ФГАОУ ВО «Севастопольский государственный университет»

Институт информационных технологий   
и управления в технических системах

ОТЧЁТ  
ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

«Исследование способов реализации алгоритмов сопоставления с образцом   
в языке Scala»

по дисциплине «Парадигмы современных языков программирования»

Выполнила:  
студентка группы ИС/м-21-1-з  
Ускова Екатерина Дмитриевна

Севастополь

2022

# ЦЕЛЬ РАБОТЫ

Исследовать особенности реализации алгоритмов сопоставления с образцом в языке Scala. Получить навыки использования case-классов и класса Option в функциональном программировании.

# Ход работы

Все задания выполнены с помощью сопоставления с образцом (match) и не используя условные выражения (if).

Написана функция типа (List[(Int, Int)]) => List[Option[Double]], которая принимает на вход список из пар двух целочисленных значений и возвращает список результатов деления первого числа пары на второе в виде Option (Some[Double]), если второй элемент пары не равен нулю, или None, если второй элемент пары равен нулю).

def division(numbers: List[(Int, Int)]): List[Option[Double]] = {  
 numbers match {  
 case Nil => Nil  
 case head :: tail =>  
 val res = head match {  
 case (\_, 0) => None  
 case \_ => Some((head.\_1).toDouble / head.\_2)  
 }  
 res :: division(tail)  
 }  
}

Написана функция типа (List[Option[Double]]) => List[String], которая принимает на вход список Option’ов типа Double (результаты работы функции из п.1) и преобразует его в список строк по следующему правилу: значения Some преобразуются в строку «Результат деления = \_\_число\_из\_Some\_\_», а значения None преобразуются в строку «Деление на ноль невозможно».

def doubleListToStr(numbers: List[Option[Double]]): List[String] = {  
 numbers match {  
 case Nil => Nil  
 case head :: tail =>  
 val res = head match {  
 case Some(number) => "Результат деления = " + number  
 case None => "Деление на ноль невозможно"  
 }  
 res :: doubleListToStr(tail)  
 }  
}

Реализовано задание из п.1, но вместо пар использован case-класс.

case class divPair(divisible: Int, divider: Int) {  
 def getResult: Option[Double] = {  
 divider match {  
 case 0 => None  
 case \_ => Some(divisible.toDouble / divider)  
 }  
 }  
}  
  
def caseDivision(numbers: List[divPair]): List[Option[Double]] = {  
 numbers match {  
 case Nil => Nil  
 case head :: tail =>  
 head.getResult :: caseDivision(tail)  
 }  
}

На основе case-классов реализована структура дерева вычислений. Написана функция, которая это дерево преобразует в строку обратной польской нотации.

abstract class Param {  
 def getData: String  
}  
case class Val(data: Int) extends Param {  
 override def getData: String = data.toString  
}  
case class Op(op: Char, param1: Param, param2: Param) extends Param {  
 override def getData: String = param1.getData + param2.getData + op.toString  
}

# Вывод

Выполняя эту работу, я исследовала особенности реализации алгоритмов сопоставления с образцом в языке Scala. Получила навыки использования case-классов и класса Option в функциональном программировании.

# Приложение А

Листинг программного кода

package main.scala  
  
object Application {  
  
 def division(numbers: List[(Int, Int)]): List[Option[Double]] = {  
 numbers match {  
 case Nil => Nil  
 case head :: tail =>  
 val res = head match {  
 case (\_, 0) => None  
 case \_ => Some((head.\_1).toDouble / head.\_2)  
 }  
 res :: division(tail)  
 }  
 }  
  
 def doubleListToStr(numbers: List[Option[Double]]): List[String] = {  
 numbers match {  
 case Nil => Nil  
 case head :: tail =>  
 val res = head match {  
 case Some(number) => "Результат деления = " + number  
 case None => "Деление на ноль невозможно"  
 }  
 res :: doubleListToStr(tail)  
 }  
 }  
  
 case class divPair(divisible: Int, divider: Int) {  
 def getResult: Option[Double] = {  
 divider match {  
 case 0 => None  
 case \_ => Some(divisible.toDouble / divider)  
 }  
 }  
 }  
  
 def caseDivision(numbers: List[divPair]): List[Option[Double]] = {  
 numbers match {  
 case Nil => Nil  
 case head :: tail =>  
 head.getResult :: caseDivision(tail)  
 }  
 }  
  
 abstract class Param {  
 def getData: String  
 }  
 case class Val(data: Int) extends Param {  
 override def getData: String = data.toString  
 }  
 case class Op(op: Char, param1: Param, param2: Param) extends Param {  
 override def getData: String = param1.getData + param2.getData + op.toString  
 }  
  
  
  
 def main (args: Array[String]): Unit = {  
// val lst: List[(Int, Int)] = List((6, 3), (3, 2), (3, 0), (10, 4))  
//  
// val dList = division(lst)  
// println(dList)  
// println(doubleListToStr(dList))  
//  
// val caseList: List[divPair] = List(divPair(6, 3), divPair(3, 2), divPair(3, 0), divPair(10, 4))  
// println(caseDivision(caseList))

val expr = "3+4\*2/(1-5)^2" // res = "342\*15-2^/+"  
 val exprTree = Op(  
 '+',  
 Val(3),  
 Op(  
 '/',  
 Op(  
 '\*',  
 Val(4),  
 Val(2)  
 ),  
 Op(  
 '^',  
 Op(  
 '-',  
 Val(1),  
 Val(5)  
 ),  
 Val(2)  
 )  
 )  
 )  
  
 println(exprTree.getData)  
 }  
}