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 № 2

по дисциплине «Функциональное программирование»

на тему:

«Основные структуры данных. Функциональные комбинаторы. Сопоставление с образцом и функциональная композиция.»

Выполнила: студентка группы БВТ1802

Лаврухина Елена Павловна

Руководитель:

Мосева Марина Сергеевна

Москва 2020

Выполнение

Код программы

1. RecursiveData

```
  sealed trait List[A]
  case class Cons[A](head: A, tail: List[A]) extends List[A]
  case class Nil[A]() extends List[A]
  /*Напишите свои решения в виде функций.*/
  object RecursiveData {
    //a) Реализуйте функцию, определяющую является ли пустым `List[Int]`.
    def ListIntEmpty(list: List[Int]): Boolean =
      list match {
        case Nil() => true
        case Cons(head, tail) => false
      }
    // используйте функцию из пункта (a) здесь, не изменяйте сигнатуру
    def testListIntEmpty(list: List[Int]): Boolean = ListIntEmpty(list)
    //b) Реализуйте функцию, которая получает head `List[Int]`или возвращает -1 в
  случае если он пустой.
    def ListIntHead(list: List [Int]): Int = list match {
      case Cons(head, tail) => head
      case Nil() => -1
    }
    // используйте функцию из пункта (a) здесь, не изменяйте сигнатуру
    def testListIntHead(list: List[Int]): Int = ListIntHead(list)
    /*c) Можно ли изменить `List[A]` так, чтобы гарантировать, что он не является
  пустым?
          Для этого нужно использовать Nil[A](head: A) вместо Nil[A]() */
    /*d) Реализуйте универсальное дерево (Tree), которое хранит значения в виде
  листьев и состоит из:
          node - левое и правое дерево (Tree)
          leaf - переменная типа A */
    sealed trait Tree[A]
    case class TCons[A](leaf: A, node: (Tree[A],Tree[A])) extends Tree[A]
    case class TNil[A]() extends Tree[A]
  }
```

2. RecursiveFunc

```
import scala.annotation.tailrec
/*Реализуйте функции для решения следующих задач.
  Примечание: Попытайтесь сделать все функции с хвостовой рекурсией, используйте
аннотацию для подтверждения.
  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 Cons(_, tail) => loop(tail, agg + 1)
      case Nil()         => agg
    }
    loop(as, 0)
  }
  /*a) Напишите функцию которая записывает в обратном порядке список:
       def reverse[A](list: List[A]): List[A] */
  def Reverse[A](list: List[A]): List[A] = {
    def rever(a: A, l: List[A]): List[A] = Cons(a, l)
    def loop(rem: List[A], num: List[A]): List[A] = rem match {
      case Nil() => num
      case Cons(x, y) => loop (y, rever (x, num) )
```

```
    }
    loop(list, Nil())
  }
  // используйте функцию из пункта (a) здесь, не изменяйте сигнатуру
  def testReverse[A](list: List[A]): List[A] = Reverse(list)
  /*b) 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] = {
    def rever(a: A, l: List[B]): List[B] = Cons(f(a), l)
    @tailrec
    def loop(l: List[A], num: List[B]): List[B] = l match {
      case Nil() => Reverse(num)
      case Cons(x,y) => loop(y,rever(x,num))
    }
    loop(list,Nil())
  }
  // используйте функцию из пункта  (b) здесь, не изменяйте сигнатуру
  def testMap[A, B](list: List[A], f: A => B): List[B] = map(list)(f)
  /*c) 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] =
    l match {
      case Nil() => r
      case Cons(h,t) => Cons(h, append(t, r))
    }
  // используйте функцию из пункта  (c) здесь, не изменяйте сигнатуру
  def testAppend[A](l: List[A], r: List[A]): List[A] = append(l,r)
  /*d) 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] = {
    def loop (rem: List[A], as:List[B], f: A => List[B]): List[B] = {
      rem match {
        case Cons (head, tail) => as match {
          case Cons (ahead, ateil)=> loop(tail, append (as, f(head)), f)
          case Nil() => loop (tail, f(head), f)
        }
        case Nil() => as
      }
    }
    loop(list, Nil() , f)
  }
  // используйте функцию из пункта  (d) здесь, не изменяйте сигнатуру
  def testFlatMap[A, B](list: List[A], f: A => List[B]): List[B] = flatMap(list)(f)
  /*e) Вопрос: Возможно ли написать функцию с хвостовой рекурсией для `Tree`s`? Если
нет, почему?
       Возможно, если дерево будет иметь отсортированный вид.
       def eval(t: Tree, env: Environmental): Int = t match {
       case Sum(l,r) => eval(l, env) + eval (r, env)
       } */
}
```

3. Compositions

```
  /*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.*/
  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()
}
/*Напишите ваши решения в тестовых функциях.*/
object Compositions {
  //a) 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
  //b) 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)
  //c) 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] = { a =>
                                      for { first <- a
                                            second <- f(first)
                                            third <- g(second)
                                            } yield h(third)
                                      }
}
```