**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 дисциплине «Функциональное программирование»

На тему:

«Функциональная композиция»

Выполнил:

студент группы БВТ1801

Задоркин Максим Александрович

Москва 2020

**Цель работы:** изучение функциональной композиции scala

**Задание:**

Решение задач из приложенных файлов

**Ход работы:**

Рекурсивные функции

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)

}

*/\* 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)

@tailrec

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 loop2(l: List[A], num: List[B]): List[B] = l **match** {

**case** Nil() => reverse(num)

**case** Cons(x,y) => loop2(y,rever(x,num))

}

loop2(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 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())

}

*// 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:*

*\* def eval(t: Tree, env: Environmental): Int = t match {*

*\* case Sum(l,r) => eval(l, env) + eval (r, env)*

*\* }*

*\* \*/*

}

Композиция:

**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 = f andThen g andThen h

*// 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] = {

val l = List(f,g,h)

var func = () => ()

**for** (item of l) {

func = func andThen l

}

func

}

}

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 a(list: List[Int]): Boolean = list **match** {

**case** Nil() => **true**

**case** \_ => **false**

}

*// используйте функцию из пункта (a) здесь, не изменяйте сигнатуру*

def testListIntEmpty(list: List[Int]): Boolean = a(list)

*// b) Реализуйте функцию, которая получает head `List[Int]`или возвращает -1 в случае если он пустой.*

def b(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 = b(list)

*// c) Можно ли изменить `List[A]` так чтобы гарантировать что он не является пустым?*

*// Нужно добавить head в класс Nil*

}

**Выводы:**

В процессе лабораторной работы были изучены основы функциональной композии языка scala.