

后缀表达式 Postfix 实验报告

13331181 龙嘉伟

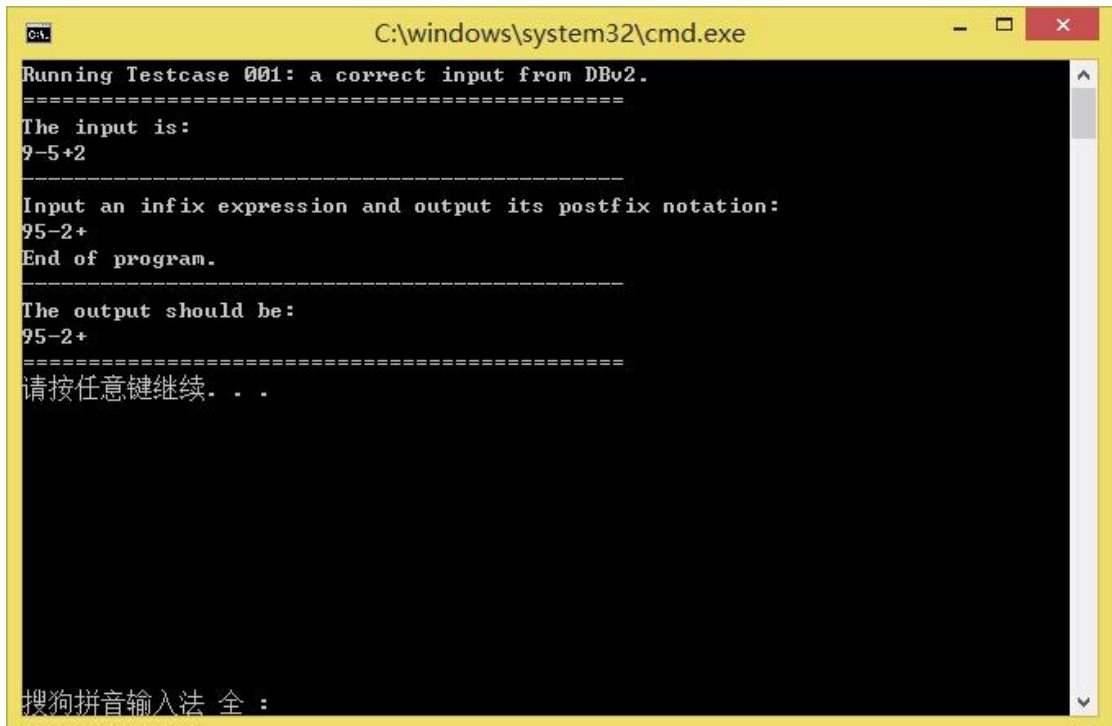
1. 比较静态成员与非静态成员

【实验结果】

静态成员:

```
class Parser {  
    static int lookahead;  
    static int count;  
};
```

Testcase1



```
C:\windows\system32\cmd.exe  
Running Testcase 001: a correct input from DBv2.  
=====  
The input is:  
9-5+2  
-----  
Input an infix expression and output its postfix notation:  
95-2+  
End of program.  
-----  
The output should be:  
95-2+  
=====  
请按任意键继续. . .  
搜狗拼音输入法 全 :
```

Testcase2

```
C:\windows\system32\cmd.exe

Running Testcase 002: a correct long input.
=====
The input is:
1-2+3-4+5-6+7-8+9-0
-----
Input an infix expression and output its postfix notation:
12-3+4-5+6-7+8-9+0-
End of program.
-----
The output should be:
12-3+4-5+6-7+8-9+0-
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

Testcase3

```
C:\windows\system32\cmd.exe

Running Testcase 003: missing an operator.
=====
The input is:
95+2
-----
Input an infix expression and output its postfix notation:
92+
<0>Syntax error at 2:lack of operator
End of program.
-----
The output should be:
92+
<0>Syntax error at 2:lack of operator
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

