《Java 应用技术》课程实验报告——表达式计算

应承峻 3170103456

1. 实验描述

编程对于可能含有以下运算符的表达式计算结果：

1. +、-、*、/、%：四则运算
2. >>、<<：移位运算
3. High、Low：常数的高、低 16 位
4. (、)：括号内优先，可嵌套

【注】允许 0x 为前缀的 16 进制数。

2. 实验思路

本程序的程序结构如图所示，当调用 eval 传入表达式后，程序会先通过 getRegular-Expression 方法将表达式进行过滤和初始化处理，然后通过 generatePostfixExpression 方法将其转换成后缀表达式，然后通过 calculatePostfixExpression 方法计算后缀表达式。

在最开始的时候，我们需要为建立一个 Operator 的静态类，它将所有合法的运算符存储在一个 HashMap 中，value 值越大代表优先级越大，其定义如下：

```
private static final Map<String, Integer> OPT_PRIORITY_MAP = new HashMap<String, Integer>() {
    {
        put(")", 0);
        put("<", 1);
        put(">", 1);
        put("<<", 2);
        put(">>", 2);
        put("+", 3);
        put("-", 3);
        put("*", 4);
        put("/", 4);
        put("%", 4);
        put("(", 9);
    }
};
```

我们还需要为该类提供一些静态方法，如 compareTo 来比较两个运算符的优先级，isValidOp 来判断是否是合法的运算符，isValidNumberCharacter 来判断是否是数字：

```
public static int compareTo(String op1, String op2) throws IllegalArgumentException {…

public static boolean isValidOp(String op) {…

public static boolean isValidNumberCharacter(char num) {…
```

接下来，我们开始对表达式进行处理。首先需要对初始表达式进行处理，处理的内容包括以下方面：

- 使用 toLowerCase 和 replaceAll 方法将字符串中所有字符(A-F, X 等)转换成小写字母，并去掉所有的空格
- 通过正则表达式"0x[0-9a-f]+"来检测所有十六进制表示的数，然后调用 parseInt 方法将其转换成 10 进制数后替换到源字符串中
- 通过正则表达式找到字符串中含有的 high()和 low()函数，并将其中的表达式取出

后，递归调用 eval 函数计算得出结果(记为 v)，最后调用 high(v)或 low(v)得到函数结果 s，并将 s 替换源字符串中的函数。

```
public static String getRegularExpression(String str) {
    str = str.toLowerCase().replaceAll("[ ]+", ""); //replace blank
    Matcher matcher = HEX_PATTERN.matcher(str);  //replace HEX number to OCT number
    while (matcher.find()) {
        String HEX = matcher.group();
        Integer OCT = Integer.parseInt(HEX.replaceAll("0x", ""), 16);
        str = str.replaceAll(HEX, OCT.toString());
    }
    matcher = FUNC_PATTERN.matcher(str);   //find operation low() or high() and replace it by its subvalue
    while (matcher.find()) {
        String item = matcher.group();
        String innerExpr;
        if (item.startsWith("high(") && item.endsWith(")")) {
            innerExpr = item.substring(5, item.length() - 1);
            str = str.replace(item, String.valueOf(high(eval(innerExpr))));
        } else if (item.startsWith("low(") && item.endsWith(")")) {
            innerExpr = item.substring(4, item.length() - 1);
            str = str.replace(item, String.valueOf(low(eval(innerExpr))));
        } else {
            throw new IllegalArgumentException("Illegal used function high() or low() !");
        }
    }
    return str;
}
```

在对表达式进行预处理后，我们需要将表达式转换成后缀表达式，转换成后缀表达式的算法是需要通过一个堆栈 Stack 来实现，其算法如下：

- 初始化堆栈 Stack 用于存放运算符，初始化 Vector 用于存放输出。

```
public static Vector<String> generatePostfixExpression(String expr) throws IllegalArgumentException {
    Vector<String> output = new Vector<>();
    Stack<String> stack = new Stack<>();
    expr = getRegularExpression(expr);
```

- 循环遍历字符串，如果是操作数则直接放入到 Vector 中，如果是运算符，则按照运算符优先级做处理，需要注意在异常时抛出异常。
  - 如果该运算符是左括号，则直接 push 入栈。如果是右括号，则将栈中的内容全部 pop 出来直到遇到左括号。

```
for (int i = 0; i < expr.length(); i++) {
    String ch = String.valueOf(expr.charAt(i));
    switch (ch) {
        case "(":
            stack.push(ch);
            break;
        case ")":
            while (!stack.empty() && !stack.peek().equals("(")) {
                output.add(stack.peek());
                stack.pop();
            }
            if (stack.empty())
                throw new IllegalArgumentException("Error: Can not find open tag correspond to ')'!");
            stack.pop();   // pop '('
            break;
```

  - 如果是加减乘除去模运算符，则比较当前栈顶的运算符和该运算符的优先级大小，如果该运算符优先级高于栈顶运算符，则将该运算符 push 入栈。否则，连续地将栈顶运算符 pop 出来，直到某一栈顶运算符的优先级低于该运算符的优先级，最后将该运算符入栈。

```
case "*":
case "/":
case "%":
case "+":
case "-":
    while (!stack.empty() && !stack.peek().equals("(") && Operator.compareTo(stack.peek(), ch) >= 0) {
        output.add(stack.peek());
        stack.pop();
    }
    stack.push(ch);
    break;
```

