**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 дисциплине «Функциональное программирование»

Выполнил: студент группы БВТ1801

Задоркин Максим Александрович

Руководитель:

Мосева Марина Сергеевна

Москва 2020

**Цель работы:**

Изучить ScalaTest. Узнать, как создать проект Scala, использующий ScalaTest(одна из основных библиотек тестирования для проектов Scala). Скомпилировать, протестировать и запустить проект с SBT.

**Задача:**

На основе ранее проделанных лабораторных работ создать проект и провести тесты для всех выполненных заданий. Так же проверить покрытие тестами кода.

**Код тестирующей программы:**

**Git:** <https://github.com/MaximZar/scalaUniversity>

**Laba1Tests.scala**

package laba1

import org.scalatest.FunSuite

class Laba1Tests extends FunSuite {

*// Functions.scala*

    test("Functions.scala testCircle") {

        val testCircle = Functions.testCircle(2.0)

        assert(testCircle == 12.566370614359172)

    }

    test("Functions.scala testRectangleCurried") {

        val testRectangleCurried = Functions.testRectangleCurried(3, 2);

        assert(testRectangleCurried == 6)

    }

    test("Functions.scala testRectangleUc") {

        val testRectangleUc = Functions.testRectangleUc(3, 2);

        assert(testRectangleUc == 6)

    }

*// HiOrder.scala*

    test("HiOrder.scala testNTimes") {

        val testNTimes = HigherOrder.testNTimes(HigherOrder.plus, 7, 9, 3)

        assert(testNTimes == 48)

    }

    test("HiOrder.scala testAnonymousNTimes") {

        val testAnonymousNTimes = HigherOrder.testAnonymousNTimes(7, 9, 2)

        assert(testAnonymousNTimes == 18)

    }

*// Patterns.scala*

    test("Patterns.scala testIntToString") {

        val testIntToString = PatternMatching.testIntToString(3)

        assert(testIntToString == "it is three")

        val testIntToString2 = PatternMatching.testIntToString(2)

        assert(testIntToString2 == "it is two")

        val testIntToString3 = PatternMatching.testIntToString(5)

        assert(testIntToString3 == "what's that")

    }

    test("Patterns.scala testIsMaxAndMoritz") {

        val testIsMaxAndMoritz = PatternMatching.testIsMaxAndMoritz("Moritez")

        assert(testIsMaxAndMoritz == **false**)

        val testIsMaxAndMoritz2 = PatternMatching.testIsMaxAndMoritz("Max")

        assert(testIsMaxAndMoritz2 == **true**)

    }

    test("Patterns.scala testIsEven") {

        val testIsEven = PatternMatching.testIsEven(8)

        assert(testIsEven == **true**)

        val testIsEven2 = PatternMatching.testIsEven(3)

        assert(testIsEven2 == **false**)

    }

    test("Patterns.scala testWinsA") {

        val winsA = PatternMatching.testWinsA(PatternMatching.Paper,PatternMatching.Rock)

        assert(winsA == PatternMatching.Win)

        val winsA2 = PatternMatching.testWinsA(PatternMatching.Rock,PatternMatching.Scissor)

        assert(winsA2 == PatternMatching.Win)

        val winsA3 = PatternMatching.testWinsA(PatternMatching.Scissor,PatternMatching.Paper)

        assert(winsA3 == PatternMatching.Win)

    }

    test("Patterns.scala testExtractMammalWeight") {

        val food = PatternMatching.Vegetables

        val mammal = new PatternMatching.Mammal("cat", food, 50)

        val testExtractMammalWeight = PatternMatching.testExtractMammalWeight(mammal)

        assert(testExtractMammalWeight == 50)

    }

    test("Patterns.scala testUpdateFood") {

        val plants = PatternMatching.Plants

        val food = PatternMatching.Vegetables

        val fish = new PatternMatching.Fish("fish", food);

        val testUpdateFood = PatternMatching.testUpdateFood(fish)

        assert(testUpdateFood.food == plants)

        val bird = new PatternMatching.Bird("fish", food);

        val testUpdateFood2 = PatternMatching.testUpdateFood(bird)

        assert(testUpdateFood2.food == plants)

    }

}

**Laba2Tests.scala**

package laba2

import org.scalatest.FunSuite

class Laba2Tests extends FunSuite {

*// RecursiveData.scala*

    test("RecursiveData.scala testListIntEmpty") {

        val list = Nil[Int]()

        val testListIntEmpty = RecursiveData.testListIntEmpty(list)

        assert(testListIntEmpty == **true**)