Testcase4

```
C:\windows\system32\cmd.exe

Running Testcase 004: missing an operand.
=====
The input is:
9-5+-2
=====
Input an infix expression and output its postfix notation:
95-2+
<0>Syntax error at 5:lack of operand

End of program.
=====
The output should be:
95-2+
<0>Syntax error at 5:lack of operand
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

非静态成员:

```
class Parser {
    int lookahead;
    static int count;
```

Testcase1

```
C:\windows\system32\cmd.exe

Running Testcase 001: a correct input from DBv2.
=====
The input is:
9-5+2
=====
Input an infix expression and output its postfix notation:
95-2+
End of program.
=====
The output should be:
95-2+
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

Testcase2

```
C:\windows\system32\cmd.exe

Running Testcase 002: a correct long input.
=====
The input is:
1-2+3-4+5-6+7-8+9-0
-----
Input an infix expression and output its postfix notation:
12-3+4-5+6-7+8-9+0-
End of program.
-----
The output should be:
12-3+4-5+6-7+8-9+0-
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

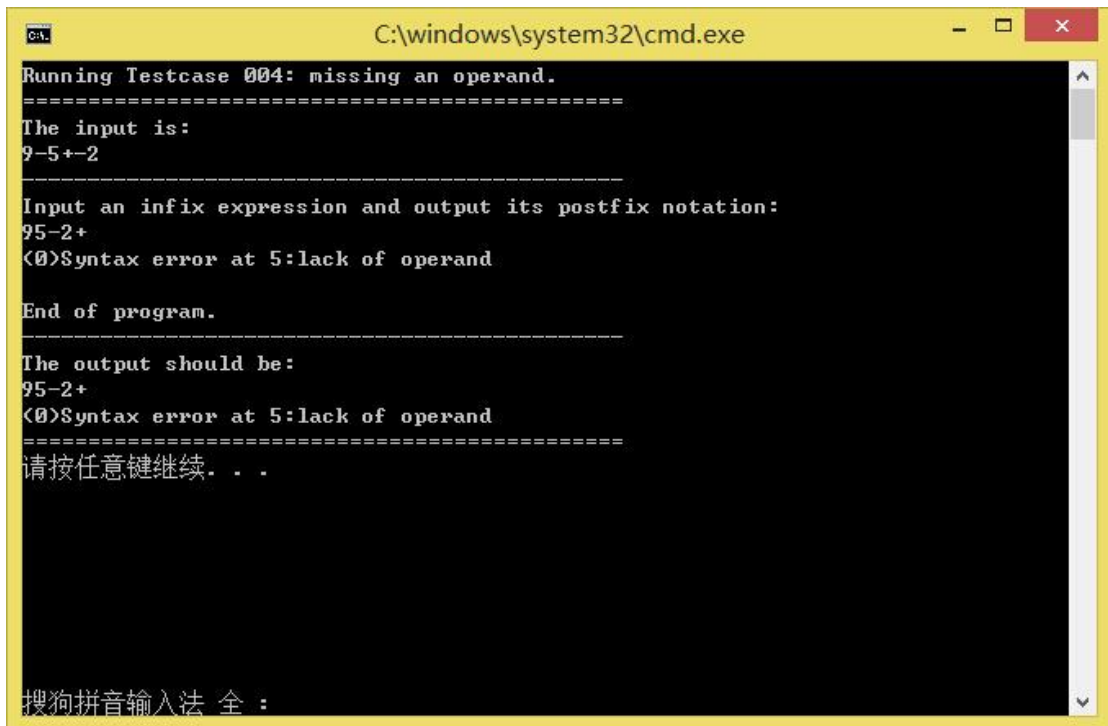
Testcase3

```
C:\windows\system32\cmd.exe

Running Testcase 003: missing an operator.
=====
The input is:
95+2
-----
Input an infix expression and output its postfix notation:
92+
<0>Syntax error at 2:lack of operator
End of program.
-----
The output should be:
92+
<0>Syntax error at 2:lack of operator
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

