编号：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实习 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 九 | 十 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |  |  |

**武汉大学计算机学院**

**本科生实验报告**

**《解释器构造》实验**

**计算器分析器构造实验**

编 号：

实习题目： 计算器分析器构造实验

专业（班）： 软件工程卓越二班

团队成员一： 颜聪（2016302580288）

团队成员二： 彭东宇（2016302580273）

团队成员三： 苏胜（2016302580279）

团队成员三： 吴勇清（2016302580280）

指导教师： 杜　卓　敏

2018 年 12 月 28 日

**郑 重 声 明**

本团队呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

团队成员签名： 日期：

**词法分析目录**

**第一部分 词法分析 ……………………… 1**

**第二部分 语法分析 ………………………**

**第三部分 错误处理 ………………………**

**第四部分 数据结构 ………………………**

**第五部分 测试部分 ………………………**

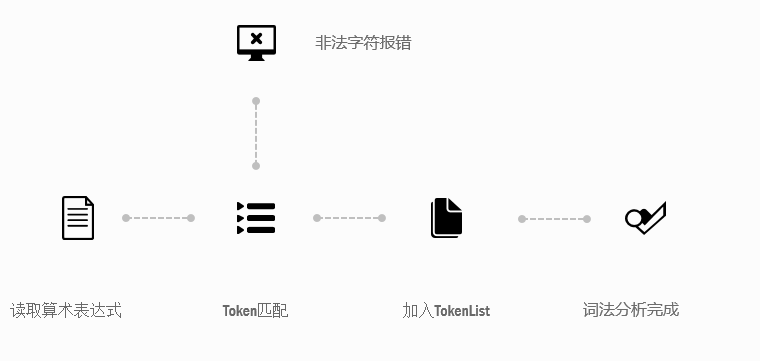
**第一部分 词法分析**

词法分析进行分析的token包含以下类型，若是其他情况，认为其是异常：

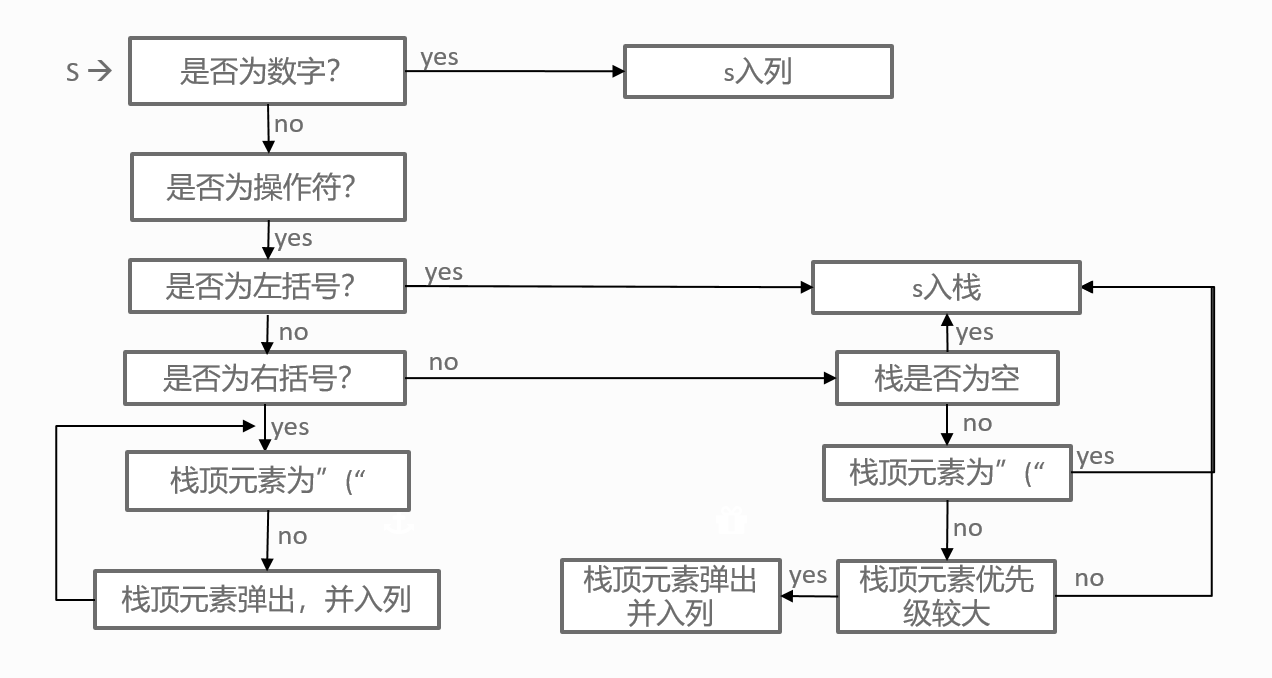
“PLUS”、“MINUS”、“MUL”、“DIV”、“LPARENT”、“RPARENT”、"NUMBER"

