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

ОТЧЁТ

по лабораторной работе № 5
«Функторы, аппликативные функторы и монады»
по дисциплине «Функциональное программирование»

Выполнил: студент группы ПМИМ-61
Резанко А. А.

Проверил: _____

Новосибирск, 2026

Лабораторная работа № 5

Функторы, аппликативные функторы и монады

Цель работы

Разобрать три ключевые абстракции функционального программирования – функтор, аппликативный функтор и монаду – на примерах собственного типа, типа Maybe и списка.

Теоретические сведения

Функтор – тип-контейнер, для которого определено отображение внутреннего значения: функция применяется к содержимому, а форма контейнера сохраняется. Аппликативный функтор добавляет две возможности: поместить обычное значение в контейнер и применить функцию, которая сама лежит в контейнере. Монада идёт дальше: следующий шаг вычисления может зависеть от результата предыдущего, поэтому в ней выражают последовательные вычисления с отказами и эффектами.

Три класса образуют иерархию: каждый следующий требует предыдущего. Их операции связаны законами – отображение тождественной функции ничего не меняет, отображение композиции равно композиции отображений, а связывание ассоциативно и имеет нейтральный элемент. Законы не проверяются компилятором, их соблюдает автор экземпляра.

Средства и порядок выполнения

Компилятор GHC 9.4.7, операционная система Ubuntu 24.04. Исходный текст программы размещён в файле Lab5.hs и приведён полностью в приложении А. Сборка и запуск:

```
ghc -O0 -Wall Lab5.hs -o lab5
./lab5
```

Ключ -Wall включает полный набор предупреждений компилятора; сборка проходит без замечаний. В программу встроено 25 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

5.1 Функтор

Экземпляр функтора для контейнера с одним значением разбирает его образцом, применяет функцию и упаковывает результат обратно. Тип отображения допускает смену типа содержимого: применение функции длины к контейнеру со строкой даёт контейнер с числом.

Для дерева отображение рекурсивно: в листе функция применяется к значению, в узле – к обоим поддеревьям. Форма дерева при этом сохраняется, меняются только значения в листьях; отдельная проверка сверяет результат с деревом, выписанным вручную.

Дерево объявлено в этом файле заново, хотя такой же тип определён в лабораторной работе № 4: каждая работа собирается отдельной программой. В реальном проекте тип вынесли бы в общий модуль, как это сделано с функциями лабораторной работы № 2 для работы № 6.

Листинг:

```
data Box a = Box a
  deriving (Show, Eq)

instance Functor Box where
  fmap f (Box x) = Box (f x)

-- Дерево из лабораторной работы № 4 объявлено заново, чтобы файл
-- оставался самостоятельным; в проекте его вынесли бы в общий модуль.
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Show, Eq)

instance Functor Tree where
  fmap f (Leaf x)   = Leaf (f x)
  fmap f (Node l r) = Node (fmap f l) (fmap f r)

sampleTree :: Tree Int
sampleTree = Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4))
```

Результат работы:

```
  fmap (*2) (Box 5)       = Box 10
  fmap length (Box "hello") = Box 5
  fmap (+1) sampleTree    = Node (Node (Leaf 2) (Leaf 3)) (Node (Leaf 4) (Leaf 5))
  fmap show sampleTree    =
```

```
  Node (Node (Leaf "1") (Leaf "2")) (Node (Leaf "3") (Leaf "4"))
fmap (*2) (Just 21)     = Just 42
fmap (*2) Nothing       = Nothing
```

5.2 Аппликативный функтор

Аппликативный экземпляр задаёт упаковку значения и применение упакованной функции к упакованному аргументу. Благодаря каррированию цепочка таких применений позволяет работать с функциями нескольких аргументов: двухместное сложение, упакованное в контейнер, применяется последовательно к двум упакованным числам.

На типе Maybe виден смысл контекста: если хотя бы один участник – отсутствующее значение, результат тоже отсутствует, причём проверять это вручную не требуется. Для списка контекст означает множественность: применение перебирает все сочетания, и пара из двухэлементного списка и двухсимвольной строки даёт четыре результата.