    }

    test("RecursiveData.scala testListIntHead") {

        val listNil = Nil[Int]()

        val listCons = Cons[Int](2, listNil)

        val testListIntHead = RecursiveData.testListIntHead(listNil)

        assert(testListIntHead == -1)

        val testListIntHead2 = RecursiveData.testListIntHead(listCons)

        assert(testListIntHead2 == 2)

    }

*// RecursiveFunc.scala*

    test("RecursiveFunc.scala testReverse") {

        val listPre = Cons[Int](2, Cons[Int](1, Nil[Int]()))

        val listPost = Cons[Int](1, Cons[Int](2, Nil[Int]()))

        val testReverse = RecursiveFunctions.testReverse(listPre)

        assert(testReverse == listPost)

    }

    test("RecursiveFunc.scala testMap") {

        val listPre = Cons[Int](2, Cons[Int](4, Nil[Int]()))

        val anonFunc = (a: Int) => a \* 2

        val testMap = RecursiveFunctions.testMap(listPre, anonFunc)

        val listResult = Cons[Int](4, Cons[Int](8, Nil[Int]()))

        assert(testMap == listResult)

    }

    test("RecursiveFunc.scala testAppend") {

        val listPre = Cons[Int](2, Cons[Int](4, Nil[Int]()))

        val listPost = Cons[Int](3, Cons[Int](1, Nil[Int]()))

        val testAppend = RecursiveFunctions.testAppend(listPre, listPost)

        val listResult = Cons[Int](2, Cons[Int](4, listPost))

        assert(testAppend == listResult)

    }

    test("RecursiveFunc.scala testFlatMap") {

        val func: List[Int] => List[Int] = \_ **match**{

**case** Cons(head, tail) => Cons(head,tail)

**case** Nil() => Nil()

        }

        val list = Cons[Cons[Int]](Cons[Int](1 , Nil[Int]()), Cons[Cons[Int]](Cons[Int](2 , Nil[Int]()), Nil[Cons[Int]]()))

        val list1 = Cons[Int](1 , Cons[Int](2 , Nil[Int]()))

        val testFlatMap = RecursiveFunctions.testFlatMap(list, func)

        assert(testFlatMap == list1)

    }

*// Compositions.scala*

    test("Compositions.scala testCompose") {

        def f(a: Int) = a + 2

        def g(a: Int) = a + 3

        def h(a: Int) = a + 4

        def result (a: Int) = a + 2 + 3 + 4

        val testCompose = Compositions.testCompose(f)(g)(h)

        assert(testCompose(2) == result(2))

    }

    test("Compositions.scala testMapFlatMap") {

        val f = (value:Int) => Some[Int](value + 1)

        val g = (value:Int) => Some[Int](value - 2)

        val h = (value: Int) => value - 13

        val res = Some[Int](-12)

        val testMapFlatMap = Compositions.testMapFlatMap(f)(g)(h)

        assert(testMapFlatMap(Some[Int](2)) == res)

    }

    test("Compositions.scala testForComprehension") {

        val f = (value:Int) => Some[Int](value + 1)

        val g = (value:Int) => Some[Int](value - 2)

        val h = (value: Int) => value - 13

        val res = Some[Int](-12)

        val testForComprehension = Compositions.testForComprehension(f)(g)(h)

        assert(testForComprehension(Some[Int](2)) == res)

    }

}

**Laba3Tests.scala**

package laba3

import org.scalatest.FunSuite

import scala.util.{Failure, Success, Try}

class Laba3Tests extends FunSuite {

*// String.scala*

    test("String.scala testUppercase") {

        assert(Strings.testUppercase("Hello") == "HELLO")

    }

    test("String.scala testInterpolations") {

        val testInterpolations = Strings.testInterpolations("Maxim", 19)

        val result = "Hi my name is Maxim and I am 19 years old."

        assert(testInterpolations == result)

    }

    test("String.scala testComputation") {

        val testComputation = Strings.testComputation(100, 200)

        val result = "Hi, \n now follows a quite hard calculation. We try ta add: \n\ta := 100 \n\tb := 200\n\n\tresult is 300"

        assert(testComputation == result)

    }

    test("String.scala testTakeTwo") {

        val testTakeTwo = Strings.testTakeTwo("qwerty")

        val result = "qw"

        assert(testTakeTwo == result)

    }

*// Sequence.scala*

    test("Sequence.scala testLastElement") {

        val seq = Seq[Int](1, 2, 3)

        val testLastElement = Sequence.testLastElement(seq)

        val result = Option[Int](3)

        assert(testLastElement == result)

    }

    test("Sequence.scala testZip") {

        val testZip = Sequence.testZip(Seq[Int](1, 2), Seq[Int](3, 4))

        val result = Seq((1, 3), (2, 4))

        assert(testZip == result)

