-- love haskell

Парадигмы фп: строгая типизация, функции высшего порядка, чистые функции, неизменяемость, статическая типизация?

Конструкции Haskell: замыкания (связь с ооп?), каррирование (преобразование функции от многих аргументов к функции с одним аргументом), вывод типов, pattern matching, классы типов

Kinds – сколько аргументов надо?

Maybe :: \* -> \*

data MapTree k v

= Leaf

| Node k v (MapTree k v) (MapTree k v)

MapTree :: \* -> \* -> \*

class Monoid a where

mempty :: a

mappend :: a -> a -> a

mconcat :: [a] -> a

class Monad m where -- m :: \* -> \*

return :: a -> m a -- return

(>>=) :: m a -> (a -> m b) -> m b -- bind

class Functor f where

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

class Functor f => Applicative f where

pure :: a -> f a

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

class Foldable t where

{-# MINIMAL foldMap | foldr #-}

fold :: Monoid m => t m -> m

foldMap :: Monoid m => (a -> m) -> t a -> m

foldr :: (a -> b -> b) -> b -> t a -> b -- сначала с первым и рекурсивно дальше

class (Functor t, Foldable t) => Traversable t where

traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

newtype Identity a = Identity { runIdentity :: a }

newtype Const a b = Const { getConst :: a }

data Writer w a = Writer { runWriter :: (a, w) } -- (текущее значение; логи)

instance Monoid w => Monad (Writer w) where

return a = Writer (a, mempty)

Writer (x, v) >>= f = let Writer (y, v') = f x

in Writer (y, v `mappend` v')

data Reader r a = Reader { runReader :: r -> a } -- принимает аргумент и обновляет значение внутри

instance Monad (Reader e) where

return a = Reader $ \\_ -> a

m >>= f = Reader $ \r -> runReader (f $ runReader m r) r

data State s a = State { runState :: s -> (a, s) } -- принимает аргумент, сохраняет значение, хранит логи

instance Monad (State s) where

return a = State $ \s -> (a, s)

oldState >>= f = State $ \s -> let (a, newState) = runState oldState s

in runState (f a) newState

newtype Cont r a = Cont { runCont :: (a -> r) -> r }

instance Monad (Cont r) where

return a = Cont ($ a)

Cont arr >>= f = Cont $ \br -> arr $ \a -> runCont (f a) br

class MonadTrans t where

lift :: (Monad m) => m a -> t m a

class (Monad m) => MonadIO m where

liftIO :: IO a -> m a

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

newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }

newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

instance (Monad m) => Monad (ReaderT r m) where

return = lift . return

m >>= f = ReaderT $ \r -> do

a <- runReaderT m r

runReaderT (f a) r

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

instance Monad m => Monad (MaybeT m) where

return x = MaybeT (return (Just x))

MaybeT action >>= f = MaybeT $ do

result <- action

case result of

Nothing -> return Nothing

Just x -> runMaybeT (f x)

class MonadTrans t where -- t :: (\* -> \*) -> \* -> \*

lift :: Monad m => m a -> t m a

instance MonadTrans MaybeT where

lift = transformToMaybeT

instance MonadTrans ReaderT where

lift m = ReaderT (const m) -- lift ma = ReaderT $ \r -> ma

data ST s a

runST :: (forall s. ST s a) -> a

stToIO :: ST RealWorld a -> IO a

class Monad m => MonadState s m where

get :: m s

put :: s -> m ()

get :: (Monad m) => StateT s m s

get = state $ \ s -> (s, s)

put :: (Monad m) => s -> StateT s m ()

put s = state $ \ \_ -> ((), s)

modify :: (Monad m) => (s -> s) -> StateT s m ()

modify f = state $ \ s -> ((), f s)

class Monad m => MonadCont m where

callCC :: ((a -> m b) -> m a) -> m a -- call-with-current-continuation

instance MonadCont (Cont r) where

callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a

type Lens s t a b = forall f . Functor f => (a -> f b) -> s -> f t

type Lens' s a = Lens s s a a

set :: Lens' s a -> a -> s -> s -- set value (setter)

set ln value thing = runIdentity $ ln (\\_ -> Identity value) thing

view :: Lens' s a -> s -> a -- lookup value (getter)

view ln th = getConst $ ln (\x -> Const x) th

over :: Lens' s a -> (a -> a) -> s -> s -- change value (modifier)

over ln f th = set ln (f (view ln th)) th

data Eval a -- Eval is monad for parallel computation

instance Monad Eval where

runEval :: Eval a -> a -- pull the result out of the monad

rpar :: a -> Eval a -- suggest to parallel, create \*spark\*

rseq :: a -> Eval a -- wait for evaluation of argument (eval it to WHNF)

class Functor w => Comonad w where

extract :: w a -> a

(<<=) :: (w a -> b) -> w a -> w b -- extend

duplicate :: w a -> w (w a)

data Identity a = Identity { runIdentity :: a }

instance Comonad Identity where

extract = runIdentity

duplicate = Identity

class Functor w => Comonad w where

extract :: w a -> a

duplicate :: w a -> w (w a)

extend :: (w a -> b) -> w a -> w b

# Ranks – глубина forall

Rank 0: Int

Rank 1: forall a . a -> Int

Rank 2: (forall a . a -> Int) -> Int -- could be enabled by Rank2Types

Rank 3: ((forall a . a -> Int) -> Int) -> Int

-- love haskell

{-# OPTIONS\_GHC -fno-warn-tabs #-}

{-# LANGUAGE TypeSynonymInstances #-}

{-# LANGUAGE GeneralizedNewtypeDeriving #-}

{-# LANGUAGE DuplicateRecordFields #-}

{-# LANGUAGE ViewPatterns #-}

{-# LANGUAGE FlexibleContexts #-}

{-# LANGUAGE FlexibleInstances #-}

{-# LANGUAGE MultiParamTypeClasses #-}

{-# LANGUAGE UndecidableInstances #-}

module Examples

( Tree (..)

