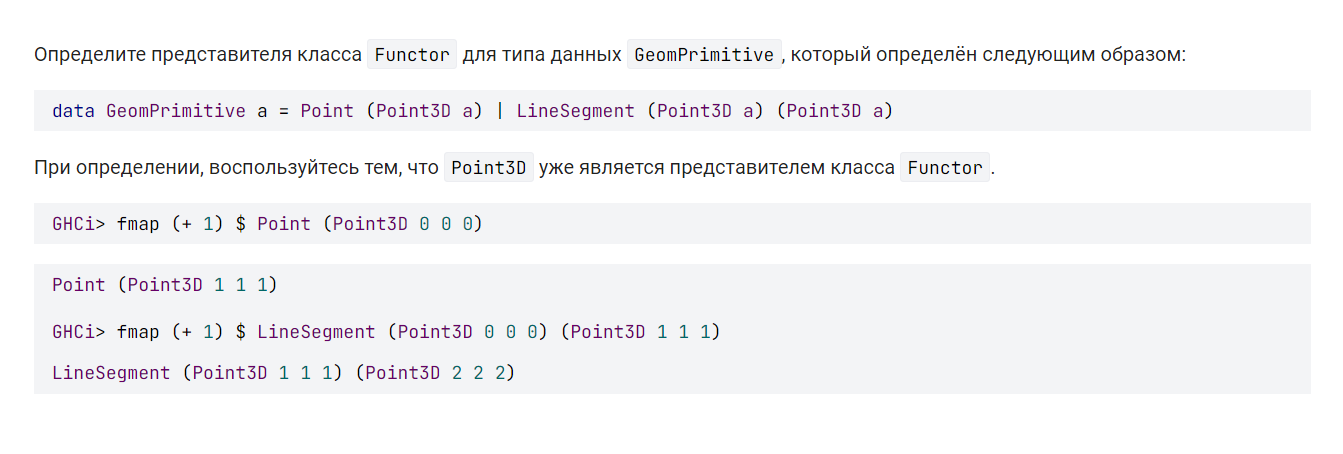


**Решение:**

instance Functor Point3D where

fmap f (Point3D x y z) = Point3D (f x) (f y) (f z)

**Задание 2:**



**Решение:**

instance Functor GeomPrimitive where

fmap f (Point (Point3D x y z)) = (Point (Point3D (f x) (f y) (f z)))

fmap f (LineSegment (Point3D x1 y1 z1) (Point3D x2 y2 z2)) = LineSegment (Point3D (f x1) (f y1) (f z1)) (Point3D (f x2) (f y2) (f z2))

**Задание 3:**



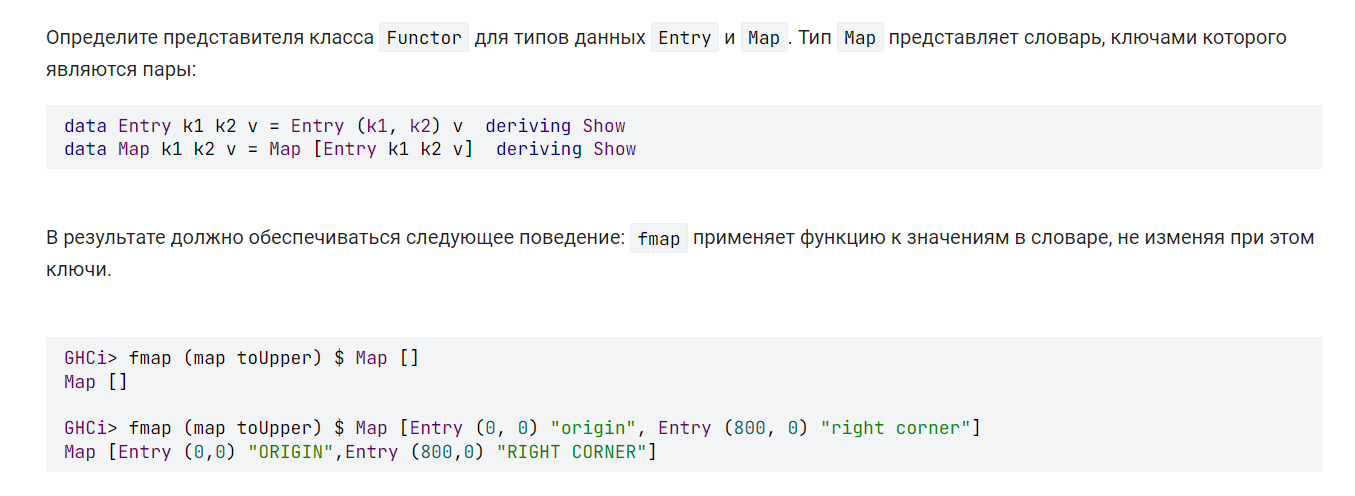
**Решение:**

instance Functor Tree where

fmap f (Leaf a) = Leaf $ fmap f a

fmap f (Branch l a r) = Branch (fmap f l) (fmap f a) (fmap f r)