Token的结构如下，string s代表token变量的值，int i表示token的类型的种别码，string name表示token类型的名称，比如s:""7777"" i:1 name :"NUMBER"

词法分析的过程如下图所示：



建立二叉树过程：



首先从用户输入当中获取算术表达式，放入char类型的数组当中：

char[] arr = input.toCharArray();

之后开始分析，如果碰到错误，会直接结束分析并报错，比如：

if(i==0&&(arr[i]!='('&&arr[i]!='-'

&&(!isNumeric(String.valueOf(arr[i]))))){

System.out.println("开头应为数字或左括号"+",position: "+i+"\n");

Controller.er+=("开头应为数字或左括号"+",position: "+i);

break;

//System.exit(-1);

}

其中的一些具体实现如下：

用于判断字符串是否为数字：

public static boolean isNumeric(String str){

Pattern pattern;

pattern = Pattern.compile("([0-9]\\d\*\\.?\\d\*)|(0\\.\\d\*[0-9])");

Matcher isNum = pattern.matcher(str);

if( !isNum.matches() ){

return false;

}

return true;

}

用于判断操作符的种类：

public static boolean isOperator(String str){

switch (str)

{

case "+":

case "-":

case "\*":

case "/":

return true;

default:

return false;

}

}

用于判断运算符的优先级：

public static int priority(String str){

switch (str)

{

case "#":

return -1;

case "(":

return 0;

case "+":

case "-":

return 1;

case "\*":

case "/":

return 2;

default:

return -1;

}

}

在分析完成之后，新建一个ArrayList并将所有token放入其中以便于进行之后的语法分析：

ArrayList<token> arrayList1 = new ArrayList<token>();

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

{

String tmp = arrayList.get(i);

token t = new token(tmp);

arrayList1.add(t);

}

return arrayList1;

**第二部分 语法分析**

语法分析的主要思路是：使用分块思路把内容按优先级分块，先使用'(' 和 ')'分块，然后使用'\*' 和 '/'号分块，分块结束；计算时从左往右计算。

采用LL(1)文法，语法规则如下：

expr —> term + expr | term - expr | term // +-从左到右依此执行

term —> factor \* term | factor/term | factor// \*/次优先，内部先左后右

factor—> digit ｜(expr) // ()最优先

建造二叉树的具体代码如下：

public class BTree {

static class TreeNode {

TreeNode leftChild;

TreeNode rightChild;

String value="#";

}

/\*\*

\* 将算术表达式转化成二叉树

\*

\* @param expression

\* 为了方便，使用字符串数组来存储表达式

\* @return 二叉树的根节点

\*/

public static TreeNode createBinaryTree(String[] expression) {

// 存储操作数的栈

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

// 存储转换后的逆波兰式的队列

Queue<String> reversePolish = new LinkedList<String>();

for (String s : expression) {

// 如果是数字

if(isDigit(s)){

reversePolish.offer(s);

// 如果是操作符

} else if(isOperator(s)){

//是左括号直接入栈

if("(".equals(s)){

opStack.push(s);

// 如果是右括号

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

// 把离上一个“（”之间的操作符全部弹出来放入逆波兰式的队列中

while(!opStack.isEmpty()){

String op = opStack.peek();

if(op.equals("(")){

opStack.pop();

break;

