

编译原理 project1 实验报告

13331024 陈炫峰 电政 1 班

1. 比较静态成员与非静态成员。

在只有一个 parser 实例运行时，将类 Parser 的成员 lookahead 声明为 static 或者非 static 对程序的正确性没有影响。

查阅资料知，在程序中任何变量或者代码都是在编译时由系统自动分配内存来存储的。在 java 类库当中有很多类成员都声明为 static，可以让用户不需要实例化对象就可以引用成员。被 static 修饰的成员，在编译时由内存分配一块内存空间，直到程序停止运行才会释放，那么就是说该类的所有对象都会共享这块内存空间。因为共享内存，该 static 变量以最后一次修改的值为准。

我的看法是声明为静态成员比较好，因为这样可以运行多个 parser 实例而且不影响程序的正确性，而且也可以在没有实例化对象的情况下直接引用静态成员。

2. 比较消除尾递归前后程序的性能。

如果一个过程体中执行的最后一条语句是对该过程的递归调用，那么这个调用就称为是尾递归的 (tail recursive)。在函数 rest() 中存在尾递归，如下图所示：

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

用 while 语句来消除尾递归，如下图所示：

```
void rest() throws IOException {  
    while (true) {  
        if (lookahead == '+') {  
            match('+');  
            term();  
            System.out.write('+');  
            continue;  
        } else if (lookahead == '-') {  
            match('-');  
            term();  
            System.out.write('-');  
            continue;  
        } else {  
            // do nothing with the input  
        }  
        break;  
    }  
}
```

比较性能的实验方案：

通过比较程序运行所需要的时间（纳秒）比较消除尾递归前后程序的性能。为了生成易于进行测试的表达式，我定义了函数 `generateExpression(int n)` 来产生长度为 `n` 的表达式，为了简洁起见，构造的表达式形如 `1+1+1+1+……`，长度为参数 `n`。

```
public static String generateExpression(int n) {
    char[] testExpr = new char[n];
    for (int i = 0; i < n; i += 2) {
        testExpr[i] = '1';
    }
    for (int i = 1; i < n; i += 2) {
        testExpr[i] = '+';
    }
    return new String(testExpr);
}
```

利用 `System.nanoTime()` 得到程序运行所需的时间，单位为纳秒(nanosecond)（代码中设置当读取完字符串时 `lookahead` 的值为 0）

```
public long expr() throws IOException {
    if (startTime == 0) {
        startTime = System.nanoTime();
    }

    if (lookahead == 0) {
        return System.nanoTime() - startTime;
    }

    term();
    rest();

    return System.nanoTime() - startTime;
}
```

