

# 后缀表达式 Postfix 实验报告

## Step1:

在这个程序当中，将 `lookahead` 定义为 `int` 不会影响程序的正确性，因为我们只定义了一个 `Parser`，这时只有一个对象，因此对象的变量就只有该对象自己能够使用。而一旦这个类有多个对象，这个时候二者就有区别了，如果 `lookahead` 是 `static` 类型的，任何一个 `Parser` 对其的修改都会影响其他所有的对象，而如果仅仅是 `int` 类型的话，那样每个 `Parser` 对象 `lookahead` 都是独立的，这时会造成读入的不一致性，因此还是用 `static` 比较好。

## Step2:

我们知道，递归会产生嵌套调用，这时候系统会不断的利用自带的函数栈进行压栈的操作，这样做不仅浪费空间，而且压栈和弹出的过程肯定不如循环来的高效，但是尾部递归有些不同，不管递归有多深，只要有尾部递归优化，栈的大小都保持不变，在尾部递归进行函数调用的时候使用下一个栈结构取代当前的栈结构，因此在不同情况下二者各有优劣，我们先从数学的角度对其进行分析：

根据上面信息我们可以知道，其实很多编译器都会自动将尾递归优化为循环，而且从本质上来看二者没有太大的区别，因此二者的时间复杂度都为  $O(n)$ 。但是尾递归涉及到函数栈的不断调用，不经过优化之前会一直递归压栈，这样一来空间复杂度大约为  $O(n)$ ，而循环每次调用栈之后就会释放掉，因此空间复杂度为  $O(1)$ 。

接下来我们利用实践方式来检验我们的理论正确性：

我们可以随机产生任意长的序列进行测试，每个操作数随机从 `0~9` 中抽取，每个操作符为 `+/-`，我们在新类 `Runner` 中实现这个函数，如下图。之后我们在 `Parser` 中调用这个函数，让 `Parser` 的解析对象为我们随机生成的任意长的字符串，我们需要记录的数据主要是在不同情况下解析完整字符串所花的时间，通过比较二者的差别来得出结论，我们利用文件输入和输出的方法将大量测试时间输出，并用 `matlab` 对得到的数据进行绘图，我们预期的结果是二者的时间大致相同，但是在数据量过大的时候，尾递归的堆栈会溢出。

我们先随机产生了一个 `10000` 长度的字符串进行尾递归，发现时间大约为 `17ms`：

```

5 public class Runner {
6     public String get_string() throws IOException {
7         String teststring = "1";
8         for (int i = 0; i < 10000; i++) {
9             char testchar = Math.random() > 0.5 ? '+' : '-';
10            teststring += testchar + new Integer((int) (Math.random()*10)).toStr
11        }
12        teststring += '\n';
13        return teststring;
14    }
15 }
16

```

Problems @ Javadoc Declaration Console

<terminated> Postfix [Java Application] C:\Program Files\Java\jdk1.8.0\_60\bin\javaw.exe (2015年11月19日 上午10:55:5  
Input an infix expression and output its postfix notation:  
16+6+2-8-6+7+2-1-3-1-7-8+1+6-9-2-5-0-8-0-1+8+7+3-6+7+0+2+2-8-0+3+8-8-4+6-0+1-4-0-9-9-  
程序运行时间: 17ms

之后我们随机产生了一个 100000 长度的字符串进行尾递归，发现栈的空间不够了：

```

        } else if (lookahead == '-') {
            match('-');
            term();
            System.out.write('-');
            rest();
        } else {
            // do nothing with the input
        }
    }

    void term() throws IOException {
        if (Character.isDigit((char) lookahead)) {
            System.out.write((char) lookahead);
            match(lookahead);
        } else throw new Error("syntax error");
    }
}

