Haskell

Функции

Выполнили:
Костюхин Алексей
Тасаков Антон
Теплов Андрей
Студенты гр. 5030102/10201

●○○○○○○○○○○○○○

Функции

```haskell
{- Определение функций -}
-- add :: Int -> (Int -> Int)
add :: Int -> Int -> Int
add x y = x + y

{- Частичная применимость -}
addFive :: Int -> Int
addFive = add 5
addFive 10  -- Результат: 15

{- Функции как аргументы -}
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
applyTwice addFive 2  -- Результат: 12

{- Лямбда-функции -}
(\x -> x + 1) 99      -- Результат: 100
(\x y -> x + y) 3 5   -- Результат: 8
```

Функциональные операторы (1|2)

```haskell
{- Определение пользовательского оператора -}
(***) :: Int -> Int -> Int
x *** y = x ^ y
infixr 5 ***
result = 2 *** 3  -- результат 8

{- Использование функций в инфиксной форме -}
divide :: Int -> Int -> Int
divide x y = x `div` y
10 `divide` 2     -- Результат 5

{- Использование операторов в префиксной форме -}
(+) 2 3                 -- Результат 5
map ((+) 10) [1, 2, 3]  -- Результат [11, 12, 13]
```

Функциональные операторы (2|2)

```haskell
{- Сечение для сложения -}
increment = (+1)
decrement = (+(-1))
{- Сечение для оператора деления -}
half = (/2)
reciprocal = (2/)
{- Сечение для оператора сравнения -}
isPositive = (>0)
isNegative = (<0)
filter isPositive [1, -2, 3, -4]
-- Результат [1, 3]
{- Сечение для оператора конкатенации -}
addPrefix = ("Hello, " ++)
addPrefix "world!"
-- Результат "Hello, world!"
```

```haskell
{- Как работает $ -}
($) :: (a -> b) -> a -> b
f $ x = f x
infixr 0 $

sqrt (1 + 2 * 3) -- Без использования $
sqrt $ 1 + 2 * 3 -- С использованием $

{- Примеры применения $ -}
map ($ 2) [(+3), (*5), (^2)]
-- Результат [5,10,4]
sum $ map (^2) $ filter even [1..10]
-- Результат 220
```

Полиморфизм функций

```haskell
{- Параметрический полиморфизм -}
identity :: a -> a
identity x = x

map :: (a -> b) -> [a] -> [b]

{- Ограниченный (ад-хок) полиморфизм -}
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

maximum :: Ord a => [a] -> a

{- Полиморфизм более высокого рода -}
fmap :: Functor f => (a -> b) -> f a -> f b
```

Условные конструкции

```haskell
{- Конструкция if-then-else -}
absolute :: Int -> Int
absolute x = if x < 0 then -x else x

{- Конструкция case ... of -}
describeList :: [a] -> String
describeList lst = case lst of
  []      -> "Пустой список"
  [x]     -> "Список с одним элементом"
  (x:xs)  -> "Список с более чем одним элементом"

{- Охранные выражения (guards) -}
bmiCategory :: Double -> String
bmiCategory bmi
  | bmi < 18.5 = "Недостаточный вес"
  | bmi < 25.0 = "Нормальный вес"
  | bmi < 30.0 = "Избыточный вес"
  | otherwise  = "Ожирение"
```

Сопоставление с образцом (1|3)

```haskell
{- Паттерн-матчинг с примитивными типами данных -}
describeNumber :: Int -> String
describeNumber 0 = "Ноль"
describeNumber 1 = "Один"
describeNumber _ = "Другое число"

{- Паттерн-матчинг со списками и строками -}
describeList :: [Int] -> String
describeList []     = "Пустой список"
describeList [x]    = "Список с одном элементом: " ++ show x
describeList [x,y]  = "Список с двумя элементами: " ++ show x ++ " и " ++ show y
describeList (x:xs) = "Список со множеством элементов, начиная с " ++ show x

describeString :: String -> String
describeString "" = "Пустая строка"
describeString [x] = "Одиночный символ: " ++ [x]
describeString (x:xs) = "Первый символ длинной строки: " ++ [x]
```