为了方便测试和绘图，两种情况下分别选取了 20 个数据来进行分析，即字符串长度从 1 实验到 39。将每次得到的时间存储在对应的 ArrayList 中，最后用 get 方法将结果输出。

```
public static void main(String[] args) throws IOException {
    ArrayList tailRecur = new ArrayList();
    ArrayList noTailRecur = new ArrayList();
    // tailRecur.clear();
    // noTailRecur.clear();

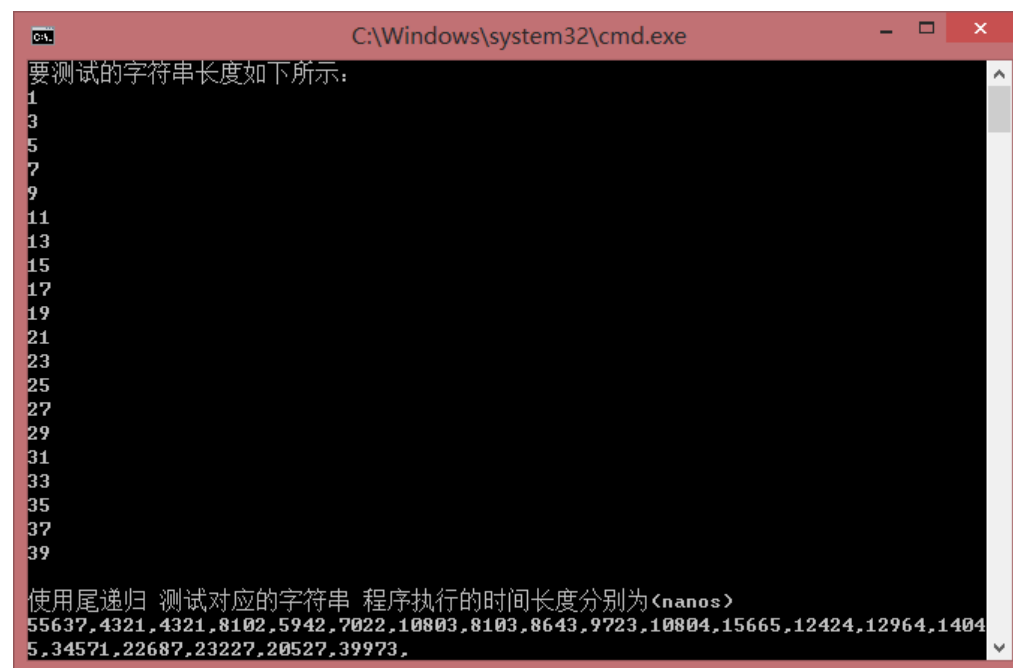
    String str;
    Long t_nano;

    System.out.println("要测试的字符串长度如下所示: ");
    for (int i = 1; i < 40; i += 2) {
        System.out.println(i);
        str = generateExpression(i);
        t_nano = new TailRecursionParser(str).expr();
        tailRecur.add(t_nano);
        t_nano = new NoTailRecursionParser(str).expr();
        noTailRecur.add(t_nano);
    }

    System.out.println("\n" + "使用尾递归 测试对应的字符串 程序执行的时间长度分别为(nanos)");
    for (int i = 0; i < tailRecur.size(); i++) {
        System.out.print((Long)tailRecur.get(i)+",");
    }

    System.out.println("\n" + "不使用尾递归 测试对应的字符串 程序执行的时间长度分别为(nanos)");
    for (int i = 0; i < noTailRecur.size(); i++) {
        System.out.print((Long)noTailRecur.get(i)+",");
    }
    System.out.println("");
}
```

运行结果如下：



```
C:\Windows\system32\cmd.exe
要测试的字符串长度如下所示:
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
33
35
37
39

使用尾递归 测试对应的字符串 程序执行的时间长度分别为(nanos)
55637.4321,4321,8102,5942,7022,10803,8103,8643,9723,10804,15665,12424,12964,1404
5,34571,22687,23227,20527,39973,
```

```

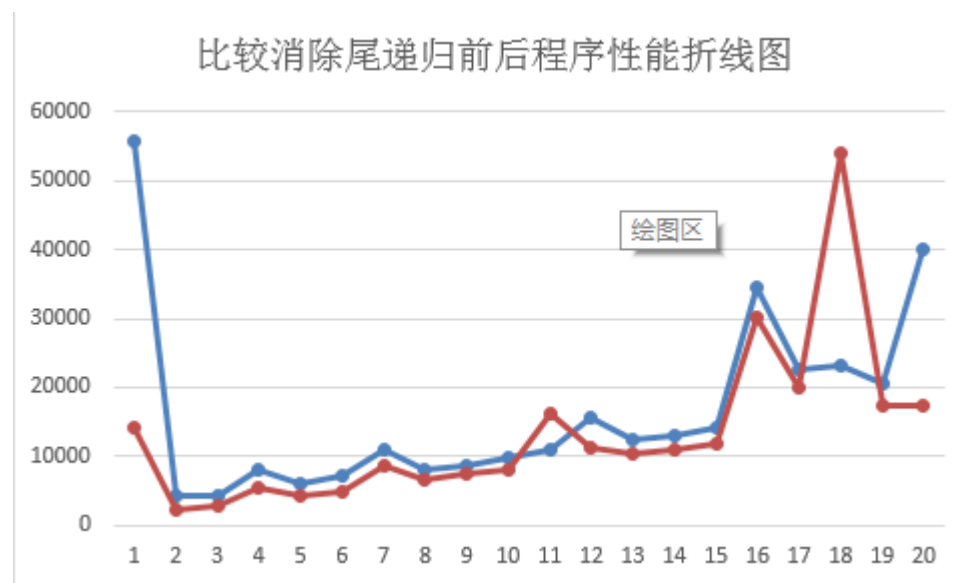
不使用尾递归 测试对应的字符串 程序执行的时间长度分别为<nanos>
14045,2160,2701,5401,4321,4862,8643,6482,7562,8103,16206,11344,10263,10804,11883
,30250,19987,54018,17286,17286,
请按任意键继续. . .

