МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет прикладной математики и информатики

# ОТЧЁТ

по лабораторной работе № 3

«Функции высшего порядка: map, filter, fold, currying»

по дисциплине «Функциональное программирование»

Выполнил: студент группы ПМИМ-61

Резанко А. А.

Проверил: _____

Новосибирск, 2026

# Лабораторная работа № 3

## Функции высшего порядка: map, filter, fold, currying

### Цель работы

Освоить функции высшего порядка – принимающие или возвращающие другие функции, – а также частичное применение и композицию.

### Теоретические сведения

Функции в Haskell – полноправные значения: их можно передавать аргументами, возвращать из других функций и хранить в структурах данных. Функция, которая этим пользуется, называется функцией высшего порядка; типичные представители – отображение списка, фильтрация и свёртка.

Каждая функция нескольких аргументов на деле принимает один аргумент и возвращает функцию от оставшихся – это каррирование. Отсюда частичное применение: передав часть аргументов, получаем готовую функцию от остальных. Композиция соединяет две функции в одну, применяя их справа налево, и вместе с частичным применением позволяет определять функции, не упоминая их аргументов.

### Средства и порядок выполнения

Компилятор GHC 9.4.7, операционная система Ubuntu 24.04. Исходный текст программы размещён в файле Lab3.hs и приведён полностью в приложении А. Сборка и запуск:

```
ghc -O0 -Wall Lab3.hs -o lab3
./lab3
```

Ключ -Wall включает полный набор предупреждений компилятора; сборка проходит без замечаний. В программу встроено 28 проверок: каждая сверяет результат функции с ожидаемым значением и печатает отметку в протоколе. Вместо экранных снимков в отчёте приведён текстовый протокол

консольного вывода – он воспроизводится запуском программы и содержит те же данные.

## Ход работы

### 3.1 Каррирование и частичное применение

Протокол GHCi показывает каррирование прямо в типах: у трёхместного умножения три стрелки, после подстановки одного аргумента остаётся две, после двух – одна. Никакого особого синтаксиса частичное применение не требует: достаточно передать меньше аргументов, чем в сигнатуре.

Именованные частично применённые версии определены без упоминания оставшихся параметров. Лямбда-выражение даёт тот же результат, что и обычное определение: запись через лямбду и через уравнение с аргументами равносильны.

Практическая польза видна в последней строке протокола: частично применённая функция передаётся в отображение списка как готовый одноместный преобразователь.

Протокол сеанса GHCi:

```
ghci> :type multiply
multiply :: Num a => a -> a -> a -> a
ghci> :type multiply 10
multiply 10 :: Num a => a -> a -> a
ghci> :type multiply 10 2
multiply 10 2 :: Num a => a -> a
ghci> multiply 10 2 7
140
ghci> :type add3 1 2
add3 1 2 :: Num a => a -> a
ghci> add3 1 2 10
13
ghci> :type map
map :: (a -> b) -> [a] -> [b]
ghci> :type filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> :type foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
ghci> :type foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
ghci> :type foldl1
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
ghci> :type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
ghci> :type flip
flip :: (a -> b -> c) -> b -> a -> c
ghci> :type doubleList
doubleList :: [Int] -> [Int]
```

```
ghci> :type (> 0)
(> 0) :: (Ord a, Num a) => a -> Bool
```

Листинг:

```
-- Все функции в Haskell каррированы: multiply принимает один аргумент
-- и возвращает функцию от оставшихся двух.
multiply :: Num a => a -> a -> a -> a
multiply x y z = x * y * z

-- Частично применённые версии
mulBy10 :: Num a => a -> a -> a
mulBy10 = multiply 10

mulBy10And2 :: Num a => a -> a
mulBy10And2 = multiply 10 2

-- Лямбда-версия трёхместного сложения
add3 :: Num a => a -> a -> a -> a
add3 = \x y z -> x + y + z
```

Результат работы:

```
multiply 2 3 4     = 24
mulBy10 3 4        = 120
mulBy10And2 7      = 140
add3 1 2 10        = 13
map (multiply 10 2) [1,2,3] = [20,40,60]
```

### 3.2 Отображение списка

Отображение применяет функцию к каждому элементу, сохраняя длину и порядок. Все три функции определены в бесточечном стиле – через частичное применение, без явного параметра-списка.

