Курс: Функциональное программирование
Практика 6. Классы типов

Разминка

Определим бинарное дерево так

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

▶ Сделайте тип `Tree a` представителем класса типов `Eq`.
▶ Реализуйте функцию `elemTree`, определяющую, хранится ли заданное значение в заданном дереве.
▶ Протестируйте функцию `elemTree`. Может ли она работать на бесконечных деревьях?
▶ Сделайте типовый оператор `Tree` представителем класса типов `Functor`.

Класс типов `Show`

Служит для представления значений типа в строковом виде

```
type ShowS = String -> String

class  Show a  where
    showsPrec :: Int    --  the operator precedence
              -> a      --  the value to be converted to a 'String'
              -> ShowS

    show      :: a   -> String

    showsPrec _ x s = show x ++ s
    show x          = shows x ""

shows           :: (Show a) => a -> ShowS
shows           =  showsPrec zeroInt
```

Рассмотрим тип списка

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

Реализуем для него представителя класса `Show`.

Версия 1, через `show`:

```
instance Show a => Show (List a) where
    show = myShowList

myShowList :: Show a => List a -> String
myShowList Nil         = "EoL"
myShowList (Cons x xs) = show x
                         ++ ";"
                         ++ myShowList xs

*Test> Cons 2 (Cons 3 (Cons 5 Nil))
2;3;5;EoL
```

Версия 2, через `shows` (на сложных типах более эффективна):

```
instance Show a => Show (List a) where
    showsPrec _ = myShowsList

myShowsList :: Show a => List a -> ShowS
myShowsList Nil         = ("EoL" ++)
myShowsList (Cons x xs) = shows x
                          . (';' :)
                          . myShowsList xs

*Test> Cons 2 (Cons 3 (Cons 5 Nil))
2;3;5;EoL
```

Имеется функция `showChar :: Char -> ShowS`, которую можно использовать вместо `(';' :)`.

► Напишите версию `instance Show` для типа `List a`, так чтобы они выводили список в следующем виде

```
*Test> Cons 2 (Cons 3 (Cons 5 Nil))
<2<3<5|>>>
```

► Напишите две версии `instance Show` для типа `Tree a`, так чтобы они выводили дерево в следующем виде

```
*Test> Branch (Leaf 1) 2 (Branch (Leaf 3) 4 (Leaf 5))
<1{2}<3{4}5>>
```

Оцените их производительность.

Класс типов `Read`

Служит для преобразования строкового представления в значения типа

```
type ReadS a = String -> [(a, String)]
class Read a where
  readsPrec    :: Int   -- the operator precedence of the enclosing context
               -> ReadS a

reads :: Read a => ReadS a
```

Например,

```
*Test> (reads "5 golden rings") :: [(Integer,String)]
[(5," golden rings")]
```

Для списков

```
myReadsList :: (Read a) => ReadS (List a)
myReadsList ('|':s) = [(Nil, s)]
myReadsList ('<':s) = [(Cons x l, u) | (x, t)     <- reads s,
                                       (l, '>':u) <- myReadsList t ]
```

```
*Test> (myReadsList "<2<3<5|>>> something else") :: [(List Int, String)]
[(Cons 2 (Cons 3 (Cons 5 Nil))," something else")]
```

▶ Напишите функцию `myReadsTree :: (Read a) => ReadS (Tree a)` так чтобы

```
*Test> (myReadsTree "<1{2}<3{4}5>> something else") :: [(Tree Int, String)]
[(Branch (Leaf 1) 2 (Branch (Leaf 3) 4 (Leaf 5))," something else")]
```