Testcase4



```
C:\windows\system32\cmd.exe
Running Testcase 004: missing an operand.
=====
The input is:
9-5+-2
=====
Input an infix expression and output its postfix notation:
95-2+
<0>Syntax error at 5: lack of operand

End of program.
=====
The output should be:
95-2+
<0>Syntax error at 5: lack of operand
=====
请按任意键继续. . .
搜狗拼音输入法 全
```

【结果分析】

将类成员 `lookahead` 声明为 `static` 和非 `static` 没有影响到程序的正确性。

经过查询资料，我了解到 `java` 中的 `static` 变量被所有的对象所共享，在内存中只有一个副本，当且仅当在类初次加载时被初始化。

在这个程序中，`Parser` 只会被实例化一次，因此 `static` 变量与非 `static` 变量没有明显的差别。

我认为在这个程序中，将 `lookahead` 声明为静态变量比较好。当加载 `Parser` 类时，就完成了 `lookahead` 的初始化，而在构造一个 `Parser` 类的对象时，就可以直接对其进行赋值，这个程序的构造函数就有对 `lookahead` 赋值的语句。

2. 比较消除尾递归前后程序的性能

【实验方案】

我使用了一个程序随机生成长度为 10001 的合法字符串，并将它保存到一个文件中。然后使用递归的 `Postfix` 程序和非递归的 `Postfix` 程序读入同一个输入文件，通过系统提供的获取时间命令获得 `expr()` 函数执行前后的时间差，以此作为程序性能的指标。将上述过程重复 50 次（每次的输入内容均不同），分别统计递归和非递归程序所花费的时间。统计的内容包括：1. 将同一个输入的递归和非递归程序运行的时间直接对比；2. 记录递归程序 50 次的运行过程中花费的不同时间出现的次数；3. 记录非递归程序 50 次的运行过程中花费的不同时间出现的次数。通过 `Excel` 的图表功能对上述统计结果进行展示。

【实验代码】

下图是 `create` 程序的代码部分，用于随机生成长度为 10001 的合法输入并保存到一个文件中。

```

1 import java.io.*;
2
3
4 public class create {
5     public static void main(String[] args) throws IOException {
6         File file = new File("D:/EclipseFile/project one/src/output.txt");
7         FileOutputStream s = new FileOutputStream(file);
8         int count = 10001;
9         String str = "";
10        int flag = 1;
11        while (count!=0) {
12            if (flag == 1) {
13                str += new Random().nextInt(10);
14                flag = 0;
15            }
16            else if (flag == 0) {
17                int tmp = new Random().nextInt(2);
18                if (tmp == 1) {
19                    str += '+';
20                }
21                else {
22                    str += '-';
23                }
24                flag = 1;
25            }
26            count--;
27        }
28        byte[] contenInBytes = str.getBytes();
29        s.write(contenInBytes);
30        s.close();
31    }
32 }
33