Листинг:

```
instance Applicative Box where
  pure = Box
  Box f <*> Box x = Box (f x)
```

Результат работы:

```
pure (*2) <*> Box 5     = Box 10
Box (+) <*> Box 3 <*> Box 4 = Box 7
Just (+3) <*> Just 5    = Just 8
Nothing <*> Just 5      = Nothing
Just (+3) <*> Nothing   = Nothing
(,) <$> [1,2] <*> "ab"  = [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

5.3 Монада Maybe

Три частичные операции – деление, квадратный корень и логарифм – возвращают не число, а значение типа Maybe: отказ отражён в типе, и обработать его обязан вызывающий.

Связывание соединяет их в цепочку. Лямбда получает результат предыдущего шага, а отсутствующее значение прерывает всю цепочку, минуя оставшиеся шаги, – ручных проверок после каждого вызова не требуется. Нотация do записывает ту же цепочку по шагам; обе версии реализованы отдельно, и проверка подтверждает их совпадение на всех шести испытательных парах.

Протокол показывает три разные причины отказа: деление на ноль, корень из отрицательного числа и логарифм нуля. Снаружи они неразличимы – результат один и тот же. Когда причина важна, вместо Maybe берут Either, как сделано в калькуляторе лабораторной работы № 6.

Листинг:

```
safeDiv :: Double -> Double -> Maybe Double
safeDiv _ 0 = Nothing
safeDiv x y = Just (x / y)

safeSqrt :: Double -> Maybe Double
safeSqrt x
  | x < 0     = Nothing
  | otherwise = Just (sqrt x)

safeLog :: Double -> Maybe Double
safeLog x
  | x <= 0    = Nothing
  | otherwise = Just (log x)

-- Цепочка через оператор связывания: лямбда принимает результат
-- предыдущего шага, Nothing прерывает вычисление целиком
safeComputeBind :: Double -> Double -> Maybe Double
safeComputeBind x y =
  safeDiv x y  >>= \d ->
  safeSqrt d   >>= \s ->
  safeLog s

-- Та же цепочка в do-нотации
safeComputeDo :: Double -> Double -> Maybe Double
safeComputeDo x y = do
  d <- safeDiv x y
  s <- safeSqrt d
  safeLog s
```

Результат работы:

```
  safeCompute   8.0  2.0 = Just 0.6931471805599453
  safeCompute   1.0  1.0 = Just 0.0
  safeCompute   1.0  0.0 = Nothing
  safeCompute  -8.0  2.0 = Nothing
  safeCompute   0.0  5.0 = Nothing
  safeCompute 100.0  4.0 = Just 1.6094379124341003
  версия на do-нотации даёт те же значения (сверено проверкой)
  причина Nothing для (1,0):  деление на ноль
  причина Nothing для (-8,2): корень из отрицательного
  причина Nothing для (0,5):  логарифм нуля
```

5.4 Монада списка

Для списка связывание означает перебор: каждый шаг нотации do перебирает элементы, а результаты собираются в один список. Пустой список в ветви условия отсеивает неподходящие сочетания, потому что конкатенация с пустым списком ничего не добавляет.

Декартово произведение записывается двумя генераторами и упаковкой пары – это в точности выделение списка, только в другой нотации.

Третий пример перебирает пифагоровы тройки: три вложенных генератора с отсевом по условию.

Ограничения на диапазоны во вложенных генераторах не случайны: каждый следующий начинается с текущего значения предыдущего, что отсекает перестановки одной и той же тройки и сокращает перебор примерно в шесть раз.

Листинг:

```
-- Список как монада перебирает все сочетания; пустой список играет роль
-- отсева, поэтому условие записывается без фильтра
pairs :: [(Int, Int)]
pairs = do
  x <- [1 .. 3]
  y <- [1 .. 3]
  if x + y == 4 then return (x, y) else []

cartesian :: [a] -> [b] -> [(a, b)]
cartesian xs ys = do
  x <- xs
  y <- ys
  return (x, y)