**Задание 4:**



**Решение:**

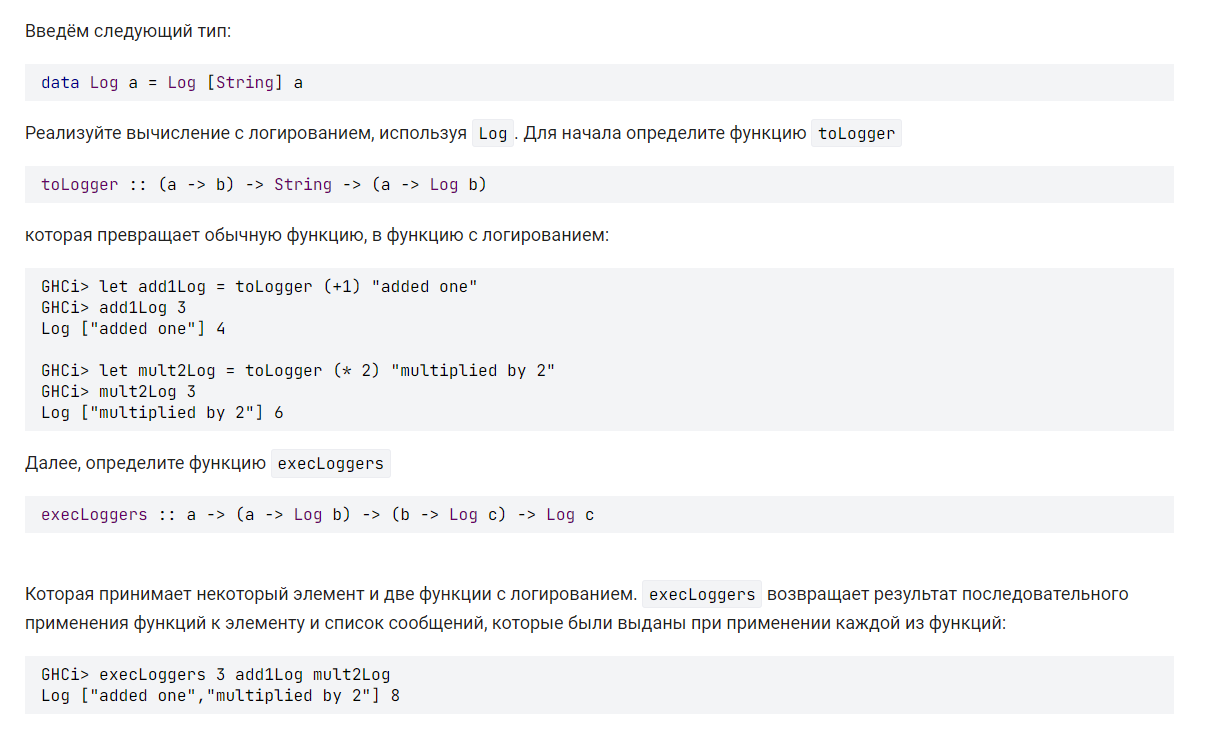
instance Functor (Entry k1 k2) where

fmap f (Entry (k1, k2) v) = Entry (k1, k2) $ f v

instance Functor (Map k1 k2) where

fmap f (Map xs) = Map (map g xs) where g = fmap f

**Задание 5:**



**Решение:**

toLogger f msg = \x -> Log [msg] (f x)

execLoggers x f g = Log ( (getMsg (f x)) ++ (getMsg (g (getValue (f x)))) ) (getValue (g (getValue (f x))))

getValue :: Log b -> b

getValue (Log \_ a) = a

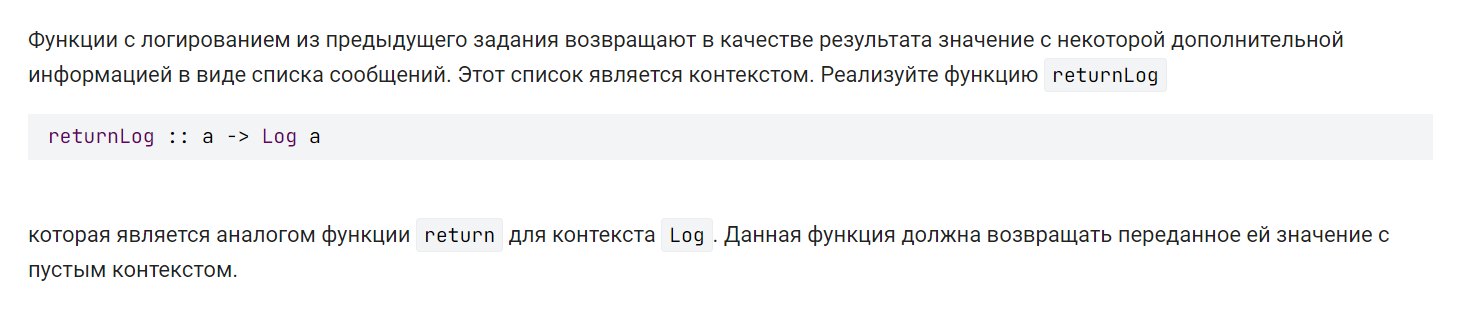
getMsg :: Log b -> [String]

getMsg (Log msg \_) = msg

add1Log = toLogger (+1) "added one"

mult2Log = toLogger (\* 2) "multiplied by 2"

