Курс: Функциональное программирование
Практика 5. Стандартные списки

Разминка

▶ На каких из следующих образцов удачно завершится сопоставление строки `"Wow"`

```
(x : y)
((:) x y)
[x, y]
(x : y : z)
[x, y, z]
(x : [y, z])
(x : y : z : [])
(x : y : z : w)
(x : y : z : w : [])
```

Какие значения будут связаны с переменными при удачном сопоставлении?

▶ Вернёт ли вызов `f "False"` что-нибудь и, если вернёт, то что?

```
f (~ t : ~ r : ~ u : ~ "e") = (:) r $ (:) u [u]
```

▶ Каковы значения следующих выражений?

```
seq ((\True y -> "AAA") undefined) 42

seq ((\True -> \y -> "AAA") undefined) 42
```

Обработка списков
Нужно реализовать функцию, не используя стандартные функции и выделение списков.

▶ Найдите количество четных элементов в заданном списке целых чисел.

```
GHCi> countEven [1..5]
2
```

▶ Сформируйте новый список целых чисел, содержащий только нечетные элементы исходного.

```
GHCi> oddElemsFrom [1..5]
[1,3,5]
```

▶ Сформируйте новый список, в котором переставлены местами четные и нечетные (по порядку следования) элементы исходного.

```
GHCi> flipNeighbours ['A'..'E']
"BADCE"
```

▶ Даны два списка целых чисел. Сформируйте список, каждый элемент которого равен разности соответствующих элементов исходных списков. Если исходные списки имеют разную длину, то предполагается, что в более коротком хранится нулевое значение.

```
GHCi> subtractLists [8,7] [2,3,4]
[6,4,-4]
```

▶ Дан список чисел. Обнулить все его элементы с нечетными порядковыми номерами, оставив остальные неизменными.

```
GHCi> nullifyOddIndexed [1..4]
[1,0,3,0]
GHCi> nullifyOddIndexed [1..5]
[1,0,3,0,5]
```

▶ Поменяйте порядок элементов списка на противоположный.

```
GHCi> revList "ABCDE"
"EDCBA"
```

Постарайтесь обеспечить линейную сложность.

```
GHCi> :set +s
GHCi> let lst = [1..100000] in lst == revList (revList lst)
True
(0.07 secs, 41,659,240 bytes)
```

▶ Сформируйте список, содержащий подсписок длины n исходного списка, начиная с k-го элемента исходного. Функция должна быть тотальной, возвращающей пересечение диапазонов индексов $[k, k + n)$ и $[0, m)$, где m — длина исходного списка.

```
GHCi> sublistOfLenFrom 3 10 [1..100]
[11,12,13]
GHCi> sublistOfLenFrom 5 (-3) [1..100]
[1,2]
```

Стандартные функции для списков
Нужно реализовать функцию, используя стандартные функции и/или выделение, но не используя явную рекурсию.

▶ Дан список чисел. Увеличить все его элементы в два раза.

```
GHCi> twiceAll [1..5]
[2,4,6,8,10]
```

▶ Дан список чисел. Увеличить все его элементы с четными значениями в два раза, оставив нечетные неизменными.

```
GHCi> twiceEven [1..5]
[1,4,3,8,5]
```

▶ Для данного список построить список пар: элемент, его порядковый номер.

```
GHCi> enumerateElems "abcd"
[('a',0),('b',1),('c',2),('d',3)]
```

▶ (Еще раз.) Дан список чисел. Обнулить все его элементы с нечетными порядковыми номерами, оставив остальные неизменными.

```
GHCi> nullifyOddIndexed' [1..4]
[1,0,3,0]
GHCi> nullifyOddIndexed' [1..5]
[1,0,3,0,5]
```

▶ Дан список чисел. Удалить из него элементы, большие заданного числа $\mathtt{k}$.

```
GHCi> delMoreThan 10 [7..15]
[7,8,9,10]
```

▶ Даны три списка чисел. Составить список сумм соответствующих элементов этих списков. Длина результирующего списка ограничена длинной самого короткого из заданных списков.

```
GHCi> sum3Short [1,2,3,4] [10,20] [100,200,300]
[111,222]
```

▶ (Еще раз.) Сформируйте список, содержащий подсписок длины $\mathtt{n}$ исходного списка, начиная с $\mathtt{k}$-го элемента исходного. Функция должна быть тотальной, возвращающей пересечение диапазонов индексов $[\mathtt{k}, \mathtt{k}+\mathtt{n})$ и $[\mathtt{0}, \mathtt{m})$, где $\mathtt{m}$ — длина исходного списка.

```
GHCi> sublistOfLenFrom' 3 10 [1..100]
[11,12,13]
GHCi> sublistOfLenFrom' 5 (-3) [1..100]
[1,2]
```

Домашнее задание

▶ (1 балл) Тип данных `Ordering` определен в стандартной библиотеке так:

```
data Ordering = LT | EQ | GT
```

Он используется при определении функций, сравнивающих элементы каких-либо типов. Если первый аргумент меньше второго, возвращается `LT`, если равен — `EQ`, если больше — `GT`.
Определим тип `LogLevel` следующим образом

```
data LogLevel = Error | Warning | Info
```

