lab3实验报告

——22336073傅小桐

实现效果

输入正确的中缀表达式，可得到相应的后缀表达式：

```
PS C:\Users\lenovo\Desktop\LabInstrument03-Postfix> ./run.bat
Input an infix expression and output its postfix notation:
1+2+3-5+6
----------------------------------------
转换为后缀表达式的结果为：12+3+5-6+
----------------------------------------
End of program.
请按任意键继续. . .
```

输入错误的中缀表达式，可检测出错误并进行报错，并找到错误发生的位置以及输出可能的后缀表达式：

```
PS C:\Users\lenovo\Desktop\LabInstrument03-Postfix> ./run.bat
Input an infix expression and output its postfix notation:
11+2
----------------------------------------
11+2
^^
语法错误，这两个运算量间缺少运算符，已自动忽略第二个运算量
----------------------------------------
转换为后缀表达式的结果为：12+
----------------------------------------
End of program.
```

```
PS C:\Users\lenovo\Desktop\LabInstrument03-Postfix> ./run.bat
Input an infix expression and output its postfix notation:
1--2+3
----------------------------------------
1--2+3
  ^
语法错误，缺少左运算量，已自动忽略此字符
----------------------------------------
转换为后缀表达式的结果为：12-3+
----------------------------------------
End of program.
```

运行testcase.bat，程序正确地运行了这几个例子：

```
PS C:\Users\lenovo\Desktop\LabInstrument03-Postfix> ./testcase.bat
Running Testcase 001: a correct input from DBv2.
==========================================
The input is:
9-5+2
------------------------------------------
Input an infix expression and output its postfix notation:
------------------------------------------
转换为后缀表达式的结果为：95-2+
------------------------------------------
End of program.
```

```
C:\Users\lenovo\Desktop\LabInstrument03-Postfix>call testcase-002.bat
Running Testcase 002: a correct long input.
==========================================
The input is:
1-2+3-4+5-6+7-8+9-0
------------------------------------------
Input an infix expression and output its postfix notation:
------------------------------------------
转换为后缀表达式的结果为：12-3+4-5+6-7+8-9+0-
------------------------------------------
End of program.
```

```
C:\Users\lenovo\Desktop\LabInstrument03-Postfix>call testcase-003.bat
Running Testcase 003: missing an operator.
==========================================
The input is:
95+2
------------------------------------------
Input an infix expression and output its postfix notation:
------------------------------------------
95+2
^^
语法错误，这两个运算量间缺少运算符，已自动忽略第二个运算量
------------------------------------------
转换为后缀表达式的结果为：92+
------------------------------------------
End of program.
```

```
C:\Users\lenovo\Desktop\LabInstrument03-Postfix>call testcase-004.bat
Running Testcase 004: missing an operand.
==========================================
The input is:
9-5+-2
------------------------------------------
Input an infix expression and output its postfix notation:
------------------------------------------
9-5+-2
    ^
语法错误，缺少左运算量，已自动忽略此字符
------------------------------------------
转换为后缀表达式的结果为：95-2+
------------------------------------------
End of program.
```

代码讲解

本程序通过 `Parser` 类实现中缀表达式转后缀表达式的逻辑，主要功能分为以下几个部分：

表达式解析主流程（`parseExpression()`）

```
public void parseExpression() throws IOException {
    parseTerm();   // 解析第一个操作数
    parseRest();   // 处理后续的运算符 + 操作数
    System.out.print("转换为后缀表达式的结果为：");
    for (char c : output) {
        System.out.print(c);
    }
}
```

- **功能**：负责整个表达式的解析和输出；
- **结构**：调用 `parseTerm()` 读取第一个数字，再用 `parseRest()` 处理后续结构。

处理“项”（操作数）：`parseTerm()`

```
private void parseTerm() throws IOException {
    while (!isDigit(currentChar)) {
        reportError("语法错误，缺少左运算量，已自动忽略此字符", currentIndex);
        advance();
        if (isEnd()) return;
    }
    output.add((char) currentChar); // 添加数字到输出
    advance();
}
```

