编译原理实验一

后缀表达式 Postfix

目录

一、 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。

二、 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。

1. 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。

2. 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。

3. 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。

4. 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。

OperNum	Time(ms)	Recursion
1000	26.61	No
1000	15.79	Yes
2000	26.25	No
2000	26.29	Yes
3000	39.46	No
3000	39.37	Yes
4000	52.96	No
4000	52.48	Yes
5000	66.51	No
5000	67.51	Yes
6000	81.15	No
6000	80.95	Yes
7000	92.39	No
7000	93.06	Yes
8000	105.72	No
8000	105.14	Yes
9000	118.11	No
9000	120.02	Yes
10000	131.3	No
10000	132.86	Yes
11000	260.73	No
java.lang.StackOverflowError, from parser with recursion, strNum=11000		
12000	317.22	No
java.lang.StackOverflowError, from parser with recursion, strNum=12000		
13000	342.88	No
java.lang.StackOverflowError, from parser with recursion, strNum=13000		
14000	370.03	No
java.lang.StackOverflowError, from parser with recursion, strNum=14000		
15000	382.53	No
java.lang.StackOverflowError, from parser with recursion, strNum=15000		

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

5. 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：

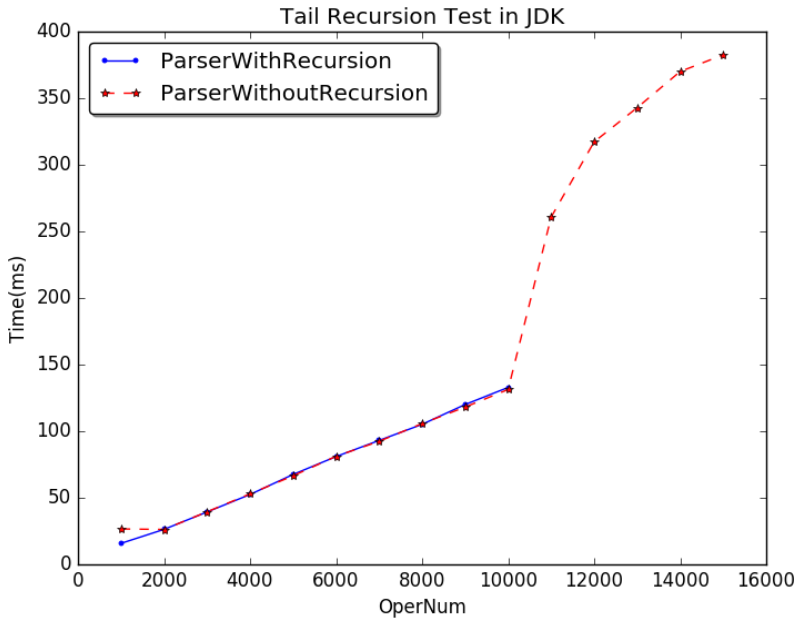

Figure 1 Parser 的“运算量个数——执行时间”折线图

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")
```

6. 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。

三、 扩展错误处理功能

1. 划分错误类型并给出错误信息

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

```
Input an infix expression and output its postfix notation:
1++3
1 (error)
The error message is: Syntax Error! Right computation is needed.
```

Figure 2 语法错误

b. 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。

```
Input an infix expression and output its postfix notation:
13+2
1 (error)
The error message is: Lexical Error! The number should only from 0~9.
```

Figure 3 词法错误，运算量只能在 0~9 中

c. 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：

```
Input an infix expression and output its postfix notation:
1*2
1 (error)
The error message is: Lexical Error! The operator can only be + or -.
```

Figure 4 词法错误，运算符只能是‘+’或‘-’

2. 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。

```
The input is:
95+2-------------------------------------------
Input an infix expression and output its postfix notation:
9 (error)
The error message is: Lexical Error! The number should only from 0~9.
The error happened at the 2 byte of the input, which is '5'
```

Figure 5 错误的定位

3. 错误恢复

本程序采用恐慌模式（Panic mode）来对错误进行恢复。具体的方法是：当遇到错误

时，记录错误的类型以及错误的位置信息，然后继续预读下一位，直到下一位符合相应的文法规则为止。

由于本实验在识别到错误时，还需要继续运行，所以，原有 Parser 使用 throw exception 的方法无法达到目的。因此，本部分实验由/src/PanicPostfix.java 来实现，通过运行脚本 panic.bat 可以编译并运行该程序。其中，错误信息使用一个 ArrayList 来保存，最后输出后缀表达式之后再打印全部的错误信息。

```
Input an infix expression and output its postfix notation:
++1--23+4$+5
12-4+5+
Syntax Error in column 1 => '+': Expression should be started with a digit.
Syntax Error in column 2 => '+': Expression should be started with a digit.
Syntax Error in column 5 => '-': Right operand is needed.
Lexical Error in column 7 => '3': Expression only supports Unit.
Lexical Error in column 10 => '$': The operator can only be + or -.

End of program.
```

Figure 6 恐慌模式错误恢复

四、 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

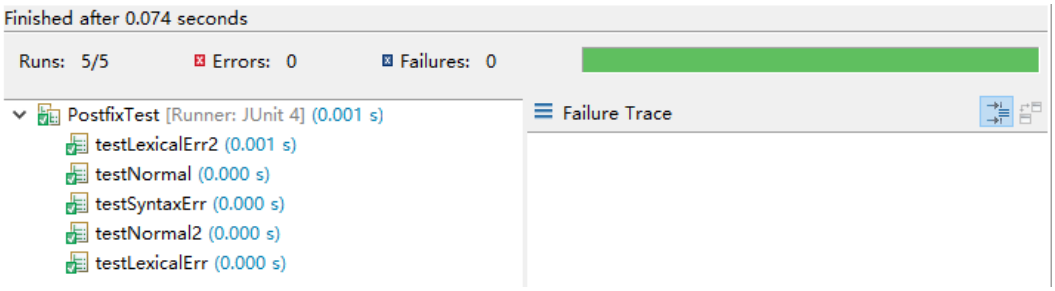

Figure 7 测试运行结果