Перевод строк в верхний регистр использует два вложенных отображения: внешнее идёт по списку строк, внутреннее – по символам каждой строки. Это прямое следствие того, что строка сама является списком.

Извлечение первых компонент пар сводится к отображению стандартного селектора и работает для пар с компонентами любых типов.

Листинг:

```
doubleList :: [Int] -> [Int]
doubleList = map (* 2)

-- Внешний map идёт по списку строк, внутренний – по символам строки
toUpperAll :: [String] -> [String]
toUpperAll = map (map toUpper)

firsts :: [(a, b)] -> [a]
firsts = map fst
```

Результат работы:

```
doubleList [1..5]  = [2,4,6,8,10]
toUpperAll         = ["MAP","FOLD","CURRY"]
firsts             = [1,2,3]
```

### 3.3 Фильтрация

Фильтрация оставляет элементы, удовлетворяющие предикату. Предикаты заданы сечениями операций сравнения: сечение – это частично применённая инфиксная операция, готовый одноместный предикат.

Отбор длинных слов показывает композицию внутри предиката: сначала вычисляется длина слова, затем результат сравнивается с порогом. Функции применяются справа налево.

Проверка простоты вынесена в отдельную функцию: перебираются делители до квадратного корня из числа, и наличие хотя бы одного делает число составным. Числа меньше двух отсекаются сразу – без этого единица считалась бы простой, поскольку делителей в пустом диапазоне нет. Проверка на числе 91, равном произведению семи и тринадцати, подтверждает, что перебор до корня достаточен.

Листинг:

```
positives :: [Int] -> [Int]
positives = filter (> 0)

longWords :: [String] -> [String]
longWords = filter ((> 5) . length)

-- Делители перебираются до корня из числа
isPrime :: Int -> Bool
isPrime n
  | n < 2     = False
  | otherwise = null [d | d <- [2 .. isqrt n], n `mod` d == 0]
  where
    isqrt = floor . sqrt . (fromIntegral :: Int -> Double)

primeNumbers :: [Int] -> [Int]
primeNumbers = filter isPrime
```

Результат работы:

```
  исходный список    = [-5,3,0,7,-2,11,4,13]
  positives          = [3,7,11,4,13]
  primeNumbers       = [3,7,11,13]
  longWords          = ["функтор", "рекурсия"]
```

### 3.4 Свёртки

Свёртка заменяет список одним значением, последовательно комбинируя элементы с накопителем. Правая свёртка группирует операции справа налево, левая – слева направо.

На сложении обе дают один результат, поскольку сложение ассоциативно и ноль нейтрален. На вычитании результаты расходятся:

правая свёртка на списке из трёх элементов даёт 2, левая – минус 6. Это включено в проверки как подтверждение различия направлений.

Два варианта разворота показывают разницу в стоимости. Версия на правой свёртке дописывает элемент в конец накопителя конкатенацией и работает квадратично; версия на левой свёртке присоединяет элемент к началу операцией cons с переставленными аргументами и работает линейно. Второй вариант – это в точности накопительный разворот из лабораторной работы № 2, записанный свёрткой.

Максимум определён через свёртку, берущую начальным значением первый элемент, поэтому на пустом списке функция не определена – в отличие от свёрток с явным начальным значением.

Последняя функция раздела демонстрирует ленивость правой свёртки: комбинирующая операция не требует накопителя, если первый аргумент уже определяет результат, поэтому вычисление обрывается на четвёртом элементе даже для бесконечного списка. Левая свёртка так не умеет: она обязана дойти до конца списка.