- **功能**：检测并添加一个合法的数字；
- **错误处理**：跳过非数字字符并报警。

去尾递归的表达式剩余部分：`parseRest()`

```
private void parseRest() throws IOException {
    while (!isEnd()) {
        if (isDigit(currentChar)) {
            reportError("语法错误，这两个运算量间缺少运算符，已自动忽略第二个运算量",
currentIndex - 1, currentIndex);
            advance();
        } else if (currentChar == '+' || currentChar == '-') {
            int op = currentChar;
            advance();
            parseTerm();          // 递归调用 parseTerm，添加操作数
            output.add((char) op); // 操作符放在操作数之后（后缀表达式）
        } else if (currentChar == ' ') {
            reportError("词法错误，此处不应该有空格，已自动忽略", currentIndex);
            advance();
        } else {
```

```
            reportError("词法错误，非法运算符，已自动忽略，只支持+与-", currentIndex);
            advance();
        }
    }
}
```

- **功能**：解析运算符和后续操作数；
- **尾递归已消除**：用 `while` 循环代替递归调用；
- **错误处理**：
  - 连续数字 → 缺少运算符；
  - 非法符号 / 空格 → 词法错误；
  - 错误不终止程序，具备一定的**错误恢复能力**。

报错位置标示（如 `reportError()`、`printPointer()`）

```
private void reportError(String message, int index) {
    printInput();          // 打印原始表达式
    printPointer(index);   // 用 ^ 标记出错位置
    System.out.println(message);
    System.out.println("----------------------------------------");
}
```

- **功能**：打印输入、错误位置（用 `^` 或 `^^`）和详细的错误说明；
- **目的**：帮助用户定位语法或词法错误。

小结

- **中缀转后缀核心算法**：通过 `parseTerm` + `parseRest` 分别处理数字和运算符，运算符延后加入，构成后缀表达式；
- **尾递归优化**：将 `parseRest()` 由递归改为循环，避免函数栈积压；
- **错误处理机制**：详细提示错误类型与位置，程序不中断，增强用户体验。

解答问题

Step 1：静态成员与非静态成员

如果类中的字段 `lookahead` 被定义为静态（`static`），说明这个字段是类级别的，**所有的 Parser 实例共享同一个 lookahead 值。**
这样做通常是为了便于将 `lookahead` 用作一个**全局状态变量**，比如多个方法调用中都能访问和修改该变量，而不依赖于具体的对象状态。

如果将 `lookahead` 改为非静态成员（即删除 `static` 关键字），它就变成了对象级别的属性，每个 `Parser` 实例拥有自己的 `lookahead`。
这在本程序中是可行的，只要相关方法都在同一个实例下运行，不会影响程序正确性。
但若存在多个对象或静态方法访问该变量，则必须使用静态成员。

Step 2：消除程序中的尾递归

原始的 `rest()` 方法如果是递归调用的（如：`rest()` 在方法最后调用自身），则属于尾递归。尾递归在每次调用时只在栈上保留当前一层的调用信息，不涉及后续处理逻辑，因此可以安全地转换为 `while` 循环，避免频繁压栈出栈，提高效率。

`parseRest()` 方法已经使用了 `while (!isEnd())` **的循环结构**，将原本的尾递归逻辑转换成了迭代形式，**成功消除了尾递归。**

例如，若原来是这样：

```
void rest() {
    if (...) {
        ...;
        rest();  // 尾递归调用
    }
}
```

现在变成：

```
void rest() {
    while (...) {
        ...;
    }
}
```

这就是典型的尾递归消除，提升了性能和可维护性。

Step 3：为程序扩展错误处理功能

程序已实现了一套较完善的错误处理机制，主要包括：

1. **空格检测**：
    - 通过判断 `currentChar == ' '` 检测非法空格；
    - 报错提示为“词法错误，此处不应该有空格”。
2. **连续数字未加运算符**：
    - 识别两个操作数之间缺少操作符的语法错误；
    - 报错提示为“两运算量间缺少运算符”。
3. **非法字符处理**：
    - 非 `+`、`-`、数字的字符会触发“非法运算符”错误；
    - 同样归类为词法错误。
4. **操作符缺少操作数**：
    - 在遇到操作符后，`parseTerm()` 会检查后续是否为合法数字；
    - 若不是，报错“缺少右运算量”或“缺少左运算量”。
5. **错误位置标注**：
    - 调用 `reportError()` 打印错误信息；
    - 使用 `^` 或 `^^` 指出具体出错字符位置；

    - 提升了用户对错误的感知能力。
6. **错误恢复（Error Recovery）**：
    - 报错后程序并不终止；
    - 通过 `advance()` 跳过当前错误字符，继续分析后续表达式；
    - 属于一种基本的错误恢复策略。

Step 4、为程序增加文档化注释

详情请见doc文件夹