ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧЕРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Санкт-Петербургский политехнический университет Петра Великого»

Институт компьютерных наук и технологий

Курсовой проект

по дисциплине «Функциональное

программирование»

Выполнил студент гр. 3530904/80001: Муллагалиева А. Ш.

Руководитель ассистент ВШПИ Лукашин А. А.

Санкт-Петербург

2019

# 1 Задание

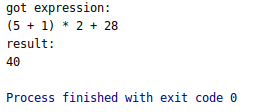
Калькулятор, поддерживающий простые арифметические операции, приоритеты и скобки.

# 2 Ход работы

## 2.1 Алгоритм решения

Используются два объекта класса Stack. Метод push помещает некоторый объект в стек сверху, при этом все другие объекты, если они есть, сдвигаются вниз. Метод pop выталкивает верхний объект из стека, при этом все другие объекты, если они есть, поднимаются вверх. Метод top возвращает самый верхний объект, не вынимая его из стека. При разборе выражения все операторы, включая скобки, в порядке появления помещаются в первый стек (actionStack), а все числа - во второй (resultStack). Перед помещением нового оператора в стек все операторы в стеке с большим или равным ему приоритетом должны быть выполнены.

## 2.2 Скриншот



# 3 Выводы

В ходе работы был изучен функциональный подход к программированию, который значительно

отличается от стандартного императивного подхода. Изучены некоторые основные алгоритмы,

используемые в функциональном программировании и произведена работа с ними.

# 4 Приложение

## 4.1 Код calc.scala

**package** calc

**import** scala.collection.mutable.Stack

**import** scala.io.Source

**abstract class** Token {

**def** pos: Int

**def** lex: String

}

**case class** Action(pos: Int, lex: String, pri: Int) **extends** Token

**case class** Result(pos: Int, lex: String, rt: Result.ResultType) **extends** Token

**object** Result **extends** Enumeration {

**type** ResultType = Value

**val** *rtString*, *rtInteger*, *rtDouble*, *rtUnknown* = Value

}

**object** Bracket **extends** Enumeration {

**type** Bracket = Value

**val** *btOpen*, *btClose*, *rtDouble*, *btNone* = Value

}

**object** Calc {

**val** *operators*=**"+-\*/^"**

**val** *spaces*=**" \n\r\t"**

**val** *delimiters*=*operators*+*spaces*+**"()"**

**var** *expr* = **""**

**var** *ind* = 0

**var** *char* = **' '**

**var** *lex* = **""**

**var** *error* = -1

**val** *actionStack* = **new** Stack[Action]

**val** *resultStack* = **new** Stack[Result]

**var** *opened* = 0

**var** *bracket* = Bracket.*btNone*

**val** *errorMsg* = *List* (

**"Неизвестная ошибка"**, *//0*

**"Остановлено пользователем"**, *//1*

**"Недопустимый символ"**, *//2*

**"Недопустимое число"**, *//3*

**"Недопустимая строка"**, *//4*

**"Недопустимая функция"**, *//5*

**"Несоответствие скобок"**, *//6*

**"Некорректное выражение"**, *//7*

**"Недопустимый аргумент"** *//8*

)

**def** isError = *error* > -1

**def** errorStr = **if** ((*error* > -1) && (*error* < *errorMsg*.length))

*errorMsg*(*error*)+**": лексема="**+*lex*+**" строка="**+*line*(*ind*)+**" позиция="**+*position*(*ind*)