```

将数据填充进 excel 中，A 列表示使用尾递归。B 列表示不使用尾递归。



绘制出改进前后两个程序各自性能的折线图，蓝色折线代表使用尾递归，红色折线代表不使用尾递归。



由折线图可以看出，排除一定范围的误差，消除尾递归之后的程序运行所需的时间更短，程序的性能更高。

查阅资料得知：编译器会自动把尾递归转换成循环，以提高程序的性能。但在 Java 语言规范中，并没有要求一定要作这种转换，因此，并不是所有的 Java 虚拟机（JVM）都会做这种转换。这就意味着在 Java 语言中采用尾递归方法将导致巨大的内存占用，而且大量的递归调用有导致堆栈溢出的危险。所以我们在编程时应该避免写出含有尾递归的程序。

3. 扩展错误处理功能

划分错误类型：

缺少运算符、缺少左运算量、缺少右运算量为语法错误。

非法的输入（即出现除了数字 0-9，加号，减号以外的字符）为词法错误。

在 term() 函数中，只能和数字 0-9 匹配，其它都为错误，需要进行捕获，捕获的错误类型分为缺少右运算分量、缺少左运算分量、输入的字符非法。

```
} else if (lookahead == '\n') {
    /**
     * 错误：运算符缺失右运算分量
     * eg:9+
     */
    errorMsg.append("错误：第" + count + "个字符\n" + "语法错误：缺少右运算分量\n");
    System.out.print("<!运算符缺少右运算分量> ");
    goAhead();
    break;
} else if (lookahead == '+' || lookahead == '-') {
    /**
     * 错误：运算符缺少左运算分量
     * eg:+9
     * 此处不需要break，因为还需要再继续调用term()函数
     */
    errorMsg.append("错误：第" + count + "个字符\n" + "语法错误：缺少左运算分量\n");
    System.out.print("<!运算符缺少左运算分量> ");
    goAhead();
    // System.out.print("lookahead:" + (char)lookahead+"\n");
} else if (count == 1) {
    /**
     * 错误：表达式首项为非法的运算量
     * eg:a+1+2
     */
    errorMsg.append("错误：第" + count + "个字符\n" + "词法错误：表达式首项应为数字\n");
    System.out.print("<!非法的运算量> ");
    goAhead();
    break;
}
```

在 rest() 函数中，出现除了+和-以外的字符为错误，需要进行捕获，捕获的错误类型分为运算量之间缺少运算符、输入的字符非法。

```
} else if (Character.isDigit((char)lookahead)) {
    /**
     * 错误：运算量间缺少运算符
     * eg:89+3
     */
    errorMsg.append("错误：第" + count + "个字符\n" + "语法错误：运算量间缺少运算符\n");
    System.out.print((char)lookahead + "<!运算量间缺少运算符> ");
    goAhead();
    continue;
} else if (Character.isAlphabetic(lookahead)) {
    /**
     * 错误：非法的输入
     * eg:1+a-2
     */
    errorMsg.append("错误：第" + count + "个字符\n" + "词法错误：非法的输入\n");
    System.out.print("<!非法的运算量> ");
    goAhead();
    continue;
}
```

实现错误定位：

定义一个 count 变量，每读入一个字符就加一，当捕获到错误时，记录 count 的值，将其输出。

实现出错恢复

在捕获了错误之后，调用 goAhead() 函数，使得向前看符号前进一个字符，继续解析其他字符直至结束。

运行测试用例的屏幕截图：

正常解析：

```
Running Testcase 001
=====
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+
=====
```

缺少右运算分量：

```
Running Testcase 002
=====
The input is:
9+
-----
Input an infix expression and output its postfix notation:
9 <!运算符缺少右运算分量> +
错误：第3个字符
语法错误：缺少右运算分量
```

