中山大學 SUN YAT-SEN UNIVERSITY

编译原理

作业名称	后缀表达式 Postfix		任课老师	万老师		
院系	软件学院		方向	通信软件	姓名	李明宽
学号	11331173		完成日期	2013 年 11 月 09 日		
QQ	736459905		E-mail	limkuan@mail2.sysu.edu.cn		

一、 实验过程

1. 比较静态成员与非静态成员：

通过在实验中测试把 lookahead 为静态成员以及非静态成员之后程序的运行结果，发现无论把 lookahead 变为静态成员还是非静态成员，对程序的正确性影响都没有影响，因为在运行的主函数中，只是实例化了一个 Parser，因此都没错。但是我觉得如果需要使用多个 Parser 分析不同的字符串，例如有图形界面，使用多线程，一个 Parser 分析的是其中一个输入框的内容，还有一个 Parser 分析另一个输入框的内容，并且他们输出到不同的地方，这样如果使用 static 就会出错，因为这两个 Parser 共用一个 lookahead，就会出现交叉错误。

使用程序进行测试，关键代码如下：

```
private void BeginActionPerformed(java.awt.event.ActionEvent evt) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                new Parser_Loop(input_1, output_1).expr();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();

     new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                new Parser_Loop(input_2, output_2).expr();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

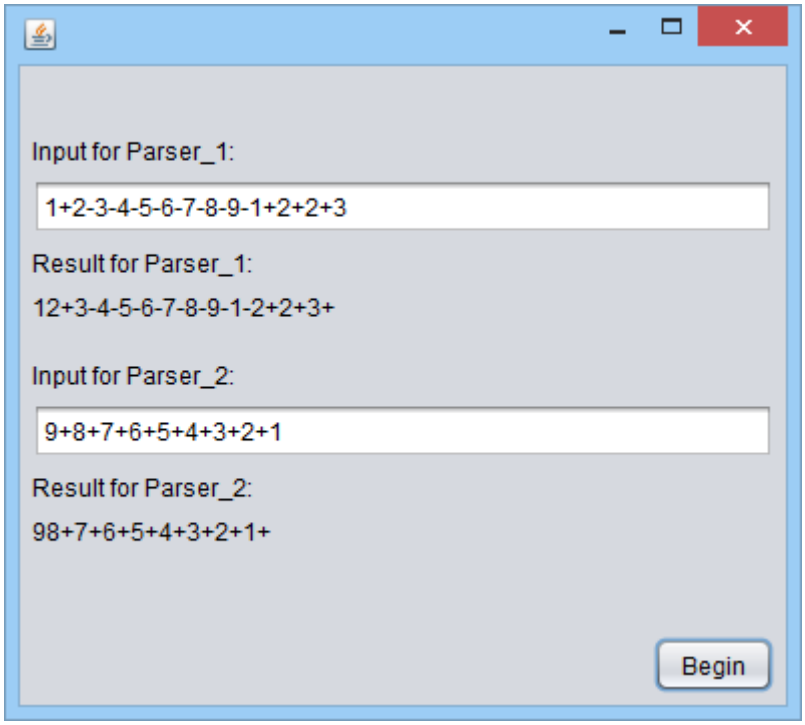
为了配合这个带图形界面程序的测试，Parser 的实现代码中做了少许修改，把输入改成了一个字符串，输出改成了向一个 JLable 添加字符。程序的测试结果如下：

- lookahead 未使用 static：（程序能正确运行）