Листинг:

```
-- Правая свёртка: f x1 (f x2 (... (f xn z)))
sumFR :: Num a => [a] -> a
sumFR = foldr (+) 0

-- Левая свёртка: f (... (f (f z x1) x2)) xn
sumFL :: Num a => [a] -> a
sumFL = foldl (+) 0

lengthFR :: [a] -> Int
lengthFR = foldr (\_ acc -> acc + 1) 0

-- Разворот правой свёрткой: дописывание в конец даёт квадратичную стоимость
reverseFR :: [a] -> [a]
reverseFR = foldr (\x acc -> acc ++ [x]) []

-- Разворот левой свёрткой: накопитель наращивается за постоянное время
reverseFL :: [a] -> [a]
reverseFL = foldl (flip (:)) []

-- foldl1 берёт первый элемент за начальное значение, поэтому на пустом
-- списке функция не определена
myMaximum :: Ord a => [a] -> a
myMaximum = foldl1 max

-- Правая свёртка не обязана доходить до конца списка: если комбинирующая
-- функция не требует накопитель, вычисление обрывается. Отсюда работа
-- с бесконечными списками.
anyGreaterThan3 :: [Int] -> Bool
anyGreaterThan3 = foldr (\x acc -> x > 3 || acc) False
```

Результат работы:

```
список            = [1,2,3,4,5]
sumFR             = 15
sumFL             = 15
lengthFR          = 5
reverseFR         = [5,4,3,2,1]
reverseFL         = [5,4,3,2,1]
myMaximum         = 5
foldr (-) 0 [1,2,3] = 2
foldl (-) 0 [1,2,3] = -6
anyGreaterThan3 [1..] = True (на бесконечном списке)
```

### 3.5 Композиция функций

Бесточечный стиль описывает функцию как конвейер преобразований, не называя обрабатываемое значение. Композиция читается справа налево: в подсчёте чётных сначала выполняется фильтрация, затем измеряется длина результата.

Показатель степени в квадрате длины строки аннотирован типом Int – иначе ключ -Wall сообщает о неоднозначности типа литерала.

Листинг:

```
squareLen :: String -> Int
squareLen = (^ (2 :: Int)) . length

totalLen :: [String] -> Int
totalLen = sum . map length

countEvens :: [Int] -> Int
countEvens = length . filter even
```

Результат работы:

```
squareLen "haskell" = 49
totalLen            = 10
countEvens          = 3
```

### 3.6 Собственные функции высшего порядка

Три стандартные функции реализованы вручную, чтобы увидеть их устройство. Отображение и фильтрация – прямая рекурсия по списку, различие лишь в том, что фильтрация решает, включать ли элемент, охраняющим условием.

Собственная правая свёртка показывает, откуда берётся направление: комбинирующая функция применяется к голове и к результату свёртки хвоста, поэтому сначала раскрывается весь список, а собирается результат с конца.

Последняя строка протокола иллюстрирует связь свёртки со структурой списка: свёртка с операцией cons и пустым списком в качестве

начального значения восстанавливает исходный список. Список и есть правая свёртка самого себя.

Листинг:

```
myMap :: (a -> b) -> [a] -> [b]
myMap _ []     = []
myMap f (x:xs) = f x : myMap f xs

myFilter :: (a -> Bool) -> [a] -> [a]
myFilter _ [] = []
myFilter p (x:xs)
  | p x       = x : myFilter p xs
  | otherwise = myFilter p xs

myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr _ z []     = z
myFoldr f z (x:xs) = f x (myFoldr f z xs)
```

Результат работы:

```
myMap (*3) [1..5]  = [3,6,9,12,15]
myFilter odd [1..10] = [1,3,5,7,9]
myFoldr (+) 0 [1..5] = 15
myFoldr (:) [] [1..5] = [1,2,3,4,5]
```