**Задание 6:**

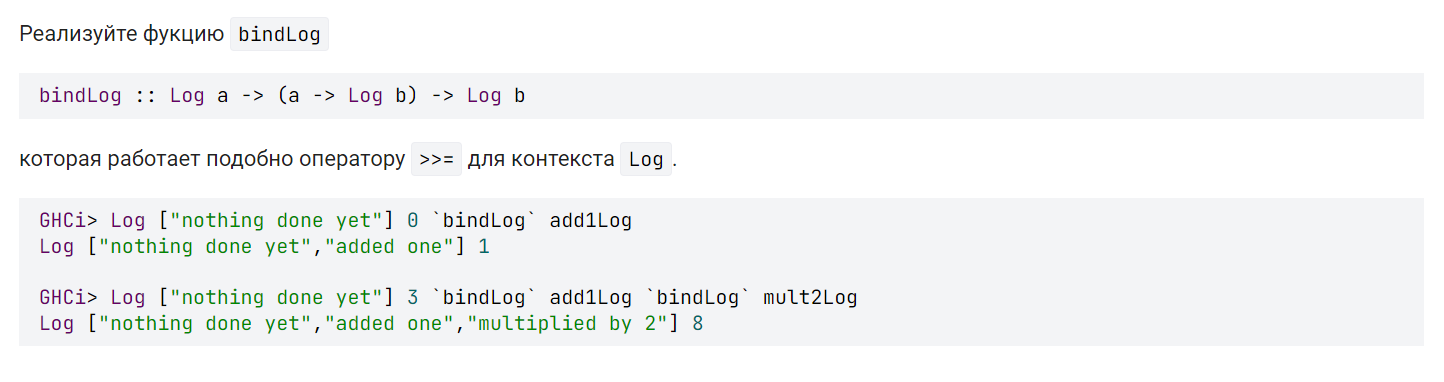


**Решение:**

returnLog :: a -> Log a

returnLog a = Log [] a

**Задание 7:**

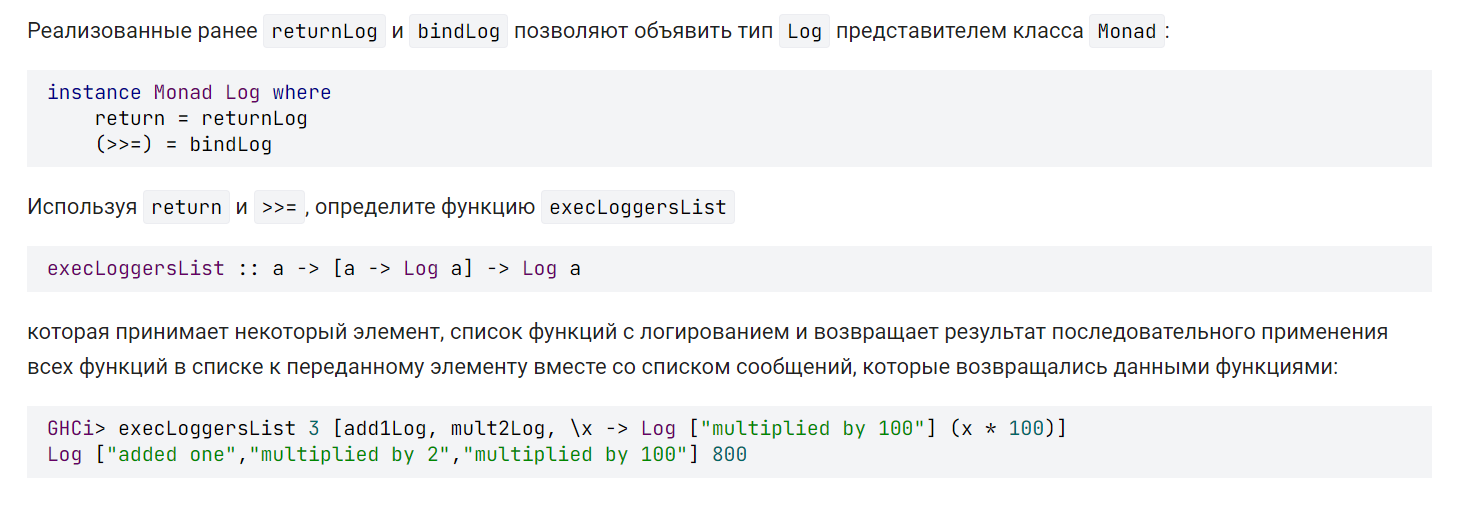


**Решение:**

bindLog :: Log a -> (a -> Log b) -> Log b

bindLog (Log s a) l = (\(Log (ms) r) -> Log (s ++ ms) r) (l a)