```
public class Parser_Loop {

   int lookahead;
   JLabel output;
   String inputString;
   int index;

   public Parser_Loop(JTextField input, JLabel output) throws IOException {
      this.output = output;
      inputString = input.getText();
      inputString += "\0";
      index = 0;
      lookahead = inputString.charAt(index);
   }
```

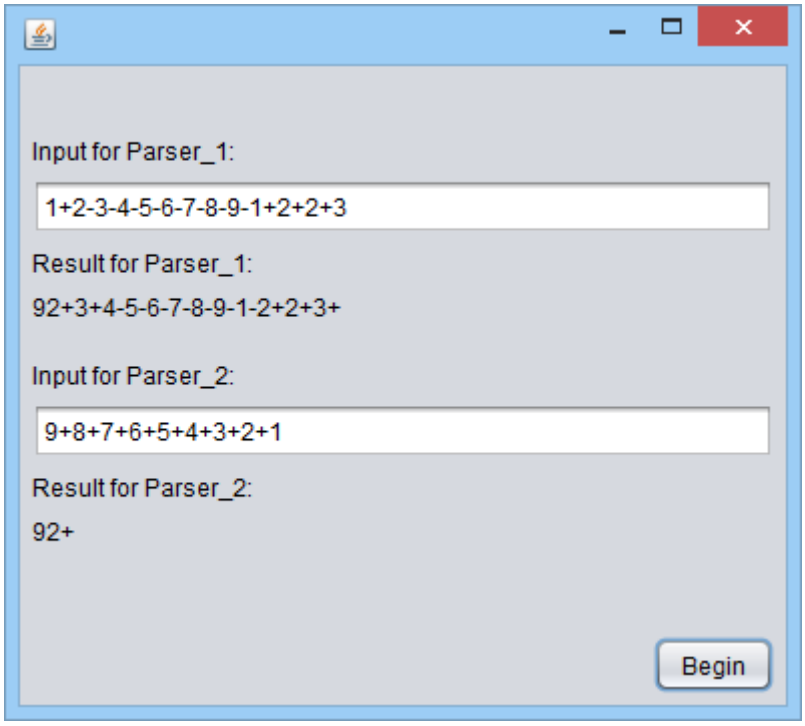


- lookahead 使用 static：（程序出现了错误）

```
public class Parser_Loop {

    static int lookahead;
    JLabel output;
    String inputString;
    int index;

    public Parser_Loop(JTextField input, JLabel output) throws IOException {
        this.output = output;
        inputString = input.getText();
        inputString += "\0";
        index = 0;
        lookahead = inputString.charAt(index);
    }
```



从上面的分析来看，我觉得还是使用非静态变量比较合理，因为每一个 Parser 都有一个自己的 lookahead 变量，而不是所有的 Parser 共用一个。但如果针对于本来的实验中只需要实例化一个 Parser，那使用静态的 lookahead 可能可以节约少许空间和时间。但使用多线程多个 Parser 一起运行的时候就会出现错误。

2. 比较消除尾递归前后程序的性能

尾递归的消除比较简单，只需要把出现尾递归的函数 rest()改成一个循环，循环的结束条件就是尾递归函数的结束条件，然后删除尾递归的那条语句，对于实验中的 Parser，消除尾递归后的函数如下图所示：

