**Курс: Функциональное программирование**
**Практика 14. Рекурсивные типы**

## Разминка

▶ Покажите, что типы `(Integer,Integer)` и `Bool -> Integer` изоморфны, реализовав взаимнообратные

```
fromP :: (Integer,Integer) -> (Bool -> Integer)
fromP = undefined

toP :: (Bool -> Integer) -> (Integer,Integer)
toP = undefined
```

Тест:

```
GHCi> (toP . fromP) (42,100)
(42,100)
GHCi> (fromP . toP) (\x -> if x then 42 else 100) True
42
GHCi> (fromP . toP) (\x -> if x then 42 else 100) False
100
```

▶ Переведите с языка структурных типов на язык Haskell приведенные ниже изоморфизмы и докажите их.

$$(C^B)^A \cong C^{A*B},$$

$$C^{A+B} \cong C^A * C^B.$$

### Эквирекурсивные типы

Тип функции неограниченной арности

$$\mu\gamma.\,\alpha \to \gamma \;=\; \mu\gamma.\,\alpha \to (\mu\gamma.\,\alpha \to \gamma)$$

В номинальном представлении $\texttt{Hungry} \;\equiv\; \mu\gamma.\,\gamma^{\alpha}$ или $\texttt{Hungry} = \alpha \to \texttt{Hungry}$. Обитатель этого типа конструируется как неподвижная точка канонического комбинатора $\mathbf{K}$

$$\texttt{eater} = \texttt{fix}\ (\lambda \mathrm{f}^{\texttt{Hungry}}\, \chi^{\alpha}.\, \mathrm{f})$$

▶ Предполагая, что `fix` имеет стандартный тип, проверьте, что `eater` имеет тип `Hungry`.

▶ Попробуйте, используя эквирекурсивное определение $\texttt{AutoA} = \texttt{AutoA} \to \alpha$, приписать типы каноническим комбинаторам $\mathbf{\Omega}$ и $\mathbf{Y}$.

### Оператор неподвижной точки для типов

Оператор неподвижной точки для типов

```
newtype Fix f = In (f (Fix f))

deriving instance Show (f (Fix f)) => Show (Fix f)
deriving instance Eq (f (Fix f)) => Eq (Fix f)

out :: Fix f -> f (Fix f)
out (In x) = x
```

Для удобства работы с рекурсивными типами, построенными с помощью `Fix`, полезно подключить расширения `StandaloneDeriving`, `FlexibleContexts`, `UndecidableInstances`.

```
GHCi> :k Fix
Fix :: (* -> *) -> *
GHCi> :t In
In :: f (Fix f) -> Fix f
```

Для описания рекурсивного типа эквивалентного, например,

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

задаём нерекурсивный тип

```
data L a l = Nil | Cons a l deriving (Eq,Show)

instance Functor (L a) where
 fmap _ Nil        = Nil
 fmap g (Cons x l) = Cons x (g l)
```

и вводим рекурсивный через неподвижную точку функтора `N` на уровне типов:

```
type List a = Fix (L a)
```

▶ Покажите, что типы `[a]` и `List a` изоморфны, реализовав взаимнообратные

```
from ::  [a] -> List a
from = undefined

to :: List a -> [a]
to = undefined
```

Тест:

```
GHCi> from "hi"
In (Cons 'h' (In (Cons 'i' (In Nil))))
GHCi> to $ In (Cons 'h' (In (Cons 'i' (In Nil))))
"hi"
GHCi> (to . from) [1,2,3]
[1,2,3]
GHCi> (from . to) $ In (Cons 1 (In (Cons 2 (In (Cons 3 (In Nil))))))
In (Cons 1 (In (Cons 2 (In (Cons 3 (In Nil))))))
```

### Катаморфизм

```
type Algebra f a = f a -> a

cata :: Functor f => Algebra f a -> Fix f -> a
cata phi (In x) = phi $ fmap (cata phi) x
```

Пример:

```
data N x = Z | S x deriving Show

instance Functor N where
  fmap g  Z    = Z
  fmap g (S x) = S (g x)

type Nat = Fix N

phiN :: Algebra N Int                -- N Int -> Int
phiN Z     = 0
phiN (S n) = succ n

natToInt :: Nat -> Int
natToInt = cata phiN
```

### Анаморфизм

```
type Coalgebra f a = a -> f a

ana :: Functor f => Coalgebra f a -> a -> Fix f
ana psi x = In $ fmap (ana psi) (psi x)
```

Пример:

```
psiN :: Coalgebra N Int            -- Int -> N Int
psiN 0 = Z
psiN n = S (n-1)
```

```
intToNat :: Int -> Nat
intToNat = ana psiN
```

### Гилеморфизм

```
hylo :: Functor f => Algebra f a -> Coalgebra f b -> (b -> a)
hylo phi psi = cata phi . ana psi
```

### Тип для двоичных чисел

Рассмотрим рекурсивный тип данных, представляющий числа в двоичной форме:

```
data Bin = Empty | Zero Bin | One Bin
```

При таком определении двоичные числа читаются справа налево

```
                Empty   == 0
            One Empty   == 1
      Zero (One Empty)  == 2
      One  (One Empty)  == 3
Zero (Zero (One Empty)) == 4
```

▶ (1 балл) Разработайте нерекурсивный функтор `B`, описывающий структуру этого типа, и определите `Bin` как неподвижную точку этого функтора.