```

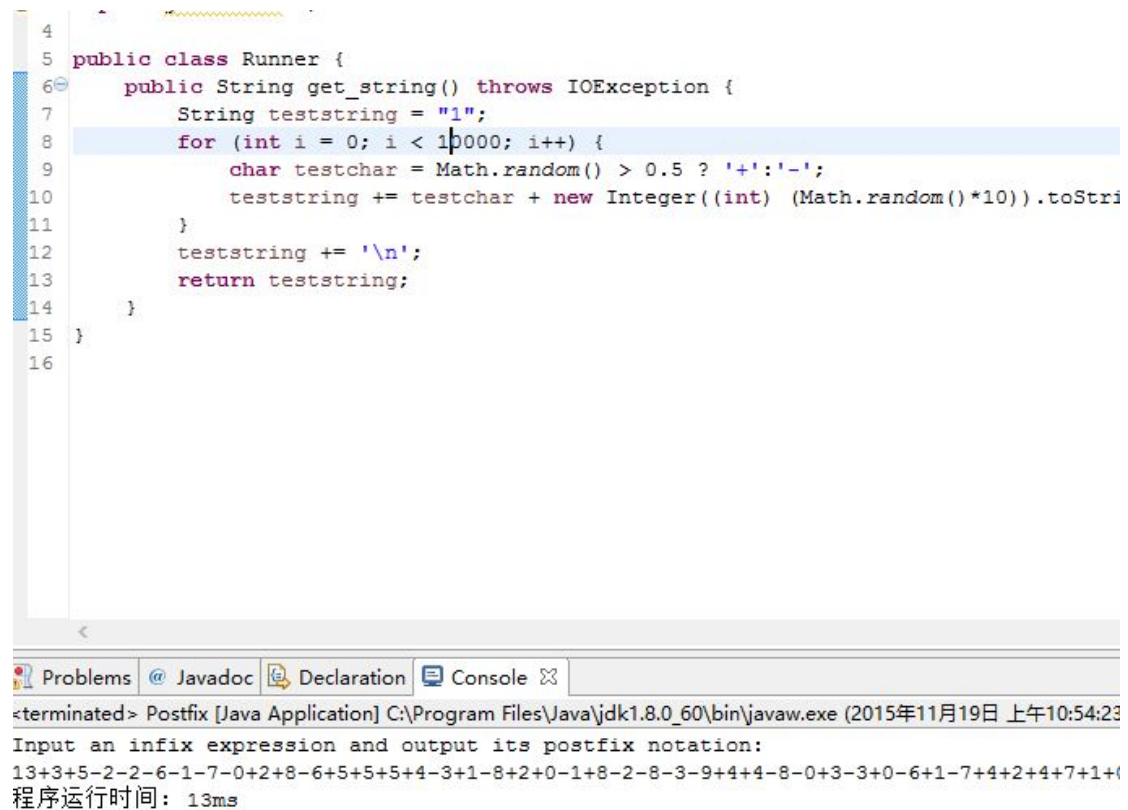
Problems @ Javadoc Declaration Console

<terminated> Postfix [Java Application] C:\Program Files\Java\jdk1.8.0\_60\bin\javaw.exe  
at Parser.rest(Postfix.java:24)  
at Parser.rest(Postfix.java:29)  
at Parser.rest(Postfix.java:29)  
at Parser.rest(Postfix.java:24)  
at Parser.rest(Postfix.java:29)

说明尾递归在不进行优化的前提下会不断进行系统压栈，进而造成内存空间不足。

接下来我们对循环的性能进行测试：

我们先对 10000 大小的字符串长度进行测试，发现时间和尾递归效率差不多，稍微快了一点：



```
4
5 public class Runner {
6     public String get_string() throws IOException {
7         String teststring = "1";
8         for (int i = 0; i < 10000; i++) {
9             char testchar = Math.random() > 0.5 ? '+' : '-';
10            teststring += testchar + new Integer((int) (Math.random()*10)).toStri
11        }
12        teststring += '\n';
13        return teststring;
14    }
15 }
16
```

Problems Javadoc Declaration Console

<terminated> Postfix [Java Application] C:\Program Files\Java\jdk1.8.0\_60\bin\javaw.exe (2015年11月19日 上午10:54:23)

Input an infix expression and output its postfix notation:

13+3+5-2-2-6-1-7-0+2+8-6+5+5+5+4-3+1-8+2+0-1+8-2-8-3-9+4+4-8-0+3-3+0-6+1-7+4+2+4+7+1+0