- 如果是遇到<或>符号需要判断后面紧跟着的字符是否是一样的，即能够构成<<或>>运算符。如果不能，则抛出异常。如果可以，就连续地 pop 出栈顶运算符，直到遇到左括号为止（因为只有右括号能够 pop 左括号）。

```
case "<":
    if (i < expr.length() - 1 && expr.charAt(i + 1) == '<') {
        while (!stack.empty() && !stack.peek().equals("(")) {
            output.add(stack.peek());
            stack.pop();
        }
        stack.push("<<");
        i++;
    } else {
        throw new IllegalArgumentException("Operator << lack of <");
    }
    break;
case ">":
    if (i < expr.length() - 1 && expr.charAt(i + 1) == '>') {
        while (!stack.empty() && !stack.peek().equals("(")) {
            output.add(stack.peek());
            stack.pop();
        }
        stack.push(">>");
        i++;
    } else {
        throw new IllegalArgumentException("Operator >> lack of >");
    }
    break;
```

- 如果不是运算符，判断是否是合法的数字字符，如果是，则不断地转换成字符串，这里使用 StringBuilder 而不是 String 来提高效率。如果不是，则抛出异常。

```
default:
    if (!Operator.isValidNumberCharacter(expr.charAt(i))) {
        throw new IllegalArgumentException("Illegal character " + ch);
    }
    StringBuilder number = new StringBuilder(ch);
    while (i < expr.length() - 1 && Operator.isValidNumberCharacter(expr.charAt(i + 1))) {
        number.append(expr.charAt(++i));
    }
    if (i < expr.length() - 1 && !Operator.isValidOp(String.valueOf(expr.charAt(i + 1)))) {
        throw new IllegalArgumentException("Illegal character " + ch);
    }
    output.add(number.toString());
```

- 遍历完字符串后，将运算符 Stack 中的所有元素 pop 出。

```
while (!stack.empty()) {
    output.add(stack.peek());
    stack.pop();
}
return output;
```

最后对后缀表达式进行计算：在这里以加减运算符为例。遍历生成的后缀 Vector，并初始化一个 Stack，当遇到数字时就把数字 push 如 Stack，当遇到运算符时，就 pop 出栈顶的两个运算符并进行运算，将结果 push 进 Stack。最终，堆栈的栈顶就是运算结果：

```
public static int calculatePostfixExpression(Vector<String> postfix) throws IllegalArgumentException {
    int output;
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < postfix.size(); i++) {
        String item = postfix.get(i);
        int x, y;
        try {
            switch (item) {
                case "+":
                    x = stack.pop();
                    y = stack.pop();
                    stack.push(y + x);
                    break;
                case "-":
                    x = stack.pop();
                    y = stack.pop();
                    stack.push(y - x);
                    break;
```

3. 实验结果

在本次实验中，为了综合测试程序对于各种运算符的计算情况和优先级处理，我们设计了一个表达式，使其能够包含所有的运算符，同时又要兼顾优先级的测试和嵌套函数的验证，因此该表达式如下：

$$(3 + 0x20) \% 0x11 << 3 - low(high(0x00ff0000) * low(3)) >> 1$$

先使用 javac Expr 来编译程序，生成 class 文件，然后执行 java Expr 后，输入表达式回车，得到运行结果：

```
PS C:\Users\Nonehyo\Desktop> javac Expr.java
PS C:\Users\Nonehyo\Desktop> java Expr
(3 + 0x20) % 0x11 << 3 - low(high(0x00ff0000) * low(3)) >> 1
32
PS C:\Users\Nonehyo\Desktop>
```

同时，为了验证结果的准确性，我们新建了一个 JavaScript 脚本，并调用 eval 函数来得到正确的结果，在此处对于 high 和 low 函数我们使用了 lambda 表达式进行定义：

Expr.java　1.js

C: > Users > Nonehyo > Desktop > 1.js > ...

```
1  high = (v) => (v >>> 16) & 0x0000ffff;
2  low = (v) => v & 0x0000ffff;
3
4  var ans = (3 + 0x20) % 0x11 << 3 - low(high(0x00ff0000) * low(3)) >> 1;
5  console.log(ans);
```

问题 2　输出　调试控制台　终端

```
[Running] node "c:\Users\Nonehyo\Desktop\tempCodeRunnerFile.js"
32

[Done] exited with code=0 in 0.225 seconds
```

经检验，程序的运行结果和标准结果一致。

4. 实验心得

在本次实验中，我通过了之前所学的数据结构的知识，使用堆栈这一数据结构来实现表达式的求值。并在其中使用了 Java 的 Regex Expression、HashMap、Vector、Stack、StringBuilder 以及泛型编程、异常处理等语法糖来辅助该功能的实现。在实验过程中，主要的难点在于字符串的处理，特别是取高位和低位的处理。在最早的算法中，我是通过标记 high(来作为起始点，)作为终止点，这样的算法有一个缺陷，当使用 high(123)*low(234)这一表达式作为测试对象时，中间截取的内容变成了 123)*low(234 而不是 123。最终的解决方案是对 high()中的表达式进行了限制，只允许出现基本的表达式而不允许出现括号，从而解决了这个问题。