▶ (1 балл) Определите B-алгебру `phiB :: B Int -> Int`, позволяющую задать катаморфизм

```
bin2int :: Bin -> Int        -- синоним  Fix B -> Int
bin2int = cata phiB
```

```
GHCi> four  = In $ Zero $ In $ Zero $ In $ One $ In Empty
GHCi> bin2int four
4
```

▶ (1 балл) Определите B-коалгебру `psiB :: Int -> B Int`, позволяющую задать анаморфизм

```
int2bin :: Int -> Bin
int2bin = ana psiB
```

```
GHCi> int2bin 31
In (One (In (One (In (One (In (One (In (One (In Empty))))))))))
```

### Тип для вычислителя выражений

Рассмотрим рекурсивный тип данных, представляющий арифметическое выражение:

```
data Expr = Num Int
          | Add Expr Expr
          | Mult Expr Expr
```

▶ (0.5 балла) Разработайте нерекурсивный функтор `E`, описывающий структуру этого типа, и определите `Expr` как неподвижную точку этого функтора.

Для тестов ниже используются следующие выражения

```
en = In . Num
e3     = en 3
ep35   = In (Add e3 (en 5))
emp357 = In (Mult ep35 (en 7))
em7p35 = In (Mult (en 7) ep35)
```

▶ (0.5 балла) Определите E-алгебру `phiE :: E Int -> Int`, позволяющую задать катаморфизм, вычисляющий выражение

```
eval :: Expr -> Int
eval = cata phiE
```

```
GHCi> eval ep35
8
GHCi> eval emp357
56
```

▶ (1 балл) Определите E-алгебру `phiEShow :: E String -> String`, позволяющую задать катаморфизм, конструирующий строковое представление выражения

```
GHCi> cata phiEShow e3
"3"
GHCi> cata phiEShow ep35
"(3+5)"
GHCi> cata phiEShow emp357
"((3+5)*7)"
```

▶ (1 балл) Определите E-алгебру `phiEShowS :: E ShowS -> ShowS`, позволяющую задать катаморфизм, конструирующий строковое представление выражения в префиксной бесскобочной записи (польской нотации)

```
GHCi> cata phiEShowS ep35 ""
"+ 3 5"
GHCi> cata phiEShowS emp357 ""
"* + 3 5 7"
GHCi> cata phiEShowS em7p35 ""
"* 7 + 3 5"
```

▶ (1 балл) Реализуйте вычислитель на стеке, используя следующие сервисные функции над стеком, реализованным через список:

```haskell
type Stack = [Int]

push :: Int -> Stack -> Stack
push a as = a : as

add :: Stack -> Stack
add  (a : b : cs) = (b + a) : cs

mult :: Stack -> Stack
mult (a : b : cs) = (b * a) : cs
```

Для этого определите E-алгебру `phiE' :: E (Stack -> Stack) -> Stack -> Stack`, позволяющую задать катаморфизм, осуществляющий вычисление

```haskell
eval' :: Expr -> Stack -> Stack
eval' = cata phiE'
```

```
GHCi> eval' emp357 []
[56]
```

### Тип для бинарного дерева

Рекурсивный тип данных, представляющий бинарное дерево:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

▶ (0.5 балла) Разработайте нерекурсивный функтор `T a`, описывающий структуру этого типа, и определите `Tree a` как неподвижную точку этого функтора. Для тестов ниже используется следующее дерево

```
{-
     5
    / \
   3   6
  / \   \
 2   4   7
-}
iB l x r = In $ Branch l x r
iL = In Leaf
testTree =
  iB
    (iB
      (iB iL
      2
      iL)
    3
      (iB iL
      4
      iL)
    )
  5
    (iB iL
    6
      (iB iL
      7
      iL)
    )
```

▶ (0.5 балла) Определите алгебру `phiTSum :: Algebra (T Integer) Integer`, катаморфизм которой суммирует значения в узлах дерева.

```
phiTSum :: Algebra (T Integer) Integer
phiTSum = undefined

treeSum :: Tree Integer -> Integer
treeSum = cata phiTSum
```

```
GHCi> treeSum testTree
27
```

▶ (1 балл) Определите алгебру `phiTInorder :: Algebra (T a) [a]`, такую что катаморфизм давал бы список всех узлов дерева, посещаемых в соответствии с inorder-стратегией обхода.

```
phiTInorder :: Algebra (T a) [a] -- T a [a] -> [a]
phiTInorder = undefined

tree2listInorder :: Tree a -> [a]
tree2listInorder = cata phiTInorder
```

```
GHCi> tree2listInorder testTree
[2,3,4,5,6,7]
```

▶ (1 балл) Определите коалгебру `psiTBST :: Ord a => Coalgebra (T a) [a]`, генерирующую на основе списка с элементами из класса типов `Ord` бинарное дерево поиска (меньшие элементы отправляются влево, большие - вправо).

```
psiTBST :: Ord a => Coalgebra (T a) [a]    -- [a] -> T a [a]
psiTBST = undefined

list2BST :: Ord a => [a] -> Tree a
list2BST = ana psiTBST
```

```
GHCi> list2BST [3,1,4]
In (Branch (In (Branch (In Leaf) 1 (In Leaf))) 3
(In (Branch (In Leaf) 4 (In Leaf))))
```

Гилеморфизм этих коалгебры и алгебры должен давать функцию, сортирующую список:

```
sort :: Ord a => [a] -> [a]
sort = hylo phiTInorder psiTBST
```

```
GHCi> sort [6,3,7,2,4,9]
[2,3,4,6,7,9]
```