程序运行时间：13ms

之后我们对 100000 长度数组进行测试，发现循环由于不用产生栈，所以不会出现内存空间不够的情况，时间大约是原来的 10 倍左右。

```

11     lookahead = test_string.charAt(indexs);
12 }
13
14 void expr() throws IOException {
15     long startTime=System.currentTimeMillis(); //获取开始时间
16     term();
17     rest();
18     long endTime=System.currentTimeMillis(); //获取结束时间
19     System.out.println("\n程序运行时间: "+(endTime-startTime)+"ms\n");
20 }
21
22 void rest() throws IOException {
23     boolean flag = true;
24     while (flag) {
25         if (lookahead == '+') {

```

<terminated> Postfix [Java Application] C:\Program Files\Java\jdk1.8.0\_60\bin\javaw.exe (2015年11月19日 上午 17-0+3-0-8-9-3-0-6+1-7-3-6+6+2-2-8-2+6-0+6-7+2+5-2-0-5-3+2-7+9+9+1+7-4-9-8+7+0-1  
程序运行时间: 107ms

End of program.

后来为了更好的比较二者之间的性能，我们测试了大量递归和循环的数据，如下图所示：

```

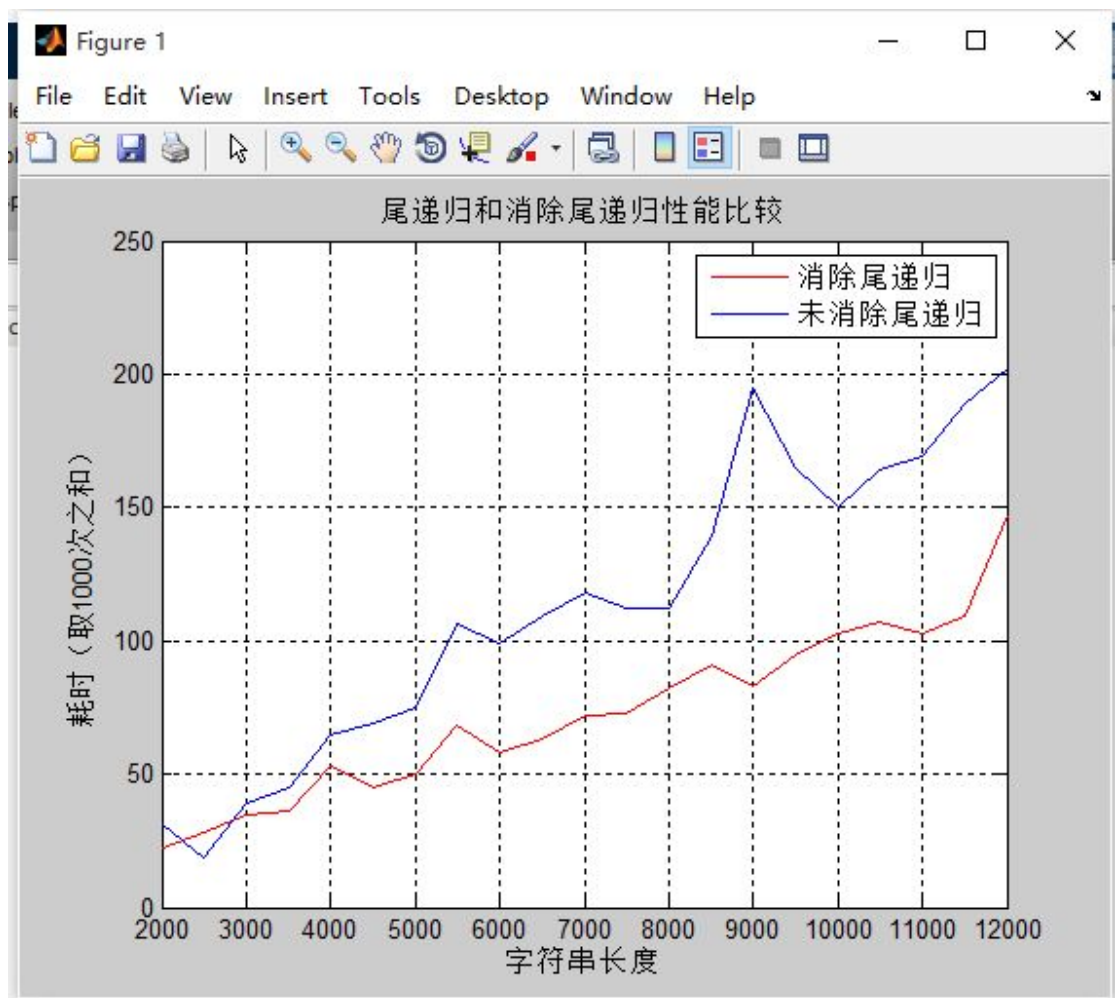
94     } catch(IOException e) {
95         System.out.println("创建data.txt文件失败, 错误信息: "+e.getMessage());
96         return;
97     }
98 }
99
100 PrintWriter pw = new PrintWriter(datafile);
101 pw.write("loop length,          loop totaltimes,          recur totaltimes\n");
102 for (int i = 1000; i < 12000; ) {
103     i += 500;
104     double loop_totaltimes = 0;
105     for (int j = 0; j < 1000; j++) {
106         loop_totaltimes += new Parser(i).expr();
107     }
108     double recur_totaltimes = 0;
109     for (int j = 0; j < 1000; j++) {
110         recur_totaltimes += new Parser(i).expr2();
111     }
112     String str = "" + i + " " + loop_totaltimes + " " + recur_totaltimes + " \n";
113     pw.write(str);
114 }
115 pw.close();
116 System.out.println("\nEnd of program.");
117 }
118