Реализуйте функцию `cmp`, сравнивающую элементы типа `LogLevel` так, чтобы имел место порядок `Error` > `Warning` > `Info`.

```
cmp :: LogLevel -> LogLevel -> Ordering
cmp = undefined
```

▶ (1 балл) Тип данных `Person` определен как запись:

```
data Person = Person { firstName :: String,
                       lastName :: String,
                       age :: Int }
```

Реализуйте функцию `abbrFirstName`, которая сокращает имя до первой буквы с точкой, то есть если имя было “John”, то после применения этой функции, оно превратится в “J.”. Однако если имя было короче двух символов, то оно не меняется.

```
abbrFirstName :: Person -> Person
abbrFirstName = undefined
```

▶ (1 балл) Следующий рекурсивный тип данных задает бинарное дерево:

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

Напишите следующие функции:
▶ вычисление суммы элементов дерева

```
treeSum :: Tree Integer -> Integer
treeSum = undefined
```

▶ вычисление максимальной высоты дерева

```
treeHeight :: Tree a -> Int
treeHeight = undefined
```

▶ (1 балл) Составить список сумм соответствующих элементов трех заданных списков. Длина результирующего списка должна быть равна длине самого длинного из заданных списков, при этом «закончившиеся» списки не должны давать вклада в суммы.

```
sum3 :: Num a => [a] -> [a] -> [a] -> [a]
sum3 = undefined
```

▶ (1 балл) Сформируйте список цифр заданного целого числа.

```
digits :: Integer -> [Integer]
digits = undefined
```

▶ (1 балл) Определите, содержит ли заданное целое число все цифры от 1 до 9. (Используйте функцию `digits` из предыдущего задания.)

```
containsAllDigits :: Integer -> Bool
containsAllDigits = undefined
```

▶ (1 балл) Определите, содержит ли заданное целое число все цифры от 1 до 9 в точности по одному разу. (Используйте функцию `digits` из предыдущего задания.)

```
containsAllDigitsOnes :: Integer -> Bool
containsAllDigitsOnes = undefined
```

▶ (1 балл) Из заданного списка выделите подсписок, состоящий из элементов с n-го номера по k-ый, считая от нуля. При этом n-ый элемент должен входить в результат, а k-ый — нет.

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

Постарайтесь обеспечить разумное поведение для произвольных целых индексов и для бесконечных списков.

▶ (1 балл) Повторите каждый элемент списка заданное число раз.

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

▶ (1 балл) Дан список (возможно бесконечный) `[a1, a2, ...]` и положительное целое число `n`. Создайте список «скользящих» подсписков длины `n`, то есть список списков следующего вида:
`[[a1, ..., an], [a2, ...., a(n+1)], [a3, ..., a(n+2)], ...]`

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

Лямбда-калькулятор (дедлайн – конец семестра)

▶ Следующий тип данных может использоваться для описания термов чистого нетипизированного лямбда-исчисления:

```
type Symb = String

infixl 2 :@

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

Например, комбинатор $\omega = \lambda x.\, xx$ в этом представлении будет иметь вид `Lam "x" (Var "x" :@ Var "x")`.
▶ (3 балла) Реализуйте алгоритм подстановки терма `n` вместо всех свободных вхождений переменной `v` в терме `m` (`m[v:=n]`)

```
subst :: Symb -> Expr -> Expr -> Expr
subst v n m = undefined
```

Не забудьте про коллизии, связанные с захватом свободных переменных, и необходимость переименования связанных переменных в этом случае.
▶ (2 балла) Реализуйте алгоритм проверки α-эквивалентности двух термов:

```
alphaEq :: Expr -> Expr -> Bool
alphaEq = undefined
```

▶ (3 балла) Реализуйте алгоритм одношаговой β-редукции, сокращающий самый левый внешний редекс в терме, если это возможно:

```
reduceOnce :: Expr -> Maybe Expr
reduceOnce = undefined
```

▶ (1 балл) Реализуйте функцию многошаговой редукции к нормальной форме, использующую нормальную стратегию редукции:

```
nf :: Expr -> Expr
nf = undefined
```

▶ (1 балл) Реализуйте алгоритм проверки β-эквивалентности двух термов:

```
betaEq :: Expr -> Expr -> Bool
betaEq = undefined
```

▶ (5 баллов) Сделайте тип данных `Expr` 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"
GHCi> cY = let {x = Var "x"; f = Var "f"; fxx = Lam "x" $ f :@
(x :@ x)} in Lam "f" $ fxx :@ fxx
GHCi> show cY
"\\f -> (\\x -> f (x x)) (\\x -> f (x x))"
```

```
GHCi> cY
\f -> (\x -> f (x x)) (\x -> f (x x))
```

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 "\\x1 x2 -> x1 x2 x2" :: Expr) == Lam "x1" (Lam "x
2" (Var "x1" :@ Var "x2" :@ Var "x2"))
True
GHCi> read (show cY) == cY
True
```

Для представителя `Read` может оказаться удобным воспользоваться функцией `lex`.