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 0
```

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
```

Имеются вспомогательные функции: `showChar :: Char -> ShowS`, которую можно использовать вместо `(';' :)`, и `showString :: String -> ShowS`, которую можно использовать вместо `("EoL" ++)`.

► 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")]
```

Домашнее задание

▶ (1 балл) Сделайте тип

```
newtype Matrix a = Matrix [[a]]
```

представителем класса типов `Show`. Строки матрицы (внутренние списки) должны изображаться как списки; каждый следующий внутренний список должен начинаться с новой строки (используйте символ `'\n'` в качестве разделителя). Пустая матрица должна выводиться как `EMPTY`.

```
GHCi> Matrix [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3]
[4,5,6]
[7,8,9]
GHCi> Matrix []
EMPTY
```

Не забывайте про существование полезных вспомогательных функций `showChar`, `showString` и `showList`.

▶ (1 балл) В модуле `Data.Complex` стандартной библиотеки реализован тип комплексных чисел `Complex a`. Сделайте тип-обертку

```
newtype Cmplx = Cmplx (Complex Double) deriving Eq
```

представителем классов типов `Show` и `Read`. Представитель класса типов `Show` должен использовать разделители вещественной и мнимой части `+i*` и `-i*`, зависящие от знака мнимой части:

```
GHCi> Cmplx $ (-2.7) :+ 3.4
-2.7+i*3.4
GHCi> Cmplx $ (-2.7) :+ (-3.4)
-2.7-i*3.4
```

Представитель класса типов `Read` должен быть точным дополнением представителя класса `Show`, то есть для любого `z :: Cmplx` должно выполняться

```
read (show z) == z
```

▶ (1 балл) Реализуйте класс типов

```
class SafeEnum a where
  ssucc :: a -> a
  spred :: a -> a
```

обе функции которого ведут себя как `succ` и `pred` стандартного класса `Enum`, однако являются тотальными, то есть не останавливаются с ошибкой на наибольшем и наименьшем значениях типа-перечисления соответственно, а обеспечивают циклическое поведение. Ваш класс должен быть расширением ряда классов типов стандартной библиотеки, так чтобы можно было написать реализацию по умолчанию его методов, позволяющую объявлять его представителей без необходимости писать какой бы то ни было код. Например, для типа `Bool` должно быть достаточно написать строку

```
instance SafeEnum Bool
```

и получить возможность вызывать

```
GHCi> ssucc False
True
GHCi> ssucc True
False
```

(Сравните это поведение со стандартной `succ` из `Enum`.)

► (2 балла) Реализуйте функцию, задающую циклическую ротацию списка.

```
rotate :: Int -> [a] -> [a]
rotate n xs = undefined
```

При положительном значении целочисленного аргумента ротация должна осуществляться влево, при отрицательном — вправо.

```
GHCi> rotate 2 "abcdefghik"
"cdefghikab"
GHCi> rotate (-2) "abcdefghik"
"ikabcdefgh"
```

Не забывайте обеспечить работоспособность вашей реализации на бесконечных списках (для сценариев, когда это имеет смысл).

► (2 балла) Найдите все сочетания по заданному числу элементов из заданного списка.

```
comb :: Int -> [a] -> [[a]]
comb = undefined
```

Например,

```
GHCi> comb 3 "abcde"
["abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde"]
```

► (5 баллов) Мы использовали тип данных `Expr` для описания термов чистого нетипизированного лямбда-исчисления:

```
type Symb = String

infixl 2 :@

data Expr = Var Symb
          | Expr :@ Expr
          | Lam Symb Expr
          deriving Eq
```

Сделайте этот тип данных представителем классов типов `Show` и `Read`. Представитель `Show` должен быть реализован так, чтобы строковое представление являлось бы валидным лямбда-термом в синтаксисе Haskell:

```
GHCi> show $ Lam "x" (Var "x" :@ Var "y")
"\\x -> x y"
```

И, наоборот, представитель `Read` должен быть реализован так, чтобы валидный в синтаксисе Haskell чистый лямбда-терм считывался бы в соответствующее выражение типа `Expr`:

```
GHCi> (read "\\x y -> x y" :: Expr) == Lam "x" (Lam "y" (Var "x" :@ Var "y"))
True
```

Для представителя `Read` может оказаться удобным воспользоваться функцией `lex`, об использовании которой можно прочитать, например, здесь `http://rsdn.ru/article/haskell/haskell_part2.xml#EOD`.