Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

*Кафедра «Математическая кибернетика и информационные технологии»*

Отчет по лабораторной работе №2

Выполнила: студентка группы БВТ1903

Лаптева Анастасия

Руководитель:

Мосева Марина Сергеевна

**Файл 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  
*/\*\*  
 flatmap. Берёт слева нашу переменную. Она у нас имеет тип Option  
 Сам тип Option либо хранит переменную в виде типа Some, либо хранит в себе None  
 flatMap смотрит налево, если слева находится None() - возвращает тоже None(). Если слева есть переменная в типе Some  
 тогда он берёт эту переменную и делает с ней функцию, которая указана справа.  
 Возвращает, соответственно, тоже Some.  
 map. Работает аналогично, но не может вернуть нам None(). map вернёт Some(None())  
 \*/*// 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 {  
 a <- \_  
 b <- f(a)  
 c <- g(b)  
 } yield h(c)  
}  
 */\*\*  
 \* for yield - конструкция в Scale  
 \* внутри for - список  
 \* yield берёт из этого списка что мы укажем.  
 \* for не цикл, а список для формирования формул. В нашем случае формула h(g(f(\_)))  
\*/*def A(x: Char): Int = x + 5  
def B(x: Int): Double = x \* 2 + 1  
def C(x: Double): Int = (x / 2).toInt  
  
def AA(x: Char): Option[Int] = *Some*(x + 5)  
def BB(x: Int): Option[Double] = *Some*(x \* 2 + 1)  
def CC(x: Double): Int = (x / 2).toInt

**Результат работы программы:**

Изображение выглядит как текст, снимок экрана, монитор, ноутбук

Автоматически созданное описание

**Файл RecursiveData**

// a) Реализуйте функцию, определяющую является ли пустым `List[Int]`.  
def listIsEmpty(list: List[Int]): Boolean = list == *Nil*()  
  
  
// используйте функцию из пункта (a) здесь, не изменяйте сигнатуру  
def testListIntEmpty(list: List[Int]): Boolean = *listIsEmpty*(list)  
  
// b) Реализуйте функцию, которая получает head `List[Int]`или возвращает -1 в случае если он пустой.  
def listIsHead(list: List[Int]): Int = list match {  
 case list: Cons[Int] => list.head  
 case \_ => -1  
}  
  
  
// используйте функцию из пункта (б) здесь, не изменяйте сигнатуру  
def testListIntHead(list: List[Int]): Int = *listIsHead*(list)  
  
// c) Можно ли изменить `List[A]` так чтобы гарантировать что он не является пустым?  
sealed trait ListNotEmpty[A]  
case class ConsNotEmpty[A](head: A, tail: ListNotEmpty[A]) extends ListNotEmpty[A]  
case class End[A](head: A) extends ListNotEmpty[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 Node[A](left: Tree[A], right: Tree[A]) extends Tree[A]  
case class Leaf[A](head: A) extends Tree[A]

**Результат работы программы:**

Изображение выглядит как текст, монитор, снимок экрана, внутренний

Автоматически созданное описание

**Файл RecursiveFunc**

/\* a) Напишите функцию которая записывает в обратном порядке список: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(origin: List[A], reverse: List[A]): List[A] = origin match {  
 case *Cons*(n, t) => loop(t, *Cons*(n, reverse))  
 case *Nil*() => reverse  
 }  
 loop(list, new Nil[A])  
}  
  
  
// 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(origin: List[A], changed: List[B]): List[B] = origin match {  
 case *Cons*(n, t) => loop(t, *Cons*(f(n), changed))  
 case *Nil*() => *reverse*(changed)  
 }  
 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] = {  
 @tailrec  
 def loop(left: List[A], right: List[A] ): List[A] = left match {  
 case *Cons*(n, t) => loop(t, *Cons*(n, right))  
 case *Nil*() => right  
 }  
 loop(*reverse*(l), 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 => B): List[B] = {  
 @tailrec  
 def loop(origin: List[A], result: List[B]): List[B] = origin match {  
 case *Cons*(n, t) => loop(t, *Cons*(f(n), result))  
 case *Nil*() => *reverse*(result)  
 }  
 loop(list,new Nil[B])  
}  
  
  
// используйте функцию из пункта (d) здесь, не изменяйте сигнатуру  
def testFlatMap[A, B](list: List[A], f: A => List[B]): List[B] = *Nil*()  
  
/\* e) Вопрос: Возможно ли написать функцию с хвостовой рекурсией для `Tree`s? Если нет, почему? \*/  
*/\*\* Если мы хотим найти какое-то значение в бинарном дереве поиска, то алгоритм можно  
 \* реализовать через хвостовую рекурсию. Если же нам необходимо проверить одновременно оба значения в Node, тогда  
 \* реализация хвостовой рекурсии может стать невыполнимой из-за необходимости запустить по функции для каждого из 2  
 \* значений  
 \*/*

**Результат работы программы:**

Изображение выглядит как текст, снимок экрана, ноутбук, внутренний

Автоматически созданное описание