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

ОТЧЁТ

по лабораторной работе № 2

«Списки и рекурсия»

по дисциплине «Функциональное программирование»

Выполнил: студент группы ПМИМ-61

Резанко А. А.

Проверил: _____

Новосибирск, 2026

Лабораторная работа № 2

Списки и рекурсия

Цель работы

Освоить работу со списками – основной структурой данных Haskell – и научиться писать рекурсивные функции их обработки.

Теоретические сведения

Список в Haskell однороден: все элементы имеют один тип. Устроен он рекурсивно – это либо пустой список, либо голова, присоединённая к хвосту операцией cons. Отсюда и способ обработки: два уравнения, одно на пустой список, другое на пару головы и хвоста.

Рекурсия заменяет циклы. Базовый случай задаёт ответ на минимальном входе, рекурсивный сводит задачу к той же задаче на более коротком списке. Вычисления ленивы: элемент списка не вычисляется, пока за ним не обратились, поэтому корректны и бесконечные списки – потребитель разворачивает ровно столько, сколько ему нужно.

Средства и порядок выполнения

Компилятор GHC 9.4.7, операционная система Ubuntu 24.04. Исходный текст программы размещён в файле Lab2.hs и приведён полностью в приложении А. Сборка и запуск:

```
ghc -O0 -Wall -main-is Lab2 Lab2.hs -o lab2
./lab2
```

Ключ -Wall включает полный набор предупреждений компилятора; сборка проходит без замечаний. В программу встроено 24 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

2.1 Базовые операции со списками

Операция cons присоединяет элемент к началу списка за постоянное время, тогда как конкатенация копирует левый аргумент и потому линейна по его длине. Это различие определяет стоимость всех последующих функций.

Функции head и tail не определены на пустом списке: их тип не отражает возможность отказа, и вызов завершается ошибкой времени выполнения. В протоколе GHCi видно, что сигнатура head несёт лишь пометку о стеке вызовов, но не обязывает обработать пустой случай.

Арифметические последовательности задают списки перечислением границ; шаг определяется вторым элементом. Взятие пяти элементов бесконечной последовательности показывает ленивость: список строится по требованию.

Протокол сеанса GHCi:

```
ghci> 1 : [2, 3, 4]
[1,2,3,4]
ghci> [1, 2] ++ [3, 4]
[1,2,3,4]
ghci> head [1, 2, 3]
1
ghci> tail [1, 2, 3]
[2,3]
ghci> length [1, 2, 3]
3
ghci> reverse [1, 2, 3]
[3,2,1]
ghci> take 2 [1, 2, 3, 4]
[1,2]
ghci> drop 2 [1, 2, 3, 4]
[3,4]
ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]
ghci> [1,3..10]
[1,3,5,7,9]
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> take 5 [1..]
[1,2,3,4,5]
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci> :type head
head :: GHC.Stack.Types.HasCallStack => [a] -> a
```

Листинг:

```
listOperations :: [(String, String)]
listOperations =
  [ ("1 : [2, 3, 4]",      show (1 : [2, 3, 4 :: Int]))
```

```
    , ("[1, 2] ++ [3, 4]",   show ([1, 2] ++ [3, 4 :: Int]))
    , ("head [1, 2, 3]",     show (head [1, 2, 3 :: Int]))
    , ("tail [1, 2, 3]",     show (tail [1, 2, 3 :: Int]))
    , ("length [1, 2, 3]",   show (length [1, 2, 3 :: Int]))
    , ("reverse [1, 2, 3]",  show (reverse [1, 2, 3 :: Int]))
    , ("take 2 [1,2,3,4]",   show (take 2 [1, 2, 3, 4 :: Int]))
    , ("drop 2 [1,2,3,4]",   show (drop 2 [1, 2, 3, 4 :: Int]))
    , ("[1..10]",            show [1 .. 10 :: Int])
    , ("[1,3..10]",          show [1, 3 .. 10 :: Int])
    , ("[2,4..20]",          show [2, 4 .. 20 :: Int])
    ]
```