**else ""**

**def** stop(error: Int) = **this**.*error*=error

**def** line(pos: Int): Int = {

**if** ((pos<1) || (pos>*expr*.length)) 1

**else** *expr*.substring(0,pos-1).count(\_==**'\n'**)+1

}

**def** position(pos: Int): Int = {

**val** posLineBreak = *expr*.lastIndexOf(**'\n'**,pos-1)

**if** (posLineBreak > -1) pos - posLineBreak **else** pos+1

}

**def** next = {

**if** (*ind* < *expr*.length) *ind*+=1

**if** (*ind* == *expr*.length) *char*=**' '**

**else** *char* = *expr*.charAt(*ind*)

}

**def** priority(op: Char): Int = op **match**{

**case '+'** | **'-'** => 1

**case '\*'** | **'/'** => 2

**case '^'** => 3

**case** \_ => 0

}

**def** open = {

**if** (*bracket*==Bracket.*btClose*) *stop*(7)

**else** {

*bracket*==Bracket.*btOpen*

*opened*+=1

*actionStack*.push(*Action*(*ind*,**"("**,0))

*next*

}

}

**def** close = {

**if** (*bracket*==Bracket.*btOpen*) *stop*(7)

**else** {

*bracket*==Bracket.*btClose*

*opened*-=1

*processOperators*

*actionStack*.pop

*next*

}

}

**def** nextNumber = {

*lex*=**""**

*bracket*==Bracket.*btNone*

**while**((**""**+*char*).matches(**"[\\d\\.]"**)) {

*lex*+=*char*

*next*

}

**if** (*delimiters*.indexOf(*char*) > -1) {

**try**{

**val** rt =

**if** (*lex*.matches(**"\\d\*"**)) {

*lex*.toInt

Result.*rtInteger*

}

**else** {

*lex*.toDouble

Result.*rtDouble*

}

*resultStack*.push(*Result*(*ind*-*lex*.length,*lex*,rt))

}

**catch** { **case** \_ : Throwable => *stop*(3)}

}

**else** *stop*(3)

}

**def** nextOperator = {

*bracket*==Bracket.*btNone*

*processOperators*

*actionStack*.push(*Action*(*ind*,**""**+*char*,*priority*(*char*)))

*next*

}

**def** overPriority(a: Action): Boolean = {

**if** (*isError*) **false**

**else if** (a.lex==**"("**) **false**

**else if** (a.lex==**")"**) **true**

**else** a.pri >= *priority*(*char*)

}

**def** processOperators = {

**if** (*actionStack*.isEmpty) *stop*(7)

**else while** (*overPriority*(*actionStack*.top)) *execOperator*

}

**def** execOperator = {

**var** unary = **false**

**if** ((*actionStack*.isEmpty) || (*resultStack*.isEmpty)) *stop*(7)

**else** {

**val** action = *actionStack*.pop

**val** result = *resultStack*.pop

**if** (*actionStack*.isEmpty) *stop*(7)

**else** {

**if** (*resultStack*.isEmpty) {

**if** ((*actionStack*.top.lex==**"("**)&&(*actionStack*.top.pos>result.pos)) *stop*(7)

**else** unary = **true**

}

**else if** ((*actionStack*.top.lex==**"("**)&&(*actionStack*.top.pos>*resultStack*.top.pos)) unary=**true**

}

**if** (!*isError*) {

**if** (unary) {

**if** (action.pos<result.pos) action.lex **match** {

**case "+"** => *resultStack*.push(result)

**case "-"** => *resultStack*.push(*minus*(result))

**case** \_ => *stop*(7)

} **else** *stop*(7)

}

**else** {

action.lex **match** {

**case "+"** => *resultStack*.push(*add*(*resultStack*.pop,result))

**case "-"** => *resultStack*.push(*substract*(*resultStack*.pop,result))

**case "\*"** => *resultStack*.push(*multiply*(*resultStack*.pop,result))

**case "/"** => *resultStack*.push(*divide*(*resultStack*.pop,result))

**case "^"** => *resultStack*.push(*power*(*resultStack*.pop,result))

**case** \_ => *stop*(7)

}

}

}

}

}

**def** isNumber(r: Result) = {

(r.rt == Result.*rtInteger*) || (r.rt == Result.*rtDouble*)

}

**def** isDouble(r1: Result, r2: Result) = {

(r1.rt == Result.*rtDouble*) || (r2.rt == Result.*rtDouble*)

