Деревья

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

list2tree :: [a] -> Tree a

list2tree

Написать функцию, строящую сбалансированное дерево в один проход

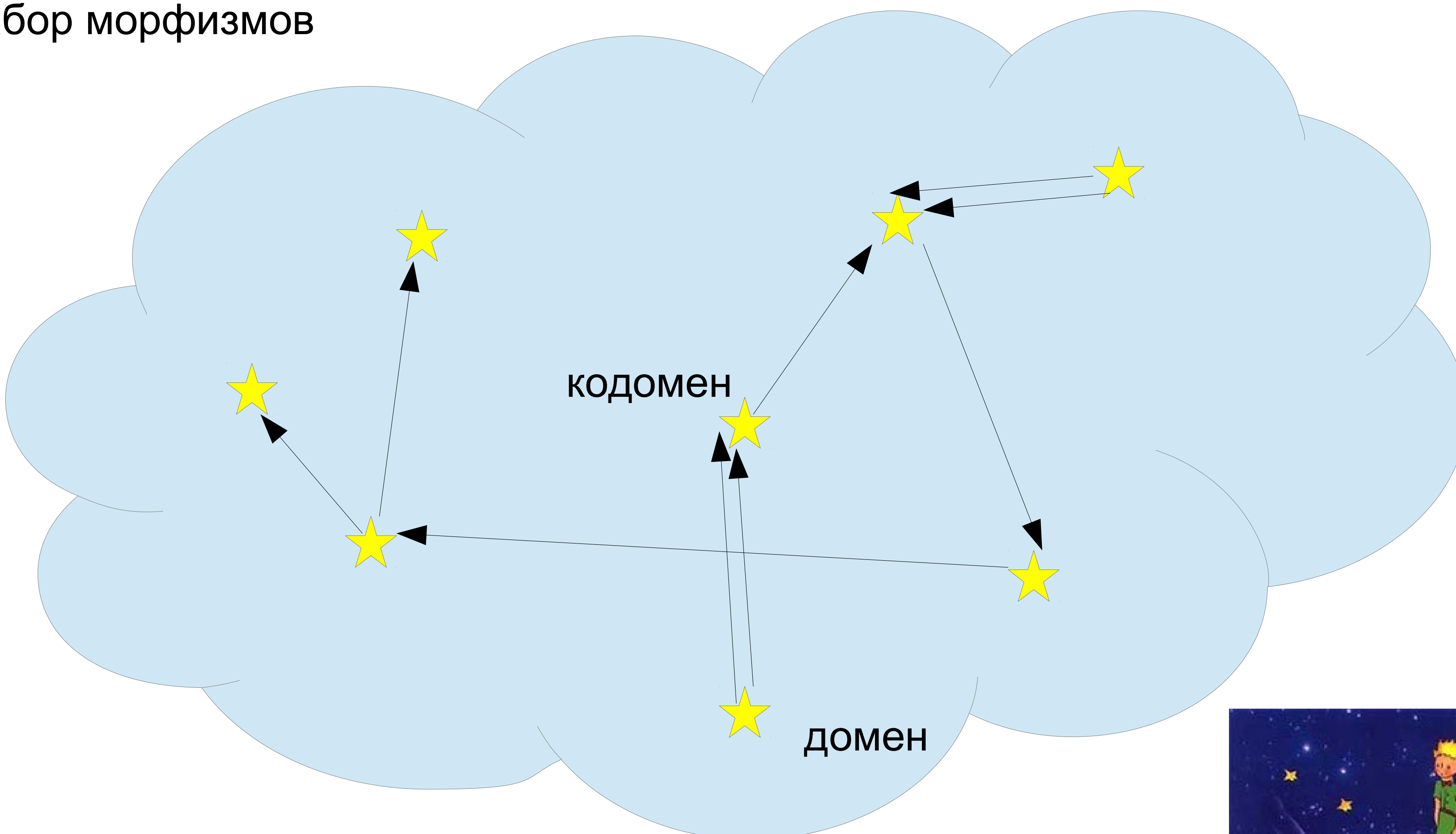
Деревья

```
list2tree xs = fst (build (length xs) xs)

build :: Int -> [a] -> (Tree a, [a])
build 0 xs = (Empty, xs)
build 1 (x:xs) = (Node x Empty Empty, xs)
build n (x:xs) = (Node x u v, xs'')
  where
    m = div (n-1) 2
    (u, xs') = build m xs
    (v, xs'') = build (n-1-m) xs'
```

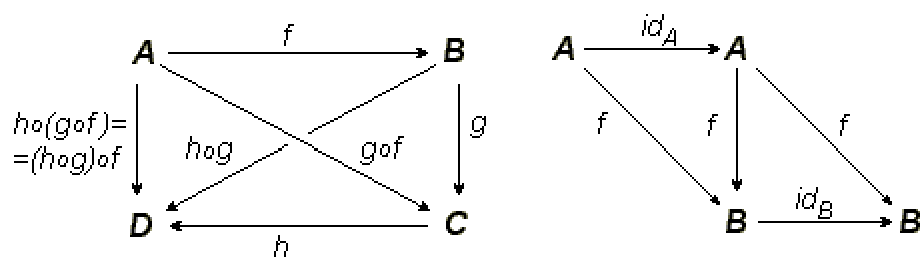
Категории

- набор объектов
- набор морфизмов



Категории

всякие аксиомы:
- композиция ассоциативна
- есть тождественный морфизм



(ц) википедия

Категории

когда домен = кодомен, это – эндоморфизм

классический пример: категория Set

объекты категории Set — это всевозможные множества, а морфизмы — это функции между этими множествами.