Результат работы:

```
1 : [2, 3, 4]          = [1,2,3,4]
[1, 2] ++ [3, 4]       = [1,2,3,4]
head [1, 2, 3]         = 1
tail [1, 2, 3]         = [2,3]
length [1, 2, 3]       = 3
reverse [1, 2, 3]      = [3,2,1]
take 2 [1,2,3,4]       = [1,2]
drop 2 [1,2,3,4]       = [3,4]
[1..10]                = [1,2,3,4,5,6,7,8,9,10]
[1,3..10]              = [1,3,5,7,9]
[2,4..20]              = [2,4,6,8,10,12,14,16,18,20]
```

2.2 Рекурсивная обработка списков

Все пять функций написаны без обращения к стандартной библиотеке – только образцы и рекурсия. Каждая устроена одинаково: уравнение для пустого списка задаёт нейтральное значение, уравнение для непустого сводит задачу к хвосту.

Нейтральные значения выбраны не произвольно: сумма пустого списка равна нулю, произведение – единице. Это единицы соответствующих операций, и благодаря им работают равенства вроде суммы объединения списков.

Разворот списка через конкатенацию дописывает по одному элементу в конец, и стоимость получается квадратичной. Рядом приведён вариант с накопителем: список проходится один раз, элементы перекладываются операцией cons, стоимость линейна. Совпадение результатов обоих вариантов и со стандартной функцией проверяется в протоколе.

Листинг:

```
-- Длина списка
myLength :: [a] -> Int
myLength []     = 0
myLength (_:xs) = 1 + myLength xs

-- Разворот списка. Каждый шаг дописывает элемент в конец через (++),
-- поэтому стоимость квадратична по длине списка.
myReverse :: [a] -> [a]
myReverse []     = []
```

```
myReverse (x:xs) = myReverse xs ++ [x]

-- Разворот с накопителем: список проходится один раз, стоимость линейна
myReverseAcc :: [a] -> [a]
myReverseAcc = go []
  where
    go acc []     = acc
    go acc (x:xs) = go (x : acc) xs

-- Проверка на палиндром
isPalindrome :: Eq a => [a] -> Bool
isPalindrome xs = xs == myReverse xs

-- Сумма элементов списка
mySum :: Num a => [a] -> a
mySum []     = 0
mySum (x:xs) = x + mySum xs

-- Произведение элементов списка
myProduct :: Num a => [a] -> a
myProduct []     = 1
myProduct (x:xs) = x * myProduct xs
```

Результат работы:

```
  исходный список      = [3,1,4,1,5,9,2,6]
  myLength             = 8
  myReverse            = [6,2,9,5,1,4,1,3]
  myReverseAcc         = [6,2,9,5,1,4,1,3]
  mySum                = 31
  myProduct            = 6480
  isPalindrome "шалаш" = True
  isPalindrome "haskell" = False
```

2.3 Фильтрация и преобразование

Отбор чётных чисел записан охраняющими условиями внутри рекурсивного уравнения: подходящий элемент присоединяется к результату обработки хвоста, неподходящий пропускается. Это ручная реализация того, что в стандартной библиотеке делает filter.

Удаление всех вхождений требует контекста Eq: тип элементов обязан поддерживать сравнение. Функция работает для любого такого типа, поэтому в протоколе она применена и к числам, и к строке – строка здесь просто список символов.