-- Пифагоровы тройки: перебор с отсевом в той же нотации
triples :: Int -> [(Int, Int, Int)]
triples n = do
  a <- [1 .. n]
  b <- [a .. n]
  c <- [b .. n]
  if a * a + b * b == c * c then return (a, b, c) else []
```

Результат работы:

```
  pairs                  = [(1,3),(2,2),(3,1)]
  cartesian [1,2] "ab"   = [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
  cartesian [] "ab"      = []
  triples 20             =
    [(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

5.5 Собственная монада

Монадический экземпляр для контейнера состоит из одной строки: содержимое извлекается образцом и передаётся функции, которая сама вернёт контейнер. Объявить его можно лишь после экземпляров функтора и аппликативного функтора – классы образуют иерархию.

Функция упаковки в монадическом экземпляре не определяется: в современном Haskell она задана в самом классе как синоним аппликативной упаковки, и собственное определение вызывает предупреждение о неканоническом экземпляре. В методичке эта строка присутствует, здесь она убрана – в листинге раздела 5.5 приведено обоснование.

Нотация do работает для нового типа сразу после объявления экземпляра: в протоколе блок с двумя связываниями и упаковкой результата даёт ожидаемое значение, хотя ничего специально для этого не писалось.

Листинг:

```
-- return отдельно не определяется: в современном Haskell он задан в классе
-- как синоним pure, и собственное определение вызывает предупреждение
-- о неканоническом экземпляре.
instance Monad Box where
  Box x >>= f = f x
```

Результат работы:

```
Box 5 >>= \x -> Box (x*2) = Box 10
do x <- Box 3; y <- Box 4; return (x+y) = Box 7
цепочка из трёх шагов   = Box 50
```

Тестирование

Полный вывод программы за один запуск, включая встроенные проверки:

```
=== 5.1. Функтор ===
  fmap (*2) (Box 5)       = Box 10
  fmap length (Box "hello") = Box 5
  fmap (+1) sampleTree    = Node (Node (Leaf 2) (Leaf 3)) (Node (Leaf 4) (Leaf 5))
  fmap show sampleTree    =
    Node (Node (Leaf "1") (Leaf "2")) (Node (Leaf "3") (Leaf "4"))
  fmap (*2) (Just 21)     = Just 42
  fmap (*2) Nothing       = Nothing

=== 5.2. Аппликативный функтор ===
  pure (*2) <*> Box 5     = Box 10
  Box (+) <*> Box 3 <*> Box 4 = Box 7
  Just (+3) <*> Just 5    = Just 8
  Nothing <*> Just 5      = Nothing
  Just (+3) <*> Nothing   = Nothing
  (,) <$> [1,2] <*> "ab"  = [(1,'a'),(1,'b'),(2,'a'),(2,'b')]

=== 5.3. Монада Maybe ===
  safeCompute   8.0  2.0 = Just 0.6931471805599453
  safeCompute   1.0  1.0 = Just 0.0
  safeCompute   1.0  0.0 = Nothing
  safeCompute  -8.0  2.0 = Nothing
  safeCompute   0.0  5.0 = Nothing
  safeCompute 100.0  4.0 = Just 1.6094379124341003
  версия на do-нотации даёт те же значения (сверено проверкой)
  причина Nothing для (1,0):  деление на ноль
  причина Nothing для (-8,2): корень из отрицательного
  причина Nothing для (0,5):  логарифм нуля

=== 5.4. Монада списка ===
  pairs                   = [(1,3),(2,2),(3,1)]
  cartesian [1,2] "ab"    = [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
  cartesian [] "ab"       = []
  triples 20              =
    [(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]

=== 5.5. Своя монада ===
  Box 5 >>= \x -> Box (x*2) = Box 10
  do x <- Box 3; y <- Box 4; return (x+y) = Box 7
  цепочка из трёх шагов   = Box 50

=== Самопроверка ===
  [OK  ] fmap (*2) (Box 5) == Box 10
  [OK  ] первый закон функтора: fmap id == id
  [OK  ] второй закон функтора: fmap (f . g) == fmap f . fmap g
  [OK  ] fmap для дерева сохраняет структуру
  [OK  ] pure (*2) <*> Box 5 == Box 10
  [OK  ] Box (+) <*> Box 3 <*> Box 4 == Box 7
  [OK  ] Nothing <*> Just 5 == Nothing
  [OK  ] Just (+3) <*> Nothing == Nothing
  [OK  ] safeDiv 1 0 == Nothing
  [OK  ] safeSqrt (-1) == Nothing
  [OK  ] safeLog 0 == Nothing
  [OK  ] safeCompute 8 2 == log 2
  [OK  ] деление на ноль обрывает цепочку
  [OK  ] корень из отрицательного обрывает цепочку
  [OK  ] логарифм нуля обрывает цепочку
  [OK  ] версии через bind и do совпадают
  [OK  ] pairs == [(1,3),(2,2),(3,1)]
  [OK  ] cartesian даёт произведение длин
  [OK  ] cartesian с пустым списком пуст
  [OK  ] triples 20 содержит (3,4,5)
  [OK  ] Box 5 >>= умножение == Box 10
  [OK  ] левая единица: return a >>= f == f a
  [OK  ] правая единица: m >>= return == m
```

```
  [OK  ] ассоциативность связывания
  [OK  ] законы монад выполняются и для Maybe

Пройдено 25 из 25 проверок.
```

