实验8

练习1

题目

求后缀表达式的值。

说明：
（1）后缀表达式中的操作数只有 0, 1, 2, ..., 9 共10 种可能。中间结果和最后结果可以是其他数。
（2）后缀表达式中的运算符只有 +, -, *, / 共4 种可能。其中 +, - 只表示双目加、减，不表示单目正、负。
（3）后缀表达式是正确的。除数不为0。

解析

类似实验6练习1，这里使用Java泛型实现链栈Stack，支持迭代并重写了toString方法。（按理来说Stack不应该支持遍历，但为了方便debug，还是加上了）（**这里定义的Stack将在之后的实验代码中多次使用**）

使用标准逆波兰式算法即可求得后缀表达式的值。（建栈，依次压入数字，遇到算术符号则弹出栈顶的两个数字，计算后再次压入栈顶，直至字符串遍历完毕）

代码

Stack.java

```
import java.util.Iterator;

// 使用Java泛型实现的链栈，支持迭代和toString
class Stack<Item> implements Iterable<Item> {
    private Node top;
    private int N;

    private class Node {
        Item data;
        Node next;

        public Node(Item data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    public boolean isEmpty() {
        return this.top == null;
    }

    public int length() {
        return this.N;
    }
```

```
    public Item peek() {
        if (this.top == null) {
            return null;
        } else {
            return this.top.data;
        }
    }

    public void push(Item data) {
        this.top = new Node(data, this.top);
        this.N++;
    }

    public Item pop() {
        if (this.top == null) {
            return null;
        } else {
            Item data = this.top.data;
            this.top = this.top.next;
            this.N--;
            return data;
        }
    }

    public String toString() {
        Node p = this.top;
        StringBuilder builder = new StringBuilder();
        builder.append("[");
        while (p != null) {
            builder.append(p.data + ", ");
            p = p.next;
        }
        builder.delete(builder.length() - 2, builder.length());
        builder.append("]");
        return builder.toString();
    }

    public Iterator<Item> iterator() {
        return new StackIterator();
    }

    private class StackIterator implements Iterator<Item> {
        private Node current = top;

        public boolean hasNext() {
            return current != null;
        }

        public void remove() {}

        public Item next() {
            Item data = current.data;
            current = current.next;
            return data;
        }
    }
}
```

Exercise_1.java

```
import java.util.Scanner;

public class Exercise_1 {
    public static double calculate_suffix_expression(String expression) {
        Stack<String> stack = new Stack<String>();
        for (int i = 0; i < expression.length(); i++) {
            char c = expression.charAt(i);
            if (c == '+' || c == '-' || c == '*' || c == '/') {
                double d2 = Double.parseDouble(stack.pop());
                double d1 = Double.parseDouble(stack.pop());
                double d3 = 0;
                if (c == '+') {
                    d3 = d1 + d2;
                } else if (c == '-') {
                    d3 = d1 - d2;
                } else if (c == '*') {
                    d3 = d1 * d2;
                } else if (c == '/') {
                    d3 = d1 / d2;
                }
                stack.push(Double.toString(d3));
            } else {
                stack.push(Character.toString(c));
            }
        }
        return Double.parseDouble(stack.pop());
    }

    public static void main(String[] args) throws Exception {
        Scanner in = new Scanner(System.in);
        String expression = in.nextLine();
        in.close();
        double result = calculate_suffix_expression(expression);
        System.out.println(result);
    }
}
```

输入

```
123+45**+
```

输出

```
101.0
```

---

*练习2

题目

求中缀表达式的值。

说明：
（1）中缀表达式中的操作数只有 0, 1, 2, ..., 9 共10 种可能。中间结果和最后结果可以是其他数。
（2）中缀表达式中的运算符只有 +, -, *, / 共4 种可能。其中 +, - 只表示双目加、减，不表示单目正、负。
（3）中缀表达式中的圆括号可以并列和嵌套。
（4）中缀表达式是正确的。除数不为0。
（5）要求不先把中缀表达式完全转化为后缀表达式，而是一边转化一边求值。

解析

原理与Dijkstra双栈算术表达式求值算法类似（建立两个栈，操作数栈和运算符栈，忽略左括号，将操作数和运算符依次压入对应栈，遇到右括号则弹出运算数栈顶的两个数字和运算符栈顶的运算符进行计算，并将结果重新压入操作数栈顶，直至遍历结束），但由于该算法仅支持用括号标注优先级的表达式，因此需要先为表达式加上括号。

