哈尔滨工业大学计算机科学与技术学院

实验报告

|  |  |
| --- | --- |
| 课程名称： | 数据结构与算法 |
| 课程类型： | 必修 |
| 实验名称： | 线性结构与应用 |
| 实验项目： | 中缀表达式的转化与运算 |
|  |  |
| 班级： | 1703005 |
| 学号： | 1170300520 |
| 姓名： | 郭子阳 |

1. 实验目的

熟悉线性结构的构造与应用

1. 实验要求及实验环境

实验要求：实现中缀表达式对后缀表达式的转换，并计算后缀表达式

实验环境：macOS，VSCode，JDK 11

1. 设计思想（本程序中用到的所有数据类型的定义，核心算法的流程图等）

栈结构定义如下：

class Stack<T> {

private ArrayList<T> list = new ArrayList<>();

public void push(T *element*) {

list.add(element);

}

public T pop() {

T element = list.get(list.size() - 1);

list.remove(list.size() - 1);

return element;

}

public ArrayList<T> popTwo() {

ArrayList<T> elements = new ArrayList<>();

elements.add(this.pop());

elements.add(this.pop());

return elements;

}

public T getTop() {

return list.get(list.size() - 1);

}

public int getSize() {

return list.size();

}

public boolean isEmpty() {

if(list.isEmpty()) {

return true;

} else {

return false;

}

}

@Override

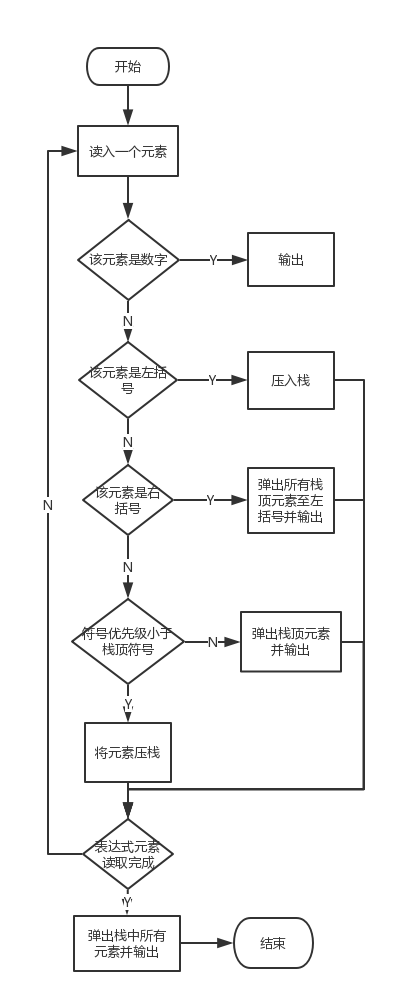
public String toString() {

return list.toString();

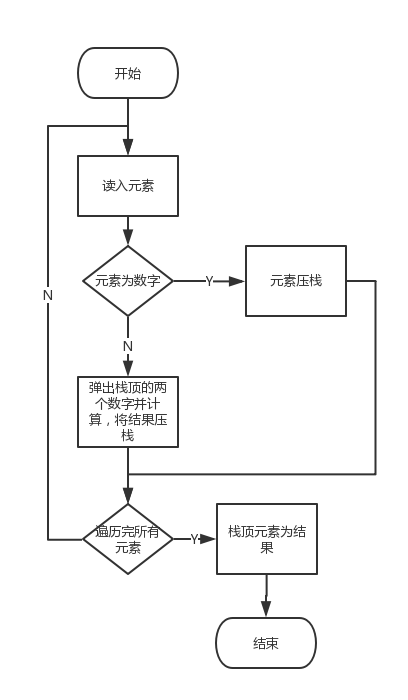
}

}

中缀转后缀算法：



后缀表达式计算：



1. 测试结果



1. 系统不足与经验体会

本次实验深刻理解了栈的设计。不足之处有无法对负数进行判断

1. 源代码

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.File;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Scanner;

public class DSExperiment {

private static ArrayList<String> infixExpressions = new ArrayList<>();

private static Scanner scanner = new Scanner(System.in);

private static Boolean logOn = false;

public static void main(String[] args) {

System.out.print("是否要开启日志模式(y or n)？");

if("y".equals(scanner.nextLine())) { //是否打开文件日志

logOn = true;

}

System.out.print("运算模式选择：\n1.读入文件\n2.实时输入\n请选择：");

if(scanner.nextInt() == 1) { //进行实时计算或者读入文件计算

readFromFile("samples.txt");

calculateExpressions();

} else {

liveCalculate();