Все 25 проверок пройдены (25 из 25), расхождений с ожидаемыми значениями нет.

Ответы на контрольные вопросы

1. В чём разница между отображением, аппликативным применением и связыванием?

Отображение применяет обычную функцию к значению внутри контекста. Контекст при этом не меняется: из контейнера получается контейнер той же формы, из отсутствующего значения – отсутствующее, из списка – список той же длины.

Аппликативное применение нужно, когда функция сама находится в контексте. Оно позволяет применять функции нескольких аргументов к нескольким значениям в контексте. Существенно, что эффекты всех участников известны заранее и не зависят друг от друга: для списков это даёт перебор всех сочетаний, для Maybe – отказ, если отсутствует хотя бы один аргумент.

Связывание допускает зависимость: функция получает уже распакованное значение и сама решает, какой контекст вернуть. Поэтому только им выражаются вычисления, в которых следующий шаг определяется результатом предыдущего, – например цепочка, обрывающаяся при первом отказе, как в задании 5.3.

2. Зачем нужен Maybe, если есть исключения?

Отсутствие значения попадает в тип функции, а значит проверяется компилятором. Вызывающий обязан разобрать оба случая, и пропустить обработку молча невозможно; исключение же нигде в сигнатуре не отражено, и о нём узнают в момент падения программы.

Функция, возвращающая Maybe, остаётся чистой: тот же аргумент даёт тот же результат, значение можно передать дальше, сохранить, отложить. Исключение – эффект, и перехватывается оно только в контексте ввода-вывода, то есть заражает им всю цепочку вызовов.

Maybe композируется: цепочка из трёх частичных операций в задании 5.3 записана без единой явной проверки. Плата – отсутствие причины отказа,

поэтому там, где причину нужно сообщить, берут Either, а исключения оставляют для действительно исключительных ситуаций вроде исчерпания памяти.

3. Можно ли считать ввод-вывод монадой? Что она оборачивает?

Да, для типа ввода-вывода определены все три экземпляра, и нотация do в лабораторной работе № 6 работает именно потому, что это монада.

Оборачивает она описание действия с внешним миром, которое при выполнении даёт значение указанного типа. Значение такого типа само по себе ничего не делает – это рецепт; выполняется он рантаймом тогда, когда входит в состав точки входа программы. Поэтому действие можно передавать аргументом и складывать в список, не запуская.