```

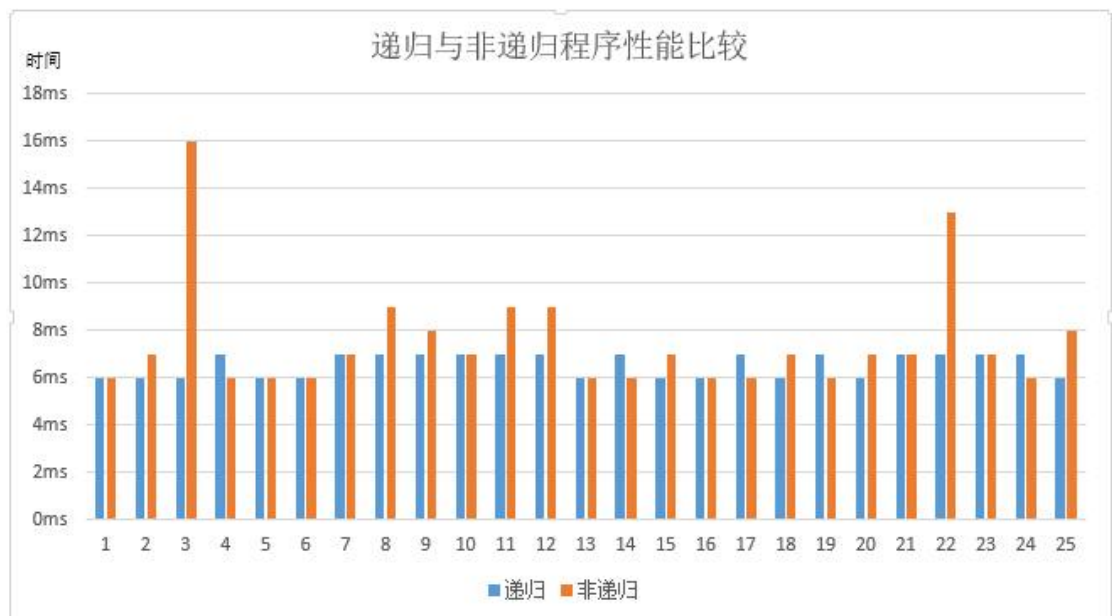
下图是批处理文件的内容，Postfix.java 是非递归版本，Postfix1.java 是递归版本。

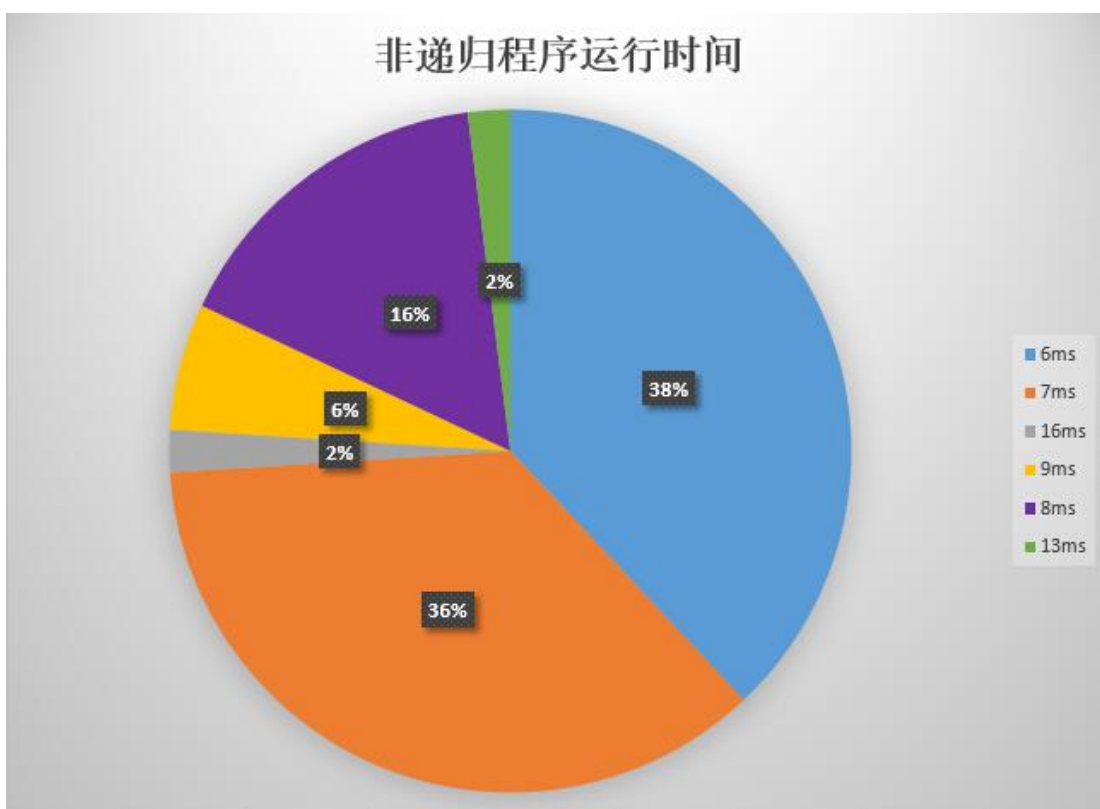
```

@echo off
javac create.java
javac Postfix.java
javac Postfix1.java
for /l %%i in (1,1,50) do (
java create
java Postfix <output.txt> out.txt
java Postfix1 <output.txt> out1.txt
)