}

if(logOn) {

System.out.println("计算完成，日志保存在log.txt中");

}

Log.log("全部计算完成", logOn);

Log.closeStream();

}

/\*\*

\* 实时输入模式

\*/

private static void liveCalculate() {

while(true) {

scanner = new Scanner(System.in);

System.out.print("请输入表达式(输入q退出)：");

String equation = scanner.nextLine();

if("q".equals(equation)) {

break;

}

equation = equation.replaceAll(" ", "");

infixExpressions.clear();

infixExpressions.add(equation);

calculateExpressions();

}

}

/\*\*

\* 从文件中读取表达式并存入ArrayList

\* @param path 文件路径

\*/

private static void readFromFile(String path) {

try{

BufferedReader bufferedReader = new BufferedReader(new FileReader(new File(path)));

String tempStr = "";

while((tempStr = bufferedReader.readLine()) != null) {

tempStr = tempStr.substring(1, tempStr.length() - 1);

infixExpressions.add(tempStr);

}

bufferedReader.close();

}catch(Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 计算ArrayList中的表达式

\*/

private static void calculateExpressions() {

for(String infixExpression : infixExpressions) {

System.out.println("The infix expression is " + infixExpression);

if(infixExpression.matches(".\*[a-zA-Z].\*")) { //如果存在字母则为表达式化简

String postfixExpression = convertToPostfix(addMultiply(infixExpression));

System.out.println("The postfix expression is " + postfixExpression);

System.out.println();

} else {

String postfixExpression = convertToPostfix(infixExpression);

System.out.println("The postfix expression is " + postfixExpression);

double ans = calculatePostfixExpression(postfixExpression);

System.out.printf("The result is %.2f\n\n", ans);

}

}

}

/\*\*

\* 为含有变量的表达式添加乘号

\* @param infixExpression 添加前的表达式

\* @return 添加完成后的表达式

\*/

private static String addMultiply(String infixExpression) {

int j = 0;

int length = infixExpression.length();

StringBuilder tempBuilder = null;

for(int i = 0; i < length - 1; i ++) {

if((infixExpression.charAt(j) >= 48 && infixExpression.charAt(j) <= 57) &&

((infixExpression.charAt(j + 1) >= 97 && infixExpression.charAt(j + 1) <= 122)

|| (infixExpression.charAt(j + 1) >= 65 && infixExpression.charAt(j + 1) <= 90))) {

tempBuilder = new StringBuilder(infixExpression);

tempBuilder.insert(j + 1, "\*");

infixExpression = tempBuilder.toString();

j ++;

}

j ++;

}

return infixExpression;

}

/\*\*

\* 将中缀表达式转化为后缀表达式

\* 1.初始化一个栈

\* 2.逐个读取元素（数字或者操作符）

\* 3.如果遇到数字，直接输出

\* 4.如果遇到操作符（不考虑括号），如果其优先级大于栈顶元素，就将栈顶弹出，并重复4步骤，否则将该操作符压入栈中（栈为空的时候也直接压栈即可）

\* 5.如果遇到左括号(“(“)，直接将其压入栈中，如果遇到右括号(“)”)，循环弹出顶栈元素，直到左括号为止（左括号也需要弹出，右括号不需要压栈），并且输出所有被弹栈顶元素（左括号除外）

\* @param infixExpression 中缀表达式

\* @return 空格分割的后缀表达式

\*/

private static String convertToPostfix(String infixExpression) {

Log.log(("开始转化中缀表达式：" + infixExpression + "\n"), logOn);

StringBuilder postfixExpression = new StringBuilder();

HashMap<String, Integer> priorityOrder = new HashMap<>();

Stack<String> convertStack = new Stack<>();

priorityOrder.put("^", 3);

priorityOrder.put("\*", 2);

priorityOrder.put("/", 2);

priorityOrder.put("+", 1);

priorityOrder.put("-", 1);

priorityOrder.put("(", 0);

ArrayList<String> elementList = splitIntoPieces(infixExpression);

for(int i = 0; i < elementList.size(); i ++) {

String element = elementList.get(i);

if("+".equals(element) || "-".equals(element) ||

"\*".equals(element) || "/".equals(element) ||

"(".equals(element) || ")".equals(element) ||

"^".equals(element)) {

if(")".equals(element)) {

String tempString = null;

while(!"(".equals(tempString = convertStack.pop())) {

postfixExpression.append(tempString + " ");

}

}else if("(".equals(element)){

convertStack.push(element);

} else{

if(convertStack.isEmpty() || priorityOrder.get(element) > priorityOrder.get(convertStack.getTop())) {

convertStack.push(element);