**Задание 8:**

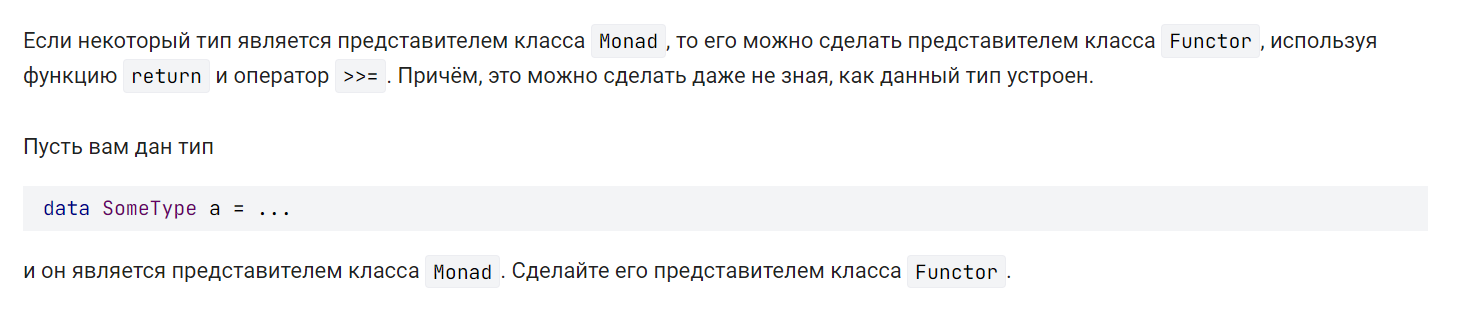


**Решение:**

execLoggersList :: a -> [a -> Log a] -> Log a

execLoggersList = foldl (>>=) . return

**Задание 9:**

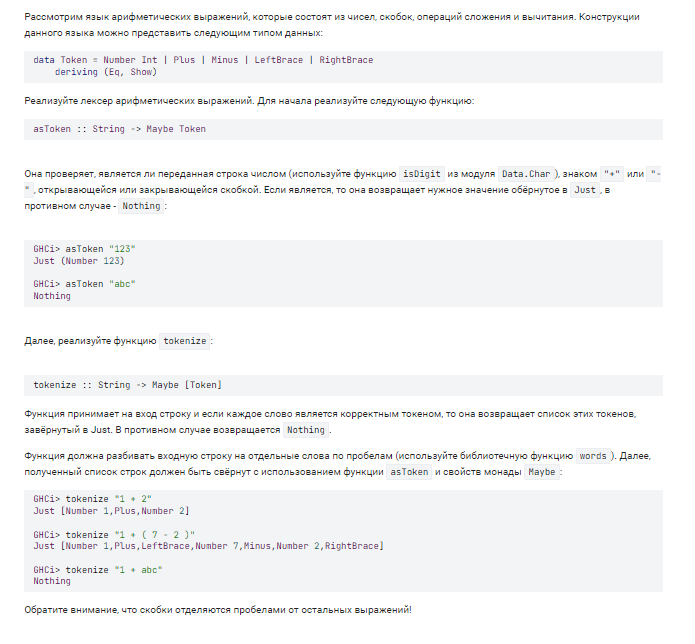


**Решение:**

instance Functor SomeType where

fmap f x = x >>= return . f

**Задание 10:**



**Решение:**

asToken :: String -> Maybe Token

asToken x = case x of

[] -> Nothing

"(" -> Just LeftBrace

")" -> Just RightBrace

"-" -> Just Minus

"+" -> Just Plus

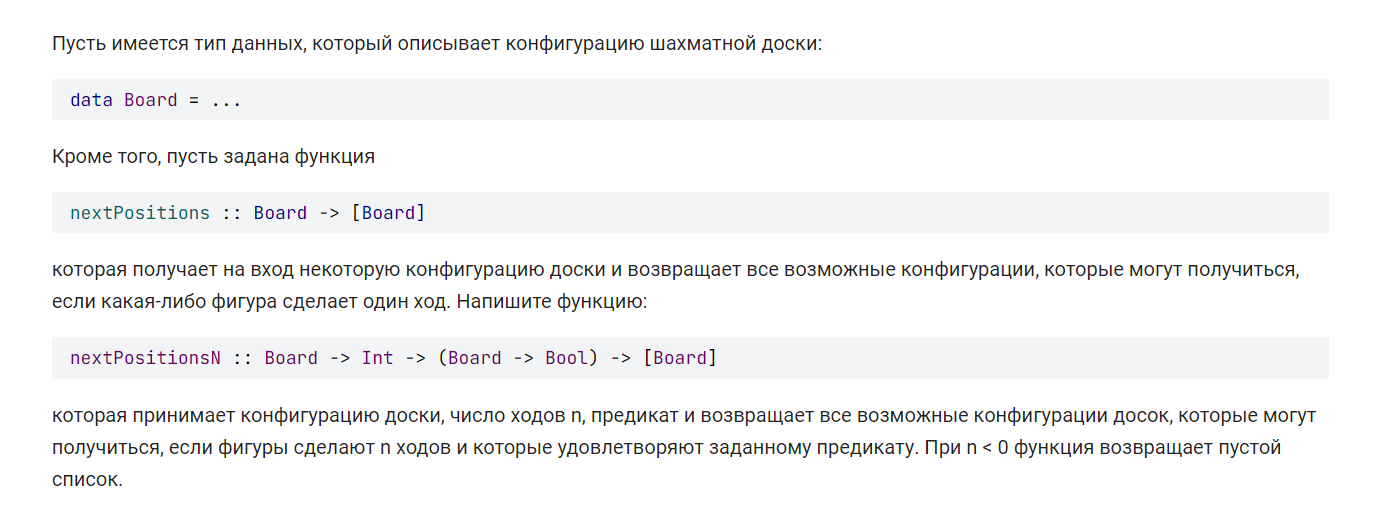
\_ | all isDigit x -> Just $ Number $ read x

| otherwise -> Nothing

tokenize :: String -> Maybe [Token]

tokenize = sequence . (map asToken) . words

**Задание 11:**



**Решение:**

nextPositionsN :: Board -> Int -> (Board -> Bool) -> [Board]

nextPositionsN b n pred

| n < 0 = []

| n == 0 = filter pred [b]