    }

    test("Sequence.scala testForAll") {

        val cond: Int => Boolean = \_ % 2 == 0

        val testForAll = Sequence.testForAll(Seq(1, 2))(cond)

        val result = **false**

        assert(testForAll == result)

    }

    test("Sequence.scala testPalindrom") {

        val testPalindrom = Sequence.testPalindrom(Seq(2, 5, 5, 2))

        val result = **true**

        assert(testPalindrom == result)

    }

    test("Sequence.scala testFlatMap") {

        val seq = Seq(1,(5, 2),3,4)

        val func = (value: Any) => Seq(Seq(value))

        val testFlatMap = Sequence.testFlatMap(seq)(func)

        val result = Seq(Seq(1), Seq((5, 2)), Seq(3), Seq(4))

        assert(testFlatMap == result)

    }

*// Maps.scala*

    test("Maps.scala testGroupUsers") {

        val users = Seq(Maps.User("Maxim", 19), Maps.User("Oleg", 18), Maps.User("Bashar", 20))

        val testGroupUsers = Maps.testGroupUsers(users)

        val result = Map("Bashar" -> 19, "Maxim" -> 19, "Oleg" -> 19)

        assert(testGroupUsers == result)

    }

    test("Maps.scala testNumberFrodos") {

        val users = Map("1" -> Maps.User("Adamas", 20),

            "2" -> Maps.User("Maxim", 19), "3" -> Maps.User("Adam", 20))

        val testNumberFrodos = Maps.testNumberFrodos(users)

        val result = 2

        assert(testNumberFrodos == result)

    }

    test("Maps.scala testUnderaged") {

        val users = Map("1" -> Maps.User("Oleg", 36),

            "2" -> Maps.User("Maxim", 19), "3" -> Maps.User("Bashar", 40))

        val testUnderaged = Maps.testUnderaged(users)

        val result = Map("1" -> Maps.User("Oleg", 36),

            "3" -> Maps.User("Bashar", 40))

        assert(testUnderaged == result)

    }

*// Adts.scala*

    test("Adts.scala testGetNth") {

        val testGetNth = Adts.testGetNth(List(1, 2, 3, 4), 2)

        val result = Option(3)

        assert(testGetNth == result)

        val testGetNth2 = Adts.testGetNth(Nil, 2)

        val result2 = **null**

        assert(testGetNth2 == result2)

    }

    test("Adts.scala testDouble") {

        val testDouble = Adts.testDouble(Option(5))

        val result = Option(10)

        assert(testDouble == result)

        val testDouble2 = Adts.testDouble(None)

        val result2 = **null**

        assert(testDouble2 == result2)

    }

    test("Adts.scala testIsEven") {

        val testIsEven = Adts.testIsEven(5)

        val result = Left("Нечетное число")

        assert(testIsEven == result)

    }

    test("Adts.scala testSafeDivide") {

        val testSafeDivide = Adts.testSafeDivide(10, 2)

        val result = Right(5)

        assert(testSafeDivide == result)

        val testSafeDivide2 = Adts.testSafeDivide(9, 0)

        val result2 = Left("Вы не можете делить на ноль")

        assert(testSafeDivide2 == result2)

    }

    test("Adts.scala testGoodOldJava") {

        val string = "test"

        val func = (s:String) => s.size

        val result = Success(4)

        val goodOldJava = Adts.testGoodOldJava(func, string)

        assert(goodOldJava == result)

    }

}

**Laba4Tests.scala**

package laba4

import org.scalatest.FunSuite

class Laba4Tests extends FunSuite {

    test("Typeclasses.scala ReversableString") {

        val string = "abc"

        val result = "cba"

        val reversableString = Typeclasses.testReversableString(string)

        assert(reversableString == result)

    }

    test("Typeclasses.scala SmashInt") {

        val a = 3

        val b = 4

        val result = 7

        val smashInt = Typeclasses.testSmashInt(a, b)

        assert(smashInt == result)

    }

    test("Typeclasses.scala SmashDouble") {

        val a = 3

        val b = 4

        val result = 12

        val smashDouble = Typeclasses.testSmashDouble(a, b)

        assert(smashDouble == result)

    }

    test("Typeclasses.scala SmashString") {

        val a = "ab"

        val b = "cd"

        val result = "abcd"