代码

```
import java.util.Scanner;

public class Exercise_2 {
    public static double calculate_infix_expression(String expression) {
        // 用括号标注优先级
        for (int i = 0; i < expression.length(); i++) {
            char c = expression.charAt(i);
            // 如果是*或/，且未加括号，加上括号以提高优先级
            if ((c == '*' || c == '/') && (i >= 2 && i <= expression.length() -
2)
                    && !(expression.charAt(i - 2) == '(' && expression.charAt(i
+ 2) == ')')) {
                String operator = expression.substring(i, i + 1); // 存储运算符
                String s1, s2, s3, s4; // 分别存储左括号前的内容、左括号至运算符的内容，
运算符至右括号的内容以及剩余部分
                if (expression.charAt(i - 1) == ')') {
                    // 若运算符前是右括号，找到与之配对的左括号
                    int count = 1;
                    int j;
                    for (j = i - 2; j >= 0; j--) {
                        // 查找配对的左括号的位置，j的末值即为其下标
                        if (expression.charAt(j) == ')') {
                            count++;
                        } else if (expression.charAt(j) == '(') {
                            count--;
                        }
                        if (count == 0) {
                            break;
                        }
                    }
                    s1 = expression.substring(0, j);
                    s2 = expression.substring(j, i);
                } else {
                    s1 = expression.substring(0, i - 1);
                    s2 = expression.substring(i - 1, i);
```

```
                }
                if (expression.charAt(i + 1) == '(') {
                    // 若运算符后是左括号，找到与之配对的右括号
                    int count = 1;
                    int j;
                    for (j = i + 2; j < expression.length(); j++) {
                        // 查找配对的右括号的位置，j的末值即为其下标
                        if (expression.charAt(j) == '(') {
                            count++;
                        } else if (expression.charAt(j) == ')') {
                            count--;
                        }
                        if (count == 0) {
                            break;
                        }
                    }
                    s3 = expression.substring(i + 1, j + 1);
                    s4 = expression.substring(j + 1, expression.length());
                } else {
                    s3 = expression.substring(i + 1, i + 2);
                    s4 = expression.substring(i + 2, expression.length());
                }
                expression = s1 + "(" + s2 + operator + s3 + ")" + s4; // 加上括号
                i++; // 由于加上了括号，运算符位置右移，因此i+1，以防止重复加括号
            }
        }
        expression = "(" + expression + ")";
        // System.out.println(expression); // 打印加完括号的表达式

        // 计算加完括号的中缀表达式
        Stack<Character> operatorStack = new Stack<Character>();
        Stack<Double> operandStack = new Stack<Double>();
        for (int i = 0; i < expression.length(); i++) {
            char c = expression.charAt(i);
            if (c == '+' || c == '-' || c == '*' || c == '/' || c == '(') {
                operatorStack.push(c);
            } else if (c != '(' && c != ')') {
                operandStack.push(Double.parseDouble(Character.toString(c)));
            }
            if (c == ')' || i == expression.length() - 1) {
                while (operatorStack.peek() != null) {
                    double d2 = operandStack.pop();
                    double d1 = operandStack.pop();
                    double d3 = 0;
                    char operator = operatorStack.pop();
                    if (operator == '+') {
                        d3 = d1 + d2;
                    } else if (operator == '-') {
                        d3 = d1 - d2;
                    } else if (operator == '*') {
                        d3 = d1 * d2;
                    } else if (operator == '/') {
                        d3 = d1 / d2;
                    }
                    operandStack.push(d3);
                    if (operatorStack.peek() != null && operatorStack.peek() ==
'(') {
                        operatorStack.pop();
```

```
                        break;
                    }
                }
            }
        }
        return operandStack.pop();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String expression = in.nextLine();
        in.close();
        double result = calculate_infix_expression(expression);
        System.out.println(result);
    }
}
```

输入

```
2*3+(7/(6+1)+(4+8)*9)*6+5
```

输出

```
1324.0
```

附

实际上，还根据书上的示例实现了普通的顺序栈SqStack和链栈LinkStack，但由于设计比较烂，使用时需要经常考虑数据类型的转换，因此所有实验的代码均未使用这两个数据类型。

Node.java

```
public class Node {
    public Object data;
    public Node next;

    public Node() {
        this(null, null);
    }

    public Node(Object data) {
        this(data, null);
    }

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```

SqStack.java

```
public class SqStack {
    private Object[] stackElem;
    private int top;

    public SqStack(int maxSize) {
        this.top = 0;
        this.stackElem = new Object[maxSize];
    }

    public void clear() {
        this.top = 0;
    }

    public boolean isEmpty() {
        return this.top == 0;
    }

    public int length() {
        return this.top;
    }

    public Object peek() {
        if (!this.isEmpty()) {
            return this.stackElem[this.top-1];
        } else {
            return null;
        }
    }

    public void push(Object x) throws Exception {
        if (this.top == this.stackElem.length) {
            throw new Exception("stack is full");
        } else {
            this.stackElem[this.top] = x;
            this.top++;
        }
    }

    public Object pop() {
        if (this.isEmpty()) {
            return null;
        } else {
            this.top--;
            return this.stackElem[this.top];
        }
    }
}
```

LinkStack.java

```
public class LinkStack {
    private Node top;

    public void clear() {
```

```
        top = null;
    }

    public boolean isEmpty() {
        return top == null;
    }

    public Object peek() {
        if (!this.isEmpty()) {
            return top.data;
        } else {
            return null;
        }
    }

    public void push(Object x) {
        Node p = new Node(x);
        p.next = top;
        top = p;
    }

    public Object pop() {
        if (isEmpty()) {
            return null;
        } else {
            Node p = top;
            top = top.next;
            return p.data;
        }
    }
}
```

心得体会

1. 利用栈可以简洁高效地实现许多算法，例如表达式计算。
2. 栈的pop()和push()操作的时间复杂度都是O(1)，效率非常高。