## Тестирование

Полный вывод программы за один запуск, включая встроенные проверки:

```
=== 3.1. Каррирование и частичное применение ===
  multiply 2 3 4     = 24
  mulBy10 3 4        = 120
  mulBy10And2 7      = 140
  add3 1 2 10        = 13
  map (multiply 10 2) [1,2,3] = [20,40,60]

=== 3.2. map ===
  doubleList [1..5]  = [2,4,6,8,10]
  toUpperAll         = ["MAP","FOLD","CURRY"]
  firsts             = [1,2,3]

=== 3.3. filter ===
  исходный список    = [-5,3,0,7,-2,11,4,13]
  positives          = [3,7,11,4,13]
  primeNumbers       = [3,7,11,13]
  longWords          = ["функтор", "рекурсия"]

=== 3.4. foldr и foldl ===
  список             = [1,2,3,4,5]
  sumFR              = 15
  sumFL              = 15
  lengthFR           = 5
  reverseFR          = [5,4,3,2,1]
  reverseFL          = [5,4,3,2,1]
  myMaximum          = 5
  foldr (-) 0 [1,2,3] = 2
  foldl (-) 0 [1,2,3] = -6
  anyGreaterThan3 [1..] = True (на бесконечном списке)

=== 3.5. Композиция функций ===
  squareLen "haskell" = 49
  totalLen           = 10
  countEvens         = 3

=== 3.6. Свои функции высшего порядка ===
  myMap (*3) [1..5]  = [3,6,9,12,15]
  myFilter odd [1..10] = [1,3,5,7,9]
  myFoldr (+) 0 [1..5] = 15
  myFoldr (:) [] [1..5] = [1,2,3,4,5]

=== Самопроверка ===
  [OK  ] multiply 2 3 4 == 24
  [OK  ] частичное применение эквивалентно полному
  [OK  ] mulBy10And2 7 == 140
  [OK  ] doubleList совпадает с поэлементным удвоением
  [OK  ] toUpperAll переводит в верхний регистр
  [OK  ] firsts совпадает с map fst
  [OK  ] positives оставляет строго положительные
  [OK  ] longWords отбирает слова длиннее пяти
  [OK  ] isPrime 13
  [OK  ] not (isPrime 1)
  [OK  ] not (isPrime 91)
  [OK  ] primeNumbers отбирает простые
  [OK  ] sumFR совпадает с sum
  [OK  ] sumFL совпадает с sum
  [OK  ] sumFR и sumFL совпадают на сложении
  [OK  ] foldr и foldl расходятся на вычитании
  [OK  ] lengthFR совпадает с length
  [OK  ] reverseFR совпадает с reverse
  [OK  ] reverseFL совпадает с reverse
  [OK  ] myMaximum совпадает с maximum
  [OK  ] правая свёртка обрывается на бесконечном списке
  [OK  ] squareLen "haskell" == 49
  [OK  ] totalLen считает суммарную длину
```

```
  [OK  ] countEvens считает чётные
  [OK  ] myMap совпадает с map
  [OK  ] myFilter совпадает с filter
  [OK  ] myFoldr совпадает с foldr
  [OK  ] myFoldr (:) [] восстанавливает список

Пройдено 28 из 28 проверок.
```

Все 28 проверок пройдены (28 из 28), расхождений с ожидаемыми значениями нет.

**Ответы на контрольные вопросы**

**1. В чём разница между правой и левой свёрткой? Когда лучше использовать каждую?**

Различие в порядке группировки. Правая свёртка раскрывает список от головы и применяет операцию к элементу и результату свёртки хвоста; левая накапливает значение слева направо, передавая накопитель первым аргументом. На ассоциативной операции с нейтральным элементом результаты совпадают, на неассоциативной – нет: в протоколе вычитание даёт 2 и минус 6.

Правая свёртка ленива по накопителю: если комбинирующая операция не смотрит на второй аргумент, вычисление обрывается. Поэтому она годится для бесконечных списков и для построения структур – отображение списка естественно выражается именно ею.

Левая свёртка обязана дойти до конца, зато не наращивает стек рекурсии по длине списка. Её берут для агрегирования в накопитель фиксированного размера: сумма, максимум, разворот. На практике используют строгий вариант левой свёртки, вычисляющий накопитель на каждом шаге, – обычная левая свёртка строит цепочку отложенных вычислений и на длинных списках расходует память.

**2. Что такое бесточечный стиль? В каких случаях он ухудшает читаемость?**

Бесточечный стиль – определение функции без упоминания её аргументов, только через композицию и частичное применение. Он подчёркивает, что функция есть конвейер преобразований, и избавляет от имён, которые ничего не добавляют.

Читаемость страдает, когда аргументов больше одного и их приходится переставлять или размножать вспомогательными комбинаторами: запись перестаёт читаться линейно, и восстановить порядок вычислений можно только по типам. Плохо он ведёт себя и при глубокой вложенности, когда

одна композиция передаётся аргументом другой. Практическое правило: пока конвейер одноместный и читается слева направо как перечисление шагов – стиль уместен; как только требуется комбинатор перестановки аргументов, яснее выписать аргумент явно.