```

ndroid

将对应的数据写入文件中得到了如下数据，之后我们利用 matlab 自带的画图功能，将递归和循环的性能进行比较，得到下图：

	loop length,	loop totaltimes,	recur totaltimes
1	1500	77.0	72.0
2	2000	56.0	89.0
3	2500	76.0	144.0
4	3000	134.0	151.0
5	3500	106.0	196.0
6	4000	129.0	201.0
7	4500	131.0	188.0
8	5000	138.0	201.0
9	5500	151.0	221.0
10	6000	69.0	159.0
11	6500	87.0	128.0
12	7000	87.0	102.0
13	7500	96.0	157.0
14	8000	106.0	130.0
15	8500	127.0	155.0
16	9000	142.0	177.0
17	9500	126.0	177.0
18	10000	124.0	180.0
19	10500	131.0	189.0
20	11000	153.0	223.0
21	11500	134.0	232.0
22	12000	146.0	263.0
23			
24			



如图所示，发现循环和尾递归处于同一个数量级，时间上相差不大，这和我们开始分析二者的时间复杂度均为  $O(n)$  是一致的，但是可以看到，当字符串长度逐渐增大的时候，循环的耗时会比尾递归要小一些，经过分析，我们认为这是因为尽管二者时间复杂度是一样的，但是尾递归涉及到栈的申请和释放，显然会耗费一些时间，这也是为什么我们要选择消除尾递归



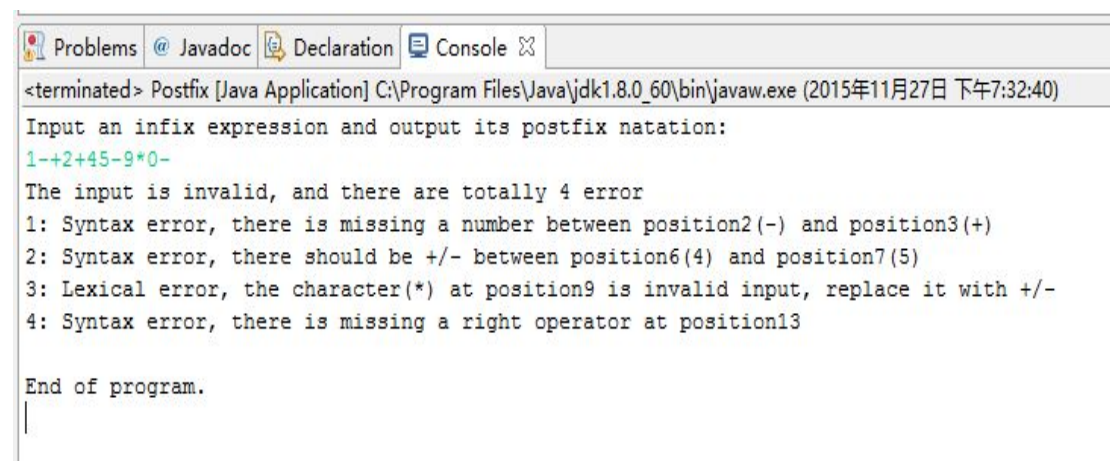
的一个原因。此外，在图上我们发现曲线不是太平滑，原因应该是取的点还是相对较少，而且计算机在同时运行多个线程，在不同时刻处理我们字符串的能力不同，但可以看到，大体的趋势还是正确的，如果我们选用 c 语言编程，或许可以在分析耗时上取得更好的效果。

