**Федеральное агентство связи**

**Ордена Трудового Красного Знамени**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский технический университет связи и информатики»**

Кафедра Математической Кибернетики и Информационных Технологий

****

**Отчет по курсовой работе**

по предмету «Функциональное программирование»

Выполнил: студент группы

БВТ1802

Самаков Владислав Владимирович

Руководитель:

Мосева Марина Сергеевна

Москва 2020

Весь исходный код работы доступен по ссылке:

[https://github.com/VladSamakov/FP-COURSEWORK](https://vk.com/away.php?utf=1&to=https%3A%2F%2Fgithub.com%2FVladSamakov%2FFP-COURSEWORK)

Задание на курсовую работу:

Необходимо реализовать проект, состоящий из выполненных лабораторных, реализовать в рамках проекта тестирование каждой из лабораторных по отдельности и совместно всего проекта.

**build.sbt**

lazy val commonSettings = Seq(  
 version := "1.0",  
 scalaVersion := "2.13.2",  
 libraryDependencies +=  
 "org.scalatest" %% "scalatest" % "3.0.8" % Test,  
)  
  
lazy val root = project  
 .in(file("."))  
 .aggregate(lab1, lab2, lab3, lab4)  
 .settings(commonSettings)  
  
lazy val lab1 = project  
 .settings(commonSettings)  
  
lazy val lab2 = project  
 .settings(commonSettings)  
  
lazy val lab3 = project  
 .settings(commonSettings)  
  
lazy val lab4 = project  
 .settings(commonSettings)

Данный build.sbt содержит корневой проект и 4 подпроекта, каждый из которых соответствует одной из выполненных лабораторных работ. Кроме того, в проекте используется библиотека Scalatest, позволяющая писать тесты для написанного кода.

**Корневой проект**

В корневом проекте файлов с исходным кодом нет, потому что все файлы с кодом относятся к лабораторным работам и, как следствие, находятся в подпроектах.

**Лабораторная работа 1**

Файл Classes.scala

package exercise1

sealed trait Animal {  
 val name: String  
 val food: String  
 def eats(food: String): Boolean=return this.food.equals(food)  
}  
case class Mammals(name: String, food: String) extends Animal  
case class Birds(name: String, food: String) extends Animal  
case class Fishes(name: String, food: String) extends Animal  
   
 object Animal{  
 sealed trait food  
  
 case object Meat extends food  
 case object Vegetables extends food  
 case object Plants extends food  
  
 val *cat*= *Mammals*("cat", "meat")  
 val *parrot*= Birds("parrot", "vegetables")  
 val *goldfish*=Fishes("goldfish", "plants")  
  
 def knownAnimal(name: String): Boolean=  
 name.equals(*cat*.name)||name.equals(*parrot*.name)||name.equals(*goldfish*.name)  
  
 def apply(name: String): Option[Animal]={  
 name match{  
 case *cat*.name=> Some(*cat*)  
 case *parrot*.name=> Some(*parrot*)  
 case *goldfish*.name=> Some(*goldfish*)  
 case other=> None  
 }  
 }  
 }

Файл Functions.scala

package exercise1

object Functions {  
 /\* a) Напишите функцию, которая рассчитывает площадь окружности  
 \*/  
def Circle(r:Double):Double=Math.*pow*(r,2)\*Math.*PI* // примените вашу функцию из пункта (a) здесь, не изменяя сигнатуру  
def testCircle(r: Double): Double = *Circle*(r)  
  
 /\* b) Напишите карированную функцию которая рассчитывает площадь прямоугольника a \* b.  
 \*/  
def RectangleCurried(a:Double)(b:Double):Double=a\*b  
def testRectangleCurried(a: Double, b: Double): Double = *RectangleCurried*(a)(b)  
  
 // c) Напишите не карированную функцию для расчета площади прямоугольника.  
def RectangleUc(a:Double,b:Double):Double=a\*b  
// примените вашу функцию из пункта (c) здесь, не изменяя сигнатуру  
def testRectangleUc(a: Double, b: Double): Double = *RectangleUc*(a, b)  
  
}

Файл HiOrder.scala

package exercise1  
*/\*\* Напишите ваши решения в виде функций. \*/*object HigherOrder {  
  
  
 /\* a) Напишите функцию, которая принимает `f: (Int, Int) => Int`, параменты `a` и `b`  
 \* и коэффициент умножения `n` и возвращает n \* f(a, b). Назовите `nTimes`.  
 \*/  
 def nTimes(f: (Int, Int)=> Int, a: Int, b: Int, n: Int): Int=n\*f(a,b)  
 // примените вашу функцию (a) здесь, не изменяйте сигнатуру  
 def testNTimes(f: (Int, Int) => Int, a: Int, b: Int, n: Int): Int = *nTimes*(f,a,b,n)  
  
 /\* b) Напишите анонимную функцию, функцию без идентификатора ((a, b) => ???) для `nTimes` которая  
 \* выполняет следующее:  
 \* if (a > b) a else b  
 \*/  
 def testAnonymousNTimes(a: Int, b: Int, n: Int): Int = *nTimes*((a: Int, b: Int)=>  
 {if (a>b) a else b}, a: Int, b: Int, n: Int)  
}

Файл Patterns.scala

package exercise1  
  
import org.scalatest.\_  
  
class PatternsTest extends FunSuite {  
  
 test("IntToString recognizes numbers from 1 to 3 inclusively") {  
 assert(PatternMatching.*IntToString*(1) == "it is one")  
 assert(PatternMatching.*IntToString*(2) == "it is two")  
 assert(PatternMatching.*IntToString*(3) == "it is three")  
 assert(PatternMatching.*IntToString*(5) == "what's that")  
 }  
  
 test("testIntToString calls IntToString with given parameters") {  
 assert(PatternMatching.*testIntToString*(1) == "it is one")  
 assert(PatternMatching.*testIntToString*(2) == "it is two")  
 assert(PatternMatching.*testIntToString*(3) == "it is three")  
 assert(PatternMatching.*testIntToString*(5) == "what's that")  
 }  
  
 test("IsMaxAndMoritz recognizes (M)max and (M)moritz") {  
 assert(PatternMatching.*IsMaxAndMoritz*("max") &&  
 PatternMatching.*IsMaxAndMoritz*("Max") &&  
 PatternMatching.*IsMaxAndMoritz*("moritz") &&  
 PatternMatching.*IsMaxAndMoritz*("Moritz") == true)  
  
 assert(PatternMatching.*IsMaxAndMoritz*("notMax") == false)  
 }  
  
 test("testIsMaxAndMoritz calls IsMaxAndMoritz with given parameter") {  
 assert(PatternMatching.*testIsMaxAndMoritz*("max") &&  
 PatternMatching.*testIsMaxAndMoritz*("Max") &&  
 PatternMatching.*testIsMaxAndMoritz*("moritz") &&  
 PatternMatching.*testIsMaxAndMoritz*("Moritz") == true)  
  
 assert(PatternMatching.*testIsMaxAndMoritz*("notMax") == false)  
 }  
  
 test("isEven returns true if number is even and false otherwise") {  
 assert(PatternMatching.*IsEven*(100) == true)  
 assert(PatternMatching.*IsEven*(13) == false)  
 }  
  
 test("testIsEven calls isEven with given parameter") {  
 assert(PatternMatching.*testIsEven*(100) == PatternMatching.*IsEven*(100))  
 assert(PatternMatching.*testIsEven*(13) == PatternMatching.*IsEven*(13))  
 }  
  