) where

import Prelude hiding (Monad, (>>=), return)

1.

Краткий список удачных конструкций Haskell

• Замыкания (closures) – их трудно полноценно реализовать в языке без сборки мусора типа C++ (upward funarg problem)

• Каррирование (currying) – то же, что для замыканий плюс проблемы с перегрузкой функций по числу аргументов,

как принято во многих императивных языках

• Вывод типов (type inference) – глобальный вывод типов накладывает серьезные ограничения на систему типов;

так, наличие наследования резко ослабляет возможности вывода типов

• Сопоставление с образцом (pattern matching)

• Классы типов (type classes)

# functions

words :: String -> [String] -- сплит по пробелу

read :: Read a => String -> a

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

2.

# list

l = [2, 1, 3]

init :: [a] -> [a] -- возвращает все, кроме последнего

drop 2 l == [3]

take 1 l == [2]

replicate 3 l == [[2,1,3], [2,1,3], [2,1,3]]

zip [1,2,3] "abc" == [(1, 'a'), (2, 'b'), (3, 'c')]

unzip [(5, True), (10, False)] == ([5, 10], [True, False])

zipWith max [10, 5, 1] [2, 1, 3] == [10, 5, 3]

x = l !! i -- x == l[i], O(i) time

# lambda

squares = map (\x -> x \* x) l

# let

z = let l = [2, 1, 3]

h = head l

in h + 10

# where

pythagoras a b = a2 + b2

where

square x = x ^ 2

a2 = square a

b2 = square b

# if

collatzSum n

| n < 0 = 0

| n == 1 = 1

| even n = n + collatzSum (n `div` 2)

| otherwise = n + collatzSum (3 \* n + 1)

# case

collatzSum n

| n < 0 = 0

| n == 1 = 1

| even n = n + collatzSum (n `div` 2)

| otherwise = n + collatzSum (3 \* n + 1)

map :: (a -> b) -> [a] -> [b]

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

foldr1 :: (a -> a -> a) -> [a] -> a

span :: (a -> Bool) -> [a] -> ([a], [a])

uncurry :: (a -> b -> c) -> (a, b) -> c

uncurry f (x, y) = f x y

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

flip f b a = f a b

(.) :: (b -> c) -> (a -> b) -> a -> c -- function composition

f . g = \x -> f (g x)

($) :: (a -> b) -> a -> b -- function application

f $ x = f x

f x y == (f x) y

f :: a -> b -> c == f :: (a -> (b -> c))

f. g . h $ j x == (f . (g . h)) $ (j x)

3.

# types

data Color = Red | Green | Blue

type String = [Char]

data User = User Int String String -- есть конструктор

newtype Message = Message String -- нет конструктора

data Vector a = Vector2D a a | Vector3D a a a

data Maybe a = Nothing | Just a -- implemented in Prelude

data Either a b = Left a | Right b

deriving (Eq, Ord, Read, Show)

eitherSecond :: [a] -> Either String a

eitherSecond (\_:x:\_) = Right x

eitherSecond \_ = Left "list doesn't have second element"

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

data User = User

{ uid :: Int

, login :: String

, password :: String

}

# kinds

Int :: \*

Maybe :: \* -> \*

Maybe String :: \*

data MapTree k v

= Leaf

| Node k v (MapTree k v) (MapTree k v)

MapTree :: \* -> \* -> \*

MapTree a :: \* -> \*

MapTree String v :: \*

data [] a = [] | a : [a] -- Defined in ‘GHC.Types’

[] :: \* -> \* -- kind of list

[] Int :: \* -- `[] a` is the same as [a]

(->) :: \* -> \* -> \* -- kind of function

foldr :: (a -> b -> b) -> b -> [a] -> b -- сначала с первым и рекурсивно дальше

foldl :: (b -> a -> b) -> b -> [a] -> b -- сначала с последним и возвращает результат

(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c

(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c

liftM :: Monad m => (a -> b) -> m a -> m b

liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c

# classes

class Eq a where

(==) :: a -> a -> Bool

(/=) :: a -> a -> Bool

class Monoid a where

mempty :: a

mappend :: a -> a -> a

mconcat :: [a] -> a

class Monad m where -- m :: \* -> \*

return :: a -> m a -- return

(>>=) :: m a -> (a -> m b) -> m b -- bind

(>>) :: m a -> m b -> m b -- then\*

instance Monad Maybe where

return = Just

Nothing >>= \_ = Nothing

Just a >>= f = f a

6.

class Functor f where

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

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

instance Functor Maybe where

fmap f (Just x) = Just (f x)

fmap \_ Nothing = Nothing

# Functor laws

1. fmap id = id

2. fmap (f . g) = fmap f . fmap g

fmap (f . g) F = fmap f (fmap g F)

class Functor f => Applicative f where

pure :: a -> f a

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

instance Applicative Maybe where

pure = Just

Nothing <\*> \_ = Nothing

Just f <\*> something = fmap f something

# Applicative laws

1. identity

pure id <\*> v = v

2. composition

pure (.) <\*> u <\*> v <\*> w = u <\*> (v <\*> w)

