# 编译原理实验一

后缀表达式Postfix

陈跃东

12330054

嵌入式软件

目录

[编译原理实验一 1](#_Toc439342766)

[一、 比较静态成员与非静态成员 2](#_Toc439342767)

[二、 比较消除尾递归前后程序的性能 2](#_Toc439342768)

[1. 实验基本思路 2](#_Toc439342769)

[2. 测试数据的获取 2](#_Toc439342770)

[3. 测试收集数据 2](#_Toc439342771)

[4. 数据的分析 2](#_Toc439342772)

[5. 数据分析结果展示 3](#_Toc439342773)

[6. 实验预期结果分析 4](#_Toc439342774)

[三、 扩展错误处理功能 4](#_Toc439342775)

[1. 划分错误类型并给出错误信息 4](#_Toc439342776)

[2. 错误的定位 4](#_Toc439342777)

[四、 单元测试 4](#_Toc439342778)

# 比较静态成员与非静态成员

实验发现，本程序中，声明为static和非static，对程序的正确性没有任何影响。

由于声明为static的变量属于类而非类的某个特定的实例的成员，所以一般情况下它有以下两个作用：

* 与final结合使用，用来作为类的常量
* 用来作为类实例的个数的计数，或者用来作为类实例的id

然而，本程序的lookahead显然不是用于实现以上两个作用。但本人还是认为lookahead声明为静态成员为好，这样有利于实现错误的定位，可以用来打印出出错的字符。

# 比较消除尾递归前后程序的性能

消除尾递归是一种很常用的编程技巧，本次实验中使用的方法是利用一个while循环来消除之前的尾递归。经过优化的编译器在实现尾递归时，其方法不是在调用栈上面添加一个新的堆栈，而是直接更新它。所以，采用显式的采用while循环来消除尾递归未必一定能提高程序执行性能。然而，遗憾的是，未必所有的编译器都会自动对尾递归进行优化。下面，就使用实验的方法来验证程序消除尾递归前后的性能差异。

## 实验基本思路

通过程序自动产生包含不同运算量的正确的表达式，作为消除尾递归前后的程序的输入。然后在同样的条件下分别执行两个程序，记录程序执行的时间。最后，分别做出两个程序的“运算量数量——执行时间”的折线图，对比分析，得出结论。

本实验中，生成数据到收集数据的过程，使用src/Compare.java来实现，它先随机生成包含1000个操作数的合法的中缀表达式，再分别将该表达式输入消除尾递归前后的两个Parser，记录开始时间和结束时间，并求出实际运行时间。完成之后，将运算量的数量增加1000，再重新执行上述步骤，直到运算量的数量为15000。得到的全部结果存储在/result.txt中。

## 测试数据的获取

本实验中使用随机数，自动生成包含1000到15000个运算量的表达式，间隔为1000，共15个输入。该过程由src/Compare.java中的CreateInfix类来完成，生成的运算式用一个String存储并返回。

## 测试收集数据

本实验中，会收集每个Parser开始执行前的时间点，以及执行结束后的时间点，然后相减求出实际运行时间。考虑到单次运行可能存在误差，所以本实验对于每个Parser都执行100次，所以这里记录的运行时间实际上是每个程序执行100次后求出的平均时间。另外还会收集输入的表达式中的运算量的个数，程序是否有递归这两个基本信息。

得到的所有数据保存在/result.txt中，每条记录以换行分隔，每条记录中的数据以空格分开，方便后续步骤对数据进行可视化分析与处理。

## 数据的分析

程序运行得到的结果如下表1所示：（*注：由于表达式是随机生成，所以重复执行，结果会存在差异*）

由表1可知，当操作数个数小于10000个时，有无尾递归的Parser的执行效率是比较接近的。但是，当操作数个数大于10000个时，含有尾递归的Parser就抛出栈溢出异常，可见JDK编译器并没有对尾递归进行优化。

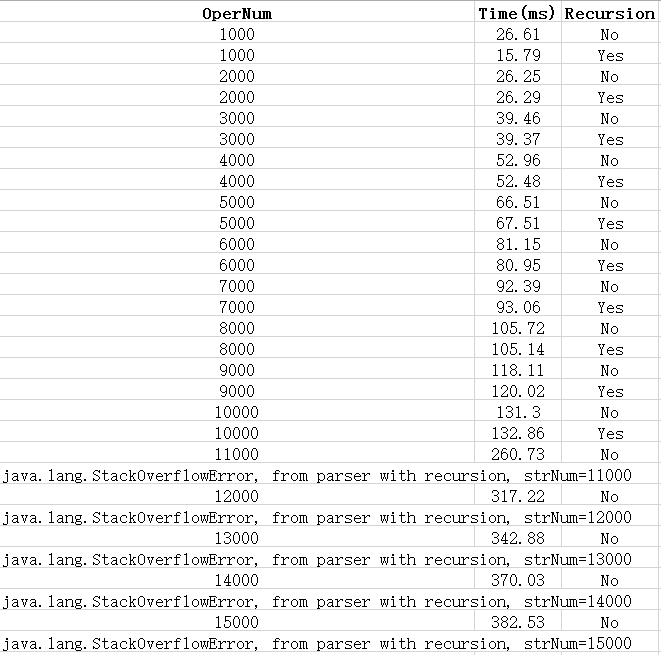


Table 1 有无尾递归的Parser的运行结果。其中OperNum指的是操作数的个数，Times指的是Parser执行100次的平均运行时间，Recursion中Yes代表有递归的Parser，而No则代表没有消除尾递归之后的Parser