 test("winsA returns rock-paper-scissor game result for first player") {  
 assert(PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Rock) == PatternMatching.Draw)  
 assert(PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Paper) == PatternMatching.Lose)  
 assert(PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Scissor) == PatternMatching.Win)  
 assert(PatternMatching.*WinsA*(PatternMatching.Paper, PatternMatching.Rock) == PatternMatching.Win)  
 assert(PatternMatching.*WinsA*(PatternMatching.Paper, PatternMatching.Paper) == PatternMatching.Draw)  
 assert(PatternMatching.*WinsA*(PatternMatching.Paper, PatternMatching.Scissor) == PatternMatching.Lose)  
 assert(PatternMatching.*WinsA*(PatternMatching.Scissor, PatternMatching.Rock) == PatternMatching.Lose)  
 assert(PatternMatching.*WinsA*(PatternMatching.Scissor, PatternMatching.Paper) == PatternMatching.Win)  
 assert(PatternMatching.*WinsA*(PatternMatching.Scissor, PatternMatching.Scissor) == PatternMatching.Draw)  
 }  
  
 test("testWinsA calls winsA with given parameters") {  
 assert(PatternMatching.*testWinsA*(PatternMatching.Rock, PatternMatching.Scissor) ==  
 PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Scissor))  
 }  
  
 test("extractMammalWeight returns mammal's weight and -1 if not mammal is passed") {  
 assert(PatternMatching.*ExtractMammalWeight*(PatternMatching.*Mammal*("cat", PatternMatching.Meat, 5)) == 5)  
 assert(PatternMatching.*ExtractMammalWeight*(PatternMatching.*Fish*("goldfish", PatternMatching.Vegetables)) == -1)  
 }  
  
 test("testExtractMammalWeight calls extractMammalWeight with given parameter") {  
 assert(PatternMatching.*testExtractMammalWeight*(PatternMatching.*Mammal*("cat", PatternMatching.Meat, 5)) == 5)  
 assert(PatternMatching.*testExtractMammalWeight*(PatternMatching.*Fish*("goldfish", PatternMatching.Vegetables)) == -1)  
 }  
  
 test("updateFood changes animal food") {  
 val fish = PatternMatching.*Fish*("goldfish", PatternMatching.Plants)  
 PatternMatching.*UpdateFood*(fish)  
 assert(fish.food == PatternMatching.Plants)  
  
 val bird = PatternMatching.*Bird*("parrot", PatternMatching.Vegetables)  
 PatternMatching.*UpdateFood*(bird)  
 assert(bird.food == PatternMatching.Plants)  
 }  
  
 test("testUpdateFood calls updateFood with given parameter") {  
 val fish = PatternMatching.*Fish*("goldfish", PatternMatching.Vegetables)  
 PatternMatching.*testUpdateFood*(fish)  
 assert(fish.food == PatternMatching.Plants)  
 }  
  
  
}

Файлы с тестами

Файл ClassesTest.scala

package exercise1  
  
import org.scalatest.FunSuite  
  
class ClassesTest extends FunSuite {  
  
 test("Animal.apply creates animal if valid parameter is passed") {  
 assert(Animal.*apply*("parrot") == (Some(Animal.*parrot*)))  
 assert(Animal.*apply*("cat") == (Some(Animal.*cat*)))  
 assert(Animal.*apply*("goldfish") == (Some(Animal.goldfish)))  
 }  
 test("Animal.apply returns None if wrong parameter is passed") {  
 assert(Animal.apply("wrong") == None)  
 }  
  
 test("KnownAnimal returns true if valid animal is passed") {  
 assert(Animal.knownAnimal("cat") && Animal.knownAnimal("parrot") &&  
 Animal.knownAnimal("goldfish") == true)  
 }  
  
 test("KnownAnimal returns false if invalid animal is passed") {  
 assert(Animal.knownAnimal("lion") == false)  
 }  
 test("Animal.eats checks if animal eats given food") {  
 val tiger = Mammals("tiger", "meat")  
 assert(tiger.eats("plants") == false)  
 assert(tiger.eats("meat") == true)  
 }  
}

Файл FunctionsTest.scala

package exercise1  
  
import org.scalatest.FunSuite  
  
class FunctionsTest extends FunSuite {  
 test("CircleArea calculates circle area") {  
 assert(Functions.*Circle*(6) == 6 \* 6 \* Math.*PI*)  
 }  
  
 test("testCircle calls circleArea with given radius") {  
 assert(Functions.*testCircle*(6) == 6 \* 6 \* Math.*PI*)  
 }  
  
 test("RectangleArea calculates area of the given rectangle") {  
 assert(Functions.*RectangleUc*(5, 10) == 50)  
 }  
  
 test("testRectangleUc calls RectangleArea with given parameters") {  
 assert(Functions.*testRectangleUc*(10, 5) == 50)  
 }  
  
 test("RectangleAreaCurried calculates area of the given rectangle") {  
 assert(Functions.*RectangleCurried*(5)(10) == 50)  
 }  
  
 test("testRectangleCurried calls RectangleAreaCurried with given parameters") {  
 assert(Functions.*testRectangleCurried*(5, 10) == 50)  
 }  
}

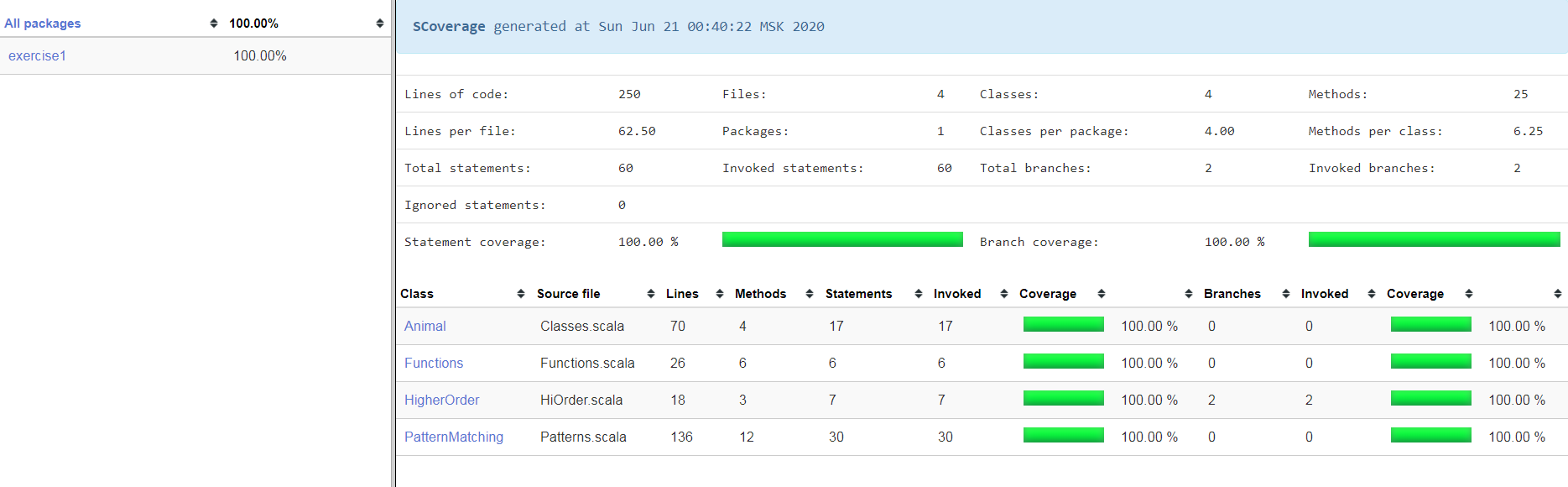
Файл HigherOrderTest.scala

package exercise1  
  
import org.scalatest.FunSuite  
  
class HigherOrderTest extends FunSuite {  
  
 test("nTimes multiplies function result by n") {  
 assert(HigherOrder.*nTimes*((a: Int, b:Int) => a + b, 3, 3, 4) == 24)  
 }  
 test("testNtimes calls nTimes with given parameters") {  
 assert(HigherOrder.*testNTimes*((a: Int, b: Int) => a + b, 3, 3, 4) == 24)  
 }  
  
 test("testAnonymousNTimes calls nTimes with given parameters and hardcoded function") {  
 assert(HigherOrder.*testAnonymousNTimes*(2, 3, 4) == 12)  
 assert(HigherOrder.*testAnonymousNTimes*(3, 2, 4) == 12)  
 }  
  
