Курс: Функциональное программирование
Практика 6. Классы типов

Разминка

Определим бинарное дерево так

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

Для тестирования используются следующие деревья

```
t1 = Node (Node (Node Leaf 5 Leaf) 2 Leaf) 3 (Node Leaf 4 Leaf)
t2 = Node (Node Leaf 2 Leaf) 3 (Node Leaf 4 Leaf)
t3 = Node Leaf 42 Leaf
tInf1 n = Node (tInf1 (n+2)) n (Node Leaf 42 Leaf)
tInf2 n = Node (tInf2 (n+2)) n (tInf2 (3*n-1))
```

▶ Реализуйте функцию `elemTree`, определяющую, хранится ли заданное значение в заданном дереве.

```
GHCi> elemTree 1 t1
False
GHCi> elemTree 42 (tInf1 3)
True
GHCi> elemTree 1 (tInf2 3)
Interrupted.
```

▶ Сделайте тип `Tree a` представителем класса типов `Eq`.

```
GHCi> tInf1 3 == tInf2 3
False
```

▶ Сделайте типовый оператор `Tree` представителем класса типов `Functor`.

```
GHCi> fmap (^2) t1
Node (Node (Node Leaf 25 Leaf) 4 Leaf) 9 (Node Leaf 16 Leaf)
GHCi> let h (Node _ v _) = v in h (fmap (+7) (tInf2 3))
10
```

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
```

```
GHCi> 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
```

```
GHCi> 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 она выводила список в следующем виде

```
GHCi> Cons 2 (Cons 3 (Cons 5 Nil))
<2<3<5|>>>
```

▶ Напишите версию `instance Show` для типа

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

так чтобы она выводила дерево в следующем виде

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

Роль первого параметра `showsPrec`

При выводе с использованием инфиксных операторов можно опускать лишние скобки, основываясь на их приоритете

```
infixr 7 :*
infixr 6 :+

data Cartesian7 a b = a :* b
  deriving Show

data Cartesian6 a b = a :+ b
  deriving Show
```

```
GHCi> 3 :+ (2 :* 5)
3 :+ 2 :* 5
GHCi> (3 :+ 2) :* 5
(3 :+ 2) :* 5
```

Первый параметр функции `showsPrec` позволяет управлять расстановкой скобок, основываясь на информации о приоритете, передаваемой через этот параметр из вызывающего окружения. Вот реализации, которые производит механизм `deriving`:

```
c7Prec = 7
instance (Show a, Show b) => Show (Cartesian7 a b)  where
  showsPrec p (x :* y) =
    showParen (p > c7Prec)
    (
    showsPrec (c7Prec + 1) x .
    showString " :* " .
    showsPrec (c7Prec + 1) y
    )

c6Prec = 6
instance (Show a, Show b) => Show (Cartesian6 a b)  where
  showsPrec p (x :+ y) =
    showParen (p > c6Prec)
```

```
    (
    showsPrec (c6Prec + 1) x .
    showString " :+ " .
    showsPrec (c6Prec + 1) y
    )
```

Теперь вывод скобок «управляем» приоритетом:

```
GHCi> showsPrec 6 (3 :+ 2) ""
"3 :+ 2"
GHCi> showsPrec 7 (3 :+ 2) ""
"(3 :+ 2)"
```

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,

```
GHCi> (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 ]
```

Здесь в генераторах активно используется сопоставление с образцом.

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

▶ Для типа

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

напишите функцию `myReadsTree :: Read a => ReadS (Tree a)`, обеспечивающую следующее поведение

```
> (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"]
```