} else{

reversePolish.offer(opStack.pop());

}

}

} else{

while(!opStack.isEmpty()){

// 如果栈顶元素为"("直接入栈

if("(".equals(opStack.peek())){

opStack.push(s);

break;

//如果栈顶元素优先级大于s

}else if(isGreat(opStack.peek(), s)){

reversePolish.offer(opStack.pop());

}else if(isGreat(s, opStack.peek())){

opStack.push(s);

break;

}

}

// 如果栈为空，直接入栈

if(opStack.isEmpty())

opStack.push(s);

}

}

}

// 将剩余的操作符入队

while(!opStack.isEmpty()){

reversePolish.offer(opStack.pop());

}

Stack<TreeNode> nodeStack = new Stack<TreeNode>();

// 将逆波兰式转化成二叉树

while(!reversePolish.isEmpty()){

String s = reversePolish.poll();

// 以当前的元素的值新建一个节点

TreeNode node = new TreeNode();

node.value = s;

// 如果是数字

if(isDigit(s)){

nodeStack.push(node);

// 如果是操作符

} else if(isOperator(s)){

//从栈里弹出两个节点作为当前节点的左右子节点

TreeNode rightNode = nodeStack.pop();

TreeNode leftNode = nodeStack.pop();

node.leftChild = leftNode;

node.rightChild = rightNode;

// 入栈

nodeStack.push(node);

}

}

return nodeStack.pop();

}

/\*\*

\* 判断是否为运算符（暂时只判断四则运算的运算符）

\*

\* @param s

\* @return

\*/

static boolean isOperator(String s) {

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

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

return true;

else

return false;

}

/\*\*

\* 判断是否为数字

\*

\* @param str

\* @return

\*/

static boolean isDigit(String str) {

Pattern pattern = Pattern.compile("([0-9]\\d\*\\.?\\d\*)|(0\\.\\d\*[0-9])");

Matcher isNum = pattern.matcher(str);

if( !isNum.matches() ){

return false;

}

return true;

}

/\*\*

\* 判断op1和op2的优先级，如果op1>op2，返回true，如果op1<=op2，返回false

\*

\* @param op1

\* @param op2

\* @return

\*/

static boolean isGreat(String op1, String op2) {

if (getPriority(op1) >=getPriority(op2))

return true;

else

return false;

}

/\*\*

\* 获取运算符的优先级

\*

\* @param op

\* @return

\*/

static int getPriority(String op) {

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

return 1;

else if ("\*".equals(op) || "/".equals(op))

return 2;

else

throw new IllegalArgumentException("Unsupported operator!");

}

/\*\*

\* 打印出还原的算术表达式

\*/

private static int depth(TreeNode pTreeRoot){

if(pTreeRoot==null){

return 0;

}

int l = depth(pTreeRoot.leftChild);

int r = depth(pTreeRoot.rightChild);

if(l > r){

return l + 1;

}else{

return r + 1;

}

}

private static void levelOrder(TreeNode pTreeNode, int level) {

if(pTreeNode == null || level < 1){

System.out.print("# ");

Controller.tree+="# ";

return ;

}

if(level == 1){

System.out.print(pTreeNode.value+ " ");

Controller.tree+=(pTreeNode.value+ " ");

return ;

}

//左子树

levelOrder(pTreeNode.leftChild, level-1);

//右子树

levelOrder(pTreeNode.rightChild, level-1);

}

public static void printFromTopToBottom(TreeNode pTreeRoot,String tree){

if(pTreeRoot==null){

return ;

}

int depth = depth(pTreeRoot);

for (int i = 1; i <= depth; ++i) {

levelOrder(pTreeRoot, i);

Controller.tree+="\n";

System.out.println("");

}

}

}

**第三部分 错误处理**

词法部分错误处理，主要是查看非法字符，一旦发现有非法字符或者表达式不符合规则就会报错。

具体相关代码如下：

if(i==0&&(arr[i]!='('&&arr[i]!='-'&&(!isNumeric(String.valueOf(arr[i]))))){

System.out.println("开头应为数字或左括号"+",position: "+i+"\n");