  
}

Файл PatternsTest.scala

package exercise1  
  
import org.scalatest.\_  
  
class PatternsTest extends FunSuite {  
  
 test("IntToString recognizes numbers from 1 to 3 inclusively") {  
 assert(PatternMatching.*IntToString*(1) == "it is one")  
 assert(PatternMatching.*IntToString*(2) == "it is two")  
 assert(PatternMatching.*IntToString*(3) == "it is three")  
 assert(PatternMatching.*IntToString*(5) == "what's that")  
 }  
  
 test("testIntToString calls IntToString with given parameters") {  
 assert(PatternMatching.*testIntToString*(1) == "it is one")  
 assert(PatternMatching.*testIntToString*(2) == "it is two")  
 assert(PatternMatching.*testIntToString*(3) == "it is three")  
 assert(PatternMatching.*testIntToString*(5) == "what's that")  
 }  
  
 test("IsMaxAndMoritz recognizes (M)max and (M)moritz") {  
 assert(PatternMatching.*IsMaxAndMoritz*("max") &&  
 PatternMatching.*IsMaxAndMoritz*("Max") &&  
 PatternMatching.*IsMaxAndMoritz*("moritz") &&  
 PatternMatching.*IsMaxAndMoritz*("Moritz") == true)  
  
 assert(PatternMatching.*IsMaxAndMoritz*("notMax") == false)  
 }  
  
 test("testIsMaxAndMoritz calls IsMaxAndMoritz with given parameter") {  
 assert(PatternMatching.*testIsMaxAndMoritz*("max") &&  
 PatternMatching.*testIsMaxAndMoritz*("Max") &&  
 PatternMatching.*testIsMaxAndMoritz*("moritz") &&  
 PatternMatching.*testIsMaxAndMoritz*("Moritz") == true)  
  
 assert(PatternMatching.*testIsMaxAndMoritz*("notMax") == false)  
 }  
  
 test("isEven returns true if number is even and false otherwise") {  
 assert(PatternMatching.*IsEven*(100) == true)  
 assert(PatternMatching.*IsEven*(13) == false)  
 }  
  
 test("testIsEven calls isEven with given parameter") {  
 assert(PatternMatching.*testIsEven*(100) == PatternMatching.*IsEven*(100))  
 assert(PatternMatching.*testIsEven*(13) == PatternMatching.*IsEven*(13))  
 }  
  
 test("winsA returns rock-paper-scissor game result for first player") {  
 assert(PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Rock) == PatternMatching.Draw)  
 assert(PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Paper) == PatternMatching.Lose)  
 assert(PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Scissor) == PatternMatching.Win)  
 assert(PatternMatching.*WinsA*(PatternMatching.Paper, PatternMatching.Rock) == PatternMatching.Win)  
 assert(PatternMatching.*WinsA*(PatternMatching.Paper, PatternMatching.Paper) == PatternMatching.Draw)  
 assert(PatternMatching.*WinsA*(PatternMatching.Paper, PatternMatching.Scissor) == PatternMatching.Lose)  
 assert(PatternMatching.*WinsA*(PatternMatching.Scissor, PatternMatching.Rock) == PatternMatching.Lose)  
 assert(PatternMatching.*WinsA*(PatternMatching.Scissor, PatternMatching.Paper) == PatternMatching.Win)  
 assert(PatternMatching.*WinsA*(PatternMatching.Scissor, PatternMatching.Scissor) == PatternMatching.Draw)  
 }  
  
 test("testWinsA calls winsA with given parameters") {  
 assert(PatternMatching.*testWinsA*(PatternMatching.Rock, PatternMatching.Scissor) ==  
 PatternMatching.*WinsA*(PatternMatching.Rock, PatternMatching.Scissor))  
 }  
  
 test("extractMammalWeight returns mammal's weight and -1 if not mammal is passed") {  
 assert(PatternMatching.*ExtractMammalWeight*(PatternMatching.*Mammal*("cat", PatternMatching.Meat, 5)) == 5)  
 assert(PatternMatching.*ExtractMammalWeight*(PatternMatching.*Fish*("goldfish", PatternMatching.Vegetables)) == -1)  
 }  
  
 test("testExtractMammalWeight calls extractMammalWeight with given parameter") {  
 assert(PatternMatching.*testExtractMammalWeight*(PatternMatching.*Mammal*("cat", PatternMatching.Meat, 5)) == 5)  
 assert(PatternMatching.*testExtractMammalWeight*(PatternMatching.*Fish*("goldfish", PatternMatching.Vegetables)) == -1)  
 }  
  
 test("updateFood changes animal food") {  
 val fish = PatternMatching.*Fish*("goldfish", PatternMatching.Plants)  
 PatternMatching.*UpdateFood*(fish)  
 assert(fish.food == PatternMatching.Plants)  
  
 val bird = PatternMatching.*Bird*("parrot", PatternMatching.Vegetables)  
 PatternMatching.*UpdateFood*(bird)  
 assert(bird.food == PatternMatching.Plants)  
 }  
  
 test("testUpdateFood calls updateFood with given parameter") {  
 val fish = PatternMatching.*Fish*("goldfish", PatternMatching.Vegetables)  
 PatternMatching.*testUpdateFood*(fish)  
 assert(fish.food == PatternMatching.Plants)  
 }  
  
  
}

Процент покрытия кода тестами:



**Лабораторная работа 2**

Файлы с исходным кодом:

Файл Compositions.scala

package exercise2  
/\*  
 \* Option представляет собой контейнер, который хранит какое-то значение   
 \* или не хранит ничего совсем, указывает, вернула ли операция результат или нет.   
 \* Это часто используется при поиске значений или когда операции могут потерпеть неудачу,   
 \* и вам не важна причина.  
 \*  
 \* Комбинаторы называются так потому, что они созданы, чтобы объединять результаты.   
 \* Результат одной функции часто используется в качестве входных данных для другой.  
 \* Наиболее распространенным способом, является использование их со стандартными структурами данных.  
 \* Функциональные комбинаторы `map` и` flatMap` являются контекстно-зависимыми.  
 \* map - применяет функцию к каждому элементу из списка, возвращается список с тем же числом элементов.  
 \* flatMap берет функцию, которая работает с вложенными списками и объединяет результаты.  
 \*/  
/\*   
 \* Напишите ваши решения в тестовых функциях.   
 \*/  
object Compositions {  
  
 /\*  
 \* Используйте данные функции. Вы можете реализовать свое решение прямо в тестовой функции.  
 \* Нельзя менять сигнатуры  
 \*/  
 def testCompose[A, B, C, D](f: A => B)  
 (g: B => C)  
 (h: C => D): A => D = h compose g compose f  
  
 /\*  
 \* Напишите функции с использованием `map` и `flatMap`. Вы можете реализовать свое решение прямо в тестовой функции.  
 \* Нельзя менять сигнатуры  
 \*/  
 def testMapFlatMap[A, B, C, D](f: A => Option[B])  
 (g: B => Option[C])  
 (h: C => D): Option[A] => Option[D] = \_.flatMap(f).flatMap(g).map(h)  
 /\*  
 \* Напишите функцию используя for. Вы можете реализовать свое решение прямо в тестовой функции.  
 \* Нельзя менять сигнатуры  
 \*/  
 def testForComprehension[A, B, C, D](f: A => Option[B])  
 (g: B => Option[C])  
 (h: C => D): Option[A] => Option[D] = for { one <- \_  
 two <- f(one)  
 three <- g(two) } yield h(three)  
  