}

**def** minus(r: Result) = *Result*(r.pos,**if** (r.lex.startsWith(**"-"**)) r.lex.substring(1) **else "-"**+r.lex,r.rt)

**def** add(r1: Result,r2: Result) = {

**if** ((*isNumber*(r1))&&(*isNumber*(r2))) {

**if** (*isDouble*(r1,r2)) *Result*(r2.pos,**""**+(r1.lex.toDouble+r2.lex.toDouble),Result.*rtDouble*)

**else** *Result*(r2.pos,**""**+(r1.lex.toInt+r2.lex.toInt),Result.*rtInteger*)

}

**else** {

*stop*(8)

**null**

}

}

**def** substract(r1: Result,r2: Result) = {

**if** ((*isNumber*(r1))&&(*isNumber*(r2))) {

**if** (*isDouble*(r1,r2)) *Result*(r2.pos,**""**+(r1.lex.toDouble-r2.lex.toDouble),Result.*rtDouble*)

**else** *Result*(r2.pos,**""**+(r1.lex.toInt-r2.lex.toInt),Result.*rtInteger*)

}

**else** {

*stop*(8)

**null**

}

}

**def** multiply(r1: Result,r2: Result) = {

**if** ((*isNumber*(r1))&&(*isNumber*(r2))) {

**if** (*isDouble*(r1,r2)) *Result*(r2.pos,**""**+(r1.lex.toDouble\*r2.lex.toDouble),Result.*rtDouble*)

**else** *Result*(r2.pos,**""**+(r1.lex.toInt\*r2.lex.toInt),Result.*rtInteger*)

}

**else** {

*stop*(8)

**null**

}

}

**def** divide(r1: Result,r2: Result) = {

**if** ((*isNumber*(r1))&&(*isNumber*(r2))) {

*Result*(r2.pos,**""**+(r1.lex.toDouble/r2.lex.toDouble),Result.*rtDouble*)

}

**else** {

*stop*(8)

**null**

}

}

**def** power(r1: Result,r2: Result) = {

**if** ((*isNumber*(r1))&&(*isNumber*(r2))) {

*Result*(r2.pos,**""**+(math.*pow*(r1.lex.toDouble,r2.lex.toDouble)),Result.*rtDouble*)

}

**else** {

*stop*(8)

**null**

}

}

**def** calculate(expr: String) = {

**this**.*expr*=expr

*error* = -1

*actionStack*.clear

*resultStack*.clear

*ind* = -1

*open*

**while** ((*ind*<expr.length)&&(!*isError*)) {

**if** (*spaces*.indexOf(*char*) != -1) *next*

**else if** ((**""**+*char*).matches(**"\\d"**)) *nextNumber*

**else if** (*operators*.indexOf(*char*) != -1) *nextOperator*

**else if** (*char*==**'('**) *open*

**else if** (*char*==**')'**) *close*

**else** *stop*(2)

}

**if** (!*isError*){

*close*

**if** (*opened*!=0) *stop*(6)

**else if** ((!*actionStack*.isEmpty)||(*resultStack*.size!=1)) *stop*(7)

}

}

**def** result = **if** (*resultStack*.size==1) *resultStack*.top.lex **else ""**

**def** main(args: Array[String]): Unit = {

**if** (args.length>0) {

**var** expr = **""**

**if** ((args(0)==**"-f"**)&&(args.length>1)) {

**try** {

expr = Source.*fromFile*(args(1)).mkString

}

**catch** { **case** \_ : Throwable => *println*(**"Ошибка чтения файла "**+args(1))}

} **else** expr = args(0)

**if** (expr!=**""**) {

*println*(**"got expression: "**)

*println*(expr)

*calculate*(expr)

**if** (*isError*) *println*(*errorStr*)

**else**

{

*println*(**"result: "**)

*println*(*result*)

}

}

}

**else** *println*(**"Отсутствуют аргументы: expression или -f filename"**)

}

}