Листинг:

```
-- Оставить только чётные числа
evens :: [Int] -> [Int]
evens [] = []
evens (x:xs)
  | even x    = x : evens xs
  | otherwise = evens xs

-- Удвоить каждый элемент
doubleAll :: Num a => [a] -> [a]
doubleAll []     = []
doubleAll (x:xs) = (2 * x) : doubleAll xs

-- Удалить все вхождения элемента
```

```
removeAll :: Eq a => a -> [a] -> [a]
removeAll _ [] = []
removeAll y (x:xs)
  | x == y    = removeAll y xs
  | otherwise = x : removeAll y xs
```

Результат работы:

```
evens     [3,1,4,1,5,9,2,6] = [4,2,6]
doubleAll [3,1,4,1,5,9,2,6] = [6,2,8,2,10,18,4,12]
removeAll 1 [3,1,4,1,5,9,2,6] = [3,4,5,9,2,6]
removeAll 'o' "молоко"    = "млк"
```

2.4 Сортировка

Быстрая сортировка выражается тремя строками: опорным берётся первый элемент, выделения списка собирают меньшие и большие, результат склеивается рекурсивными вызовами. Такая запись близка к словесному описанию алгоритма.

У краткости есть цена. Хвост просматривается дважды – отдельно для меньших и для больших элементов, – а склейка результатов конкатенацией добавляет копирование. На уже отсортированном списке выбор первого элемента опорным даёт худший случай с квадратичной стоимостью, в отличие от классической версии с обменами на месте.

Условие в первом выделении нестрогое, во втором строгое: элементы, равные опорному, попадают в левую часть ровно один раз, так что длина результата совпадает с длиной исходного списка. Это проверяется отдельно, вместе с идемпотентностью сортировки.

Листинг:

```
-- Быстрая сортировка. Опорным выбирается первый элемент, список
-- разбивается двумя выделениями, порядок равных элементов сохраняется.
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) =
  let smaller = [x | x <- xs, x <= p]
      larger  = [x | x <- xs, x > p]
  in quicksort smaller ++ [p] ++ quicksort larger
```

Результат работы:

```
quicksort [3,1,4,1,5,9,2,6] = [1,1,2,3,4,5,6,9]
quicksort строк = ["айва", "банан", "груша", "яблоко"]
quicksort [] = []
```

2.5 Генерация последовательностей

Отрезок целых строится прямой рекурсией с условием остановки, когда начало превышает конец; на пустом отрезке результат пуст.

Решето Эратосфена записано как рекурсивное просеивание: голова списка заведомо проста, а из хвоста выбрасываются её кратные. Вспомогательная функция вынесена в where и не видна снаружи. Для аргументов меньше двух исходный отрезок пуст, и результат тоже пуст.

Числа Фибоначчи строятся с парой соседних значений в накопителе, поэтому список получается за линейное время. Прямая рекурсия по определению пересчитывала бы одни и те же значения многократно и работала бы экспоненциально долго. Корректность проверяется не только сверкой с эталоном, но и рекуррентным соотношением на двадцати числах.

Листинг:

```
-- Список целых от a до b включительно
range :: Int -> Int -> [Int]
range a b
  | a > b     = []
  | otherwise = a : range (a + 1) b

-- Простые числа до n решетом Эратосфена
primes :: Int -> [Int]
primes n = sieve [2 .. n]
  where
    sieve []     = []
    sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]

-- Первые n чисел Фибоначчи. Пара соседних чисел передаётся в накопителе,
-- поэтому список строится за линейное время, а не за экспоненциальное,
-- как при прямой рекурсии по определению.
fibList :: Int -> [Int]
fibList n = go n 0 1
  where
    go k a b
      | k <= 0    = []
      | otherwise = a : go (k - 1) b (a + b)
```

Результат работы:

```
  range 3 12  = [3,4,5,6,7,8,9,10,11,12]
  range 5 5   = [5]
  range 7 2   = []
  primes 50   = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
  primes 1    = []
  fibList 15  = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
```