```
public void expr() throws IOException {
    term();
    while (true) {
        if (lookahead == '+') {
            match('+');
            term();
            System.out.write('+');
        } else if (lookahead == '-') {
            match('-');
            term();
            System.out.write('-');
        } else {
            break;
        }
    }
}
```

为了比较使用循环和尾递归对程序性能的影响，我使用了实验检测的方法。

首先为了能够产生大量的随机正确的中缀表达式，新建了一个 ProduceInfix 类，用来产生给定长度的中缀表达式，它的核心代码如下：

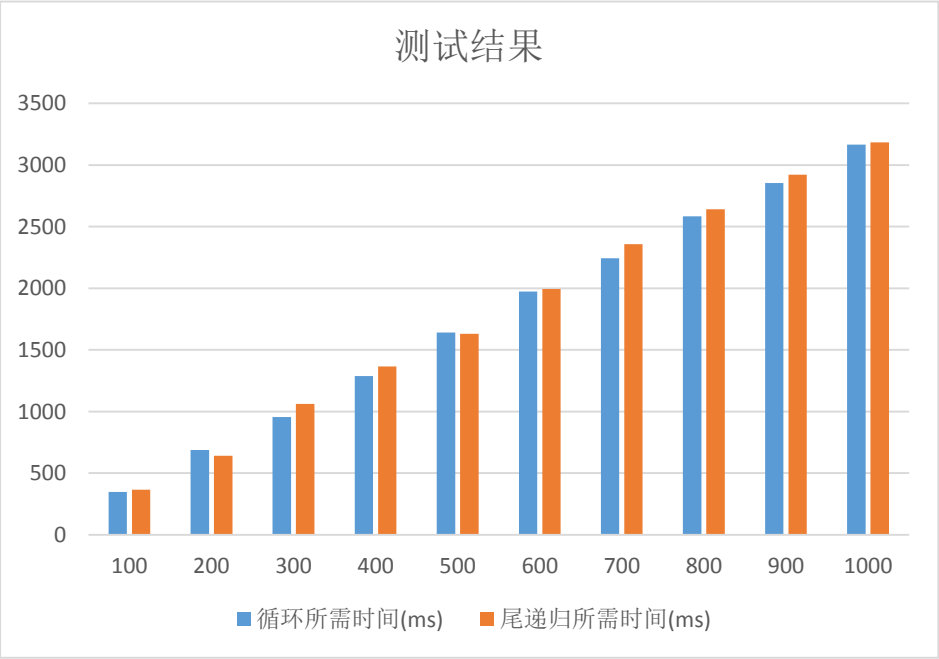
```
public static String getRandomInfix(int length) {
    if (length % 2 == 0) {
        throw new IllegalArgumentException("Length of infix must be an odd number!");
    }
    String infix = new String();
    for (int i = 0; i < length; i++) {
        if (i % 2 == 0) {
            infix += getRandomNum();
        } else {
            infix += getRandomOpt();
        }
    }
    return infix;
}
```

在比较这两个效率的类里边，使用了一些小技巧，例如使用输入流重定向到一个字符串中(System.setIn(InputStream))，使得对于程序而言，使用 System.in.read()实际上就是从给定的字符串中得到下一个字符，代码如下：

```
System.setIn(new ByteArrayInputStream(infixStrings.get(i).getBytes()));
new Parser_Recursive().expr();
```

总的测试总共进行了 10 次比较，每次测试都让循环和尾部递归的程序处理了长度为 1001 的字符串，第一次测试有 100 个测试样例，第二次有 200 个，以此类推，产生的结果如下：

测试用例数	循环所需时间(ms)	尾递归所需时间(ms)
100	347	367
200	689	641
300	957	1062
400	1287	1365
500	1642	1630
600	1975	1995
700	2245	2359
800	2585	2640
900	2853	2921
1000	3167	3184



从测试结果中来看，其实使用尾递归的时间和使用循环的时间相差不大，我个人觉得有以下几个原因：

a) 程序中主要的耗时都花费在输入输出的读写上面，程序控制（循环 or 递归）所需的时间相对于输入输出的用时可以忽略，而两个都进行了同样大量的输入输出操作，因此相差的时间不多。

b) 还有一种可能是 Java 在编译成中间代码的时候已经自动检测到尾递归并且实现了消除，也就是说其实一个尾递归的程序在运行的时候其实还是使用循环来进行执行的，因此两者的时间差别不大并且都是使用循环所需的时间。

总的来说，对于运行时间而言，使用尾部递归与循环的结果都一样，但如果对于使用的内存而言，尾部递归可能需要更多的存储空间，因为递归调用需要使用堆栈来存储当前状态，因此这个使用循环可以节省程序堆栈空间。

3. 扩展错误处理功能

我使用了一个 ExceptionManager 的类，并且把所有的错误都加到这个类的一个变量中，Parser 执行结束后从这个类中取出字符串，如果有错那就取出显示错误的字符串，如果没有错误就取出后缀表达式。程序的实现过程中，对 Parser 类进行了较多的修改，但是 Parser 本来的一些结构是基于循环的 Parser 类。

1. 指出错误类型：在 Parser 中识别字符串的过程中多个地方进行判断，如果应该是数字的位置出现了操作符，那就是缺少操作符的语法错误，如果应该是操作符的位置出现了数字，那就是缺少操作数的语法错误，如果处理过程中出现了不在字符集{0-9, +, -}中的字符，那就出现了词法错误。

2. 指出错误位置：我使用的方法是把匹配字符后输出的操作放到 ExceptionManager 中，如果没有错那就往存储后缀表达式的字符串中添加该字符，并往错误位置的字符串中添加一个空格，这样如果出错那往错误位置的字符串添加一个^字符，那就能够使用^指出错误的位置。

3. 错误恢复：如果使用 throws 抛出异常，如果没有相应的 catch 语句，程序会马上终止执行，而我使用的方式是出现的所有错误都使用一个字符串 String 记录，最后程序结束的时候(遇到'\0')再把所有的错误输出。

4. 扩展功能：通过一些处理，如果出现了错误的时候尝试恢复正确的后缀表达式，在运行结束的时候输出恢复后缀表达式的结果，也就是程序中的输出"Do you mean the expression ...."

5. 注意：要使用等宽字体，否则使用^指示的错误位置可能会不准确！

6. 运行结果：

   - 两个错误，缺少操作数和操作符结尾。