3. homomorphism

pure f <\*> pure x = pure (f x)

4. interchange

u <\*> pure y = pure ($ y) <\*> u

class Foldable t where

{-# MINIMAL foldMap | foldr #-}

fold :: Monoid m => t m -> m

foldMap :: Monoid m => (a -> m) -> t a -> m

foldr :: (a -> b -> b) -> b -> t a -> b -- сначала с первым и рекурсивно дальше

class Applicative f => Alternative f where

empty :: f a

(<|>) :: f a -> f a -> f a

instance Alternative Maybe where

empty = Nothing

Nothing <|> r = r

l <|> \_ = l

class (Functor t, Foldable t) => Traversable t where

traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

instance Traversable Maybe where

traverse \_ Nothing = pure Nothing

traverse f (Just x) = Just <$> (f x)

5.

newtype Identity a = Identity { runIdentity :: a }

deriving (Eq, Ord)

newtype Const a b = Const { getConst :: a }

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

7.

data Writer w a = Writer { runWriter :: (a, w) } -- (текущее значение; логи)

instance Monoid w => Monad (Writer w) where

return a = Writer (a, mempty)

Writer (x, v) >>= f = let Writer (y, v') = f x

in Writer (y, v `mappend` v')

data Reader r a = Reader { runReader :: r -> a } -- принимает аргумент и обновляет значение внутри

ask :: Reader e e

asks :: (e -> a) -> Reader e a

local :: (e -> b) -> Reader b a -> Reader e a

instance Monad (Reader e) where

return a = Reader $ \\_ -> a

m >>= f = Reader $ \r -> runReader (f $ runReader m r) r

data State s a = State { runState :: s -> (a, s) } -- принимает аргумент, сохраняет значение, хранит логи

instance Monad (State s) where

return a = State $ \s -> (a, s)

oldState >>= f = State $ \s -> let (a, newState) = runState oldState s

in runState (f a) newState

# пример state - stack

type Stack = [Int]

pop :: State Stack Int

pop = state $ \(x:xs) -> (x, xs)

push :: Int -> State Stack ()

push x = state $ \xs -> ((), x:xs)

stackOps :: State Stack Int

stackOps = pop >>= \x -> push 5 >> push 10 >> return x

newtype Cont r a = Cont { runCont :: (a -> r) -> r }

cont :: ((a -> r) -> r) -> Cont r a

instance Monad (Cont r) where

return a = Cont ($ a)

Cont arr >>= f = Cont $ \br -> arr $ \a -> runCont (f a) br

class MonadTrans t where

lift :: (Monad m) => m a -> t m a

class (Monad m) => MonadIO m where

liftIO :: IO a -> m a

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

newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }

newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

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

-- ST monad

data ST s a

runST :: (forall s. ST s a) -> a

stToIO :: ST RealWorld a -> IO a

class Monad m => MonadState s m where

get :: m s

put :: s -> m ()

class Monad m => MonadState s m | m -> s where

get :: m s

get = state (\s -> (s, s))

put :: s -> m ()

put s = state (\\_ -> ((), s))

state :: (s -> (a, s)) -> m a

state f = do

s <- get

let ~(a, s') = f s

put s'

return a

get :: (Monad m) => StateT s m s

get = state $ \ s -> (s, s)

put :: (Monad m) => s -> StateT s m ()

put s = state $ \ \_ -> ((), s)

modify :: (Monad m) => (s -> s) -> StateT s m ()

modify f = state $ \ s -> ((), f s)

gets :: (Monad m) => (s -> a) -> StateT s m a

gets f = state $ \ s -> (f s, s)

class MonadFish m where

returnFish :: a -> m a

(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)

class MonadJoin m where

returnJoin :: a -> m a

join :: m (m a) -> m a

class Monad m => MonadCont m where

callCC :: ((a -> m b) -> m a) -> m a -- call-with-current-continuation

instance MonadCont (Cont r) where

callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a

callCC f = cont $ \c -> runCont (f (\x -> cont $ \\_ -> c x)) c

8. IO

{-# LANGUAGE MagicHash #-} # для оптимизации ???

newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

# do notation, >>

(>>) :: IO a -> IO b -> IO b

(action1 >> action2) world0 =

let (\_, world1) = action1 world0

(b, world2) = action2 world1

in (b, world2)

putStrLn :: String -> IO ()

main = do putStrLn "What is your name?"

putStrLn "How old are you?"

putStrLn "Nice day!"

main = (putStrLn "What is your name?") >>

(putStrLn "How old are you?") >>

(putStrLn "Nice day!")

# do, <-

(>>=) :: IO a -> (a -> RealWorld -> (b, RealWorld)) -> IO b

(action1 >>= action2) world0 =

let (a, world1) = action1 world0

(b, world2) = action2 a world1

in (b, world2)

getLine :: IO String

main = do s <- getLine

putStrLn s

main = getLine >>= \s -> putStrLn s

# example!

main = do putStr "What is your name?"

a <- readLn

putStr "How old are you?"

b <- readLn

print (a,b)

==

main = putStr "What is your name?"

>> readLn

>>= \a -> putStr "How old are you?"

>> readLn

>>= \b -> print (a,b)

# изменяемые данные IORef

import Data.Array.IO

main = do arr <- newArray (1,10) 37 :: IO (IOArray Int Int)

a <- readArray arr 1

writeArray arr 1 64

b <- readArray arr 1

print (a, b)

# exceptions

throwIO :: Exception e => e -> IO a

catch :: Exception e => IO a -> (e -> IO a) -> IO a

--- что-то неинтересное???

