

编译原理实验

后缀表达式 *Postfix*

实验报告

目 录

1 静态变量与非静态.....	1
2 消除尾递归.....	1
3 错误处理.....	4

1. 静态变量与非静态变量

1 静态变量与实例变量的区别：

类的静态变量在内存中只有一个，java 虚拟机在加载类的过程中为静态变量分配内存，静态变量位于方法区，被类的所有实例共享。静态变量可以直接通过类名进行访问，其生命周期取决于类的生命周期。

而实例变量取决于类的实例。每创建一个实例，java 虚拟机就会为实例变量分配一次内存，实例变量位于堆区中，其生命周期取决于实例的生命周期。

2 思考使用静态变量的原因

经过实验发现，使用静态变量和实例变量对此程序运行结果没有任何影响。

使用静态变量，可以使 lookahead 共享给所有 Parser 类。

猜想，有可能为了方便扩充一些静态函数时，调用 lookahead。

2. 消除尾递归

1. 消除尾递归后的代码：

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

2. 速度比较：

利用 Testcase5.java 生成一个较长的 infix 文件。

```
public class Testcase5 {
    private static final int TIMES = 100;
    private static final String INFIXPATH = "../testcases/tc-005.infix";
    private static final String POSTFIXPATH = "../testcases/tc-005.postfix";
    public static void main(String[] args) throws IOException {
        int a = TIMES;

        try {
            FileOutputStream fos = new FileOutputStream(INFIXPATH);
            DataOutputStream dos = new DataOutputStream(fos);
            dos.writeBytes("0");
            while((a-- > 0) {
                for(int i = 0; i < 10; i++) {
                    if(i%2==0)
                        dos.writeBytes("-");
                    else
                        dos.writeBytes("+");
                    String s = "" + i;
                    dos.writeBytes(s);
                }
            }
            dos.writeBytes("\n");
            dos.close();
            fos.close();
            fos = new FileOutputStream(POSTFIXPATH);
            dos = new DataOutputStream(fos);
            dos.writeBytes("0");
            a = TIMES;
            while((a-- > 0) {
                for(int i = 0; i < 10; i++) {
                    String s = "" + i;
                    dos.writeBytes(s);
                    if(i%2==0)
                        dos.writeBytes("-");
                    else
                        dos.writeBytes("+");
                }
            }
            dos.writeBytes("\n");
            dos.close();
            fos.close();
        }
    }
}
```

tc-005.infix

[illegible]

在原函数中加入计时:

```
System.out.println("Input an infix expression and output its postfix notation:");
long startTime=System.currentTimeMillis();
new Parser().expr();
long endTime=System.currentTimeMillis();
System.out.println("\nEnd of program.");
float excTime=(float)(endTime-startTime);
System.out.println("Using time:" + excTime + "msec.");
```

双击 testcase-005.bat:

```

@echo off
rem : Produce tc-005.infix and tc-005.postfix
cd bin
java Testcase5
cd ..

@echo Running Testcase 005: long testcase in Tail recursion
@echo =====
@echo The input is:
type testcases\tc-005.infix
@echo -----
cd bin

rem : Run the testcase with input direction
java Postfix2 < ..\testcases\tc-005.infix

rem : Compare the expected output
@echo -----
@echo The output should be:
type ..\testcases\tc-005.postfix

cd ..
@echo =====
pause

@echo Running Testcase 005: long testcase in cycling
@echo =====
@echo The input is:
type testcases\tc-005.infix
@echo -----
cd bin

rem : Run the testcase with input direction
java Postfix < ..\testcases\tc-005.infix

rem : Compare the expected output
@echo -----
@echo The output should be:
type ..\testcases\tc-005.postfix

cd ..

```

尾递归:

```

End of program.
Using time:26.0msec.

```

循环:

```

End of program.
Using time:25.0msec.

```

经比较发现，尾递归与循环基本上在时间复杂度上没有差别。

3. 原因分析：

经过思考不难发现，尾递归比起一般的递归，没有回溯这个过程，栈空间的使用优化了很多。而且他在函数末尾调用自身，相当于每次递归只是对参数的更新，与循环一样。所以尾递归与循环基本上拥有相同的时间复杂度。

