Курс: Функциональное программирование
Практика 7. Свёртки

Моноиды

Моноид — это множество с ассоциативной бинарной операцией над ним (`mappend`) и нейтральным элементом для этой операции (`mempty`). Для `mappend` есть инфиксный синоним

```
infixr 6 <>

(<>) :: Monoid m => m -> m -> m
(<>) = mappend
```

Эндоморфизм (стрелка из типа в него же) можно упаковать так

```
newtype Endo a = Endo { appEndo :: a -> a }
```

Эндоморфизм образует моноид относительно композиции.

▶ Напишите

```
instance Monoid (Endo a) where
 mempty = ???
 Endo f `mappend` Endo g = ???
```

▶ Определите тип функции

```
fn = mconcat $ map Endo [(+5),(*3),(^2)]
```

и вычислите значение выражения

```
appEndo fn 2
```

▶ Можно ли написать представителя моноида для типа `Maybe a`? Сколько разных вариантов можно реализовать?

Свёртки

▶ Устно вычислите значения выражений и проверьте результат в GHCi:

```
foldl (/) 480 [3,2,5,2]
foldr (/) 2 [8,12,24,4]
```

► Используя `foldr1`, напишите функцию, конструирующую из списка строк строку, разделённую запятыми.

```
> f ["ab","cde","fgh"]
"ab,cde,fgh"
```

► Напишите реализацию следующих функций стандартной библиотеки через свёртки `foldr` или `foldl` (можно использовать `foldr1` или `foldl1`):

```
or' :: [Bool] -> Bool
or' = fold? undefined undefined

length' :: [a] -> Int
length' = fold? undefined undefined

maximum' :: Ord a => [a] -> a
maximum' = fold? undefined undefined

head' :: [a] -> a
head' = fold? undefined undefined

last' :: [a] -> a
last' = fold? undefined undefined

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = fold? undefined undefined

map' :: (a -> b) -> [a] -> [b]
map' f = fold? undefined undefined
```

Иногда реализация через свертку требует некоторого трюка с дополнительным параметром:

```
take' :: Int -> [a] -> [a]
take' n xs = foldr step ini xs n
 where
  step :: a -> (Int -> [a]) -> Int -> [a]
  step x g 0 = []
  step x g n = x : g (n - 1)
  ini :: Int -> [a]
  ini = const []
```

Работает это так

```
take' 2 "abc" ~>                                  -- take' def
foldr step ini "abc" 2 ~>>                        -- foldr def (3 times)
step 'a' (step 'b' (step 'c' ini)) 2 ~>           -- step def (2)
'a' : ( (step 'b' (step 'c' ini)) (2 - 1) ) ~>    -- step def (2)
'a' : 'b' : ( (step 'c' ini) 0 ) ~>               -- step def (1)
'a' : 'b' : []
```

Foldr-map fusion law

Найдем какие требования нужно наложить на `g`, `w`, `h`, `f` и `v`, чтобы они удовлетворяли уравнению

```
foldr g w . map h = foldr f v          -- (Eqv)
```

(1) На пустом списке это дает

```
foldr g w (map h []) = foldr f v []    -- map def
foldr g w []         = foldr f v []    -- foldr def
w                    = v                                  -- (result 1)
```

(2) На непустом списке

```
foldr g w (map h (x:xs))       = foldr f v (x:xs)
foldr g w (h x : map h xs)     = foldr f v (x:xs)   -- map def
g (h x) (foldr g w (map h xs)) = f x (foldr f v xs) -- foldr def
g (h x) (foldr f v xs)         = f x (foldr f v xs) -- (Eqv, as IH)
g (h x)                        = f x
g . h                          = f                       -- (result 2)
```

Получаем знаменитый foldr-map fusion law

```
foldr g w . map h = foldr (g . h) w
```

Домашнее задание

▶ (1 балл) Используя `unfoldr`, реализуйте функцию, которая возвращает в обратном алфавитном порядке список символов, попадающих в заданный парой диапазон. Попадание символа `x` в диапазон пары `(a,b)` означает, что `x >= a` и `x <= b`.

```
revRange :: (Char,Char) -> [Char]
revRange = unfoldr fun

fun = undefined
```

▶ (1 балл) Напишите реализации функций из стандартной библиотеки `tails, inits :: [a] -> [[a]]` через свёртку `foldr`:

```
tails' :: [a] -> [[a]]
tails' = foldr fun ini
fun = undefined
ini = undefined

inits' :: [a] -> [[a]]
inits' = foldr fun' ini'
fun' = undefined
ini' = undefined
```

► (1 балл) Напишите две реализации функции обращения списка `reverse :: [a] -> [a]` — через свёртки `foldr` и `foldl`:

```
reverse' :: [a] -> [a]
reverse' = foldr fun' ini'
fun' = undefined
ini' = undefined

reverse'' :: [a] -> [a]
reverse'' = foldl fun'' ini''
fun'' = undefined
ini'' = undefined
```

► (2 балла) Напишите реализацию оператора «безопасного» поиска элемента списка по индексу `(!!!)` через `foldr`:

```
GHCi> [1..10] !!! 5
Just 6
GHCi> [1..10] !!! (-1)
Nothing
GHCi> [1..10] !!! 100
Nothing
```

Используйте технику дополнительного параметра:

```
(!!!) :: [a] -> Int -> Maybe a
xs !!! n = foldr fun ini xs n
fun = undefined
ini = undefined
```

► (2 балла) Напишите реализацию `foldl` через `foldr`:

```
foldl'' :: (b -> a -> b) -> b -> [a] -> b
foldl'' f v xs = foldr (fun f) ini xs v
fun = undefined
ini = undefined
```

(Используйте технику дополнительного параметра.)

► (3 балла) Для реализации свертки двоичных деревьев нужно выбрать алгоритм обхода узлов дерева (см., например, `http://en.wikipedia.org/wiki/Tree_traversal`).

Сделайте двоичное дерево

```
data Tree a = Nil | Branch (Tree a) a (Tree a)  deriving (Eq, Show)
```

представителем класса типов `Foldable`, реализовав симметричную стратегию (in-order traversal). Реализуйте также три другие стандартные стратегии (pre-order traversal, post-order traversal и level-order traversal), упаковав дерево в типы-обертки

```
newtype Preorder a = PreO (Tree a) deriving (Eq, Show)
newtype Postorder a = PostO (Tree a) deriving (Eq, Show)
newtype Levelorder a = LevelO (Tree a) deriving (Eq, Show)
```

и сделав эти обертки представителями класса `Foldable`.