这说明尾递归是一种效率极高的递归，通过老赵的博客和自己的分析，其实很多编译器都可以自动将尾递归优化成循环，这是由于函数在递归调用之前已经把所有的计算任务已经完毕了，他只要把得到的结果全交给子函数就可以了，无需保存什么，子函数其实可以不需要再去创建一个栈帧，直接把就着当前栈帧，把原先的数据覆盖即可。这样看来，其实尾递归和循环的效率大致是在同一个数量级上的，不过尾递归可能需要  $O(n)$  左右的空间复杂度，还是避免不了堆栈和堆栈溢出的问题，这也是为什么我们用循环代替递归的一个原因。

### Step3:

通过学习书上几种错误恢复策略，我们可以利用错误产生式这样类似于 Yacc 的实现规则来定义在简单加减乘除过程中可能出现的错误，并且一旦出现错误，可以根据错误表达式给出的恢复方案进行恢复，然后直接通过 lookahead 读取下一个字符，进而继续分析下去，一次发现多个错误。

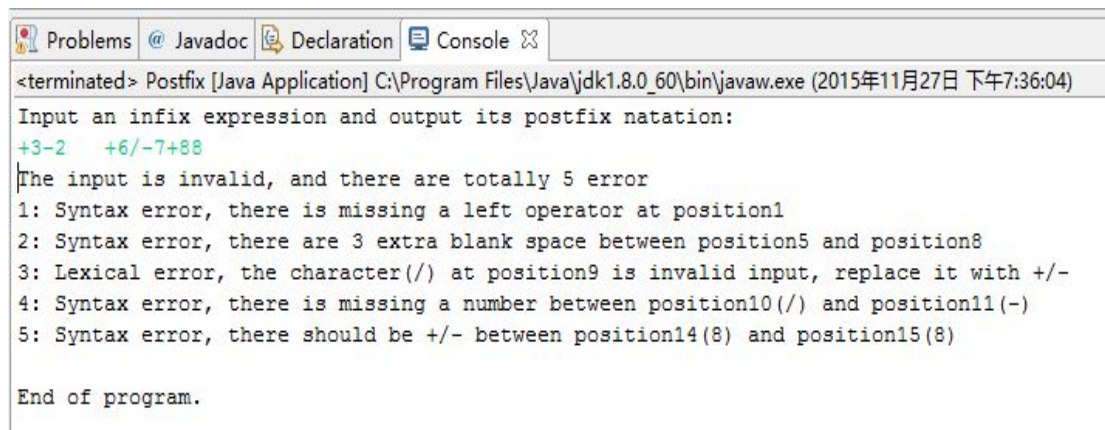
由于设计的程序可以一次检测出多个错误，所以我们设计了一条包含各种典型错误的中缀表达式 `1-+2+45-9*0-` 所以，得到如下错误提示信息：



```
<terminated> Postfix [Java Application] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (2015年11月27日 下午7:32:40)
Input an infix expression and output its postfix notation:
1-+2+45-9*0-
The input is invalid, and there are totally 4 error
1: Syntax error, there is missing a number between position2(-) and position3(+)
2: Syntax error, there should be +/- between position6(4) and position7(5)
3: Lexical error, the character(*) at position9 is invalid input, replace it with +/-
4: Syntax error, there is missing a right operator at position13

End of program.
```