категория Hask.
объекты — это типы данных, возможные в языке Haskell: Int, [], Tree, a, b

а морфизмы — функции языка Haskell: Int → Int, (a → b) → a → b, Tree → Int

Категории

```
class  Category (~>)  where
  (.) :: (a ~> b) -> (b ~> c) -> (a ~> c)
  id  :: a ~> a
```

законы:

1. ∀ a, b, c : a . (b . c) = (a . b) . c
2. ∀ m : m . id = m = id . m

Функторы

функтор F — это отображение между категориями, такое, что структура категории сохраняется или, другими словами, это гомоморфизм между двумя категориями

При этом функтор отображает объекты первой категории в объекты второй, а морфизмы первой — в морфизмы второй категории. К тому же накладываются определённые ограничения — аксиомы.

Если функтор отображает категорию саму в себя, такой функтор называется эндофунктором.

Функторы

Любой конструктор выполняет преобразования
[]: a → [a]
Maybe: b → Maybe b: Just b | Nothing
data C c = …
C: c → C c
Если теперь такое отображение объектов (конструктор типов C c одним параметром) дополнить отображением морфизмов, то получим функтор, действующий из Hask в Hask, эндофунктор на категории Hask.

морфизм is a -> b

Согласно аксиомам функторов, морфизм между объектами a и b должен отображаться в морфизм между объектами (C a) и (C b). Таким образом, отображение морфизмов должно быть функцией следующего вида:

```
fmap :: (a -> b) -> (C a -> C b)
```

Функторы

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


1. ∀ f, g : fmap f . fmap g = fmap (f . g)
2. fmap id = id
   id x = x
   fmap id x = id x
```

Функторы

```
> map (+1) [1,2,3,4,5]
[2,3,4,5,6]


instance Functor [] where
  fmap = map
```

Функторы

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

```
(read . show $ 4)::Int
fmap (show . (+2)) [1..4]
(fmap show . fmap (+2)) [1..4]
```

Функторы

```
instance Functor Tree where
    fmap g EmptyTree = EmptyTree
    fmap g (Node a l r) = Node (g a) (fmap g l) (fmap g r)
```

Функторы

```
> fmap length (list2tree [“goodbye”,”cruel”,”world”])
```

Функторы

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap _ Nothing  = Nothing

(fmap show . fmap (+2)) $ Just 4
(fmap show . fmap (+2)) $ Nothing
```

Функторы

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap _ Nothing  = Nothing

Доказать, что
fmap (f . g) = fmap f . fmap g

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

Понимание функтора

```
:t fmap

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

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

:t fmap (show . (1+))
… :: (Functor f, Num b, Show b) => f b -> f String

:t fmap (replicate 3)
… :: Functor f => f a -> f [a]

fmap (replicate 3) [1,2,3]
fmap (replicate 3) $ Just 2

fmap (replicate 3) Just 2 ?
```

Аппликативные функторы

```
let a = fmap (*) [1,2,3,4]
:t a
a :: [Integer -> Integer]
fmap (\f -> f 9) a
[9,18,27,36]

import Control.Applicative

class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Аппликативные функторы (Maybe)

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something

Just (+3) <*> Just 9
pure (+3) <*> Just 9
Just (++"hahah") <*> Nothing
Nothing <*> Just "woot"

pure (+) <*> Just 3 <*> Just 5
pure (\x y z -> x + y + z) <*> Just 3 <*> Just 5 <*> Just 2
```

Аппликативные функторы (Maybe)

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

pure (\x y z -> x + y + z) <*> Just 3 <*> Just 5 <*> Just 2
(\x y z -> x + y + z) <$> Just 3 <*> Just 5 <*> Just 2
```

Аппликативные функторы (List)

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]

[(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]

[(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]

[ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]

(*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

Аппликативные функторы (ZipList)

```
instance Applicative ZipList where
        pure x = ZipList (repeat x)
        ZipList fs <*> ZipList xs =
          ZipList (zipWith (\f x -> f x) fs xs)

getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [1,1,1]
[2,3,4]

getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [1,1..]
[2,3,4]

getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Аппликативные функторы

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b

liftA2 (:) (Just 1) (Just [2])

liftA3 (,,) [1] [2] [3]

sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs

sequenceA [Just 3, Just 2, Just 1]
sequenceA [Just 3, Nothing, Just 1]
sequenceA [(+3),(+2),(+1)] 3
sequenceA [[1,2,3],[4,5,6]]
sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]

sequenceA через свертку !
```

Аппликативные функторы

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

import Data.Traversable

```
and $ map (\f -> f 7) [(>4),(<10),odd]
True

and $ sequenceA [(>4),(<10),odd] 7
True

:t getLine
getLine :: IO String

sequenceA [getLine, getLine, getLine]
```

Аппликативные функторы

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
```