# text

import qualified Data.Text as T

-- From pack

myTStr1 :: T.Text

myTStr1 = T.pack ("foo" :: String)

# ByteString

{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString as S

import qualified Data.ByteString.Char8 as S8

-- From pack

bstr1 :: S.ByteString

bstr1 = S.pack ("foo" :: String)

-- From overloaded string literal.

bstr2 :: S8.ByteString

bstr2 = "bar"

# из дз interactive

type MMap = StateT (M.Map T.Text T.Text) IO

update :: T.Text -> MMap ()

update a = do

myMap <- get

liftIO $ putStrLn $ "input new value for '" ++ T.unpack a ++ "' property (previous: '" ++ show (M.lookup a myMap) ++ "')"

b <- liftIO $ getLine

modify (M.insert a (T.pack b))

liftIO $ putStrLn $ "property " ++ T.unpack a ++ " with value " ++ b ++ " updated"

config

9.

Reader has ask but doesn't have put

State has put but doesn't have ask

# Transformers!

foo :: ReaderT Int (State [Int]) Int -- or StateT [Int] (Reader Int) Int

foo i = do

baseCounter <- ask

let newCounter = baseCounter + i

put [baseCounter, newCounter]

return newCounter

newtype MaybeIO a = MaybeIO {

runMaybeIO :: IO (Maybe a)

}

instance Monad MaybeIO where

return x = MaybeIO (return (Just x))

MaybeIO action >>= f = MaybeIO $ do

result <- action

case result of

Nothing -> return Nothing

Just x -> runMaybeIO (f x)

# MaybeT Transformer

newtype MaybeT m a = MaybeT {

runMaybeT :: m (Maybe a)

}

instance Monad m => Monad (MaybeT m) where

return x = MaybeT (return (Just x))

MaybeT action >>= f = MaybeT $ do

result <- action

case result of

Nothing -> return Nothing

Just x -> runMaybeT (f x)

transformToMaybeT :: Monad m => m a -> MaybeT m a

transformToMaybeT action = MaybeT $ do

result <- action

return (Just result)

transformToMaybeT :: Monad m => m a -> MaybeT m a

transformToEitherT :: Monad m => m a -> EitherT l m a

class MonadTrans t where -- t :: (\* -> \*) -> \* -> \*

lift :: Monad m => m a -> t m a

instance MonadTrans MaybeT where

lift = transformToMaybeT

newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

instance (Monad m) => Monad (ReaderT r m) where

return = lift . return

m >>= f = ReaderT $ \r -> do

a <- runReaderT m r

runReaderT (f a) r

instance MonadTrans ReaderT where

lift m = ReaderT (const m) -- lift ma = ReaderT $ \r -> ma

class (Monad m) => MonadIO m where

liftIO :: IO a -> m a

instance MonadIO IO where

liftIO = id

instance MonadIO m => MonadIO (StateT s m) where

liftIO = lift . liftIO

instance MonadIO m => MonadIO (ReaderT r m) where

liftIO = lift . liftIO

-- | The CoroutineT monad is just ContT stacked with

-- a StateT containing the suspended coroutines.

newtype CoroutineT r m a = CoroutineT

{ runCoroutineT' :: ContT r (StateT [CoroutineT r m ()] m) a

} deriving (Functor, Applicative, Monad, MonadCont, MonadIO)

10. speed

import Data.List (foldl')

main :: IO ()

main = print $ foldl' (+) 0 [1..10^7] -- 10^8 ok

ламбда-выражение:

\* в нормальной форме = нет бета-редукций (применения или подстановки)

\* в головной нормальной форме = применение без бета-редукции или лямбда с телом в головной нф

\* слабая головная н.ф. = головная или абстракция??

-- ну такое, про скорость и изменяемые массивы...

11. template haskell

-- давайте запилим функции, которые исполняются на стадии компиляции

{-# LANGUAGE TemplateHaskell #-}

module FstN where

import Language.Haskell.TH

data Q a

fstN :: Int -> Q Exp

fstN n = do

x <- newName "x"

return $ LamE [TupP $ VarP x : replicate (n - 1) WildP] (VarE x)

ghci> runQ [| \(x,\_,\_) -> x |]

LamE [TupP [VarP x\_1,WildP,WildP]] (VarE x\_1)

# TH quotes

Expression quotes

[| \x -> x + 1 |] :: Q Exp

Type quotes

[t| Int -> Int |] :: Q Type

Pattern quotes

[p| xs@(x:r) |] :: Q Pat

Declaration quotes

[d| data Pair a = Pait a a |] :: Q [Dec]

# Lens -- давайте сделаем нормальные конструкторы

-- вот это больно

setHouse person value =

person { address = (address person) { house = value } }

setStreet person value =

person { address = (address person) { street = value } }

newPerson1 = setHouse person 45

newPerson2 = setStreet person "New Street"

-- решение

data Lens' obj field = Lens'