```
Input an infix expression and output its postfix notation:
1++2-2+4+
 ^       ^
It has below errors(from left to right):
Syntax Error '+' : Lack of operand!
Syntax Error ' ' : Lack of operand!
Do you mean the expression '12+2-4+1+' ?

End of program.
```

- 各种错误集合：

```
Input an infix expression and output its postfix notation:
1+2-334+a2-4+12+-b+4+12+2+3+c
     ^^ ^^    ^^ ^    ^     ^
It has below errors(from left to right):
Syntax Error '3' : Lack of operators!
Syntax Error '4' : Lack of operators!
Lexical Error 'a' : input character isn't in character set!
Syntax Error '2' : Lack of operators!
Syntax Error '2' : Lack of operators!
Syntax Error '-' : Lack of operand!
Lexical Error 'b' : input character isn't in character set!
Syntax Error '2' : Lack of operators!
Lexical Error 'c' : input character isn't in character set!
Do you mean the expression '12+3-1+4-1+1+4+1+2+3+1+' ?

End of program.
```

- 各种错误集合：

```
Input an infix expression and output its postfix notation:
1+2 +3-123 42+1-1+1+3-12
   ^    ^^^^^          ^
It has below errors(from left to right):
Lexical Error ' ' : input character isn't in character set!
Syntax Error '2' : Lack of operators!
Syntax Error '3' : Lack of operators!
Lexical Error ' ' : input character isn't in character set!
Syntax Error '4' : Lack of operators!
Syntax Error '2' : Lack of operators!
Syntax Error '2' : Lack of operators!
Do you mean the expression '12+3+1-1+1-1+3+1-' ?

End of program.
```

- 对空字符串的处理：

```
Input an infix expression and output its postfix notation:

^
It has below errors(from left to right):
Syntax Error: Empty expression!
Do you mean the expression '11+' ?

End of program.
```

4. 单元测试

在这个程序中对 Parser 类进行了单元测试。

- 测试用例的设计：首先测试对于输入的字符串比较输出的字符串是否为正确的后缀表达式，手动输入几个正确的没有错误的中缀字符串表达式，然后判断后缀表达式是否为预想中的结果。接下来手动输入各种错误的表达式作为 Parser 的输入，判断 ExceptionManager 的 isError 参数是否为 True。
- 测试结果：我使用了 Eclipse IDE，进行单元测试结果如下：

二、 实验心得

见文本文件<readme.txt>