3. 错误检测

1. 错误类型检测设计思路

错误类型检测其实很简单，因为可以简单划分成两种：在符号处错误和在数字处错误，而符号处理的函数是 `rest()`，数字处理函数是 `term()`。分别在对应的函数检测对应的错误即可。符号和数字在各自的地方又有两种错误，缺失和错误字符，只要检测他的下一个字符是否为应该对应的字符就可知道：

```
void term() throws IOException {
    if (Character.isDigit((char)lookahead)) {
        if(errorNum == 0) {
            output = output + (char)lookahead;
        }
        if(wrongState == false)
            errorPos = errorPos+(char)' ';
        else
            wrongState = false;
        match(lookahead);
    } else if(lookahead == '+' || lookahead == '-') {
        errorNum++;
        errorPos = errorPos + (char)'^';
        errorList.add("Miss a digit!!");
        wrongState = true;
    } else {
        if(wrongState == false) {
            errorNum++;
            errorPos = errorPos + (char)'^';
            errorList.add("No this digit!!");
            match(lookahead);
        }
    }
}
```

```

void rest() throws IOException {
    while(true) {
        if (lookahead == '+') {
            if(wrongState == false)
                errorPos = errorPos+(char)' ';
            else
                wrongState = false;
            match('+');
            term();
            if(errorNum == 0) {
                output = output + (char)'+';
            }
        } else if (lookahead == '-') {
            if(wrongState == false)
                errorPos = errorPos+(char)' ';
            else
                wrongState = false;
            match('-');
            term();
            if(errorNum == 0) {
                output = output + (char) '-';
            }
        } else if(lookahead == 13){
            break;
        } else {
            errorNum++;
            if(Character.isDigit((char)lookahead)) {
                errorList.add("Miss a operator!!");
                errorPos = errorPos + (char)'^';
                wrongState = true;
            } else {
                errorList.add("Operator don't exist!!");
                errorPos = errorPos + (char)'^';
                match(lookahead);
            }
            term();
        }
    }
}
}

```

2.错误位置检测设计思路

首先要输出一个错误的位置必须得到完整的输入串。于是在开头和 match 处读入字符

时将其放入 input 队列。

```

/**
 * record the input.
 */
private String input;

```

```

void match(int t) throws IOException {
    if (lookahead == t) {
        lookahead = System.in.read();
        input = input + (char)lookahead;
    } else {
        throw new Error("syntax error");
    }
}

```

利用 errorPos 队列来记录错误的位置。利用 errorList 记录错误位置对应的错误。

```
/**
 * record the position of the error.
 */
private String errorPos;
```


```
/**
 * record the error type.
 */
private ArrayList<String> errorList;
```

具体操作为：

- 1.当 term 或 rest 检测到正确的字符时，往 errorPos 末尾加入空格“ ”；当检测到错误是往 errorPos 末尾加入上箭头“ ^”，并把对应的错误类型放入 errorList 队列，错误书 errorNum++。
- 2.到最后输出错误时，从 i=0 到 errorNum-1 遍历输出 input,然后在下一行输出 errorPos 的第 i 个“ ^”，以及输出 errorList 中的第 i 错误类型。

```
void printPostfix() {
    System.out.println(output+"\n");
    for(int i = 0; i < errorNum; i++) {
        System.out.println("\nThere is an error:\"\" + errorList.get(i) + "\" at:");
        System.out.println(input);
        int n = i;
        for(int j = 0; j < errorPos.length(); j++) {
            if(errorPos.charAt(j) != '^') {
                System.out.write(errorPos.charAt(j));
            } else {
                if(n == 0) {
                    System.out.write(errorPos.charAt(j));
                    break;
                } else {
                    System.out.write(' ');
                    n--;
                }
            }
        }
        System.out.println("\n");
    }
}
```

3.测试用例

双击  testcase-006.bat

```
@echo off
@echo Running Testcase 006: a wrong expression
@echo =====
@echo The input is:
type testcases\tc-006.infix
@echo -----
cd bin

rem : Run the testcase with input direction
java Postfix < ..\testcases\tc-006.infix

rem : Compare the expected output
@echo -----
@echo The output should be:
type ..\testcases\tc-006.postfix

cd ..
@echo =====
pause
```

其中 tc-006.infix 为:

```
1+2+34+5+6+-7+a+9*0+1
```

运行结果:

```
Running Testcase 006: a wrong expression
=====
The input is:
1+2+34+5+6+-7+a+9*0+1
-----
Input an infix expression and output its postfix notation:
12+3+

There is an error:"Miss a operator!!" at:
1+2+34+5+6+-7+a+9*0+1
      ^

There is an error:"Miss a digit!!" at:
1+2+34+5+6+-7+a+9*0+1
              ^

There is an error:"No this digit!!" at:
1+2+34+5+6+-7+a+9*0+1
                  ^

There is an error:"Operator don't exist!!" at:
1+2+34+5+6+-7+a+9*0+1
                  ^

End of program.
Using time:17.0msec.
-----
The output should be:
12+3+(error)
=====
请按任意键继续. . .
```