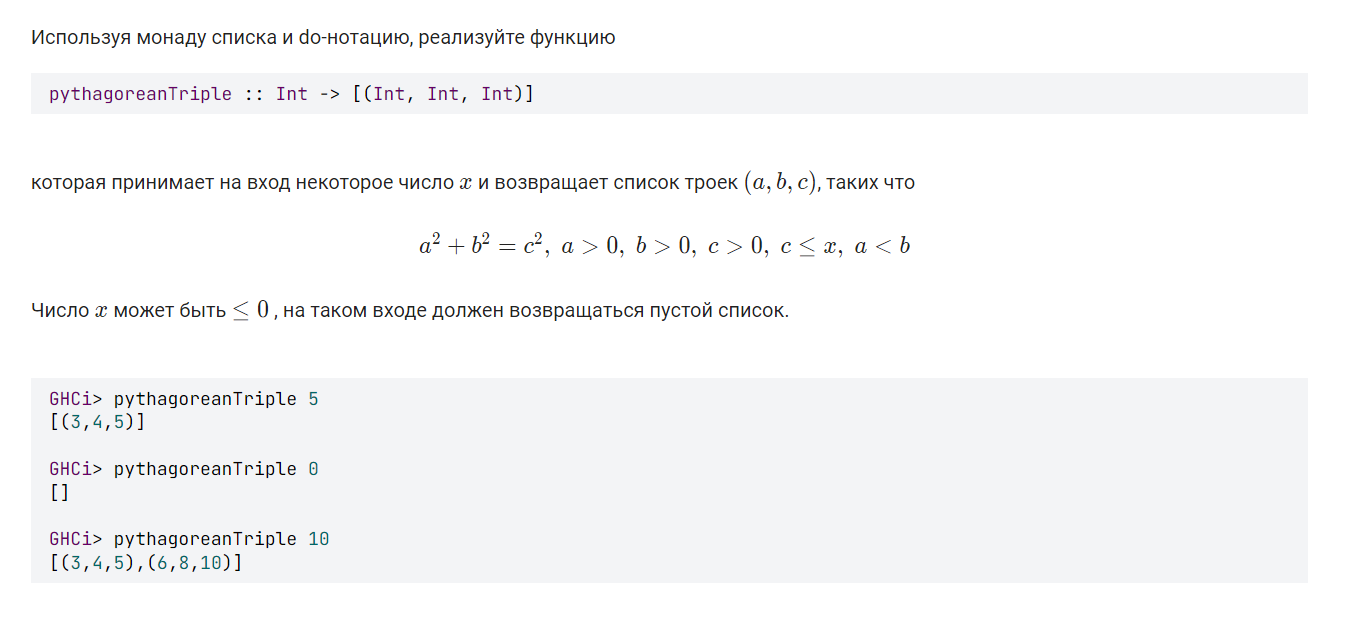
| otherwise = do

move <- nextPositions b

restMoves <- nextPositionsN move (n - 1) pred

return restMoves

**Задание 12:**



**Решение:**

pythagoreanTriple :: Int -> [(Int, Int, Int)]

pythagoreanTriple x

| x <= 0 = []

| otherwise = do

b <- [1..x]

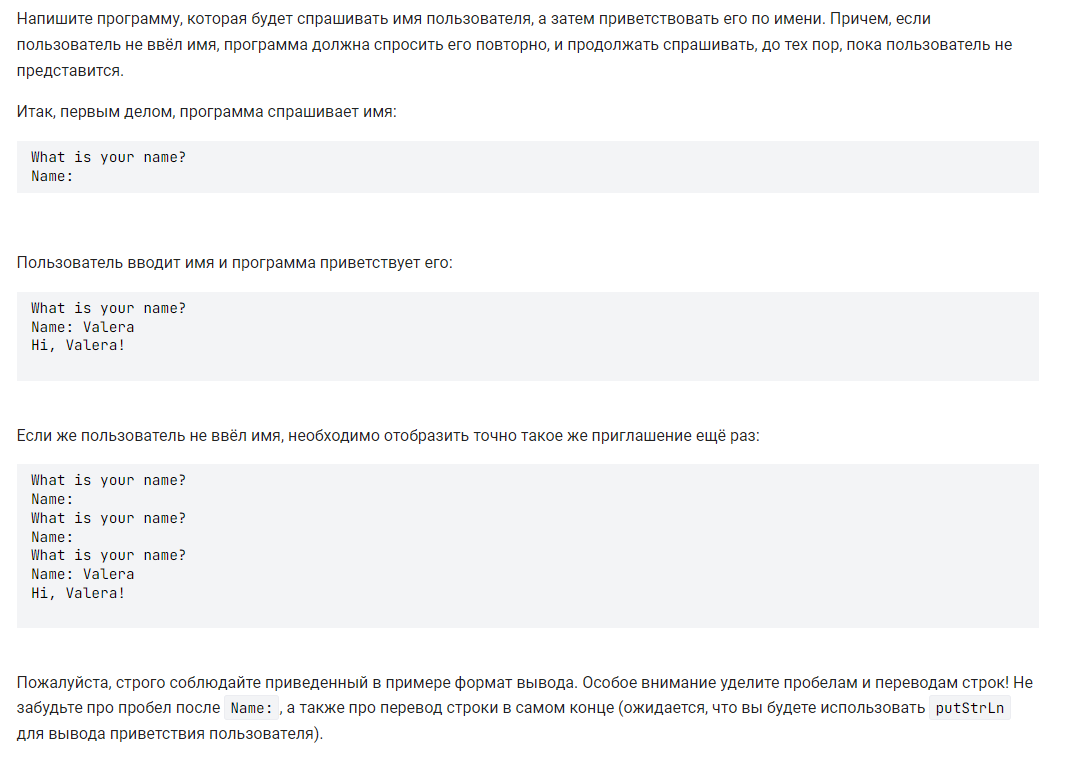
a <- [1..b-1]

c <- [1..x]

True <- return $ (a^2 + b^2) == c^2

return (a,b,c)

**Задание 13:**



**Решение:**

main' :: IO ()

main' = do

putStr $ "What is your name?\nName: "