```

【实验结果】





【结果分析】

按照常理而言，一个递归程序会比与其等价的非递归程序低效。因此，实验的预期结果应该是递归版本程序所化时间多于非递归版本。但根据 50 次的运行结果，非递归版本程序运行时间普遍等于或大于递归版本程序，而且在某些情况下所花的时间远大于同一

输入下的递归版本程序。就递归版本而言，6ms 和 7ms 出现的次数占到 94%，大于 7ms 的仅占 4%。而非递归版本中，6ms 和 7ms 出现的次数仅占 74%，其余占比的运行时间均大于 7ms。

为什么会有这个出乎意料的结果呢？经过在网上查找相关资料，我认为是这个程序的逻辑太简单，导致编译器所作的优化大于非递归之于递归的优势，因此非递归版本的效率反而低于递归版本。

3. 扩展错误处理功能

【算法分析】

(1) 如何给出明确地出错信息：

首先是要区分遇到的是词法错误还是语法错误。当输入的字符不是‘0’到‘9’或‘-’或‘+’时，可以归类为词法错误。而语法错误包括运算符、运算数的缺失。

`expr()`函数调用了 `term()`函数来识别第一个数字，然后调用 `rest()`来识别剩下的部分。当一个串的第一个输入不是数字时，就出现了一个错误，而这个错误的识别在 `term()`函数里面实现。当第一个输入为‘+’或‘-’时，说明缺少了一个运算数，属于语法错误。当第一个输入为非‘0’到‘9’或‘+’或‘-’时，说明出现了一个非法字符输入，属于词法错误。

当一个串的第一个输入是数字时，将其输出，并转到 `rest()`的执行。`rest()`函数判断下一个输入是否为‘+’和‘-’，如果不是这两个字符，就判断是否为数字，如果是数字，就说明缺少了运算符，属于语法错误。如果是非数字字符，说明出现了一个非法字符输入，属于词法错误。而当 `rest()`判断输入字符为‘+’或‘-’时，就会匹配字符并调用 `term()`进行预读，`term()`的识别过程已说明，不再赘述。

(2) 如何实现错误的定位：

对于错误的定位，我使用了一个位置标记，记录当前读到串的哪一位，而当发现错误时，这个位置标记就能确定错误的定位。

(3) 如何实现出错后的恢复：

这是实现错误处理功能难度最大的部分。我采用了恢复策略是恐慌模式的错误恢复，即当发现错误时，丢弃零个或多个输入字符，直到输入合法为止。这个策略的好处在于能尽量多地发现原输入串的错误，并且不引入新的错误。在发现错误时，先将其记录下来，直到整个串读入结束后再进行错误信息的输出。

这个方法的关键是发现错误时要丢弃字符并继续判断后续字符是否能够与之前的字符组成合法的搭配。

例如输入为“12-3+4”，由于在‘1’后缺少了一个运算符，读到‘2’时就发现了这个错误，要继续扫描下去并且不给后面带来新的错误，最直接的办法就是把‘2’丢弃。而丢弃的实现在 `rest()`函数中，当出现了一个数字之后是另一个数字的输入，会先记录这个错误，并直接读入下一位，继续执行判断。