Связывание задаёт порядок: второй шаг строится по результату первого, и именно оно превращает набор рецептов в последовательность. Отличие от Maybe лишь в природе контекста – там контекстом была возможность отказа, здесь взаимодействие с внешним миром.

4. Сформулируйте законы монад простыми словами

Левая единица: упаковать значение и сразу передать его функции – то же самое, что вызвать функцию напрямую. Упаковка не должна добавлять ничего своего.

Правая единица: если после вычисления только упаковать его результат, ничего не изменится. Пустой шаг в конце цепочки не влияет на результат.

Ассоциативность: способ расстановки скобок в цепочке связываний не важен – сгруппировать первые два шага или последние два даёт один результат. Важен порядок шагов, а не их группировка; благодаря этому закону нотация do читается как линейная последовательность.

Все три закона проверяются в программе для собственного типа, а для Maybe к ним добавлена проверка ассоциативности на цепочке из деления и квадратного корня.

Выводы

Разобрана иерархия из трёх классов: объявлены экземпляры функтора, аппликативного функтора и монады для собственного типа, а также функтор для рекурсивного дерева. Каждый следующий класс требует предыдущего, и порядок объявления это отражает.

На типе Maybe показано, ради чего вводится монада: цепочка из трёх частичных операций записана без явных проверок, причём отказ на любом шаге прерывает вычисление целиком. Нотация do оказалась синтаксической записью той же цепочки связываний – обе версии дают одинаковый результат на всех испытательных данных.

Монада списка показала, что тот же интерфейс описывает не только отказ, но и перебор: декартово произведение и поиск пифагоровых троек записываются связыванием так же, как выделением списка. Законы монад проверены вычислительно для двух типов. Программа собирается без предупреждений при ключе -Wall и проходит все 25 встроенных проверок.

