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

БВТ1802

Самаков Владислав Владимирович

Руководитель:

Мосева Марина Сергеевна

Москва 2020

Задания к работе указаны в файлах с кодом.

Выполнение:

Compositions.scala

```scala
/** 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 {

  // 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] = {

    for { first <- _
          second <- f(first)
          third <- g(second) } yield h(third)

  }

}
```

RecursiveFunc.scala

```scala
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 подствержения.
* 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)
  }

  /* 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] = {
    @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 (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] = {
    @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  (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] = {
    @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  (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[List[B]] = {
    @tailrec
    def loop(rem: List[A], result: List[List[B]])(f: A => List[B]): List[List[B]] =
rem match {
      case x :: tail => loop(tail, result :+ f(x))(f)
      case Nil       => result
    }
    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[List[B]] =
FlatMap(list)(f)

  /* e) Вопрос: Возможно ли написать функцию с хвостовой рекурсией для `Tree`s? Если
нет, почему? */

      // Нет. Одним из признаков хвостовой рекурсии являтся рекурсия в одном
направлении, что невозможно для древовидной структуры.
}
```

RecursiveData.scala

```
import scala.collection.immutable.List



/** 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 x :: tail   => true
    case Nil         => 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 x :: tail   => x
    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?

  def ListNotEmpty[A](head: A, list: List[A]) : List[A] = list match {
    case Nil          => head :: list
    case x :: tail    => list
  }

  /* d) Реализуйте универсальное дерево (Tree) которое хранит значения в виде листьев
и состоит из:
   *      node - левое и правое дерево (Tree)
   *      leaf - переменная типа A
   */

  class Tree[A](LeftNode: Tree[A], RightNode: Tree[A], leaf: A) {}

}
```