Сопоставление с образцом (2|3)

```haskell
{- Паттерн-матчинг с кортежами -}
addVectors :: (Int, Int) -> (Int, Int) -> (Int, Int)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

describeTriple :: (Int, Int, Int) -> String
describeTriple (0, 0, 0) = "Все нули!"
describeTriple (x, 0, 0) = "Первое число не ноль"
describeTriple (0, y, 0) = "Второе число не ноль"
describeTriple (0, 0, z) = "Третье число не ноль"
describeTriple _         = "Смешанные значения"
```

Сопоставление с образцом (3|3)

```haskell
{- Паттерн-матчинг с пользовательскими типами -}
data Maybe a = Just a | Nothing
describeMaybe :: Maybe Int -> String
describeMaybe Nothing = "Нет значения"
describeMaybe (Just x) = "Значение: " ++ show x

data Shape = Circle Float | Rectangle Float Float
area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rectangle w h) = w * h
```

Рекурсия

```haskell
{- Прямая рекурсия -}
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
factorial 5     -- Результат 120

{- Хвостовая рекурсия -}
factorialTail :: Integer -> Integer
factorialTail n = go n 1
  where
    go 0 acc = acc
    go n acc = go (n - 1) (n * acc)
factorialTail 5 -- Результат 120
```

```haskell
{- Косвенная рекурсия -}
isEven :: Integer -> Bool
isEven 0 = True
isEven n = isOdd (n - 1)

isOdd :: Integer -> Bool
isOdd 0 = False
isOdd n = isEven (n - 1)

isEven 4 -- Результат True
isOdd 3  -- Результат False
```

Функции высшего порядка (1|4)

```haskell
{- Функция map -}
map :: (a -> b) -> [a] -> [b]
-- (a -> b) — функция, которая преобразует элемент типа a в элемент типа b.
-- [a] — исходный список элементов типа a.
-- [b] — новый список элементов типа b.

-- Пример
doubleList :: [Int] -> [Int]
doubleList = map (* 2)  -- Удваивает каждый элемент в списке
doubleList [1, 2, 3]    -- Результат [2, 4, 6]
```

Функции высшего порядка (2|4)

```haskell
{- Функция filter -}
filter :: (a -> Bool) -> [a] -> [a]
-- (a -> Bool) — предикат, который проверяет выполнение условия для каждого элемента.
-- [a] — исходный список.
-- [a] — новый список, содержащий только те элементы, для которых предикат вернул True.

-- Пример
evenNumbers :: [Int] -> [Int]
evenNumbers = filter even    -- Оставляет только четные числа
evenNumbers [1, 2, 3, 4, 5]  -- Результат [2, 4]
```

Функции высшего порядка (3|4)

```haskell
{- Функции foldr и foldl -}
foldr :: (a -> b -> b) -> b -> [a] -> b
-- (a -> b -> b) — функция, которая принимает текущий элемент списка a
--                 и промежуточное значение b, возвращая обновленное значение b.
-- b — начальное значение аккумулятора.
-- [a] — исходный список.
-- b — окончательное свернутое значение.

-- Пример foldr
sumList :: [Int] -> Int
sumList = foldr (+) 0   -- Складывает все элементы списка
sumList [1, 2, 3, 4]    -- Результат: 10

-- Пример foldl
productList :: [Int] -> Int
productList = foldl (*) 1   -- Умножает все элементы списка
productList [1, 2, 3, 4]    -- Результат: 24
```

Функции высшего порядка (4|4)

```haskell
{- Функция zipWith -}
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
-- (a -> b -> c) — функция, которая объединяет элементы двух списков.
-- [a] и [b] — исходные списки.
-- [c] — новый список, полученный объединением.

-- Пример
addLists :: [Int] -> [Int] -> [Int]
addLists = zipWith (+)        -- Складывает соответствующие элементы двух списков
addLists [1, 2, 3] [4, 5, 6]  -- Результат: [5, 7, 9]
```