**3. Объясните тип операции композиции**

Операция композиции имеет тип, принимающий функцию из промежуточного типа в итоговый, функцию из исходного типа в промежуточный, значение исходного типа, и возвращающий итоговый. Протокол GHCi подтверждает эту сигнатуру.

Читать её следует так: две функции стыкуются по промежуточному типу – выход правой обязан совпасть со входом левой. Порядок применения обратен порядку записи: сначала работает правая функция, её результат поступает в левую. Благодаря каррированию третий аргумент можно не передавать, и тогда композиция возвращает готовую функцию – на этом и строится бесточечный стиль.

**4. Как работает перестановка аргументов и зачем она нужна?**

Комбинатор перестановки принимает двухместную функцию и возвращает функцию с обменянными местами первыми двумя аргументами; его тип виден в протоколе GHCi.

Он нужен, когда частично применить требуется не первый аргумент, а второй, или когда чужая функция ожидает аргументы в неудобном порядке. Показательный случай – разворот списка левой свёрткой: свёртка передаёт накопитель первым аргументом, а операция cons ожидает его вторым, и перестановка устраняет это несоответствие без написания лямбды.

**Выводы**

Освоены функции высшего порядка: отображение, фильтрация и обе свёртки применены к типовым задачам, а затем реализованы вручную через рекурсию, что показало их внутреннее устройство.

Разобрано каррирование: тип функции в протоколе GHCi прямо показывает, что подстановка аргумента превращает трёхместную функцию в двухместную. Частичное применение и композиция позволили записать часть функций без упоминания аргументов.

На развороте списка и на свёртке бесконечного списка показано, что выбор между правой и левой свёрткой – это не стилистика, а решение о стоимости и о применимости: одна работает с бесконечными данными, другая не наращивает стек. Программа собирается без предупреждений при ключе -Wall и проходит все 28 встроенных проверок.