{ view' :: obj -> field

, update' :: (field -> field) -> obj -> obj

}

set' :: Lens' obj field -> field -> obj -> obj

set' ln newValue thing = (update' ln) (\\_ -> newValue) thing

# examples

addressLens :: Lens' Person Address

addressLens = Lens' address (\fn thing -> thing { address = fn (address thing) })

cityLens :: Lens' Address String

cityLens = Lens' city (\fn thing -> thing { city = fn (city thing) })

type Lens s t a b = forall f . Functor f => (a -> f b) -> s -> f t

type Lens' s a = Lens s s a a

set :: Lens' s a -> a -> s -> s -- set value (setter)

set ln value thing = runIdentity $ ln (\\_ -> Identity value) thing

view :: Lens' s a -> s -> a -- lookup value (getter)

view ln th = getConst $ ln (\x -> Const x) th

over :: Lens' s a -> (a -> a) -> s -> s -- change value (modifier)

over ln f th = set ln (f (view ln th)) th

# real Lens

import Control.Lens

type Lens' a b = forall f . (Functor f) => (b -> f b) -> (a -> f a)

makeLenses ''Game

# lens laws

-- Get-Put

>>> forall $ \whole -> set l (view l whole) whole == whole

True

-- Put-Get

>>> forall $ \whole part -> view l (set l part whole) == part

True

-- Put-Put

>>> forall $ \whole part1 part2 ->

set l part2 (set l part1) whole = set l part2 whole

# prisms -- что это??

preview :: Prism' s a -> s -> Maybe a

review :: Prism' s a -> a -> s

\_Left :: Prism' (Either a b) a

\_Just :: Prism' (Maybe a) a

ghci> preview \_Left (Left "hi")

Just "hi"

ghci> preview \_Left (Right "hi")

Nothing

ghci> review \_Left "hi"

Left "hi"

ghci> preview \_Just (Just "hi")

Just "hi"

ghci> review \_Just "hi"

Just "hi"

ghci> Left "hi" ^? \_Left

Just "hi"

12. parallelism

Haskell parallelism advantages:

1. Pure чистый

2. Deterministic статический (определенный?)

3. Flexible гибкий

data Eval a -- Eval is monad for parallel computation

instance Monad Eval where

runEval :: Eval a -> a -- pull the result out of the monad

rpar :: a -> Eval a -- suggest to parallel, create \*spark\*

rseq :: a -> Eval a -- wait for evaluation of argument (eval it to WHNF)

-- сравнение rpar rseq

-- стратегии

-- Par Monad

13. comonads

class Functor w => Comonad w where

extract :: w a -> a

(<<=) :: (w a -> b) -> w a -> w b -- extend

duplicate :: w a -> w (w a)

-- example

data Identity a = Identity { runIdentity :: a }

instance Comonad Identity where

extract = runIdentity

duplicate = Identity

We can emulate some OOP patterns

1. Initial state

2. Builder pattern

3. Iterator pattern (or infinite streams)

4. Command pattern

-- пропуск!!

# Traced Comonad

newtype Traced m a = Traced { runTraced :: m -> a }

instance Monoid m => Comonad (Traced m) where

extract (Traced ma) = ma mempty

extend f (Traced ma) = Traced $ \m -> f (Traced $ \m' -> ma (m <> m'))

type ConfigBuilder = Traced [Option] Config

profile :: ConfigBuilder -> Config

profile builder = runTraced builder ["-prof", "-auto-all"]

goFaster :: ConfigBuilder -> Config

goFaster builder = runTraced builder ["-O2"]

# Store comonad

data Store s a = Store (s -> a) s

class Functor w => Comonad w where

extract :: w a -> a

duplicate :: w a -> w (w a)

duplicate = extend id

extend :: (w a -> b) -> w a -> w b

extend f = fmap f . duplicate

#if \_\_GLASGOW\_HASKELL\_\_ >= 708

{-# MINIMAL extract, (duplicate | extend) #-}

#endif

instance Comonad (Store s) where

extract (Store f s) = f s

extend f (Store g s) = Store (f . Store g) s

type Lens s a = a -> Store s a

# Comonad transformers

class ComonadTrans t where

lower :: Comonad w => t w a -> w a

15.

applyTwo :: forall b c . (forall a . [a] -> [a]) -> [b] -> [c] -> ([b], [c])

applyTwo f x y = (f x, f y)

ghci> applyTwo id [2,1,3] [True,False]

([2,1,3],[True,False])

ghci> applyTwo reverse [True,False] "patak"

([False,True],"katap")

# Ranks

Rank 0: Int

Rank 1: forall a . a -> Int

Rank 2: (forall a . a -> Int) -> Int -- could be enabled by Rank2Types

Rank 3: ((forall a . a -> Int) -> Int) -> Int

The rank of a type describes the depth at which universal quantifiers appear

in a contravariant position, i.e. to the left of a function arrow.

A function type has rank n + 1 when its argument has rank n.

Int -> Int -- rank 0

forall a . a -> a -- rank 1

(forall a . a -> a) -> Int -- rank 2

forall a . Int -> a -> a -- rank 1

forall a b . a -> b -> a -- rank 1

forall a b . a -> b -> a -- rank 1

(a -> a) -> (forall b . b -> b) -> (c -> c) -- rank 2

-- лееньь

16. Idris

-- Unary multiplication

mult : Nat -> Nat -> Nat

mult Z y = Z

mult (S k) y = plus y (mult k y)

-- sum

sum : (single : Bool) -> isSingleton single -> Nat

sum True x = x

sum False [] = 0

sum False (x :: xs) = x + sum False xs

-- reverse

reverse : List a -> List a

reverse xs = revAcc [] xs where

revAcc : List a -> List a -> List a

revAcc acc [] = acc

revAcc acc (x :: xs) = revAcc (x :: acc) xs

Вариант 3

**1**.Укажите преимущества чистоты и иммутабельности.

* Проще реализовать многопоточность.
* Проще доказать, что код работает, так как каждая функция не взаимодействует с внешним миром.
* Чистая функция всегда возвращает одинаковый результат, если вызвана от одинакового набора аргументов. Например map для списка можно выполнять параллельно.
* Можно менять порядок вычислений аргументов