}

Файл RecurciveData.scala

package exercise2  
import scala.annotation.tailrec  
import scala.collection.immutable.List   
/\*  
 \*Напишите свои решения в виде функций.   
 \*/  
object RecursiveData {  
  
 /\*  
 \*Реализуйте функцию, определяющую является ли пустым `List[Int]`.  
 \*/  
 def ListIntEmpty(list: List[Int]) : Boolean = list match {  
 case x :: tail => false  
 case *Nil* => true  
 }  
  
 /\*  
 \* Используйте функцию из пункта (a) здесь, не изменяйте сигнатуру.  
 \*/  
 def testListIntEmpty(list: List[Int]): Boolean = *ListIntEmpty*(list)  
  
 /\*  
 \*Реализуйте функцию, которая получает head `List[Int]`или возвращает -1 в случае если он пустой.  
 \*/  
 def ListIntHead(list: List[Int]) : Int = list match {  
 case x :: tail => x  
 case *Nil* => -1  
 }  
  
 /\*  
 \*Используйте функцию из пункта (a) здесь, не изменяйте сигнатуру.  
 \*/  
 def testListIntHead(list: List[Int]): Int = *ListIntHead*(list)  
  
 /\*  
 \*Можно ли изменить `List[A]` так чтобы гарантировать что он не является пустым?  
 \*/  
 def ListNotEmpty[A](head: A, list: List[A]) : List[A] = list match {  
 case *Nil* => head :: list  
 case x *::* tail => list  
 }  
  
}

Файл RecursiveFunc.scala

package exercise2  
import scala.annotation.tailrec  
import scala.collection.immutable.List   
 /\*  
 \* Реализуйте функции для решения следующих задач.  
 \* Примечание: Попытайтесь сделать все функции с хвостовой рекурсией, используйте аннотацию для подстверждения.  
 \* рекурсия будет хвостовой если:  
 \* 1. рекурсия реализуется в одном направлении  
 \* 2. вызов рекурсивной функции будет последней операцией перед возвратом  
\*/  
object RecursiveFunctions {  
  
 def length[A](as: List[A]): Int = {  
 @tailrec  
 def loop(rem: List[A], agg: Int): Int = rem match {  
 case x *::* tail => loop(tail, agg + 1)  
 case *Nil* => agg  
 }  
  
 loop(as, 0)  
 }  
  
 /\*   
 \* Напишите функцию которая записывает в обратном порядке список:  
 \* def reverse[A](list: List[A]): List[A]  
 \*/  
 def reverse[A](list: List[A]): List[A] = {  
 @tailrec  
 def loop(rem: List[A], result: List[A]): List[A] = rem match {  
 case x *::* tail => loop(tail, x :: result)  
 case *Nil* => result  
 }  
 loop(list, *Nil*)  
 }  
 /\*  
 \* Используйте функцию из пункта здесь, не изменяйте сигнатуру  
 \*/  
 def testReverse[A](list: List[A]): List[A] = *reverse*(list)  
  
 /\*   
 \* Напишите функцию, которая применяет функцию к каждому значению списка:  
 \* def map[A, B](list: List[A])(f: A => B): List[B]  
 \*/  
 def Map[A, B](list: List[A])(f: A => B): List[B] = {  
 @tailrec  
 def loop(rem: List[A], result: List[B])(f: A => B): List[B] = rem match {  
 case x *::* tail => loop(tail, result :+ f(x))(f)  
 case *Nil* => result  
 }  
 loop(list, *Nil*)(f)  
 }  
 /\*  
 \* Используйте функцию из пункта здесь, не изменяйте сигнатуру  
 \*/  
 def testMap[A, B](list: List[A], f: A => B): List[B] = *Map*(list)(f)  
   
 /\*   
 \* Напишите функцию, которая присоединяет один список к другому:  
 \* def append[A](l: List[A], r: List[A]): List[A]  
 \*/  
 def Append[A](l: List[A], r: List[A]) : List[A] = {  
 @tailrec  
 def loop(rem: List[A], result: List[A]) : List[A] = rem match {  
 case x *::* tail => loop(tail, result :+ x)  
 case *Nil* => result  
 }  
 loop(r, l)  
 }  
 /\*  
 \* Используйте функцию из пункта здесь, не изменяйте сигнатуру  
 \*/  
 def testAppend[A](l: List[A], r: List[A]): List[A] = *Append*(l, r)  
  
 /\*   
 \* Напишите функцию, которая применяет функцию к каждому значению списка:  
 \* def flatMap[A, B](list: List[A])(f: A => List[B]): List[B]  
 \*   
 \* она получает функцию, которая создает новый List[B] для каждого элемента типа A в   
 \* списке. Поэтому вы создаете List[List[B]].   
 \*/  
 def FlatMap[A, B](list: List[A])(f: A => List[B]): List[B] = {  
 @tailrec  
 def loop(rem: List[A], result: List[B])(f: A => List[B]): List[B] = rem match {  
 case x :: tail => loop(tail, result ++ f(x))(f)  
 case *Nil* => result  
 }  
 loop(list, *Nil*)(f)  
 }  
 /\*  
 \* Используйте функцию из пункта здесь, не изменяйте сигнатуру  
 \*/  
 def testFlatMap[A, B](list: List[A], f: A => List[B]): List[B] = *FlatMap*(list)(f)  
  
 /\*   
 \* Вопрос: Возможно ли написать функцию с хвостовой рекурсией для `Tree`s? Если нет, почему?   
 \*  
 \* Нельзя написать функцию с хвостовой рекурсией для `Tree`s, так как хвостовая рекурсия - это  
 \* рекурсия в одном направлении, что не подходит для древовидной структуры  
 \*/  
}

Файл CompositionTest.scala

package exercise2  
  
  
import org.scalatest.FunSuite  
  
class CompositionTest extends FunSuite {  
  
 test("testCompose should compose given functions") {  
 assert(Compositions.*testCompose*((i:Int) => "test" \* i)((i: String) => i \* 2)  
 ((i:String) => i.dropRight(3))(2) == "testtesttestt")  
 }  
  
 test("testFlatMap should compose given functions") {  
 assert(Compositions.*testMapFlatMap*((i:Int) => if (i > 0) Some(i) else None)  
 ((i:Int) => if (i > 10) Some(i) else None)  
 ((i:Int) => i \* 2)(Some(-1)) == None)  
 }  
  
 test("testForComprehension should compose given functions") {  
 assert(Compositions.*testForComprehension*((i:Int) => if (i > 0) Some(i) else None)  
 ((i:Int) => if (i > 10) Some(i) else None)  
 ((i:Int) => i \* 2)(Some(11)) == Some(22))  
 }  
  