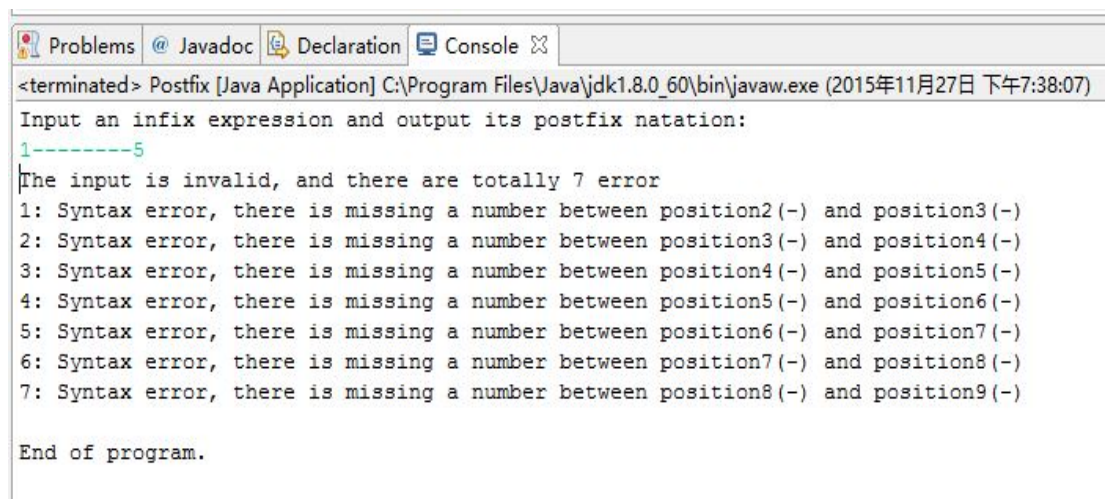
可以看到，位置 2 和位置 3 两个运算符之间缺少运算分量，位置 6 和位置 7 两个运算数之间缺少运算符，以及表达式结尾缺少右运算分量，这些都属于语法错误，而位置 9 上符号 \* 不能被识别，属于词法错误，程序都会按照扫描顺序依次给出错误的位置和类别，一次运行可发现多个错误，接下来我们再给出几个例子：



```
<terminated> Postfix [Java Application] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (2015年11月27日 下午7:36:04)
Input an infix expression and output its postfix natation:
+3-2 +6/-7+88
The input is invalid, and there are totally 5 error
1: Syntax error, there is missing a left operator at position1
2: Syntax error, there are 3 extra blank space between position5 and position8
3: Lexical error, the character(/) at position9 is invalid input, replace it with +/-
4: Syntax error, there is missing a number between position10(/) and position11(-)
5: Syntax error, there should be +/- between position14(8) and position15(8)

End of program.
```

可以看到，表达式开头就缺少左操作符，属于语法错误，如果表达式中间存在着一个或者多个空格，比如位置 5 和位置 8 之间存在着 3 个空格，同样会被认为是语法错误，位置 9 出现了/这样一个不能识别的符号，我们会认为用户想输入的是+或者-，所以这样一来，位置 9 和位置 10 之间就缺少运算分量，结尾处两个连续的运算分量之间缺少运算符。



```
<terminated> Postfix [Java Application] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (2015年11月27日 下午7:38:07)
Input an infix expression and output its postfix natation:
1-----5
The input is invalid, and there are totally 7 error
1: Syntax error, there is missing a number between position2(-) and position3(-)
2: Syntax error, there is missing a number between position3(-) and position4(-)
3: Syntax error, there is missing a number between position4(-) and position5(-)
4: Syntax error, there is missing a number between position5(-) and position6(-)
5: Syntax error, there is missing a number between position6(-) and position7(-)
6: Syntax error, there is missing a number between position7(-) and position8(-)
7: Syntax error, there is missing a number between position8(-) and position9(-)

End of program.
```