**2.** Задан тип данных data Role a = A { name :: String, role :: a} | B { name :: String, roles :: [a] }. Напишите конструкцию синтаксичеcким сахаром для которой является данный Record Syntax.

    data Role a = A String a | B String [a]

    name :: Role a -> String

    name (A name \_) = name

    role :: Role a -> a

    role (A \_ role) = role

    …

**3.** В чём разница между newtype и data?

        newtype - это лишь обертка, которая соптимизируется

        Data - не будет оптимизироваться, лол

        Ну и ньютайп ток над одним типом можно хуячить

data User = User Int String String -- есть конструктор

newtype Message = Message String -- нет конструктора

**4.** Напишите несколько функций, полезных при работе с монадами (7 >= хорошо)

halt(0)

    liftM2, liftM3, liftM4, liftM5, liftM6, liftM7

    return, >>=, join, maybe, either, >>, <<=

    >=>, <=< ?

**5.** Напишите type class Functor и его реализацию для  ((->) r)

    class Functor f where  
       fmap :: (a -> b) -> f a -> f b

instance [Functor](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#Functor) ((->) [r](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#local-1627396341)) where  
  [fmap](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#fmap) = ([.](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#.))

   fmap f ((->) r x) = (->) r (f x)  ???

**6.** Напишите type class Applicative и его реализацию Maybe.

class Functor f => Applicative f where

   (<\*>) :: f (a -> b) -> f a -> f b

   pure :: a -> f a

instance Applicative Maybe where

   Nothing <\*> \_ = Nothing

   \_ <\*> Nothing = Nothing

   (Just f) <\*> (Just x) = Just (f x)

   pure = Just

    7. Напишите реализацию Foldable для списка.

class Foldable t where

 {-# MINIMAL foldMap | foldr #-}

    fold    :: Monoid m => t m -> m

    foldMap :: Monoid m => (a -> m) -> t a -> m

    foldr :: (a -> b -> b) -> b -> t a -> b

instance Foldable [] where

   foldr \_ zero [] =  zero

   foldr f zero (x:xs) = f x (foldr f zero xs)

    8. Реализуйте traverse через sequence.

        traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

        sequence :: Monad m => t (m a) -> m (t a)

        traverse f = sequence . fmap f

    9. Напишите определение типа данных Writer и его инстанс Monad.

        newtype Writer w a = Writer { runWriter :: (a, w) }

              instance (Monoid m) => Monad (Writer m) where

            return = Writer (,mempty)

                              Writer (a, w) >>= f = let (res, log) = runWriter $ f a in

                        Writer (res, w <> log)

    10. Напишите теоретический инстанс монады для IO

        data IO a :: \* -> \*

        instance Monad IO  where  
              (>>=)     = bindIO

    11. Нарисуйте табличку отличий обычных типов и их трансформеров для Writer, Reader, State

    Writer - (a, w) - m (a, w)

    Reader - (e -> a) - (e -> m a)

    State - (s -> (a, s)) -> (s -> m (a, s))

    12. Что такое STRef и в чём отличие от IORef?

        И то, и другое хранит ссылку на изменяемую переменную, но IORef позволяет сделать больше, например IO действия. Прикол в том, что из ST монады, в отличие IO можно “выходить”, это позволяет писать функции, использующие мутабельное дерьмо, но имеющие чистый интерфейс.

13. В чём разница между [||] и $()?

        $() -- вклейка, абстрактное синтаксическое дерево подставляется на место шаблона

            main = putStrLn $ "Localtime: " ++ $(localtimeTemplate)

        [||] --цитирующие скобки, получают абстрактное синтаксическое дерево для указанного кода

            [| "string" |] :: Q Exp

14. Напишите определение типов Lens s t a b и Lens’ s a.

type Lens s t a b = forall f . Functor f => (a -> f b) -> s -> f t

type Lens' s a spsr= Lens s s a a

15. Что такое spark?

    Такая штука, которая умеет создавать дешёвые задачи, класть их в очередь. Когда  ядро освобождается оно выполняет очередную задачу.  Это не потоки.

    Программа распределённой обработкислабоструктурированных данных?

16. Что такое nix и чем он отличается от остальных build tools (в рамках Haskell)?

    Nix - это пакетный менеджер. Он может быть использован для каких угодно пакетов.

    Применительно к Haskell он хорош тем, что stack умеет с ним интегрироваться. Это

    позволяет добавлять к проекту какие-то не-хаскель пакеты (например, системные библиотеки) в качестве зависимостей. Я думаю, что самый ключевой элемент все-таки -- это haskellNG и автоинтеграция с hackage.

17. Напишите функцию, которая принимает функцию, два произвольных списка и возвращает пару из списков, применяя функцию к переданным двум.

    fun :: ([a] -> [b]) -> [a] -> [a] -> ([b], [b])

    fun f x y = (map f x, map f y) кудах а как же бифункторы!!

18. Напишите тип callCC.

callCC :: ((a -> m b) -> m a) -> m a

19. Напишите instance Comonad для (,) e, реализовав функцию duplicate

class Functor w => Comonad w where

 duplicate :: w a -> w (w a)

instance Comonad ((,)e) where  
 duplicate p = (fst p, p)

20. Напишите при помощи Arrows функцию, которая принимает число и возвращает пару из числа умноженного на 2 и поделённого целочисленно на 2.