}

Файл RecursiveDataTest.scala

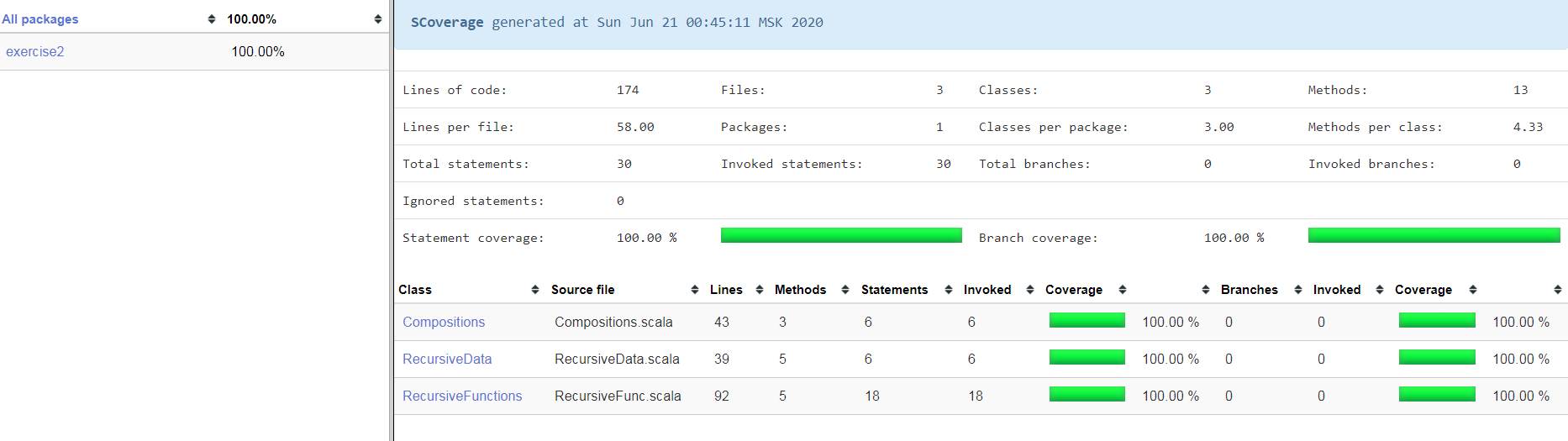
package exercise2  
  
import org.scalatest.FunSuite  
  
class RecursiveDataTest extends FunSuite{  
  
 test("ListIntEmpty returns true if list is empty and false otherwise") {  
 assert(RecursiveData.*ListIntEmpty*(*Nil*) == true)  
 assert(RecursiveData.*ListIntEmpty*(*List*(1, 2, 3)) == false)  
 }  
  
 test("testListIntEmpty returns true if list is empty and false otherwise") {  
 assert(RecursiveData.*testListIntEmpty*(*Nil*) == true)  
 assert(RecursiveData.*testListIntEmpty*(*List*(1, 2, 3)) == false)  
 }  
  
 test("ListIntHead returns first element of the list and -1 if the list is empty") {  
 assert(RecursiveData.*ListIntHead*(*List*(1, 2, 3)) == 1)  
 assert(RecursiveData.*ListIntHead*(*Nil*) == -1)  
 }  
  
 test("testListIntHead returns first element of the list and -1 if the list is empty") {  
 assert(RecursiveData.*testListIntHead*(*List*(1, 2, 3)) == 1)  
 assert(RecursiveData.*testListIntHead*(*Nil*) == -1)  
 }  
  
 test("ListNotEmpty returns adds element to the list if it's empty") {  
 assert(RecursiveData.*ListNotEmpty*(1, *Nil*) == *List*(1))  
 assert(RecursiveData.*ListNotEmpty*(1, *List*(1, 2, 3)) == *List*(1, 2, 3))  
 }  
  
}

Файл RecurciveFuncTest.scala

package exercise2  
  
import org.scalatest.FunSuite  
import RecursiveFunctions.\_  
  
import scala.collection.immutable.List  
  
class RecursiveFuncTest extends FunSuite {  
  
 test("length returns lenght of the passed list") {  
 assert(length(List(1, 2, 3)) == 3)  
 }  
  
 test("reverse takes a list and returns same but reversed list") {  
 assert(*reverse*(List(1, 2, 3)) == List(3, 2, 1))  
 }  
  
 test("testReverse takes a list and returns same but reversed list") {  
 assert(*testReverse*(List(1, 2, 3)) == List(3, 2, 1))  
 }  
  
 test("Map should apply given function to each element of the passed list") {  
 assert(*Map*(List(1, 2, 3))((x: Int) => x \* 2) == List(2, 4, 6))  
 }  
  
 test("testMap should apply given function to each element of the passed list") {  
 assert(*testMap*(List(1, 2, 3), (x: Int) => x \* 2) == List(2, 4, 6))  
 }  
  
 test("Append concatenates two lists") {  
 assert(*Append*(List(1, 2, 3), List(4, 5, 6)) == List(1, 2, 3, 4, 5, 6))  
 }  
  
 test("testAppend concatenates two lists") {  
 assert(*testAppend*(List(1, 2, 3), List(4, 5, 6)) == List(1, 2, 3, 4, 5, 6))  
 }  
  
 test("FlatMap applies given function to each element of the passed list") {  
 assert(*FlatMap*(List(2, 3, 4))((x:Int) => List.range(1, x)) == List(1, 1, 2, 1, 2, 3))  
 }  
  
 test("testFlatMap applies function to each element of the passed list") {  
 assert(testFlatMap(List(2, 3, 4), (x:Int) => List.range(1, x)) == List(1, 1, 2, 1, 2, 3))  
 }  
}

Результаты:

Покрытие кода тестами:



**Лабораторная работа 3**

Файлы с исходным кодом:

Файл Adts.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Adts.\_  
  
  
class AdtsTest extends FunSuite{  
  
 test("testGetNth should return n-th element of the list") {  
 assert(*testGetNth*(*List*(1, 2, 3, 4), 2) == *Some*(3))  
 }  
  
 test("Double should return doubled number and None if None is passed") {  
 assert(*Double*(*Some*(2)) == *Some*(4))  
 assert(*Double*(None) == None)  
 }  
  
 test("testDouble should return double number and None if None is passed") {  
 assert(*testDouble*(*Some*(2)) == *Some*(4))  
 assert(*testDouble*(None) == None)  
 }  
  
 test("IsEven should return passed number if it's even and string \"Odd number\" otherwise") {  
 assert(*IsEven*(6) == *Right*(6))  
 assert(*IsEven*(7) == *Left*("Odd number"))  
 }  
  
 test("testIsEven should return passed number if it's even and string \"Odd number\" otherwise") {  
 assert(*testIsEven*(6) == *Right*(6))  
 assert(*testIsEven*(7) == *Left*("Odd number"))  
 }  
  
 test("SafeDivide should divide two numbers unless divisor is 0") {  
 assert(*SafeDivide*(20, 2) == *Right*(10))  
 assert(*SafeDivide*(20, 0) == *Left*("Cannot divide by zero"))  
 }  
  
 test("testSafeDivide should divide two numbers unless divisor is 0") {  
 assert(*testSafeDivide*(20, 2) == *Right*(10))  
 assert(*testSafeDivide*(20, 0) == *Left*("Cannot divide by zero"))  
 }  
  
 def impureFunc(str: String): Int = {  
 10/0  
 }  
  
 def pureFunc(str: String): Int = {  
 10  
 }  
  
 test("GoodOldJava should return Try[Int]") {  
 assert(*GoodOldJava*(impureFunc, "Hello, world!") != util.*Success*(10))  
 }  
  
 test("testGoodOldJava should return Try[Int]") {  
 assert(*testGoodOldJava*(pureFunc, "Hello, world!") == util.*Success*(10))  
 }  
  
}

Файл Maps.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Maps.\_  
  
class MapsTest extends FunSuite {  
 val *Frodo* = *User*("Frodo", 15)  
 val Legolas1 = User("Legolas", 1000)  
 val Legolas2 = User("Legolas", 500)  
  
 val map = Map("Frodo" -> Frodo, "Legolas" -> Legolas1, "Legolas" -> Legolas2)  
 val seq = Seq(Frodo, Legolas1, Legolas2)  
  
 test("testGroupUsers should group users by name and calculate average age of each group") {  
 assert(testGroupUsers(seq) == Map("Frodo" -> 15, "Legolas" -> 750))  
 }  
  
 test("testNumberFrodos counts \"Adam\" count in passed map[String->User]") {  
 assert(testNumberFrodos(Map("Frodo" -> Frodo, "Adam" -> User("Adam", 40))) == 1)  
 }  
  
 test("testUnderaged removes from the map all users under 35 years old") {  
 assert(testUnderaged(map) == Map("Legolas" -> Legolas1, "Legolas" -> Legolas2))  
 }  
  