缺少左运算分量：

```
Running Testcase 003
=====
The input is:
+9
-----
Input an infix expression and output its postfix notation:
<!运算符缺少左运算分量>
错误：第1个字符
语法错误：缺少左运算分量
```

词法错误:

```
Running Testcase 004:
=====
The input is:
a+1+2
-----
Input an infix expression and output its postfix notation:
<?!非法的运算量> 1+2+
错误: 第1个字符
词法错误: 表达式首项应为数字
```

运算量间缺少运算符:

```
Running Testcase 005
=====
The input is:
89+3
-----
Input an infix expression and output its postfix notation:
89 <?!运算量间缺少运算符> 3+
错误: 第2个字符
语法错误: 运算量间缺少运算符
```

联合测试:

```
Input an infix expression and output its postfix notation:
1+2-a+89
12+ <?!非法的运算量> -8+9 <?!运算量间缺少运算符>
错误: 第5个字符
词法错误: 非法的输入
错误: 第8个字符
语法错误: 运算量间缺少运算符
```

```
Input an infix expression and output its postfix notation:
+1+a+89-
<?!运算符缺少左运算分量> 1 <?!非法的运算量> +8+9 <?!运算量间缺少运算符> <?!运算符
缺少右运算分量> -
错误: 第1个字符
语法错误: 缺少左运算分量
错误: 第4个字符
词法错误: 非法的输入
错误: 第7个字符
语法错误: 运算量间缺少运算符
错误: 第9个字符
语法错误: 缺少右运算分量
```

```
Input an infix expression and output its postfix notation:
b+s+12-4+-3
<?!非法的运算量> <?!非法的运算量> +1+2 <?!运算量间缺少运算符> 4- <?!运算符缺少左运
算分量> 3+
错误: 第1个字符
词法错误: 表达式首项应为数字
错误: 第3个字符
词法错误: 非法的输入
错误: 第6个字符
语法错误: 运算量间缺少运算符
错误: 第10个字符
语法错误: 缺少左运算分量
```

4. JUnit 单元测试

由于不熟悉 windows 下 JUnit 测试，所以这部分是在 linux 下测试的。修改比较性能时使用的 NoTailRecursionParser 文件，新增了 str 变量，用来存储输出的字符串。

单元测试类如下，通过 InputStreamReader 和 BufferedReader 读取 testcases 文件夹里的测试文件，将.infix 作为 parse 的参数读入，返回正确转换后的后缀表达式或者“error”字符串，再读取对应的.postfix 文件内容进行 assert 断言。

测试样例为两个正确的输入和三个错误的输入，对于正确的输入，parse 会返回转换后的后缀表达式，对于错误的输入，parse 会返回 “error”。

```
public class parserTest {
    @Test
    public void parserTest() throws FileNotFoundException, IOException {
        FileInputStream fis1, fis2;
        String fileName;
        String fileNameBase = "testcases/tc-00";
        String input, ans;
        NoTailRecursionParser parse;

        for (int i = 1; i <= 5; i++) {
            fileName = fileNameBase + i + ".infix";
            fis1 = new FileInputStream(fileName);
            fileName = fileNameBase + i + ".postfix";
            fis2 = new FileInputStream(fileName);

            InputStreamReader isr = new InputStreamReader(fis1);
            BufferedReader br = new BufferedReader(isr);
            input = br.readLine();

            isr = new InputStreamReader(fis2);
            br = new BufferedReader(isr);
            ans = br.readLine();

            parse = new NoTailRecursionParser(input);
            parse.expr();
            String str = parse.getStr();
            //System.out.println("\nans: "+ans);
            //System.out.println("str: "+str);
            assertEquals(ans, str);
        }
    }
}
```

测试结果:

```
conan@Conan:~/program/my project1/Postfix/JUnitTest$ sh build.sh
conan@Conan:~/program/my project1/Postfix/JUnitTest$ sh run.sh
JUnit version 4.9
.
Time: 0.007

OK (1 test)

conan@Conan:~/program/my project1/Postfix/JUnitTest$
```