Тестирование

Полный вывод программы за один запуск, включая встроенные проверки:

```
=== 2.1. Базовые операции со списками ===
  1 : [2, 3, 4]          = [1,2,3,4]
  [1, 2] ++ [3, 4]       = [1,2,3,4]
  head [1, 2, 3]         = 1
  tail [1, 2, 3]         = [2,3]
  length [1, 2, 3]       = 3
  reverse [1, 2, 3]      = [3,2,1]
  take 2 [1,2,3,4]       = [1,2]
  drop 2 [1,2,3,4]       = [3,4]
  [1..10]                = [1,2,3,4,5,6,7,8,9,10]
  [1,3..10]              = [1,3,5,7,9]
  [2,4..20]              = [2,4,6,8,10,12,14,16,18,20]

=== 2.2. Рекурсивная обработка списков ===
  исходный список      = [3,1,4,1,5,9,2,6]
  myLength             = 8
  myReverse            = [6,2,9,5,1,4,1,3]
  myReverseAcc         = [6,2,9,5,1,4,1,3]
  mySum                = 31
  myProduct            = 6480
  isPalindrome "шалаш" = True
  isPalindrome "haskell" = False

=== 2.3. Фильтрация и преобразование ===
  evens     [3,1,4,1,5,9,2,6] = [4,2,6]
  doubleAll [3,1,4,1,5,9,2,6] = [6,2,8,2,10,18,4,12]
  removeAll 1 [3,1,4,1,5,9,2,6] = [3,4,5,9,2,6]
  removeAll 'о' "молоко"    = "млк"

=== 2.4. Сортировка ===
  quicksort [3,1,4,1,5,9,2,6] = [1,1,2,3,4,5,6,9]
  quicksort строк = ["айва", "банан", "груша", "яблоко"]
  quicksort [] = []

=== 2.5. Генерация последовательностей ===
  range 3 12  = [3,4,5,6,7,8,9,10,11,12]
  range 5 5   = [5]
  range 7 2   = []
  primes 50   = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
  primes 1    = []
  fibList 15  = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]

=== Самопроверка ===
  [OK  ] myLength совпадает с length
  [OK  ] myLength [] == 0
  [OK  ] myReverse совпадает с reverse
  [OK  ] myReverseAcc даёт тот же результат
  [OK  ] двойной разворот возвращает исходный список
  [OK  ] isPalindrome "шалаш"
  [OK  ] not (isPalindrome "haskell")
  [OK  ] mySum совпадает с sum
  [OK  ] myProduct совпадает с product
  [OK  ] myProduct [] == 1
  [OK  ] evens оставляет только чётные
  [OK  ] evens совпадает с filter even
  [OK  ] doubleAll удваивает
  [OK  ] removeAll убирает все вхождения
  [OK  ] removeAll сохраняет остальные
  [OK  ] quicksort сортирует
  [OK  ] quicksort сохраняет длину
  [OK  ] quicksort идемпотентен
  [OK  ] range 3 12 задаёт отрезок
  [OK  ] range на пустом отрезке
  [OK  ] primes 50 совпадает с эталоном
  [OK  ] primes 1 пуст
```

```
  [OK  ] fibList 10 совпадает с эталоном
  [OK  ] каждое число Фибоначчи – сумма двух предыдущих

Пройдено 24 из 24 проверок.
```

Все 24 проверок пройдены (24 из 24), расхождений с ожидаемыми значениями нет.

Ответы на контрольные вопросы

1. Почему разворот списка через конкатенацию имеет квадратичную сложность? Как его оптимизировать?

Конкатенация копирует левый аргумент: её стоимость пропорциональна длине первого списка. В рекурсивном развороте на каждом шаге уже развёрнутый хвост становится левым аргументом, а справа дописывается один элемент. Для списка длины n хвосты имеют длины 0, 1, …, n−1, и суммарная стоимость составляет n(n−1)/2 операций, то есть растёт квадратично.

Оптимизация – накопитель. Результат переносится во второй аргумент вспомогательной функции, и очередной элемент присоединяется к нему операцией cons за постоянное время. Список проходится один раз, стоимость линейна. В работе такой вариант реализован рядом с исходным, и проверка подтверждает совпадение результатов. Та же идея выражается левой свёрткой с операцией cons с переставленными аргументами – это разбирается в лабораторной работе № 3.