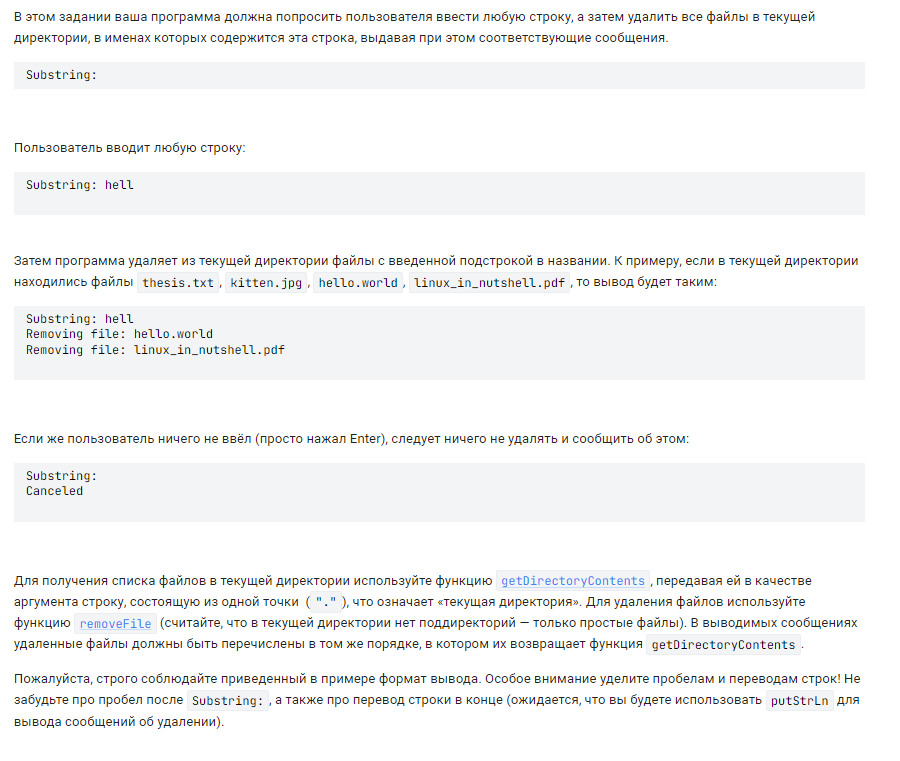
name <- getLine

if null name

then main'

else putStrLn $ "Hi, " ++ name ++ "!"

**Задание 14:**



**Решение:**

main' :: IO ()

main' = do

putStr $ "Substring: "

pattern <- getLine

if null pattern

then putStrLn "Canceled"

else getFiles pattern >>= mapM\_ deleteFile

getFiles :: String -> IO [FilePath]

getFiles pattern =

liftM (filter (L.isInfixOf pattern)) $ getDirectoryContents "."

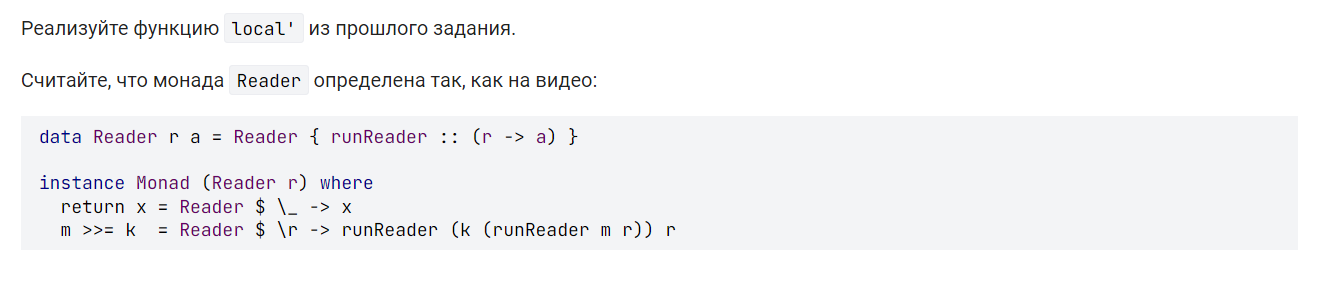
deleteFile :: FilePath -> IO ()

deleteFile path = do

putStrLn $ "Removing file: " ++ path

removeFile path

**Задание 15:**

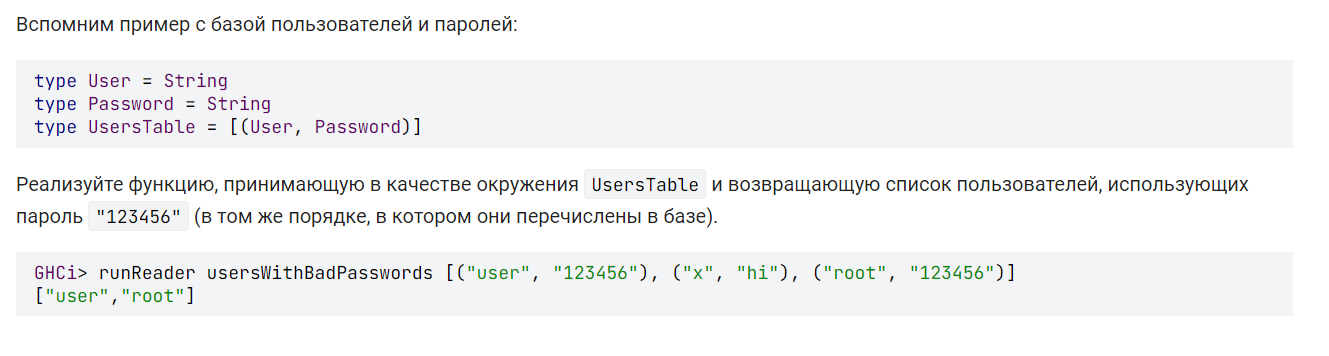


**Решение:**

local' :: (r -> r') -> Reader r' a -> Reader r a

local' f m = Reader $ (runReader m) . f

**Задание 16:**



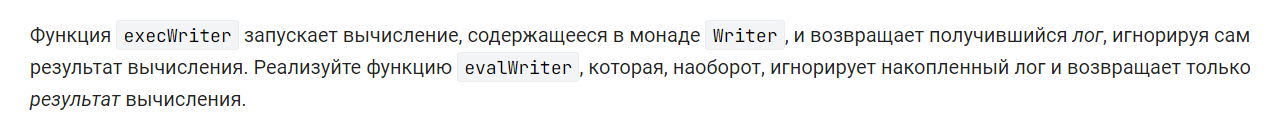
**Решение:**

usersWithBadPasswords :: Reader UsersTable [User]