如上图，如果我同样的错误循环的出现，程序会不厌其烦的指出每一个错误，并且对每一个错误都列出错误的位置和类别，比如每两个相邻的运算符之间应当插入一个运算数。

相应处理错误的代码如下（部分截图）：

```

3      boolean flag = true;
4      while (flag) {
5          if (lookahead == '+') {
6              match('+');
7              term();
8              //System.out.write('+');
9              output_Postfix += '+';
10         } else if (lookahead == '-') {
11             match('-');
12             term();
13             //System.out.write('-');
14             output_Postfix += '-';
15         } else if ((char)lookahead == '\n' || lookahead == 13) {
16             flag = false;
17         } else if (Character.isDigit((char)lookahead)) {
18             String temp = "Syntax error, there should be +/- between position" + position.toString() + "(" + input_Infix.charAt(position-1) + ") and position";
19             error_list.add(temp);
20             state = false;
21             term();
22         } else if ((char)lookahead == ' ') {
23             Integer space_num = 0;
24             while ((char)lookahead == ' ') {
25                 match(lookahead);
26                 space_num++;
27             }
28             state = false;
29             position++;
30             String temp = "Syntax error, there are " + space_num.toString() + " extra blank space between position" + new Integer(position-space_num).toString() + " and position";
31             error_list.add(temp);
32             rest();
33         } else {
34             String temp = "Lexical error, the character(" + (char)lookahead + ") at position" + position.toString() + " is invalid input, replace it with +/-";
35             error_list.add(temp);
36             match(lookahead);
37             state = false;
38             term();
39         }
40     }
41 }

```



```

void term() throws IOException {
    if (Character.isDigit((char)lookahead)) {
        //System.out.write((char)lookahead);
        Character temp_char = (char)lookahead;
        output_Postfix += temp_char.toString();
        match(lookahead);
    } else {
        if ((char)lookahead == ' ') {
            Integer space_num = 0;
            while ((char)lookahead == ' ') {
                match(lookahead);
                space_num++;
            }
            state = false;
            position++;
            String temp = "Syntax error, there are "+ space_num.toString()+" extra blank space between position"+new Integer(position-space_num).toString();
            position--;
            error_list.add(temp);
            term();
        } else if ((char)lookahead == '+' || (char)lookahead == '-') {
            if (position == 0) {
                String temp = "Syntax error, there is missing a left operator at position" + new Integer(position+1).toString();
                error_list.add(temp);
            } else {
                String temp = "Syntax error, there is missing a number between position" + position.toString() + "(" + input_Infix.charAt(position) + ")";
                error_list.add(temp);
            }
            state = false;
        } else if ((char)lookahead == '\n' || lookahead == 13) {
            position++;
            String temp = "Syntax error, there is missing a right operator at position" + position.toString();
            error_list.add(temp);
            state = false;
        } else {
            String temp = "Lexical error, the character(" + (char)lookahead + ") at position"+position.toString()+" is invalid input, replace it with a valid character";
            error_list.add(temp);
            match(lookahead);
            state = false;
        }
    }
}

```

## Step4:

通过学习网上提供的 javadoc 注释规范，我们可以用：

@author 标明开发该类模块的作者

@version 标明该类模块的版本

@see 参考转向，也就是相关主题

@param 对方法中某参数的说明

@return 对方法返回值的说明

@exception 对方法可能抛出的异常进行说明

这样可以增加我们代码的可读性，同时也可以更好的管理和维护代码。之后我们通过 Eclipse 自带的工具将文档化注释转化为相应的 HTML 文档并保存起来。对应的文档可以查看实验软装置中 doc 文件夹。

## Step5:

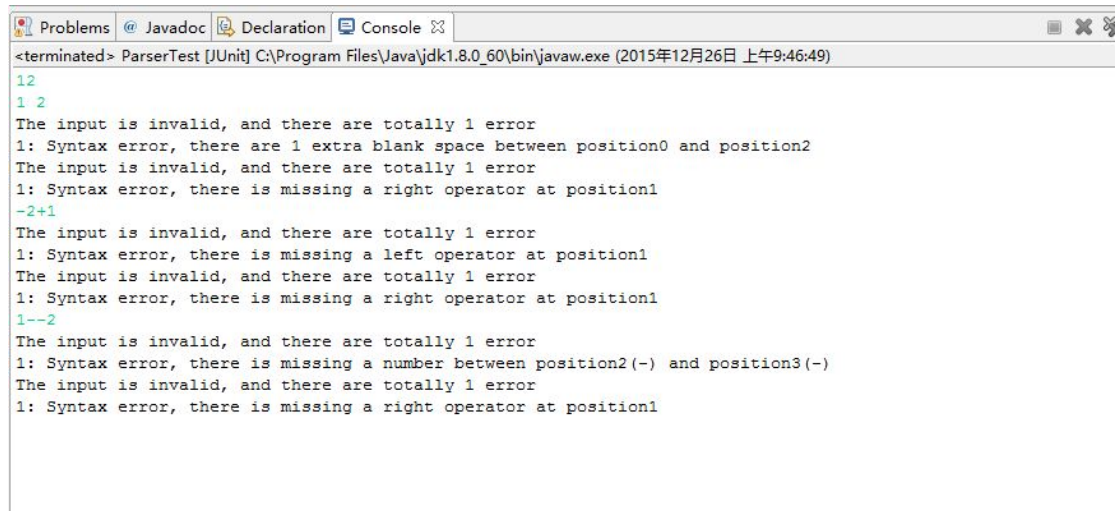
通过利用网上开源的 JUnit 测试工具，我们对整个程序可以分模块的验证正确性，通过上网查找教程或者直接从 Github 上下载源码，我们最自然的想法是对所有可能出错的地方设计对应的测试用例来验证其真确性。

