**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 лабораторной работе**

по предмету «Функциональное программирование»

Выполнил: студент группы

Митрохин Ярослав Игоревич

Руководитель:

Мосева Марина Сергеевна

Москва 2020

*Цель работы:* реализовать представленные классы, учитывая заданный синтаксис.

*Ход работы:*

*Реализованный класс Compositions(код программы):*

sealed trait Option[A]  
{  
 def map[B](f: A => B): Option[B]  
 def flatmap[B](f: A => Option[B]):Option[B]  
}  
case class Some[A](a: A) extends Option[A]  
{  
 def map[B](f: A => B): Option[B] = *Some*(f(a))  
 def flatmap[B](f: A => Option[B]):Option[B] = f(a)  
}  
case class None[A]() extends Option[A]  
{  
 def map[B](f: A => B): Option[B] = *None*()  
 def flatmap[B](f: A => Option[B]):Option[B] = *None*()  
}  
object Compositions {  
 def testCompose[A,B,C,D](f: A=>B)(g: B=>C)(h: C=>D): A => D = h compose g compose f  
 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  
 //def testForComprehension[A,B,C,D](f: A=>Option[B])(g: B=>Option[C])(h: C=>D): Option[A] => Option[D] = \_ //непонятно}

*Реализовать класс RecursiveFunc(код программы):*

import scala.annotation.tailrec  
  
object RecursiveFunctions {  
  
 def length[A](as: List[A]): Int = {  
 @tailrec  
 def loop(rem: List[A], agg: Int): Int = rem match{  
 case *Cons*(\_,tail) => loop(tail, agg+1)  
 case *Nil*() => agg  
 }  
 loop(as,0)  
 }  
  
 def main(args: Array[String]): Unit = {  
 val a = *Cons*(1, *Cons*(2, *Cons*(3, *Cons*(4, *Nil*()))))  
 val b = *Cons*(4, *Cons*(3, *Cons*(2, *Cons*(1, *Nil*()))))  
 val c = *Cons*(*Cons*(4, *Cons*(2, *Nil*())), *Cons*(*Cons*(5, *Cons*(4, *Nil*())), *Nil*()))  
 val sw = (x: Int) => x.toDouble  
 val de = (l: List[Int]) => l  
 *println*(*flatMap*(c)(de))  
 }  
  
  
 def reverse[A](list: List[A]): List[A] =  
 {  
 def rev(a: A, l: List[A]): List[A] = *Cons*(a,l)  
 @tailrec  
 def loop(rem: List[A],num: List[A]): List[A] = rem match {  
 case *Nil*() => num  
 case *Cons*(x,y) => loop(y,rev(x,num))  
 }  
 loop(list,*Nil*())  
 }  
  
 def testReverse[A](list: List[A]): List[A] = *reverse*(list)  
  
 def map[A,B](list: List[A])(f: A=>B): List[B] =  
 {  
 def rev(a: A, l: List[B]): List[B] = *Cons*(f(a),l)  
 @tailrec  
 def loop2(l: List[A],num: List[B]): List[B] = l match {  
 case *Nil*() =>*reverse*(num)  
 case *Cons*(x,y) => loop2(y,rev(x,num))  
 }  
 loop2(list,*Nil*())  
 }  
  
 def testMap[A,B](list: List[A], f: A=>B): List[B] = *map*(list)(f)  
  
 def append[A](l: List[A],r: List[A]): List[A] = l match  
 {  
 case *Nil*() => r  
 case *Cons*(x,y)=>*Cons*(x,*append*(y,r))  
 }  
  
 def testAppend[A](l: List[A],r:List[A]): List[A]= *append*(l,r)  
  
 def flatMap[A,B](list: List[A])(f: A=>List[B]): List[B] =  
 {  
 def rev(a: A, l: List[B]): List[B] = *append*(f(a),l)  
 @tailrec  
 def loop2(l: List[A],num: List[B]): List[B] = l match {  
 case *Nil*() =>*reverse*(num)  
 case *Cons*(x,y) => loop2(y,rev(x,num))  
 }  
 loop2(list,*Nil*())  
 }  
   
 def testFlatMap[A,B](list: List[A], f:A=>List[B]): List[B] = *flatMap*(list)(f)  
}  
//e) Возможно ли написать функцию с хвостовой рекурсией для 'Tree's? Если нет, почему?  
// Да, можно, например: def eval(t: Tree, env: Environmental): Int = t match {case Sum(l,r)=>eval(l,env)+eval(r,env) - начение суммы двух выражений просто сумма двух выражений просто сумма значений этих выражений  
// case Var(n)=>env(n)  
// case Const(v)=>v}

*Реализовать класс RecursiveData(код программы):*

sealed trait List[A]  
case class Cons[A](head: A, tail: List[A]) extends List[A]  
case class Nil[A]() extends List[A]  
  
sealed trait Tree[A]  
case class Leaf[A](value: A) extends Tree[A]  
case class Node[A](left: Tree[A],right: Tree[A]) extends Tree[A]  
  
object RecursiveData {  
  
 def ListIntEmpty(list: List[Int]): Boolean = list match{  
 case Nil() => true  
 case \_ => false  
}  
 def testListIntEmpty(list: List[Int]): Boolean = *ListIntEmpty*(list)  
  
 def ListIntHead(list: List[Int]): Int = list match{  
 case Cons(head,tail) => head  
 case Nil() => -1}  
 def testListIntHead(list: List[Int]): Int = *ListIntHead*(list)  
 //c) Можно ли изменить List[A] так чтобы гарантировать что он не является пустым?  
 // Можно, если воспользоваться типом ListBuffer, либо добавить значение "head" в класс Nil}

*Выводы:* реализовал заданные классы, изучил конструкцию Option[], научился создавать списки и деревья, реализовал стандартные функции scala. В конце получен корректный результат.