ПРИЛОЖЕНИЕ А. Полный листинг программы (Lab5.hs)

```
-- Лабораторная работа № 5. Функторы, аппликативные функторы и монады
-- Резанко А. А., ПМИМ-61
--
-- Сборка:  ghc -O0 -Wall Lab5.hs -o lab5
-- Запуск:  ./lab5

module Main where

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

-- ====================== 5.1. Функтор ======================

data Box a = Box a
  deriving (Show, Eq)

instance Functor Box where
  fmap f (Box x) = Box (f x)

-- Дерево из лабораторной работы № 4 объявлено заново, чтобы файл
-- оставался самостоятельным; в проекте его вынесли бы в общий модуль.
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Show, Eq)

instance Functor Tree where
  fmap f (Leaf x)   = Leaf (f x)
  fmap f (Node l r) = Node (fmap f l) (fmap f r)

sampleTree :: Tree Int
sampleTree = Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4))

-- ====================== 5.2. Аппликативный функтор ======================

instance Applicative Box where
  pure = Box
  Box f <*> Box x = Box (f x)

-- ====================== 5.5. Своя монада ======================

-- return отдельно не определяется: в современном Haskell он задан в классе
-- как синоним pure, и собственное определение вызывает предупреждение
-- о неканоническом экземпляре.
instance Monad Box where
  Box x >>= f = f x

-- ====================== 5.3. Монада Maybe ======================

safeDiv :: Double -> Double -> Maybe Double
safeDiv _ 0 = Nothing
safeDiv x y = Just (x / y)

safeSqrt :: Double -> Maybe Double
safeSqrt x
  | x < 0     = Nothing
  | otherwise = Just (sqrt x)

safeLog :: Double -> Maybe Double
safeLog x
  | x <= 0    = Nothing
  | otherwise = Just (log x)

-- Цепочка через оператор связывания: лямбда принимает результат
-- предыдущего шага, Nothing прерывает вычисление целиком
safeComputeBind :: Double -> Double -> Maybe Double
safeComputeBind x y =
  safeDiv x y  >>= \d ->
  safeSqrt d   >>= \s ->
  safeLog s
```

```
-- Та же цепочка в do-нотации
safeComputeDo :: Double -> Double -> Maybe Double
safeComputeDo x y = do
  d <- safeDiv x y
  s <- safeSqrt d
  safeLog s

-- ====================== 5.4. Монада списка ======================

-- Список как монада перебирает все сочетания; пустой список играет роль
-- отсева, поэтому условие записывается без фильтра
pairs :: [(Int, Int)]
pairs = do
  x <- [1 .. 3]
  y <- [1 .. 3]
  if x + y == 4 then return (x, y) else []

cartesian :: [a] -> [b] -> [(a, b)]
cartesian xs ys = do
  x <- xs
  y <- ys
  return (x, y)

-- Пифагоровы тройки: перебор с отсевом в той же нотации
triples :: Int -> [(Int, Int, Int)]
triples n = do
  a <- [1 .. n]
  b <- [a .. n]
  c <- [b .. n]
  if a * a + b * b == c * c then return (a, b, c) else []

-- ====================== 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 "5.1. Функтор"
  printf "  fmap (*2) (Box 5)       = %s\n" (show (fmap (* 2) (Box (5 :: Int))))
  printf "  fmap length (Box \"hello\") = %s\n" (show (fmap length (Box "hello")))
  printf "  fmap (+1) sampleTree    = %s\n" (show (fmap (+ 1) sampleTree))
  printf "  fmap show sampleTree    =\n    %s\n" (show (fmap show sampleTree))
  printf "  fmap (*2) (Just 21)     = %s\n" (show (fmap (* 2) (Just (21 :: Int))))
  printf "  fmap (*2) Nothing       = %s\n"
         (show (fmap (* 2) (Nothing :: Maybe Int)))

  section "5.2. Аппликативный функтор"
  printf "  pure (*2) <*> Box 5     = %s\n"
         (show (pure (* 2) <*> Box (5 :: Int)))
  printf "  Box (+) <*> Box 3 <*> Box 4 = %s\n"
         (show (Box (+) <*> Box (3 :: Int) <*> Box 4))
  printf "  Just (+3) <*> Just 5    = %s\n"
         (show (Just (+ 3) <*> Just (5 :: Int)))
  printf "  Nothing <*> Just 5      = %s\n"
         (show ((Nothing :: Maybe (Int -> Int)) <*> Just 5))
  printf "  Just (+3) <*> Nothing   = %s\n"
         (show (Just (+ 3) <*> (Nothing :: Maybe Int)))
  printf "  (,) <$> [1,2] <*> \"ab\"  = %s\n"
         (show ((,) <$> [1, 2 :: Int] <*> "ab"))

  section "5.3. Монада Maybe"
  let cases = [(8, 2), (1, 1), (1, 0), (-8, 2), (0, 5), (100, 4)]