        Не было в программе?!

    21. Напишите тип Free и укажите его kind.

    22. Какие различные виды family бывают?

    23. Что такое Functional Dependencies? Назовите какой-нибудь известный вам type class, в котором присутствуют функциональные зависимости.

ШОК СПИСОК РАСПИДОРАСИЛО

1. Перечислите основные парадигмы ФП и Haskell, отличительные особенности (>=5 - хорошо)

Ленивость, иммутабельность, чистота, строгая типизация, функции высшего порядка, никому не нужность(это не точно)

Удачные конструкции: замыкания, каррирование, вывод типов, классы типов, pattern matching

1. Напишите тип следующей функции в наиболее общем виде:

f a = map (\* a) . map (uncurry (+)) . map (\x -> (x, x))

        f a = map (\* a) . (uncurry (+)) .(\x -> (x, x))

        f a = map \y -> (\* a) (uncurry (+) ((\x -> (x, x)) y)

        f a = map (\y -> a \* (y + y))

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

1. Напишите, что такое DatatypeContexts? Приведите пример (не из презентации).

        Когда ты хочешь наложить ограничения на тип в определении type class

        Deprecated в хаскеле начиная с 7 версии

Ex: data Functor f => Data f  = Data (f Int)

1. Напишите реализацию Monad для Maybe

        return = Just

        (Just a) >>= f = Just $ f a

Nothing >>= \_ = Nothing

1. Напишите type class Functor и его реализацию для Maybe

class Functor f where  
 fmap :: (a -> b) -> f a -> f b

fmap f (Just a)  = Just $ f a

fmap \_ Nothing = Nothing

1. Какие две реализации Applicative есть для списка и чем они отличаются?

    Можно сделать декартово произведение [a->b] \* [a] (реализовано в ZipList)

    Можно применить почленно fs <\*> xs = [f x | f <- fs, x <- xs]

1. Напишите реализацию Foldable для Maybe

instance Foldable Maybe where

   foldr f zero Nothing = zero

   foldr f zero (Just x) = f x zero

1. Напишите type class Traversable

        class ([Functor](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#Functor) [t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175), [Foldable](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Foldable.html#Foldable) [t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175)) => [Traversable](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#Traversable) [t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) where  
                                  traverse :: [Applicative](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#Applicative) [f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654176) => ([a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654177) -> [f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654176) [b](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654178)) -> [t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) [a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654177) -> [f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654176) ([t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) [b](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654178))  
                                  [traverse](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#traverse) [f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654186) = [sequenceA](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#sequenceA) [.](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#.) [fmap](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#fmap) [f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654186)                                  sequenceA :: [Applicative](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#Applicative) [f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654179) => [t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) ([f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654179) [a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654180)) -> [f](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654179) ([t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) [a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654180))  
                                 [sequenceA](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#sequenceA) = [traverse](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#traverse) [id](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#id)                                  mapM :: [Monad](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#Monad) [m](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654181) => ([a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654182) -> [m](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654181) [b](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654183)) -> [t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) [a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654182) -> [m](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654181) ([t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) [b](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654183))  
                                 [mapM](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#mapM) = [traverse](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#traverse)                                  sequence :: [Monad](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#Monad) [m](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654184) => [t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) ([m](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654184) [a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654185)) -> [m](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654184) ([t](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654175) [a](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#local-1627654185))  
                                 [sequence](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#sequence) = [sequenceA](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Traversable.html#sequenceA)

1. Напишите определение типа данных State и его инстанс Monad.

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

**instance** **Monad** (**State** **s**) **where**  
       return a       = **State** $ \s -> (a, s)  
       oldState >>= f = **State** $ \s -> **let** (a, newState) = runState oldState s  
                                                         **in** runState (f a) newState

1. Почему действия с “внешним миром” требуют монаду?

        Требуется зафиксировать порядок вычисления -- ??

Внешний мир меняется, и функции с одинаковыми параметрами могут вернуть   разные значения ??

1. Задан тип ReaderT Int (MaybeT (Reader Double)) String. Напишите тип содержимого, раскрыв обёртки по их конструкторам

        newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

        data Reader r a = Reader {  runReader :: r -> a }

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

        Принимает Int, хранит Reader (Double -> Maybe String) -- не знаю

1. Что такое Deforestation?

            Переподвешивание узлов в дереве для быстрого доступа к ним??

[Short cut fusion](https://wiki.haskell.org/Short_cut_fusion) on list structures

**Short cut fusion** is an optimizer method that merges some function calls into one.

E.g.

**map** f **.** **map** g

can be substituted by

**map** (f **.** g)

and

**filter** p **.** **filter** q

can be substituted by

**filter** (\x **->** q x **&&** p x)

It can also help to remove intermediate data structures.

1. Напишите коротко, что позволяет делать Template Haskell

        Генерировать и выполнять хаскель код на этапе компиляции

1. Напишите коротко, что такое линзы и зачем они нужны

        Класс, которые позволяет делать красивые конструкторы и удобно с ними работать, обращаться к проекциям типа?

1. В чем разница между rpar, rseq  и rdeepseq?

rpar :: a -> Eval a  -- suggest to parallel, create \*spark\*

rseq :: a -> Eval a  -- wait for evaluation of argument (eval it to WHNF)

1. Что такой acid-state и что он позволяет делать?