2. Что такое ленивые вычисления и как они влияют на работу с бесконечными списками?

Ленивость означает, что выражение не вычисляется в момент связывания с именем, а откладывается до момента, когда результат понадобится потребителю. Вычисление идёт до слабой головной нормальной формы – ровно настолько, чтобы стал известен внешний конструктор значения.

Для списков это значит, что хвост остаётся невычисленным, пока к нему не обратились. Поэтому бесконечный список – законное значение: взятие первых пяти элементов последовательности натуральных чисел завершается за пять шагов, как показано в протоколе GHCi. По той же причине работают короткие замыкания логических операций и обрыв правой свёртки, разобранный в следующей работе.

Обратная сторона – отложенные вычисления занимают память и могут накапливаться. Классический пример – левая свёртка без принудительного вычисления накопителя: к концу списка выстраивается цепочка отложенных операций длиной во весь список.

3. Как работает выделение списка в реализации быстрой сортировки?

Выделение списка состоит из выражения-результата, одного или нескольких генераторов и охранных условий. Генератор перебирает элементы источника, связывая каждый с именем, охранное условие отсеивает неподходящие, а выражение слева строит элемент результата.

В быстрой сортировке используются два выделения с одним генератором и одним условием каждое: первое собирает элементы, не превосходящие опорного, второе – строго большие. Такое выделение равносильно вызову filter с соответствующим предикатом, а более сложные формы с несколькими генераторами равносильны цепочке связываний в монаде списка – это показано в лабораторной работе № 5.

Выводы

Освоена рекурсивная обработка списков: девять функций написаны без обращения к стандартной библиотеке, только через сопоставление с образцом и рекурсию, и сверены с библиотечными аналогами.

На развороте списка показано, как выбор способа записи меняет асимптотику: версия с конкатенацией квадратична, версия с накопителем линейна при том же результате. Аналогично числа Фибоначчи с накопителем строятся линейно, тогда как прямая рекурсия по определению работала бы экспоненциально.

Модуль назван Lab2 и не объявляет список экспорта, поэтому его функции доступны другим модулям; в лабораторной работе № 6 они используются для расчёта статистики по файлу. Программа собирается без предупреждений при ключе -Wall и проходит все 24 встроенные проверки.