## ПРИЛОЖЕНИЕ А. Полный листинг программы (Lab3.hs)

```
-- Лабораторная работа № 3. Функции высшего порядка: map, filter, fold, currying
-- Резанко А. А., ПМИМ-61
--
-- Сборка:  ghc -O0 -Wall Lab3.hs -o lab3
-- Запуск:  ./lab3

module Main where

import Data.Char (toUpper)
import Data.List (intercalate)
import System.IO (hSetEncoding, stdout, utf8)
import Text.Printf (printf)

-- Стандартная функция show экранирует кириллицу числовыми кодами,
-- поэтому списки строк печатаются своей функцией.
showStrings :: [String] -> String
showStrings xs = "[" ++ intercalate ", " (map quote xs) ++ "]"
  where quote w = "\"" ++ w ++ "\""

-- ====================== 3.1. Каррирование ======================

-- Все функции в Haskell каррированы: multiply принимает один аргумент
-- и возвращает функцию от оставшихся двух.
multiply :: Num a => a -> a -> a -> a
multiply x y z = x * y * z

-- Частично применённые версии
mulBy10 :: Num a => a -> a -> a
mulBy10 = multiply 10

mulBy10And2 :: Num a => a -> a
mulBy10And2 = multiply 10 2

-- Лямбда-версия трёхместного сложения
add3 :: Num a => a -> a -> a -> a
add3 = \x y z -> x + y + z

-- ====================== 3.2. map ======================

doubleList :: [Int] -> [Int]
doubleList = map (* 2)

-- Внешний map идёт по списку строк, внутренний – по символам строки
toUpperAll :: [String] -> [String]
toUpperAll = map (map toUpper)

firsts :: [(a, b)] -> [a]
firsts = map fst

-- ====================== 3.3. filter ======================

positives :: [Int] -> [Int]
positives = filter (> 0)

longWords :: [String] -> [String]
longWords = filter ((> 5) . length)

-- Делители перебираются до корня из числа
isPrime :: Int -> Bool
isPrime n
  | n < 2     = False
  | otherwise = null [d | d <- [2 .. isqrt n], n `mod` d == 0]
  where
    isqrt = floor . sqrt . (fromIntegral :: Int -> Double)

primeNumbers :: [Int] -> [Int]
primeNumbers = filter isPrime
```

```haskell
-- ==================== 3.4. foldr и foldl ====================

-- Правая свёртка: f x1 (f x2 (... (f xn z)))
sumFR :: Num a => [a] -> a
sumFR = foldr (+) 0

-- Левая свёртка: f (... (f (f z x1) x2)) xn
sumFL :: Num a => [a] -> a
sumFL = foldl (+) 0

lengthFR :: [a] -> Int
lengthFR = foldr (\_ acc -> acc + 1) 0

-- Разворот правой свёрткой: дописывание в конец даёт квадратичную стоимость
reverseFR :: [a] -> [a]
reverseFR = foldr (\x acc -> acc ++ [x]) []

-- Разворот левой свёрткой: накопитель наращивается за постоянное время
reverseFL :: [a] -> [a]
reverseFL = foldl (flip (:)) []

-- foldl1 берёт первый элемент за начальное значение, поэтому на пустом
-- списке функция не определена
myMaximum :: Ord a => [a] -> a
myMaximum = foldl1 max

-- Правая свёртка не обязана доходить до конца списка: если комбинирующая
-- функция не требует накопитель, вычисление обрывается. Отсюда работа
-- с бесконечными списками.
anyGreaterThan3 :: [Int] -> Bool
anyGreaterThan3 = foldr (\x acc -> x > 3 || acc) False

-- ==================== 3.5. Композиция функций ====================

squareLen :: String -> Int
squareLen = (^ (2 :: Int)) . length

totalLen :: [String] -> Int
totalLen = sum . map length

countEvens :: [Int] -> Int
countEvens = length . filter even

-- ==================== 3.6. Свои функции высшего порядка ====================

myMap :: (a -> b) -> [a] -> [b]
myMap _ []     = []
myMap f (x:xs) = f x : myMap f xs

myFilter :: (a -> Bool) -> [a] -> [a]
myFilter _ [] = []
myFilter p (x:xs)
  | p x       = x : myFilter p xs
  | otherwise = myFilter p xs

myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr _ z []     = z
myFoldr f z (x:xs) = f x (myFoldr f z xs)

-- ==================== Протокол выполнения ====================

section :: String -> IO ()
section t = printf "\n=== %s ===\n" t

check :: String -> Bool -> IO Bool
check label ok = do
  printf "  [%s] %s\n" (if ok then "OK  " else "FAIL" :: String) label
  return ok

main :: IO ()
main = do
```

```
hSetEncoding stdout utf8

section "3.1. Каррирование и частичное применение"
printf "  multiply 2 3 4     = %d\n" (multiply (2 :: Int) 3 4)
printf "  mulBy10 3 4        = %d\n" (mulBy10 (3 :: Int) 4)
printf "  mulBy10And2 7      = %d\n" (mulBy10And2 (7 :: Int))
printf "  add3 1 2 10        = %d\n" (add3 (1 :: Int) 2 10)
printf "  map (multiply 10 2) [1,2,3] = %s\n"
