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»

на тему:

«Язык программирования «Scala» ЛР 2»

Выполнил: студент группы

БВТ1802

Дворянинов Павел Владимирович

Руководитель:

Мосева Марина Сергеевна

Москва 2020

Выполнение

File 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 extends App{
  /*
   * 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 {
                                                                    first <- _

                                                                second <- f(first)

                                                            third <- g(second) } yield h(third)

  println(testCompose((i:Int) => "Compose" * i)((i: String) => i * 2)((i:String)
  => i.dropRight(3))(2))

  println(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)))

  println(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)))
}
```

File RecursiveData.scala

```
import scala.annotation.tailrec
import scala.collection.immutable.List
/*
 * Напишите свои решения в виде функций.
 */
object RecursiveData extends App {

  /*
   * 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)
```

```
  /*
   * 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
  }

  println("ListNotEmpty(Nil): \t" + ListNotEmpty(1, Nil))
  println("ListNotEmpty(List(1, 2, 3): \t" + ListNotEmpty(1, List(1, 2, 3)))
  println("ListIntHead(List(1, 2, 3): \t" + testListIntHead(List(1, 2, 3)))
}
```

File Recursive.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  extends App {

  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)
  }
```

```scala
/*
 * 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[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 здесь, не изменяйте сигнатуру
   */
  def testFlatMap[A, B](list: List[A], f: A => List[B]): List[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, потому что
   * хвостовая рекурсия - это
   * рекурсия в одном направлении, что не подходит для древовидной структуры
   */
  println("Reverse Test List(1, 2, 3):\t " + testReverse(List(1, 2, 3)))
  println("Append List(1, 2, 3) and List(4, 5, 6):\t " +
  testAppend(List(1, 2, 3), List(4, 5, 6)))
}
```

Результаты работы программы

File Compositions.scala

```
D:\4 семестр\(экзамен) ФП>scala LAB_2.scala
ComposeComposeComposeComp
None
Some(22)
```

File RecursiveData.scala

```
D:\4 семестр\(экзамен) ФП>scala LAB_2.scala
ListNotEmpty(Nil):              List(1)
ListNotEmpty(List(1, 2, 3):     List(1, 2, 3)
ListIntHead(List(1, 2, 3):      1
```

File Recursive.scala

```
D:\4 семестр\(экзамен) ФП>scala LAB_2.scala
Reverse Test List(1, 2, 3):      List(3, 2, 1)
Append List(1, 2, 3) and List(4, 5, 6):  List(1, 2, 3, 4, 5, 6)
```