ПРИЛОЖЕНИЕ А. Полный листинг программы (Lab2.hs)

```
-- Лабораторная работа № 2. Списки и рекурсия
-- Резанко А. А., ПМИМ-61
--
-- Модуль объявлен с именем Lab2, чтобы функции можно было использовать
-- в лабораторной работе № 6 (задание 6.5).
--
-- Сборка:  ghc -O0 -Wall -main-is Lab2 Lab2.hs -o lab2
-- Запуск:  ./lab2

module Lab2 where

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

-- Стандартная функция show экранирует кириллицу числовыми кодами
-- (\1072 вместо буквы), поэтому списки строк печатаются своей функцией.
showStrings :: [String] -> String
showStrings xs = "[" ++ intercalate ", " (map quote xs) ++ "]"
  where quote w = "\"" ++ w ++ "\""

-- ===================== 2.1. Базовые операции со списками =====================

listOperations :: [(String, String)]
listOperations =
  [ ("1 : [2, 3, 4]",      show (1 : [2, 3, 4 :: Int]))
  , ("[1, 2] ++ [3, 4]",   show ([1, 2] ++ [3, 4 :: Int]))
  , ("head [1, 2, 3]",     show (head [1, 2, 3 :: Int]))
  , ("tail [1, 2, 3]",     show (tail [1, 2, 3 :: Int]))
  , ("length [1, 2, 3]",   show (length [1, 2, 3 :: Int]))
  , ("reverse [1, 2, 3]",  show (reverse [1, 2, 3 :: Int]))
  , ("take 2 [1,2,3,4]",   show (take 2 [1, 2, 3, 4 :: Int]))
  , ("drop 2 [1,2,3,4]",   show (drop 2 [1, 2, 3, 4 :: Int]))
  , ("[1..10]",            show [1 .. 10 :: Int])
  , ("[1,3..10]",          show [1, 3 .. 10 :: Int])
  , ("[2,4..20]",          show [2, 4 .. 20 :: Int])
  ]

-- ===================== 2.2. Рекурсивная обработка списков =====================

-- Длина списка
myLength :: [a] -> Int
myLength []     = 0
myLength (_:xs) = 1 + myLength xs

-- Разворот списка. Каждый шаг дописывает элемент в конец через (++),
-- поэтому стоимость квадратична по длине списка.
myReverse :: [a] -> [a]
myReverse []     = []
myReverse (x:xs) = myReverse xs ++ [x]

-- Разворот с накопителем: список проходится один раз, стоимость линейна
myReverseAcc :: [a] -> [a]
myReverseAcc = go []
  where
    go acc []     = acc
    go acc (x:xs) = go (x : acc) xs

-- Проверка на палиндром
isPalindrome :: Eq a => [a] -> Bool
isPalindrome xs = xs == myReverse xs

-- Сумма элементов списка
mySum :: Num a => [a] -> a
mySum []     = 0
mySum (x:xs) = x + mySum xs

-- Произведение элементов списка
```

```
myProduct :: Num a => [a] -> a
myProduct []     = 1
myProduct (x:xs) = x * myProduct xs

-- ==================== 2.3. Фильтрация и преобразование ====================

-- Оставить только чётные числа
evens :: [Int] -> [Int]
evens [] = []
evens (x:xs)
  | even x    = x : evens xs
  | otherwise = evens xs

-- Удвоить каждый элемент
doubleAll :: Num a => [a] -> [a]
doubleAll []     = []
doubleAll (x:xs) = (2 * x) : doubleAll xs

-- Удалить все вхождения элемента
removeAll :: Eq a => a -> [a] -> [a]
removeAll _ [] = []
removeAll y (x:xs)
  | x == y    = removeAll y xs
  | otherwise = x : removeAll y xs

-- ==================== 2.4. Сортировка ====================

-- Быстрая сортировка. Опорным выбирается первый элемент, список
-- разбивается двумя выделениями, порядок равных элементов сохраняется.
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) =
  let smaller = [x | x <- xs, x <= p]
      larger  = [x | x <- xs, x > p]
  in quicksort smaller ++ [p] ++ quicksort larger

-- ==================== 2.5. Генерация последовательностей ====================

-- Список целых от a до b включительно
range :: Int -> Int -> [Int]
range a b
  | a > b     = []
  | otherwise = a : range (a + 1) b

-- Простые числа до n решетом Эратосфена
primes :: Int -> [Int]
primes n = sieve [2 .. n]
  where
    sieve []     = []
    sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]

-- Первые n чисел Фибоначчи. Пара соседних чисел передаётся в накопителе,
-- поэтому список строится за линейное время, а не за экспоненциальное,
-- как при прямой рекурсии по определению.
fibList :: Int -> [Int]
fibList n = go n 0 1