        Add ACID guarantees to any serializable Haskell data structure (сериализация данных с гарантиями)

        Атомарность, согласованность, изолированность, долговечность

1. Напишите ранг функций и их запись типа с forall:   
   String -> Maybe Int -- rank 0  
   Either a b -> Either c d -- rank 0  
   (forall a . a -> [a]) -> (forall b . [b] -> b) -- rank 2
2. Зачем нужен callCC?

callCC :: ((a -> m b) -> m a) -> m a

    вызывает функцию с текущим продолжением в качестве аргумента.

    callCC (call-with-current-continuation) calls a function with the current continuation as its argument. Provides an escape continuation mechanism for use with Continuation monads. Escape continuations allow to abort the current computation and return a value immediately. They achieve a similar effect to [throwError](http://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Error.html#v:throwError) and [catchError](http://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Error.html#v:catchError) within an [Error](http://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Error.html#t:Error) monad. Advantage of this function over calling return is that it makes the continuation explicit, allowing more flexibility and better control (see examples in [Control.Monad.Cont](http://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Cont.html)).

The standard idiom used with callCC is to provide a lambda-expression to name the continuation. Then calling the named continuation anywhere within its scope will escape from the computation, even if it is many layers deep within nested computations.

1. Задан тип data Stream a = a :> Stream a. Напишите instance Comonad для него через функцию extend.
2. Напишите при помощи Arrows функцию, которая вместо первого элемента кортежа, являющегося списком, подставляет его длину
3. Напишите тип функции liftF и ее реализацию
4. Приведите пример type family или data family (не из презентации)
5. Что такое Functional Dependencies? Назовите какой-нибудь известный вам type class, в котором присутствуют функциональные зависимости.

        Вариант 26

1. Перечислите основные парадигмы ФП и Haskell, отличительные особенности (>=5)

было

1. Напишите функцию с типом, которая принимает список пар чисел и оставляет только такие, что сумма чисел в паре чётная

    fun :: [(a, a)] -> [(a, a)]

    fun = filter (\(a,b) -> even (a + b))

1. Что такое Monoid? Укажите известные вам инстансы Monoid

    Class Monoid a where

        Mempty : a

        Mappend :: a -> a -> a

        mconcat :: [a] -> a

    Instances: [a], Maybe a, Identity a

1. Перечислите несколько типов данных, являющихся монадой (>= 5)

    Maybe, Either, Reader, Writer, State, Identity

1. Напишите type class Functor и его реализацию для списка

class Functor f where

    Fmap :: (a -> b) -> f a -> f b

instance Functor [a] where

    Fmap= map

1. Напишите type class Applicative и его реализацию для Maybe

    class Functor f => Applicative f where

        pure :: a

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

    instance Applicative (Maybe) where

        pure = Just

        (Just f) <\*> x = fmap f x

        Nothing <\*> \_ = Nothing

1. Напишите реализацию Foldable для Maybe

class Foldable t where

    foldMap :: (a -> m) -> t a -> m

    foldr :: (a -> b -> b) -> b -> t a -> b

    fold :: t m -> m

instance Foldable Maybe where

    foldr f y (Just x) = f x y

    foldr \_ z Nothing = z

1. Был раньше
2. Напишите определение типа данных Reader и его инстанс Monad

    data Reader r a = Reader { runReader :: r -> a }

    instance Monad (Reader r) where

        return a = Reader $ \\_ -> a

        m >>= f = Reader $ \r -> runReader (f (runReader m r)) r

1. Почему действия с внешним миров требуют монаду

было

1. Напишите класс MonadTrans и реализуйте его для WriterT

    class MonadTrans t where

        lift ::(Monad m) ->  m a -> t m a

    data Writer w m a = Writer {runWriter :: m (a, w)}

    instance MonadTrans (WriterT w) where

        lift x =  сложно

1. Был раньше
2. Как можно посмотреть AST-дерево для выражений?

    report :: Bool -> String -> Q ()

1. Было раньше
2. Что такое Strategy?

type Strategy a = a -> Eval a

Стратегия параллелизма??

1. Что такое FFI?

    Foreign function interface

1. Как определяется ранг функции

    Глубина forall в дереве

    The rank of a type describes the depth at which universal quantifiers appear

in a contravariant position, i.e. to the left of a function arrow.

A function type has rank n + 1 when its argument has rank n.

1. Напишите ConstT и его инстанс Monad.
2. Задан тип data Stream a = a :> Stream a. Напишите instance Comonad для него через функцию duplicate
3. Напишите при помощи Arrows функцию, которая вместо первого элемента кортежа, являющегося списком, подставляет его длину
4. Напишите Free и реализуйте instance Monad для него
5. Укажите отличие type family от data family
6. Что такое GADT и в чём их отличие от ADT?

* Напишите тип данных Difference List и реализацию функции append для него. В чем преимущества этого класса над обычным списком?

newtype DList a = DL { unDL :: [a] -> [a] }

append       :: DList a -> DList a -> DList a  
    append xs ys = DL (unDL xs . unDL ys)

* Какими способами в haskell реализуется параллелизм и что они значат.

    rpar :: a -> Eval a  -- suggest to parallel, create \*spark\*

rseq :: a -> Eval a  -- wait for evaluation of argument (eval it to WHNF)

Комбинация стратегий?

Spark -- не параллелить работы, а правильно их распределить

Неточный параллелизм?

* Что такое stack и чем отличается от других?

    Система сборки проекта, умеет собирать и тестить

* Напишите тип ContT, как Cont выражен через ContT и инстант MonadTrans для ContT
* Напишите instance Comonad для (,) e, реализовав функцию extend
* Что такое стрелки, что они позволяют делать?