}

Файл Sequence.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Sequence.\_  
  
class SequenceTest extends FunSuite {  
  
 test("testLastElement returns last element of the sequence") {  
 assert(testLastElement(Seq(1, 2, 3, 4)) == Some(4))  
 }  
  
 test("testZip zips combines two sequences") {  
 assert(testZip(Seq(1, 2), Seq(3, 4)) == Seq((1, 3), (2, 4)))  
 }  
  
 test("testForAll checks codition for all elements") {  
 assert(testForAll(Seq(1, 2, 3, 4))((x: Int) => x < 5) == true)  
 }  
  
 test("testPalindrom returns true if sequence is palindrom and false otherwise") {  
 assert(testPalindrome(Seq(1, 2, 3, 4)) == false)  
 assert(testPalindrome(Seq(1, 2, 2, 1)) == true)  
 }  
  
 test("testFlatMap should apply function to each element of the sequence") {  
 assert(testFlatMap(Seq(1, 2, 3, 4))((x: Int) => Seq.range(1, x)) == Seq(1, 1, 2, 1, 2, 3))  
 }  
}

Файл Strings.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Strings.\_  
  
class StringsTest extends FunSuite {  
  
 test("testUppercase returns string with all capital letters") {  
 assert(testUppercase("testString") == "TESTSTRING")  
 }  
  
 test("testInterpolation inserts given name and age into string") {  
 assert(testInterpolations("Legolas", 750) == "Hi, my name is Legolas and I am 750 years old.")  
 }  
  
 test("testComputation inserts numbers into string") {  
 assert(testComputation(5, 6) == "Hi,\n" +  
 "now follows a quite hard calculation. We try to add:\n" +  
 " a := 5\n" +  
 " b := 6\n\n" +  
 " return 5 + 6")  
 }  
  
 test("testTakeTwo takes first two characters of the string") {  
 assert(testTakeTwo("Substring") == "Su")  
 }  
}

Файлы с тестами:

Файл AdtsTest.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Adts.\_  
  
  
class AdtsTest extends FunSuite{  
  
 test("testGetNth should return n-th element of the list") {  
 assert(*testGetNth*(*List*(1, 2, 3, 4), 2) == Some(3))  
 }  
  
 test("Double should return doubled number and None if None is passed") {  
 assert(*Double*(Some(2)) == Some(4))  
 assert(*Double*(None) == None)  
 }  
  
 test("testDouble should return double number and None if None is passed") {  
 assert(*testDouble*(Some(2)) == Some(4))  
 assert(*testDouble*(None) == None)  
 }  
  
 test("IsEven should return passed number if it's even and string \"Odd number\" otherwise") {  
 assert(*IsEven*(6) == *Right*(6))  
 assert(*IsEven*(7) == *Left*("Odd number"))  
 }  
  
 test("testIsEven should return passed number if it's even and string \"Odd number\" otherwise") {  
 assert(*testIsEven*(6) == *Right*(6))  
 assert(*testIsEven*(7) == *Left*("Odd number"))  
 }  
  
 test("SafeDivide should divide two numbers unless divisor is 0") {  
 assert(*SafeDivide*(20, 2) == *Right*(10))  
 assert(*SafeDivide*(20, 0) == *Left*("Cannot divide by zero"))  
 }  
  
 test("testSafeDivide should divide two numbers unless divisor is 0") {  
 assert(*testSafeDivide*(20, 2) == *Right*(10))  
 assert(*testSafeDivide*(20, 0) == *Left*("Cannot divide by zero"))  
 }  
  
 def impureFunc(str: String): Int = {  
 10/0  
 }  
  
 def pureFunc(str: String): Int = {  
 10  
 }  
  
 test("GoodOldJava should return Try[Int]") {  
 assert(*GoodOldJava*(impureFunc, "Hello, world!") != util.Success(10))  
 }  
  
 test("testGoodOldJava should return Try[Int]") {  
 assert(*testGoodOldJava*(pureFunc, "Hello, world!") == util.Success(10))  
 }  
  
}

Файл MapsTest.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Maps.\_  
  
class MapsTest extends FunSuite {  
 val *Frodo* = *User*("Frodo", 15)  
 val *Legolas1* = *User*("Legolas", 1000)  
 val *Legolas2* = *User*("Legolas", 500)  
  
 val *map* = *Map*("Frodo" -> *Frodo*, "Legolas" -> *Legolas1*, "Legolas" -> *Legolas2*)  
 val *seq* = *Seq*(*Frodo*, *Legolas1*, *Legolas2*)  
  
 test("testGroupUsers should group users by name and calculate average age of each group") {  
 assert(*testGroupUsers*(*seq*) == *Map*("Frodo" -> 15, "Legolas" -> 750))  
 }  
  
 test("testNumberFrodos counts \"Adam\" count in passed map[String->User]") {  
 assert(*testNumberFrodos*(*Map*("Frodo" -> *Frodo*, "Adam" -> *User*("Adam", 40))) == 1)  
 }  
  
 test("testUnderaged removes from the map all users under 35 years old") {  
 assert(*testUnderaged*(*map*) == *Map*("Legolas" -> *Legolas1*, "Legolas" -> *Legolas2*))  
 }  
  
}

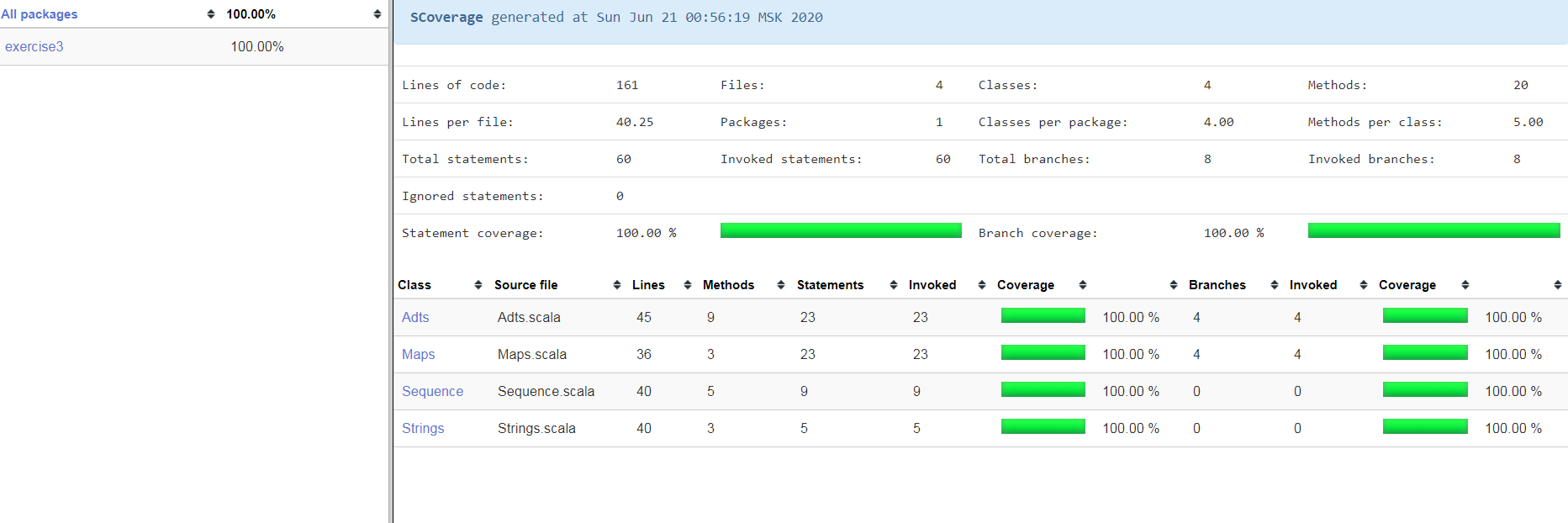
Файл SequenceTest.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Sequence.\_  
  