Controller.er+=("开头应为数字或左括号"+",position: "+i);

break;

//System.exit(-1);

}

if(!isNumeric(String.valueOf(arr[i-1])))

{

Controller.er+=("表达式格式错误"+"position:"+i+"\n");

System.out.println("表达式格式错误"+"position:"+i);

break;

//System.exit(-1);

}

if(i==arr.length-1||!isNumeric(String.valueOf(arr[i+1])))

{

Controller.er+=("表达式格式错误"+"position:"+i+"\n");

System.out.println("表达式格式错误"+"position:"+i);

break;

//System.exit(-1);

}

if(i==arr.length-1)

{

Controller.er+=("缺少操作数 "+",position: "+(i)+"\n");

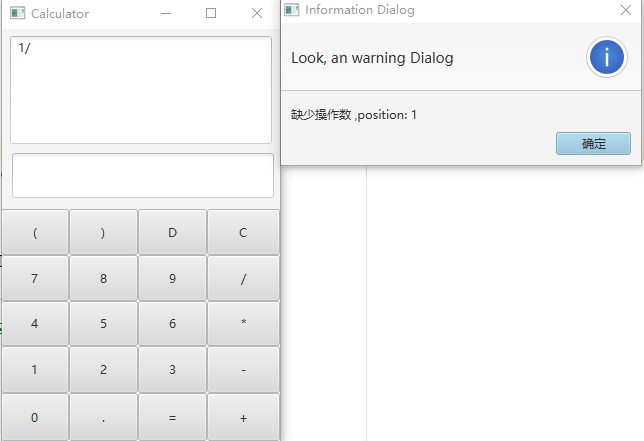
System.out.println("缺少操作数 "+",position: "+(i));

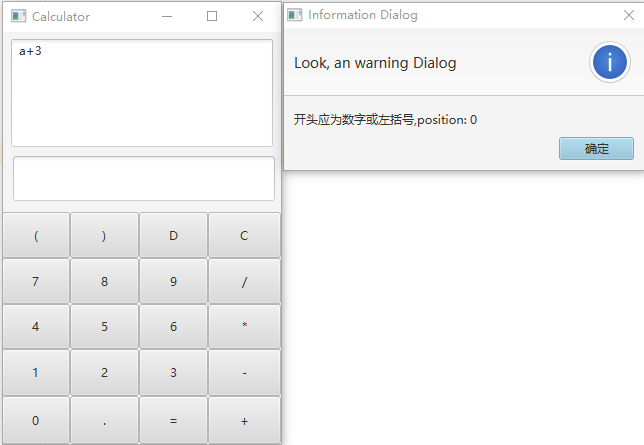
break;

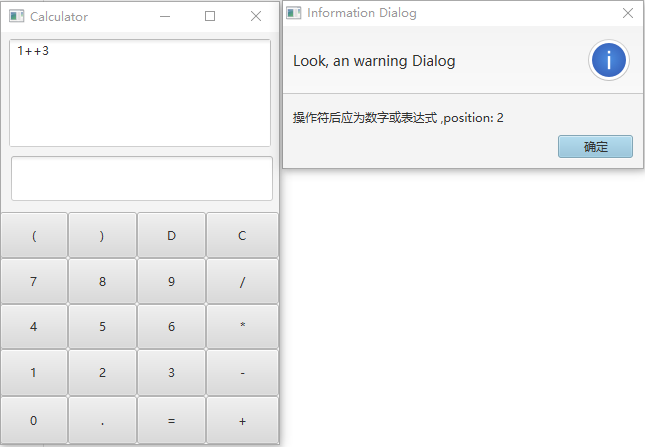
//System.exit(-1);