```

```
            :: [(Double, Double)]
  mapM_ (\(x, y) -> printf "  safeCompute %5.1f %4.1f = %s\n"
                      x y (show (safeComputeBind x y)))
        cases
  printf "  версия на do-нотации даёт те же значения (сверено проверкой)\n"
  printf "  причина Nothing для (1,0):  деление на ноль\n"
  printf "  причина Nothing для (-8,2): корень из отрицательного\n"
  printf "  причина Nothing для (0,5):  логарифм нуля\n"

  section "5.4. Монада списка"
  printf "  pairs                   = %s\n" (show pairs)
  printf "  cartesian [1,2] \"ab\"    = %s\n" (show (cartesian [1, 2 :: Int] "ab"))
  printf "  cartesian [] \"ab\"       = %s\n" (show (cartesian ([] :: [Int]) "ab"))
  printf "  triples 20              =\n    %s\n" (show (triples 20))

  section "5.5. Своя монада"
  printf "  Box 5 >>= \\x -> Box (x*2) = %s\n"
         (show (Box (5 :: Int) >>= \x -> Box (x * 2)))
  printf "  do x <- Box 3; y <- Box 4; return (x+y) = %s\n"
         (show (do { x <- Box (3 :: Int); y <- Box 4; return (x + y) }))
  printf "  цепочка из трёх шагов   = %s\n"
         (show (Box (2 :: Int) >>= \a -> Box (a + 3) >>= \b -> Box (b * 10)))

  section "Самопроверка"
  results <- sequence
    [ check "fmap (*2) (Box 5) == Box 10" (fmap (* 2) (Box (5 :: Int)) == Box 10)
    , check "первый закон функтора: fmap id == id"
        (fmap id (Box (7 :: Int)) == Box 7)
    , check "второй закон функтора: fmap (f . g) == fmap f . fmap g"
        (fmap ((+ 1) . (* 2)) (Box (5 :: Int)) == (fmap (+ 1) . fmap (* 2)) (Box
5))
    , check "fmap для дерева сохраняет структуру"
        (fmap (+ 1) sampleTree
           == Node (Node (Leaf 2) (Leaf 3)) (Node (Leaf 4) (Leaf 5)))
    , check "pure (*2) <*> Box 5 == Box 10"
        ((pure (* 2) <*> Box (5 :: Int)) == Box 10)
    , check "Box (+) <*> Box 3 <*> Box 4 == Box 7"
        ((Box (+) <*> Box (3 :: Int) <*> Box 4) == Box 7)
    , check "Nothing <*> Just 5 == Nothing"
        (((Nothing :: Maybe (Int -> Int)) <*> Just 5) == Nothing)
    , check "Just (+3) <*> Nothing == Nothing"
        ((Just (+ 3) <*> (Nothing :: Maybe Int)) == Nothing)
    , check "safeDiv 1 0 == Nothing" (safeDiv 1 0 == Nothing)
    , check "safeSqrt (-1) == Nothing" (safeSqrt (-1) == Nothing)
    , check "safeLog 0 == Nothing" (safeLog 0 == Nothing)
    , check "safeCompute 8 2 == log 2"
        (fmap (\v -> abs (v - log 2) < 1e-12) (safeComputeBind 8 2) == Just True)
    , check "деление на ноль обрывает цепочку" (safeComputeBind 1 0 == Nothing)
    , check "корень из отрицательного обрывает цепочку"
        (safeComputeBind (-8) 2 == Nothing)
    , check "логарифм нуля обрывает цепочку" (safeComputeBind 0 5 == Nothing)
    , check "версии через bind и do совпадают"
        (and [safeComputeBind x y == safeComputeDo x y | (x, y) <- cases])
    , check "pairs == [(1,3),(2,2),(3,1)]" (pairs == [(1, 3), (2, 2), (3, 1)])
    , check "cartesian даёт произведение длин"
        (length (cartesian [1, 2, 3 :: Int] "abcd") == 12)
    , check "cartesian с пустым списком пуст" (null (cartesian ([] :: [Int]) "ab"))
    , check "triples 20 содержит (3,4,5)" ((3, 4, 5) `elem` triples 20)
    , check "Box 5 >>= умножение == Box 10"
        ((Box (5 :: Int) >>= \x -> Box (x * 2)) == Box 10)
    , check "левая единица: return a >>= f == f a"
        ((return (5 :: Int) >>= \x -> Box (x * 2)) == Box 10)
    , check "правая единица: m >>= return == m"
        ((Box (5 :: Int) >>= return) == Box 5)
    , check "ассоциативность связывания"
        (let m = Box (5 :: Int); f x = Box (x + 1); g x = Box (x * 2)
         in ((m >>= f) >>= g) == (m >>= \x -> f x >>= g))
    , check "законы монад выполняются и для Maybe"
        (let m = Just (5 :: Double)
             f x = safeDiv x 2
             g = safeSqrt
```

```
       in (m >>= return) == m
          && (return (5 :: Double) >>= f) == f 5
          && ((m >>= f) >>= g) == (m >>= \x -> f x >>= g))
  ]
printf "\nПройдено %d из %d проверок.\n"
       (length (filter id results)) (length results)
```