又例如输入为“1-2+-3”，由于‘+’后紧接着‘-’，缺少了一个运算数。‘+’是在 `rest()`中被识别的，然后进入了‘+’分支，调用了 `term()`函数进行预读，而读入‘-’后发现这是一个语法错误，记录下来后多读一位，然后递归调用 `term()`判断输入字符，直到输入为数字才会将其输出，将控制返回到 `rest()`函数中，继续执行程序。

对于非法字符的处理，是最为简单的，直接舍弃并不会对程序的正确性带来影响。

【实验结果】


```
C:\windows\system32\cmd.exe

Running Testcase 001: a correct input from DBv2.
=====
The input is:
9-5+2
-----
Input an infix expression and output its postfix notation:
95-2+
End of program.
-----
The output should be:
95-2+
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

上图是一个正确的输入。

```
C:\windows\system32\cmd.exe

Running Testcase 002: a correct long input.
=====
The input is:
1-2+3-4+5-6+7-8+9-0
-----
Input an infix expression and output its postfix notation:
12-3+4-5+6-7+8-9+0-
End of program.
-----
The output should be:
12-3+4-5+6-7+8-9+0-
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

上图是一个正确的较长的输入。

```
C:\windows\system32\cmd.exe

Running Testcase 003: missing an operator.
=====
The input is:
95+2
-----
Input an infix expression and output its postfix notation:
92+
<0>Syntax error at 2:lack of operator

End of program.
-----
The output should be:
92+
<0>Syntax error at 2:lack of operator
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

上图是缺少一个运算符的输入。

```
C:\windows\system32\cmd.exe

Running Testcase 004: missing an operand.
=====
The input is:
9-5+-2
-----
Input an infix expression and output its postfix notation:
95-2+
<0>Syntax error at 5:lack of operand

End of program.
-----
The output should be:
95-2+
<0>Syntax error at 5:lack of operand
=====
请按任意键继续. . .

搜狗拼音输入法 全 :
```

上图是缺少一个运算数的输入。

```
C:\windows\system32\cmd.exe

Running Testcase 005: illegal character.
=====
The input is:
6- 2+&2-5-!3
=====
Input an infix expression and output its postfix notation:
62-2+5-3-
<0>Lexical error at 3:illegal character
<1>Lexical error at 6:illegal character
<2>Lexical error at 11:illegal character

End of program.
=====
The output should be:
62-2+5-3-
<0>Lexical error at 3:illegal character
<1>Lexical error at 6:illegal character
<2>Lexical error at 11:illegal character
=====
请按任意键继续. . .
搜狗拼音输入法 全 :
```

上图是出现三个非法字符的输入。

```
C:\windows\system32\cmd.exe

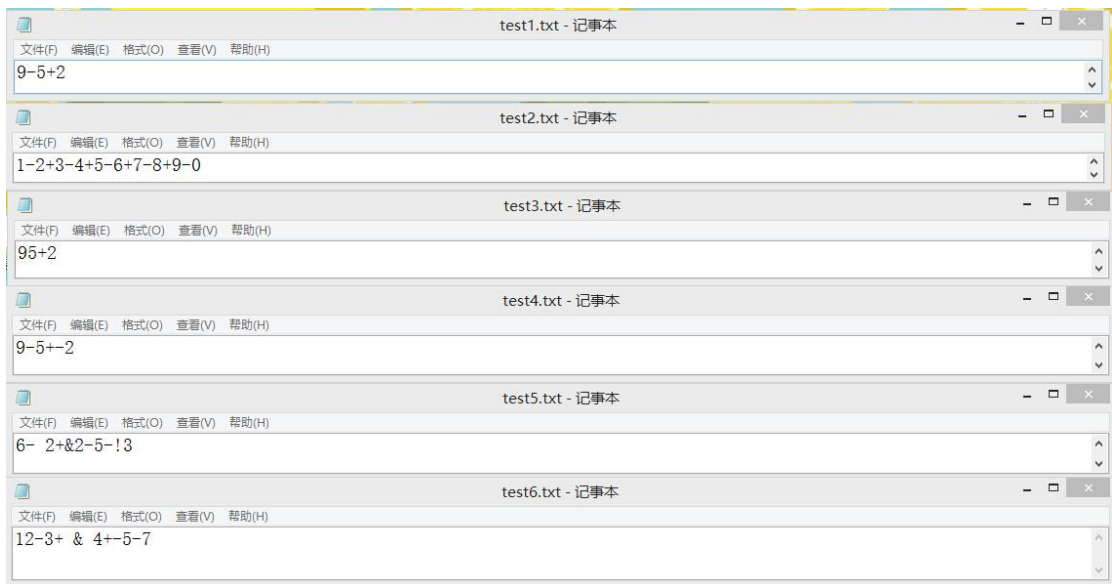
Running Testcase 006: different kinds of errors
=====
The input is:
12-3+ & 4+-5-7
=====
Input an infix expression and output its postfix notation:
13-4+5+7-
<0>Syntax error at 2:lack of operator
<1>Lexical error at 6:illegal character
<2>Lexical error at 7:illegal character
<3>Lexical error at 8:illegal character
<4>Syntax error at 11:lack of operand