class SequenceTest extends FunSuite {  
  
 test("testLastElement returns last element of the sequence") {  
 assert(*testLastElement*(*Seq*(1, 2, 3, 4)) == Some(4))  
 }  
  
 test("testZip zips combines two sequences") {  
 assert(*testZip*(*Seq*(1, 2), *Seq*(3, 4)) == *Seq*((1, 3), (2, 4)))  
 }  
  
 test("testForAll checks codition for all elements") {  
 assert(*testForAll*(*Seq*(1, 2, 3, 4))((x: Int) => x < 5) == true)  
 }  
  
 test("testPalindrom returns true if sequence is palindrom and false otherwise") {  
 assert(*testPalindrome*(*Seq*(1, 2, 3, 4)) == false)  
 assert(*testPalindrome*(*Seq*(1, 2, 2, 1)) == true)  
 }  
  
 test("testFlatMap should apply function to each element of the sequence") {  
 assert(*testFlatMap*(*Seq*(1, 2, 3, 4))((x: Int) => *Seq*.range(1, x)) == *Seq*(1, 1, 2, 1, 2, 3))  
 }  
}

Файл StringsTest.scala

package exercise3  
  
import org.scalatest.FunSuite  
import Strings.\_  
  
class StringsTest extends FunSuite {  
  
 test("testUppercase returns string with all capital letters") {  
 assert(*testUppercase*("testString") == "TESTSTRING")  
 }  
  
 test("testInterpolation inserts given name and age into string") {  
 assert(*testInterpolations*("Legolas", 750) == "Hi, my name is Legolas and I am 750 years old.")  
 }  
  
 test("testComputation inserts numbers into string") {  
 assert(*testComputation*(5, 6) == "Hi,\n" +  
 "now follows a quite hard calculation. We try to add:\n" +  
 " a := 5\n" +  
 " b := 6\n\n" +  
 " return 5 + 6")  
 }  
  
 test("testTakeTwo takes first two characters of the string") {  
 assert(*testTakeTwo*("Substring") == "Su")  
 }  
}

Результаты:



**Лабораторная работа 4**

Файлы с исходным кодом:

Файл TypeClasses.scala

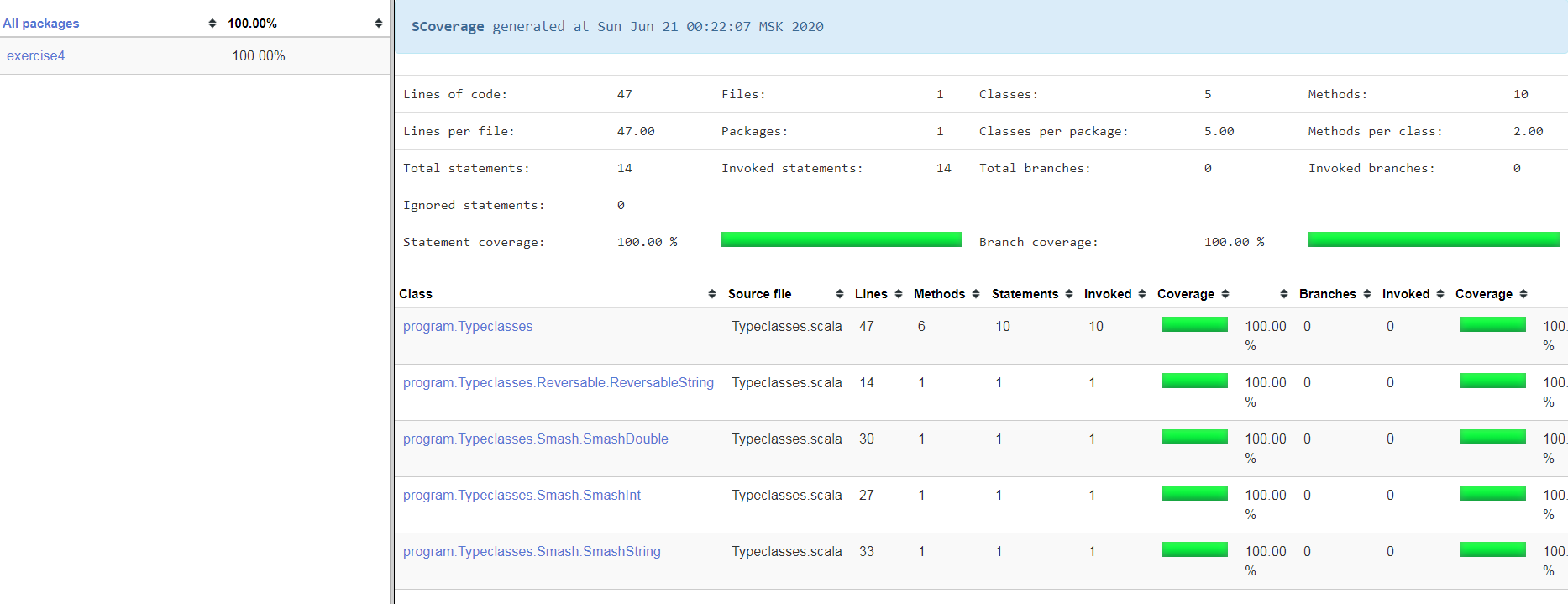
package exercise4  
object program {  
  
 object Typeclasses {  
  
 // a) Определите тайп-класс Reversable, который представляет в обратном порядке значения.  
 trait Reversable[T] {  
 def reverse(x: T): T  
 }  
  
 // b) Реализуйте функцию Reverse для String.  
 object Reversable {  
 implicit object ReversableString extends Reversable[String] {  
 def reverse(str: String) : String = str.reverse  
 }  
 }  
 def reverse[T](str: T)(implicit rev: Reversable[T]): T = rev.reverse(str)  
 // примените тайп-класс-решение из пункта (a) здесь  
 def testReversableString(str: String): String = *reverse*(str)  
  
 // c) Определите тайп-класс Smash таким образом чтобы в нем была функция smash, которая выполняет операцию со значениями одного типа.  
 trait Smash[T] {  
 def smash(a: T, b: T): T  
 }  
 object Smash {  
 implicit object SmashInt extends Smash[Int] {  
 def smash(a: Int, b: Int): Int = a + b  
 }  
 implicit object SmashDouble extends Smash[Double] {  
 def smash(a: Double, b: Double): Double = a \* b  
 }  
 implicit object SmashString extends Smash[String] {  
 def smash(a: String, b:String): String = a.concat(b)  
 }  
 }  
  
 // d) Реализуйте функции Smash для типа Int и Double.  
 // Используйте сложение для типа Int у умножение для типа Double.  
 def smash[T](a: T, b: T)(implicit sm : Smash[T]) : T = sm.smash(a, b)  
 // примените тайп-класс-решение из пункта (d) здесь  
 def testSmashInt(a: Int, b: Int): Int = smash(a, b)  
 // примените тайп-класс-решение из пункта (d) здесь  
 def testSmashDouble(a: Double, b: Double): Double = smash(a, b)  
  
 // e) Реализуйте функцию Smash для типа String. Необходимо выполнить конкатенацию строк, которые будут получены в качестве параметра.  
 // примените тайп-класс-решение из пункта (d) здесь  
 def testSmashString(a: String, b: String): String = smash(a, b)  
 }  
  
}

Файлы с тестами:

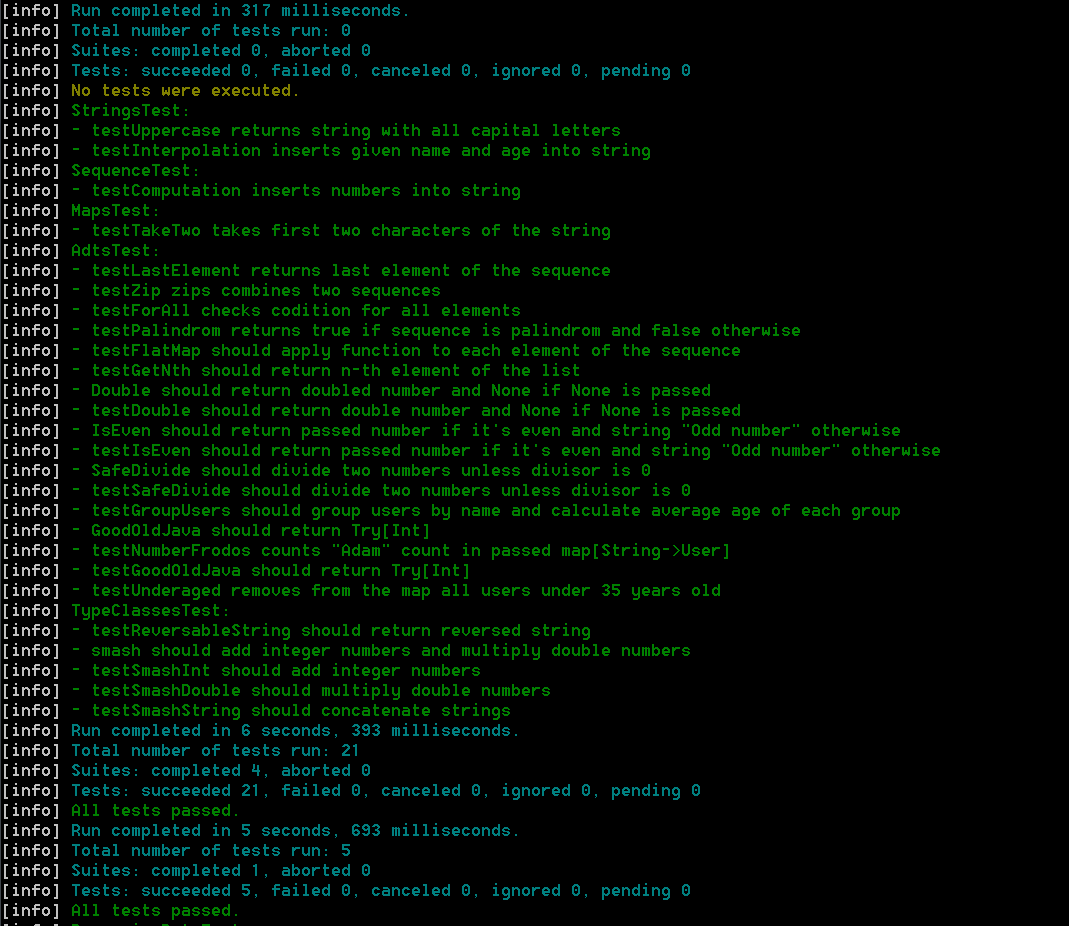
Файл TypeClassesTest.scala

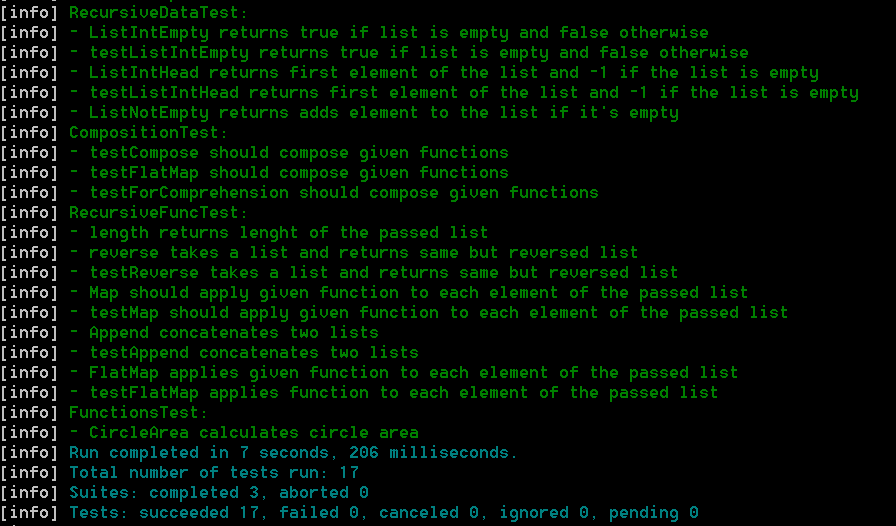
package exercise4  
  
import org.scalatest.FunSuite  
import exercise4.program.Typeclasses.\_  
  
class TypeClassesTest extends FunSuite {  
  
 test("testReversableString should return reversed string") {  
 assert(*testReversableString*("reverse") == "esrever")  
 }  
  
 test("smash should add integer numbers and multiply double numbers") {  
 assert(smash(32, 3) == 35)  
 assert(smash(2.5, 4.0) == 10.0)  
 assert(smash("Concat", "String") == "ConcatString")  
 }  
  
 test("testSmashInt should add integer numbers") {  
 assert(testSmashInt(32, 3) == 35)  
 }  
  
 test("testSmashDouble should multiply double numbers") {  
 assert(testSmashDouble(2.5, 4.0) == 10.0)  
 }  
  
 test("testSmashString should concatenate strings") {  
 assert(testSmashString("Concat", "String") == "ConcatString")  
 }  
  
  
  
}

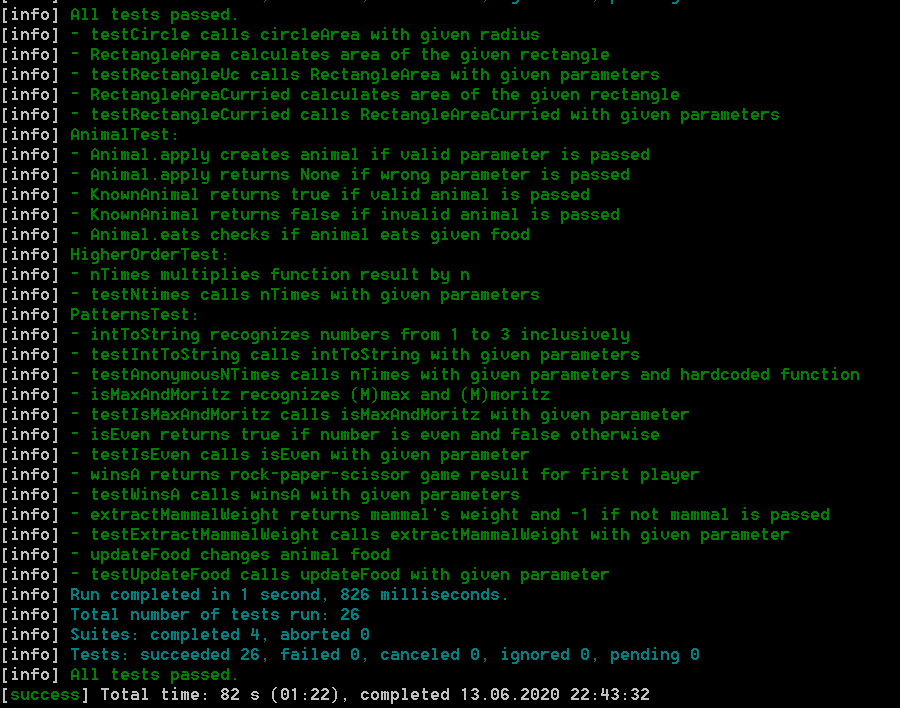
Результаты:



Тестирование всего проекта:







Вывод

Я познакомилась с sbt, scalatest и scoverage – инструментами для разработки и тестирования кода на языке программирования scala. Сделанные лабораторные работы были полностью протестированы. В процессе тестирования были исправлены допущенные ранее ошибки.