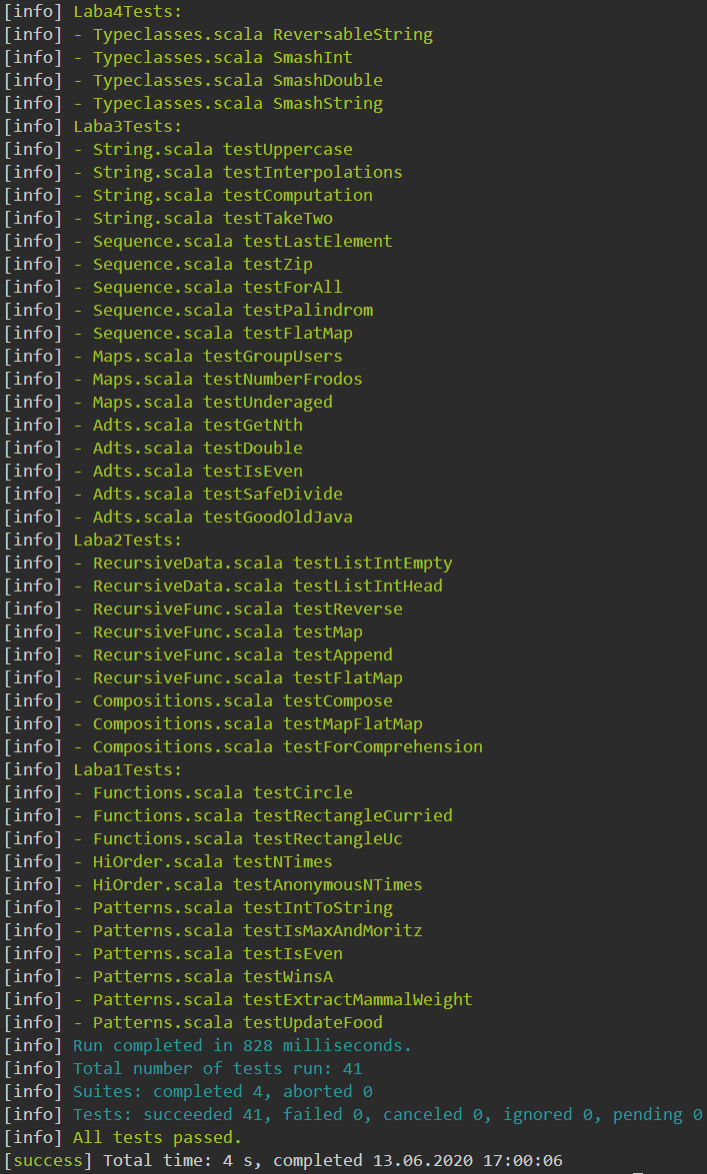
        val smashString = Typeclasses.testSmashString(a, b)

        assert(smashString == result)

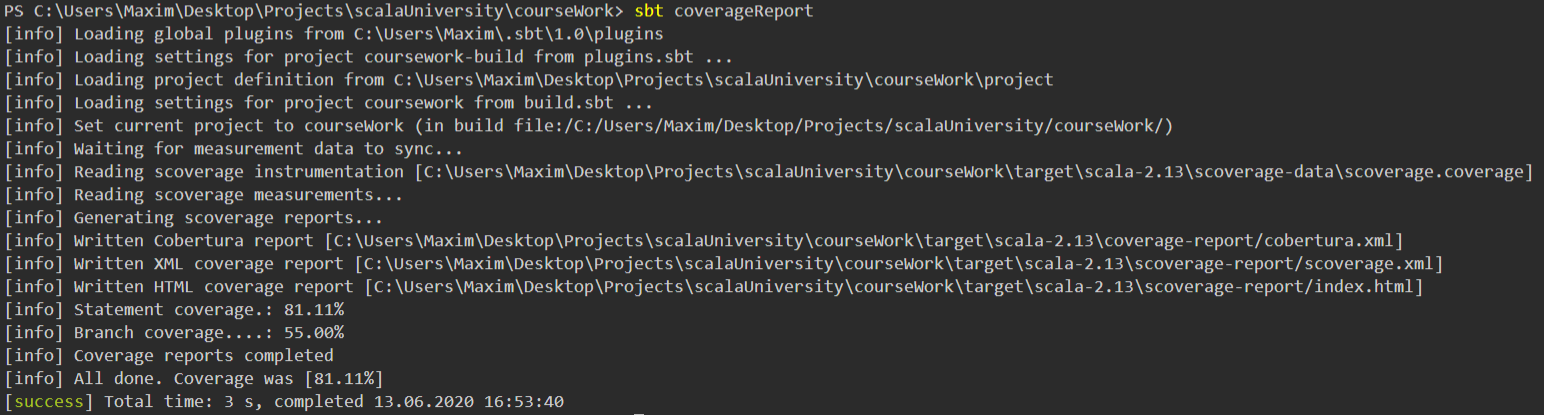
    }

}

**Итоговое тестирование лабораторных работ:**



**Coverage тест:**



Минимально необходимый порог: 73%  
Результаты coverage теста: 81.11%

**Вывод:**

Изучена библиотека ScalaTest. В процессе изучения были освоены навыки тестирования языка scala и работа coverage тестов.