**Курс: Функциональное программирование**
**Практика 5'. Типы данных**

### Разминка

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

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

### Полиморфные типы

▶ Разработайте контейнерные типы данных `EvenC` и `OddC` похожие на списки, однако позволяющие хранить только четное и нечетное число элементов соответственно. Реализуйте «библиотечные» функции `headE`, `headO`, `tailE`, `tailO`, `concat2EE`, `concat2OO`, `concat2EO`, `concat2OE`. Реализуйте контейнеры 2 способами — независимым и взаимно-рекурсивным. Сравните сложность реализации и эффективность «библиотечных» функций.

### Кодирование просто типизированного по Черчу лямбда-исчисления

▶ (Stepik, 1 балл) Для типа данных `Type`, кодирующего тип для просто типизированного лямбда-исчисления

```
type Symb = String

infixr 3 :->
data Type = TVar Symb
          | Type :-> Type
    deriving (Eq,Show)
```

реализуйте функции, возвращающие арность и порядок типа:

```
arity :: Type -> Int
arity = undefined

order :: Type -> Int
order = undefined
```

Проверка

```
GHCi> [a,b,c] = map (TVar . pure) "abc"
GHCi> arity $ (b :-> c) :-> (a :-> b) :-> a :-> c
3
GHCi> order $ (b :-> c) :-> (a :-> b) :-> a :-> c
2
```

Следующий тип данных может использоваться для кодирования термов чистого лямбда-исчисления, просто типизированного в стиле Черча:

```
infixl 4 :@
data Term = Var Symb
          | Term :@ Term
          | Lam Symb Type Term
    deriving (Eq,Show)
```

Например, комбинатор $\omega = \lambda x^{a}.\, x\, x$ в этом представлении будет иметь вид `Lam "x" (Tvar "a") (Var "x" :@ Var "x")`.

▶ (Stepik, 1 балл) Для типа данных `Term` реализуйте функции, возвращающие списки свободных и связанных переменных терма:

```
freeVars :: Term -> [Symb]
freeVars = undefined

boundVars :: Term -> [Symb]
boundVars = undefined
```

Для последней функции нужно обеспечить для переменной такое количество вхождений в результирующий список, сколько раз эта переменная связана в терме.

```
GHCi> x = Var "x"
GHCi> a = TVar "a"
GHCi> boundVars $ Lam "x" a (x :@ x)
["x"]
GHCi> boundVars $ Lam "x" a (x :@ Lam "x" a x)
```

```
["x","x"]
GHCi> boundVars $ x :@ Lam "x" a (x :@ Lam "x" a x)
["x","x"]
```

▶ (Stepik, 2 балла) Задавая контексты с помощью синонима типа

```
type Env = [(Symb,Type)]
```

реализуйте алгоритм вывода типа для комбинаторов в просто типизированном лямбда-исчислении в стиле Черча

```
infer0 :: Term -> Maybe Type
infer0 = infer []

infer :: Env -> Term -> Maybe Type
infer = undefined
```

Проверка

```
GHCi> [f,g,x] = map (Var . pure) "fgx"
GHCi> [a,b,c] = map (TVar . pure) "abc"
GHCi> infer0 (Lam "f" (b:->c) $ Lam "g" (a:->b) $ Lam "x" a $ f
:@ (g :@ x))
Just ((TVar "b" :-> TVar "c") :-> ((TVar "a" :-> TVar "b") :->
(TVar "a" :-> TVar "c")))
GHCi> infer0 (Lam "f" b $ Lam "g" c $ Lam "x" a $ f :@ (g :@ x))
Nothing
```

### 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
```

## 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 $\alpha$-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` (или ее современным аналогом `lexP`).