End of program.
=====
The output should be:
13-4+5+7-
<0>Syntax error at 2:lack of operator
<1>Lexical error at 6:illegal character
<2>Lexical error at 7:illegal character
<3>Lexical error at 8:illegal character
<4>Syntax error at 11:lack of operand
=====
搜狗拼音输入法 全 :
```

上图是缺少一个运算符、一个运算数且出现三个非法字符的输入。

4. 单元测试

【测试输入】



【测试代码】

```
1 import static org.junit.Assert.*;
2
3 import java.io.IOException;
4
5 import org.junit.Before;
6 import org.junit.Test;
7
8 public class MyPostfixTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void test1() throws IOException {
16         MyParser p = new MyParser("D://test1.txt");
17         p.expr();
18         assertEquals(p.getResult(), "95-2+");
19         assertEquals(p.getErrorNum(), 0);
20     }
21
22     @Test
23     public void test2() throws IOException {
24         MyParser p = new MyParser("D://test2.txt");
25         p.expr();
26         assertEquals(p.getResult(), "12-3+4-5+6-7+8-9+0-");
27         assertEquals(p.getErrorNum(), 0);
28     }
29
30     @Test
31     public void test3() throws IOException {
32         MyParser p = new MyParser("D://test3.txt");
33         p.expr();
34         assertEquals(p.getResult(), "92+");
35         assertEquals(p.getErrorNum(), 1);
36     }
37
38     @Test
39     public void test4() throws IOException {
40         MyParser p = new MyParser("D://test4.txt");
41         p.expr();
42         assertEquals(p.getResult(), "95-2+");
43         assertEquals(p.getErrorNum(), 1);
44     }
45
46     @Test
47     public void test5() throws IOException {
48         MyParser p = new MyParser("D://test5.txt");
49         p.expr();
50         assertEquals(p.getResult(), "62-2+5-3-");
51         assertEquals(p.getErrorNum(), 3);
52     }
53
54     @Test
55     public void test6() throws IOException {
56         MyParser p = new MyParser("D://test6.txt");
57         p.expr();
58         assertEquals(p.getResult(), "13-4+5+7-");
59         assertEquals(p.getErrorNum(), 5);
60     }
61 }
```

【测试结果】

Markers

Properties

Servers

Data Source Explorer

Snippets

Problems

Console

JUnit

Finished after 0.03 seconds

Runs: 6/6

Errors: 0

Failures: 0

MyPostfixTest [Runner: JUnit 4] (0.001 s)

Failure Trace

test1 (0.001 s)

test2 (0.000 s)

test3 (0.000 s)

test4 (0.000 s)

test5 (0.000 s)

test6 (0.000 s)