  where
    go k a b
      | k <= 0    = []
      | otherwise = a : go (k - 1) b (a + b)

-- ==================== 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 "2.1. Базовые операции со списками"
  mapM_ (\(e, v) -> printf "  %-22s = %s\n" e v) listOperations

  section "2.2. Рекурсивная обработка списков"
  let xs = [3, 1, 4, 1, 5, 9, 2, 6] :: [Int]
  printf "  исходный список      = %s\n" (show xs)
  printf "  myLength             = %d\n" (myLength xs)
  printf "  myReverse            = %s\n" (show (myReverse xs))
  printf "  myReverseAcc         = %s\n" (show (myReverseAcc xs))
  printf "  mySum                = %d\n" (mySum xs)
  printf "  myProduct            = %d\n" (myProduct xs)
  printf "  isPalindrome \"шалаш\" = %s\n" (show (isPalindrome "шалаш"))
  printf "  isPalindrome \"haskell\" = %s\n" (show (isPalindrome "haskell"))

  section "2.3. Фильтрация и преобразование"
  printf "  evens     %s = %s\n" (show xs) (show (evens xs))
  printf "  doubleAll %s = %s\n" (show xs) (show (doubleAll xs))
  printf "  removeAll 1 %s = %s\n" (show xs) (show (removeAll 1 xs))
  printf "  removeAll 'o' \"молоко\"    = \"%s\"\n" (removeAll 'o' "молоко")

  section "2.4. Сортировка"
  printf "  quicksort %s = %s\n" (show xs) (show (quicksort xs))
  printf "  quicksort строк = %s\n"
         (showStrings (quicksort ["груша", "яблоко", "банан", "айва"]))
  printf "  quicksort [] = %s\n" (show (quicksort ([] :: [Int])))

  section "2.5. Генерация последовательностей"
  printf "  range 3 12  = %s\n" (show (range 3 12))
  printf "  range 5 5   = %s\n" (show (range 5 5))
  printf "  range 7 2   = %s\n" (show (range 7 2))
  printf "  primes 50   = %s\n" (show (primes 50))
  printf "  primes 1    = %s\n" (show (primes 1))
  printf "  fibList 15  = %s\n" (show (fibList 15))

  section "Самопроверка"
  results <- sequence
    [ check "myLength совпадает с length" (myLength xs == length xs)
    , check "myLength [] == 0" (myLength ([] :: [Int]) == 0)
    , check "myReverse совпадает с reverse" (myReverse xs == reverse xs)
    , check "myReverseAcc даёт тот же результат" (myReverseAcc xs == myReverse xs)
    , check "двойной разворот возвращает исходный список"
        (myReverse (myReverse xs) == xs)
    , check "isPalindrome \"шалаш\"" (isPalindrome "шалаш")
    , check "not (isPalindrome \"haskell\")" (not (isPalindrome "haskell"))
    , check "mySum совпадает с sum" (mySum xs == sum xs)
    , check "myProduct совпадает с product" (myProduct xs == product xs)
    , check "myProduct [] == 1" (myProduct ([] :: [Int]) == 1)
    , check "evens оставляет только чётные" (all even (evens xs))
    , check "evens совпадает с filter even" (evens xs == filter even xs)
    , check "doubleAll удваивает" (doubleAll xs == map (2 *) xs)
    , check "removeAll убирает все вхождения" (notElem 1 (removeAll 1 xs))
    , check "removeAll сохраняет остальные"
        (myLength (removeAll 1 xs) == myLength xs - 2)
    , check "quicksort сортирует" (quicksort xs == [1, 1, 2, 3, 4, 5, 6, 9])
    , check "quicksort сохраняет длину" (myLength (quicksort xs) == myLength xs)
    , check "quicksort идемпотентен" (quicksort (quicksort xs) == quicksort xs)
    , check "range 3 12 задаёт отрезок" (range 3 12 == [3 .. 12])
    , check "range на пустом отрезке" (null (range 7 2))
    , check "primes 50 совпадает с эталоном"
        (primes 50 == [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47])
    , check "primes 1 пуст" (null (primes 1))
    , check "fibList 10 совпадает с эталоном"
        (fibList 10 == [0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
    , check "каждое число Фибоначчи – сумма двух предыдущих"
        (let f = fibList 20
         in and (zipWith3 (\a b c -> a + b == c) f (drop 1 f) (drop 2 f)))
    ]
```

```
printf "\nПройдено %d из %d проверок.\n"
       (length (filter id results)) (length results)
```