## 数据分析结果展示

利用步骤4中得到的数据，使用python的matplotlib库，分别对有无递归的Parser做出“运算量个数——执行时间”的折线图，如下所示：

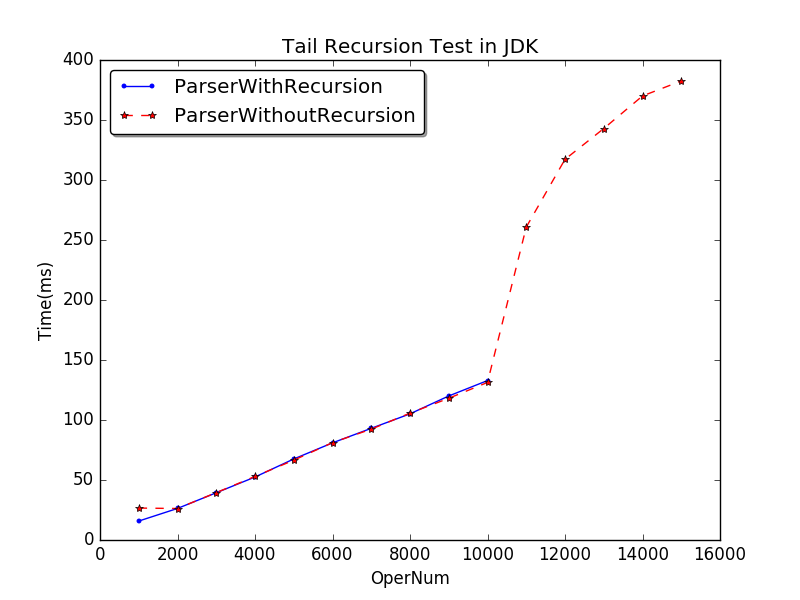


Figure Parser的“运算量个数——执行时间”折线图

注：作图的python程序是/plotResult.py，具体的运行方法是，首先，本地要配置有python环境，然后安装numpy以及matplotlib库，在项目的根路径下进入python命令行终端，然后，执行如下命令。

|  |
| --- |
| >>> import plotResult  >>> plotResult.plotPoint("result.txt") |

## 实验预期结果分析

结合第4步的数据以及第5步的可视化展示分析可知，当操作数个数规模不大时，对Parser执行消除尾递归操作之后，在性能上并没有明显的提升。但是，当操作数个数较多时（本人实验机器上，操作数个数大于10000个），没有消除尾递归的Parser便会抛出栈溢出异常，而消除尾递归之后的Parser则能够正常运行并获得结果。这也证实了Java编译器在编译代码过程中，并没有自动地对尾递归进行优化，所以，在写Java程序时，消除尾递归是很有必要的。

# 扩展错误处理功能

## 划分错误类型并给出错误信息

1. 在term()函数中发现的是语法错误(Syntax Error)，原因是缺乏右运算量。

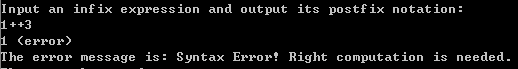


Figure 语法错误

1. 在rest()函数中发现的是词法错误(Lexical Error)，如果当时lookahead是数字，那么就说明出现了连续两个数字，出错的原因便是运算量只能在0~9中。

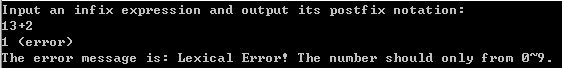


Figure 词法错误，运算量只能在0~9中

1. 而如果lookahead是其他符号，那就说明应该是运算符出错，出错的原因便是运算符只能是‘+’或‘-’，如下：

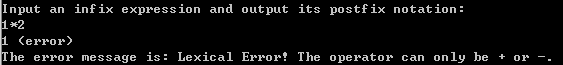


Figure 词法错误，运算符只能是‘+’或‘-’

## 错误的定位

在程序中增加一个静态成员变量，用来记录当前读入字符的位置，出错时便可以打印出出错的位置；然后打印出lookahead，便是出错的字符。

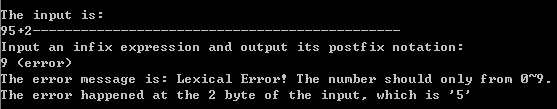


Figure 错误的定位

# 单元测试

本部分实验使用的是Eclipse内嵌的Junit模块，和标准JDK下的程序可能略有差异，但核心代码部分是一致的。参考链接:

<https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml>

*注：代码为/src/PostfixTest.java，必须使用Eclipse运行*

测试样例有5个，分别是：

* 两个正确的测试样例，当得出的后缀表达式与正确的相同时，assert正确
* 一个语法错误测试样例，当出现缺乏右运算量的语法错误异常，抛出”Syntax Error! Right computation is needed.”信息时，assert正确
* 一个词法错误测试样例，当出现运算量不是0~9的词法错误异常，抛出” Lexical Error! The number should only from 0~9.”信息时，assert正确
* 一个词法错误测试样例，当出现运算符不是+或-的词法错误异常，抛出” Lexical Error! The operator can only be + or -.”信息时，assert正确

测试程序/PostfixTest位于/src目录下，包含以上全部测试用例。运行结果是全部都正确通过测试，符合预期。测试结果如下图

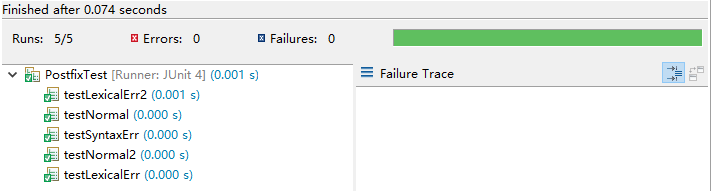


Figure 测试运行结果