       (show (map (multiply 10 2) [1, 2, 3 :: Int]))

section "3.2. map"
printf "  doubleList [1..5]  = %s\n" (show (doubleList [1 .. 5]))
printf "  toUpperAll         = %s\n" (show (toUpperAll ["map", "fold", "curry"]))
printf "  firsts             = %s\n"
       (show (firsts [(1 :: Int, 'a'), (2, 'b'), (3, 'c')]))

section "3.3. filter"
let nums = [-5, 3, 0, 7, -2, 11, 4, 13] :: [Int]
printf "  исходный список    = %s\n" (show nums)
printf "  positives          = %s\n" (show (positives nums))
printf "  primeNumbers       = %s\n" (show (primeNumbers nums))
printf "  longWords          = %s\n"
       (showStrings (longWords ["fold", "функтор", "map", "рекурсия", "тип"]))

section "3.4. foldr и foldl"
let ys = [1, 2, 3, 4, 5] :: [Int]
printf "  список             = %s\n" (show ys)
printf "  sumFR              = %d\n" (sumFR ys)
printf "  sumFL              = %d\n" (sumFL ys)
printf "  lengthFR           = %d\n" (lengthFR ys)
printf "  reverseFR          = %s\n" (show (reverseFR ys))
printf "  reverseFL          = %s\n" (show (reverseFL ys))
printf "  myMaximum          = %d\n" (myMaximum ys)
printf "  foldr (-) 0 [1,2,3] = %d\n" (foldr (-) 0 [1, 2, 3 :: Int])
printf "  foldl (-) 0 [1,2,3] = %d\n" (foldl (-) 0 [1, 2, 3 :: Int])
printf "  anyGreaterThan3 [1..] = %s (на бесконечном списке)\n"
       (show (anyGreaterThan3 [1 ..]))

section "3.5. Композиция функций"
printf "  squareLen \"haskell\" = %d\n" (squareLen "haskell")
printf "  totalLen           = %d\n" (totalLen ["один", "два", "три"])
printf "  countEvens         = %d\n" (countEvens nums)

section "3.6. Свои функции высшего порядка"
printf "  myMap (*3) [1..5]  = %s\n" (show (myMap (* 3) [1 .. 5 :: Int]))
printf "  myFilter odd [1..10] = %s\n" (show (myFilter odd [1 .. 10 :: Int]))
printf "  myFoldr (+) 0 [1..5] = %d\n" (myFoldr (+) 0 [1 .. 5 :: Int])
printf "  myFoldr (:) [] [1..5] = %s\n" (show (myFoldr (:) [] [1 .. 5 :: Int]))

section "Самопроверка"
results <- sequence
  [ check "multiply 2 3 4 == 24" (multiply (2 :: Int) 3 4 == 24)
  , check "частичное применение эквивалентно полному"
      (mulBy10 (3 :: Int) 4 == multiply 10 3 4)
  , check "mulBy10And2 7 == 140" (mulBy10And2 (7 :: Int) == 140)
  , check "doubleList совпадает с поэлементным удвоением"
      (doubleList [1 .. 5] == [2, 4, 6, 8, 10])
  , check "toUpperAll переводит в верхний регистр"
      (toUpperAll ["map", "fold"] == ["MAP", "FOLD"])
  , check "firsts совпадает с map fst"
      (firsts [(1 :: Int, 'a'), (2, 'b')] == [1, 2])
  , check "positives оставляет строго положительные" (all (> 0) (positives nums))
  , check "longWords отбирает слова длиннее пяти"
      (longWords ["fold", "функтор", "тип"] == ["функтор"])
  , check "isPrime 13" (isPrime 13)
  , check "not (isPrime 1)" (not (isPrime 1))
  , check "not (isPrime 91)" (not (isPrime 91))
  , check "primeNumbers отбирает простые" (primeNumbers nums == [3, 7, 11, 13])
  , check "sumFR совпадает с sum" (sumFR ys == sum ys)
  , check "sumFL совпадает с sum" (sumFL ys == sum ys)
  , check "sumFR и sumFL совпадают на сложении" (sumFR ys == sumFL ys)
```

```
  , check "foldr и foldl расходятся на вычитании"
      (foldr (-) 0 [1, 2, 3 :: Int] /= foldl (-) 0 [1, 2, 3 :: Int])
  , check "lengthFR совпадает с length" (lengthFR ys == length ys)
  , check "reverseFR совпадает с reverse" (reverseFR ys == reverse ys)
  , check "reverseFL совпадает с reverse" (reverseFL ys == reverse ys)
  , check "myMaximum совпадает с maximum" (myMaximum ys == maximum ys)
  , check "правая свёртка обрывается на бесконечном списке"
      (anyGreaterThan3 [1 ..])
  , check "squareLen \"haskell\" == 49" (squareLen "haskell" == 49)
  , check "totalLen считает суммарную длину"
      (totalLen ["один", "два", "три"] == 10)
  , check "countEvens считает чётные" (countEvens nums == 3)
  , check "myMap совпадает с map"
      (myMap (* 3) [1 .. 5 :: Int] == map (* 3) [1 .. 5])
  , check "myFilter совпадает с filter"
      (myFilter odd [1 .. 10 :: Int] == filter odd [1 .. 10])
  , check "myFoldr совпадает с foldr"
      (myFoldr (+) 0 [1 .. 5 :: Int] == foldr (+) 0 [1 .. 5])
  , check "myFoldr (:) [] восстанавливает список"
      (myFoldr (:) [] [1 .. 5 :: Int] == [1 .. 5])
  ]
printf "\nПройдено %d из %d проверок.\n"
       (length (filter id results)) (length results)
```