usersWithBadPasswords = asks $ map fst . filter isBad where

isBad = ("123456" ==) . snd

**Задание 17:**

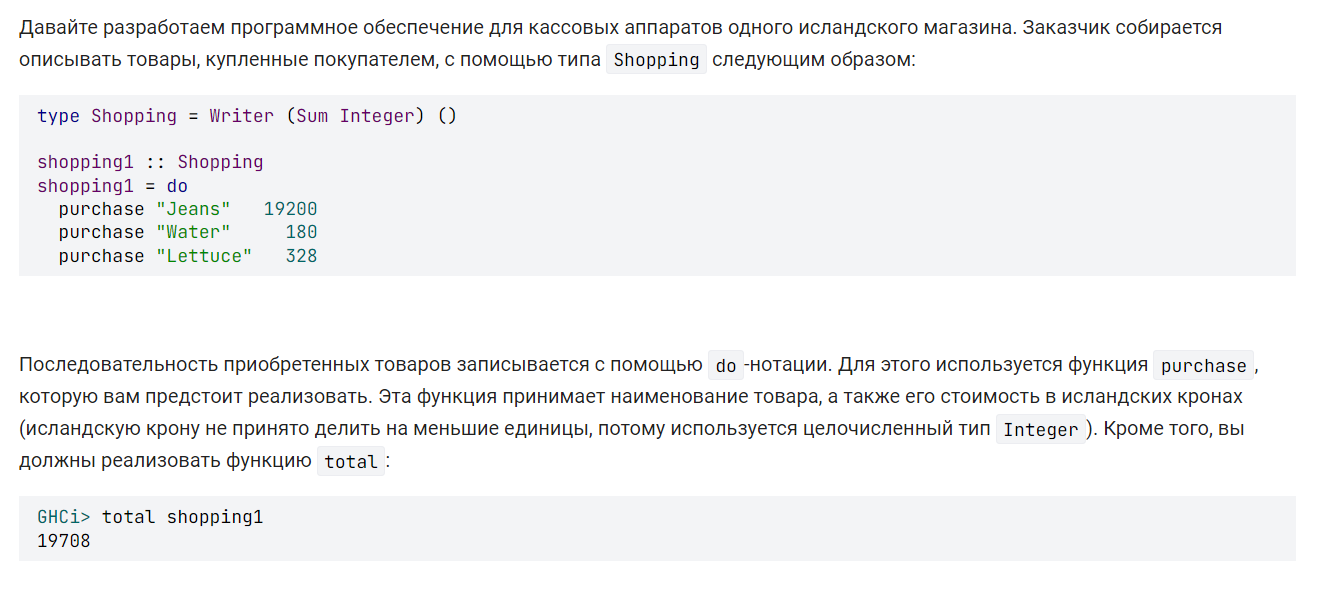


**Решение:**

evalWriter :: Writer w a -> a

evalWriter = fst . runWriter

**Задание 18:**



**Решение:**

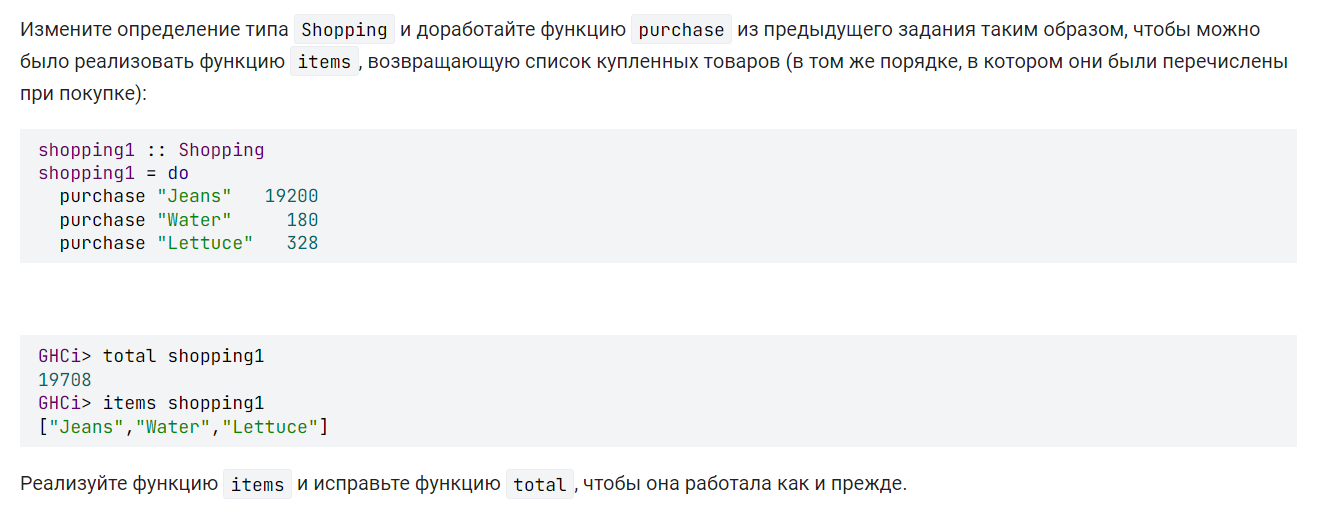
purchase :: String -> Integer -> Shopping

purchase \_ cost = writer ((), Sum cost)

total :: Shopping -> Integer

total = getSum . execWriter

**Задание 19:**



**Решение:**

type Shopping = Writer ([(String, Integer)]) ()

purchase :: String -> Integer -> Shopping

purchase item price = writer ((), [(item, price)])