} else {

postfixExpression.append(convertStack.pop() + " ");

i --;

continue;

}

}

}else {

postfixExpression.append(element + " ");

}

Log.log(element, convertStack.toString(), logOn);

}

while(!convertStack.isEmpty()) {

postfixExpression.append(convertStack.pop() + " ");

}

Log.log(("成功转化为后缀表达式：" + postfixExpression.toString().trim() + "\n\n"), logOn);

return postfixExpression.toString().trim();

}

/\*\*

\* 讲中缀表达式按照元素分割，存入ArrayList

\* @param infixExpression 中缀表达式

\* @return 元素List

\*/

private static ArrayList<String> splitIntoPieces(String infixExpression) {

ArrayList<String> elementList = new ArrayList<>();

outer:for(int i = 0; i < infixExpression.length(); i ++) {

for(int j = i; j < infixExpression.length(); j ++) {

if(infixExpression.charAt(j) == '+' || infixExpression.charAt(j) == '-' ||

infixExpression.charAt(j) == '\*' || infixExpression.charAt(j) == '/' ||

infixExpression.charAt(j) == '(' || infixExpression.charAt(j) == ')' ||

infixExpression.charAt(j) == '^') {

if(infixExpression.charAt(i) == '+' || infixExpression.charAt(i) == '-' ||

infixExpression.charAt(i) == '\*' || infixExpression.charAt(i) == '/' ||

infixExpression.charAt(i) == '(' || infixExpression.charAt(i) == ')' ||

infixExpression.charAt(i) == '^') {

elementList.add(infixExpression.substring(i, j + 1));

i = j;

break;

} else {

elementList.add(infixExpression.substring(i, j));

i = j - 1;

break;

}

}else if(j == infixExpression.length() - 1) {

elementList.add(infixExpression.substring(i, j + 1));

break outer;

}

}

}

return elementList;

}

/\*\*

\* 计算后缀表达式的值

\* 1.初始化一个栈

\* 2.遍历后缀表达式的所有元素

\* 3.如果遇到数字，就将其直接压入栈

\* 4.如果遇到运算符，就将顶栈的两个元素弹出，将执行该运算之后结果在压入栈内

\* 5.遍历结束后，栈内的唯一一个元素就是计算结果。

\* @param postfixExpression 后缀表达式

\* @return 后缀表达式的结果

\*/

private static double calculatePostfixExpression(String postfixExpression) {

Log.log(("开始计算后缀表达式：" + postfixExpression + "\n"), logOn);

String[] elementArray = postfixExpression.split(" ");

Stack<Double> calculateStack = new Stack<>();

Double tempAns = 0.0;

for(int i = 0; i < elementArray.length; i ++) {

String element = elementArray[i];

if("+".equals(element) || "-".equals(element) ||

"\*".equals(element) || "/".equals(element) ||

"^".equals(element) ) {

ArrayList<Double> tempList = calculateStack.popTwo();

if("+".equals(element)) {

tempAns = tempList.get(1) + tempList.get(0);

calculateStack.push(tempAns);

}else if("-".equals(element)) {

tempAns = tempList.get(1) - tempList.get(0);

calculateStack.push(tempAns);

}else if("\*".equals(element)) {

tempAns = tempList.get(1) \* tempList.get(0);

calculateStack.push(tempAns);

}else if("/".equals(element)) {

tempAns = tempList.get(1) / tempList.get(0);

calculateStack.push(tempAns);

}else if("^".equals(element)) {

tempAns = Math.pow(tempList.get(1), tempList.get(0));

calculateStack.push(tempAns);

}

} else {

calculateStack.push(Double.valueOf(element));

}

Log.log(element, calculateStack.toString(), logOn);

}

Log.log("计算完成，结果是：" + calculateStack.getTop() + "\n\n", logOn);

return calculateStack.getTop();

}

}

/\*\*

\* 泛型栈类

\* @param <T>

\*/

class Stack<T> {

private ArrayList<T> list = new ArrayList<>();

public void push(T element) {

list.add(element);

}

public T pop() {

T element = list.get(list.size() - 1);

list.remove(list.size() - 1);

return element;

}

public ArrayList<T> popTwo() {

ArrayList<T> elements = new ArrayList<>();

elements.add(this.pop());

elements.add(this.pop());

return elements;

}

public T getTop() {

return list.get(list.size() - 1);

}

public int getSize() {

return list.size();

}

public boolean isEmpty() {

if(list.isEmpty()) {

return true;

} else {

return false;

}

}

@Override

public String toString() {

return list.toString();

}

}