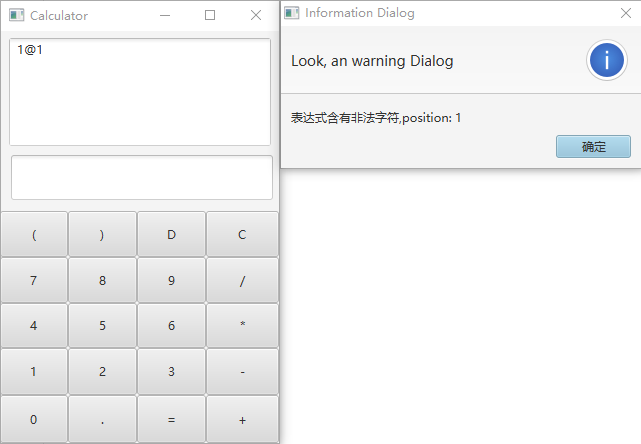
}

以上只是一部分的错误处理代码，部分词法错误处理展示如下：









语法分析报错：

if(tempResult.compareTo(BigDecimal.ZERO)==0)

{

System.out.println("除数不能为0!!");

Controller.er+=("除数不能为0!!");

return null;

}

if(!head.name.equals("RPARENT"))

factor = expr();

else{

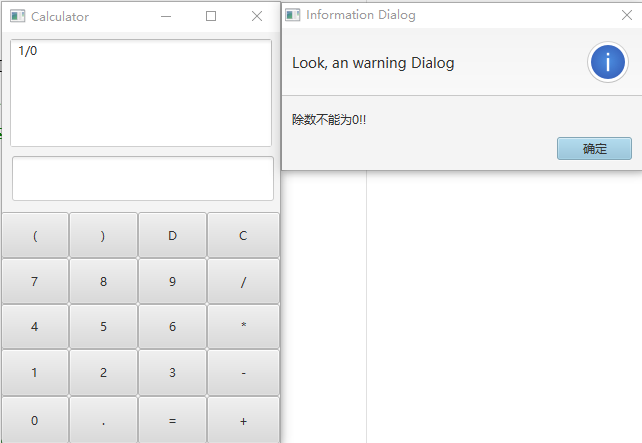
System.out.println("括号内无数字");

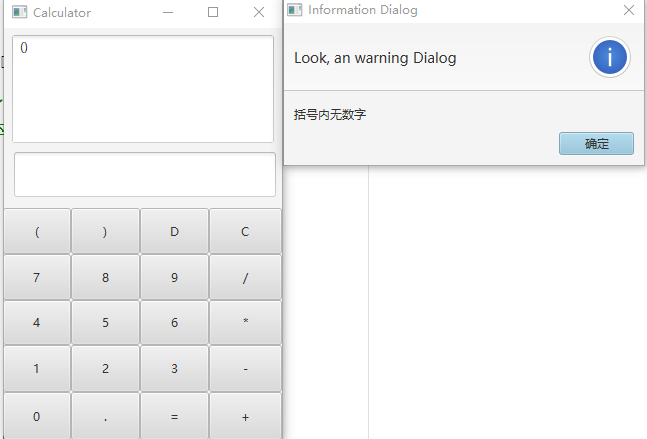
Controller.er+=("括号内无数字");

return null;

//System.exit(-1);

}





**第四部分 数据结构**

TreeNode节点的代码如下：3

static class TreeNode {

TreeNode leftChild;

TreeNode rightChild;

String value="#";

}

Token的代码如下：

public class token {

public token(String value){

this.s=value;

foo(value);

}

String s;//token变量的值

int i;//token的类型的种别码

String name;//token类型的名称 如 s ""7777"" i 1 name "NUMBER"

public void foo(String value) {

switch (value) {

case "+":

this.i = 2;

this.name = "PLUS";

break;

case "-":

this.i = 3;

this.name = "MINUS";

break;

case "\*":

this.i = 4;

this.name = "MUL";

break;

case "/":

this.i = 5;

this.name = "DIV";

break;

case "(":

this.i = 6;

this.name = "LPARENT";

break;

case ")":

this.i = 7;

this.name = "RPARENT";

break;

case ";":

this.i = 8;

this.name = "SEMICOLON";

break;

default:

this.i = 1;

this.name = "NUMBER";

}

}

}

**第五部分 测试部分**

为了使树美观，用#代替空子树，测试的结果如下：