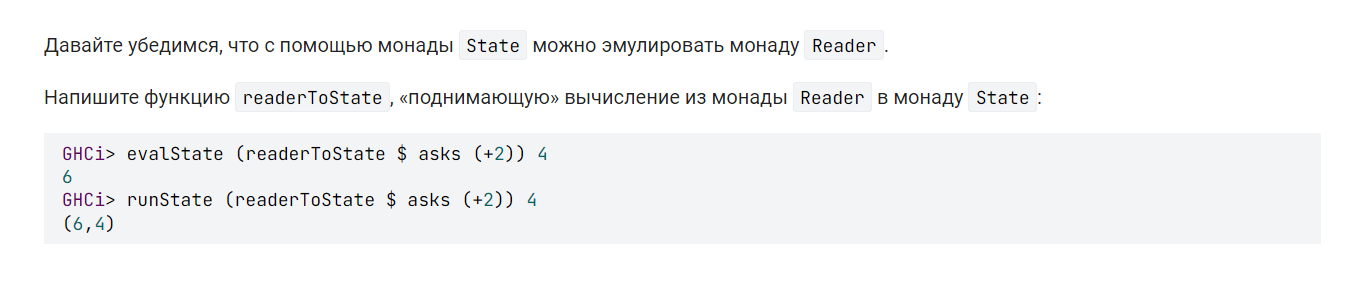
total :: Shopping -> Integer

total = sum . (map snd) . execWriter

items :: Shopping -> [String]

items = (map fst) . execWriter

**Задание 20:**

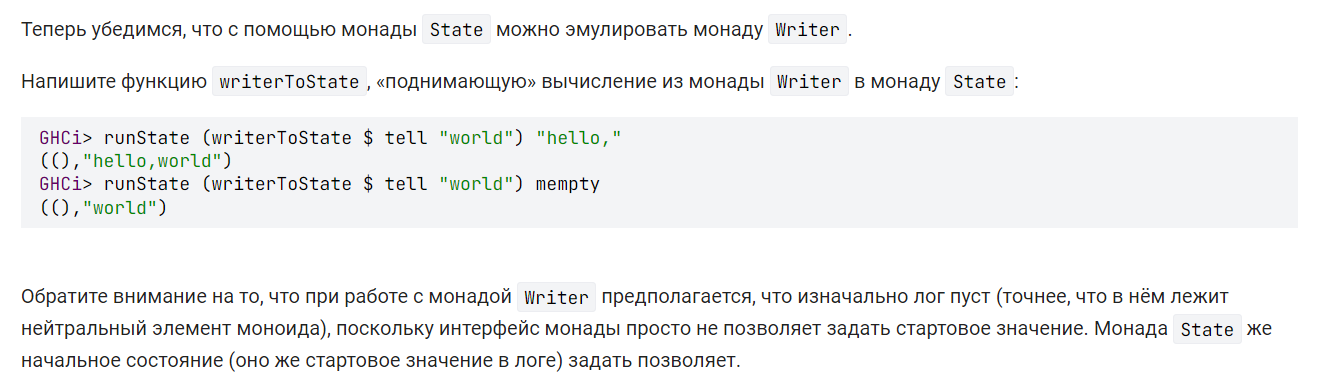


**Решение:**

readerToState :: Reader r a -> State r a

readerToState m = state $ \e -> (runReader m e, e)

**Задание 21:**



**Решение:**

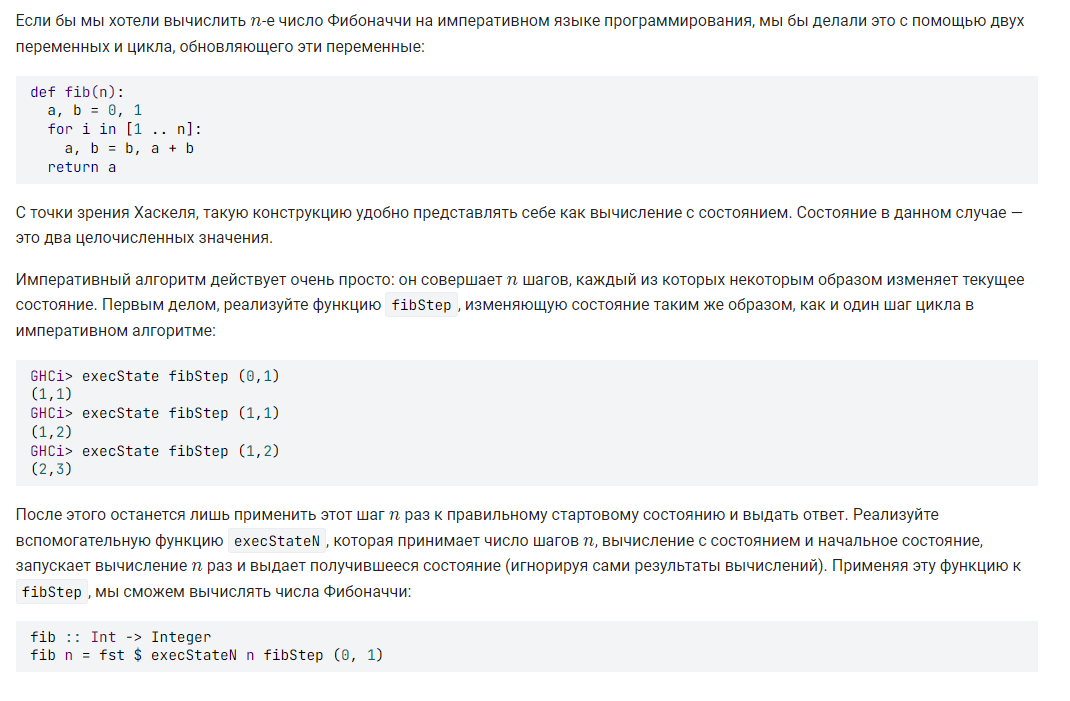
writerToState :: Monoid w => Writer w a -> State w a

writerToState m = let

(a, w) = runWriter m

in state $ \e -> (a, e `mappend` w)

**Задание 22:**



**Решение:**

fibStep :: State (Integer, Integer) ()

fibStep = do

(a, b) <- get

put (b, a + b)

execStateN :: Int -> State s a -> s -> s

execStateN n m = execState (replicateM n m)

**Сертификат:**