首先我们先设计对应的用例，我们需要验证程序对于不同类型错误输入的响应，为了方便起见，我们用简单的例子来反映程序的错误恢复能力，输入的例子大概有 12(缺少运算符)、1 2(字符串之间有空格)、-2+1(缺少左运算分量)、1--2(两个运算符之间缺少运算分量)、1-(缺少右运算分量)、r-1(存在着不能识别的符号)，这样一些典型的错误来检查程序的对应错误检测功能，如下图：

The screenshot shows an IDE with a JUnit test runner on the left and Java code on the right. The test runner shows a successful run of the ParserTest class with four tests: testTerm (6.536 s), testRest (0.000 s), testMatch (12.662 s), and testExpr (23.038 s). The code on the right is the ParserTest class, which contains several test methods. The testExpr method is currently selected and highlighted in blue.

```
8 public class ParserTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void testExpr() throws IOException {
16         //input the string 12
17         assertEquals(new Parser().expr(), "Invalid, totally 1 error");
18         //input the string 1 2
19         assertEquals(new Parser().expr(), "Invalid, totally 1 error");
20         //input the string -2+1
21         assertEquals(new Parser().expr(), "Invalid, totally 1 error");
22         //input the string 1--2
23         assertEquals(new Parser().expr(), "Invalid, totally 1 error");
24         //input the string 1-
25         assertEquals(new Parser().expr(), "Invalid, totally 1 error");
26         //input the string r-1
27         assertEquals(new Parser().expr(), "Invalid, totally 1 error");
28     }
29
30     @Test
31     public void testRest() throws IOException {
32         assertEquals(new Parser().rest(), true);
33     }
34
35     @Test
36     public void testTerm() throws IOException {
37         assertEquals(new Parser().term(), true);
38     }
39
40     @Test
41     public void testMatch() throws IOException {
42         Parser myparser = new Parser();
43         int tempint = (int) (Math.random()*255);
44         myparser.lookahead = tempint;
45         assertEquals(myparser.match(tempint), true);
46     }
47 }
```

可以看到，程序中的各个函数(`term()`、`rest()`、`match()`、`expr()`)均能够按照要求正常工作，通过 `assertEquals` 函数我们就能够分模块的对所有用例逐一检验，如下图：



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a JUnit test named ParserTest. The output shows several test cases where the input is invalid due to syntax errors, and the program correctly identifies these errors. The errors are as follows:

```
<terminated> ParserTest [JUnit] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (2015年12月26日 上午9:46:49)
12
1 2
The input is invalid, and there are totally 1 error
1: Syntax error, there are 1 extra blank space between position0 and position2
The input is invalid, and there are totally 1 error
1: Syntax error, there is missing a right operator at position1
-2+1
The input is invalid, and there are totally 1 error
1: Syntax error, there is missing a left operator at position1
The input is invalid, and there are totally 1 error
1: Syntax error, there is missing a right operator at position1
1--2
The input is invalid, and there are totally 1 error
1: Syntax error, there is missing a number between position2(-) and position3(-)
The input is invalid, and there are totally 1 error
1: Syntax error, there is missing a right operator at position1
```

这样，我们通过使用 JUnit 就验证了程序对缺少运算符、字符串之间有空格、缺少左运算分量、两个运算符之间缺少运算分量、缺少右运算分量、存在着不能识别的符号这些错误均能正常识别。