实验报告

1. 静态成员与非静态成员比较

1. 在实验中，将 `Parser` 类的 `lookahead` 成员从静态改为非静态后，程序的正确性未受影响。

   修改前：

```
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:
12-3+4-5+6-7+8-9+0-
End of program.
------------------------------------------------
The output should be:
12-3+4-5+6-7+8-9+0-
================================================
请按任意键继续. . .
```

   修改后：

```
class Parser {
    int lookahead;
```

```
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:
12-3+4-5+6-7+8-9+0-
End of program.
------------------------------------------------
The output should be:
12-3+4-5+6-7+8-9+0-
================================================
请按任意键继续. . .
```

2. 声明`lookahead`成员变量是静态的，这样`lookahead`这一成员变量在这个对象的所有实例当中就都是共享的，此时不同实例对`lookahead`的修改就都是同步的。在明确变量不应该被共享的情况下将变量设置为静态，可能会导致多个实例同时修改、同时读取的错误。由于java中类的静态变量不需要初始化一个实例也可以直接访问，所以这样设置可能是为了方便？但是修改前后正确性实际没有被影响，原因在于原代码仅创建了一个 `Parser` 实例，所有操作均在该实例内完成。
3. 静态成员属于类级别，所有实例共享，若存在多个实例会导致状态混乱。而非静态成员为实例级别，每个实例独立管理状态，避免潜在的并发问题。虽然静态变量可以在不创建实例的前提下方便访问，但是本次实验的实现都是基于创建对象实例的，所以本次实验中`lookahead`声明为非静态成员更合理，确保程序的封装性和线程安全等。

2. 消除尾递归的性能比较（可选）

尾递归消除后，通过循环替代递归调用，减少了方法调用栈的开销。理论上，循环效率更高。实验方案如下：

- **实验准备**：

- 源代码的尾递归实现`RecursiveParser.java`和循环实现`LoopParser.java`，其中由于最终代码还实现了错误纠正、可能影响性能，故这里的测试比较用代码`LoopParser.java`只修改了循环实现，同时将源代码`RecursiveParser.java`修改为单个类实现、以避免封装解封装带来的额外开销
- 测试类`Benchmark.java`，主要用于以上两种实现的比较、同时统计并输出两种实现对不同长度表达式的计算时间
- 测试数据生成类`DataGenerator.java`，用于生成不同长度的随机表达式

- **测试数据**：生成超长随机表达式，例如`1+2-3+...`循环拼接
  - 选择`100, 1_000, 10_000, 50_000, 100_000, 500_000, 1_000_000`作为测试长度序列
  - 随机生成的表达式，表达式生成`.txt`文件存储在同一目录下

    `DataGenerator.java`：

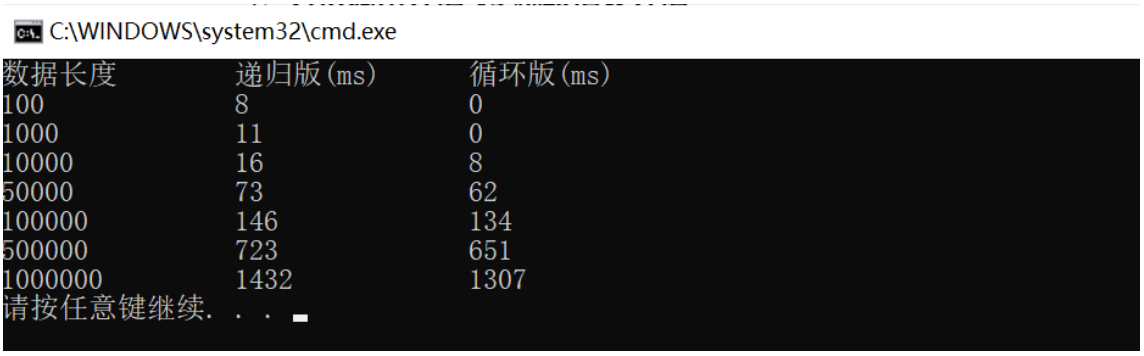
```java
import java.io.*;
import java.util.Random;

public class DataGenerator {
    public static void main(String[] args) throws IOException {
        int[] lengths = {100, 1_000, 10_000, 50_000, 100_000, 500_000, 1_000_000}; // 测试数据长度
        for (int len : lengths) {
            generateExpression(len);
        }
    }

    private static void generateExpression(int length) throws IOException {
        try (FileWriter fw = new FileWriter("test_"+length+".txt")) {
            Random rand = new Random();
            boolean needDigit = true;

            for (int i = 0; i < length; ) {
                if (needDigit) {
                    fw.write(rand.nextInt(10) + '0');
                    i++;
                    needDigit = false;
                } else {
                    fw.write(rand.nextBoolean() ? '+' : '-');
                    i++;
                    needDigit = true;
                }
            }
            fw.write(rand.nextInt(10) + '0');
        }
    }
}
```

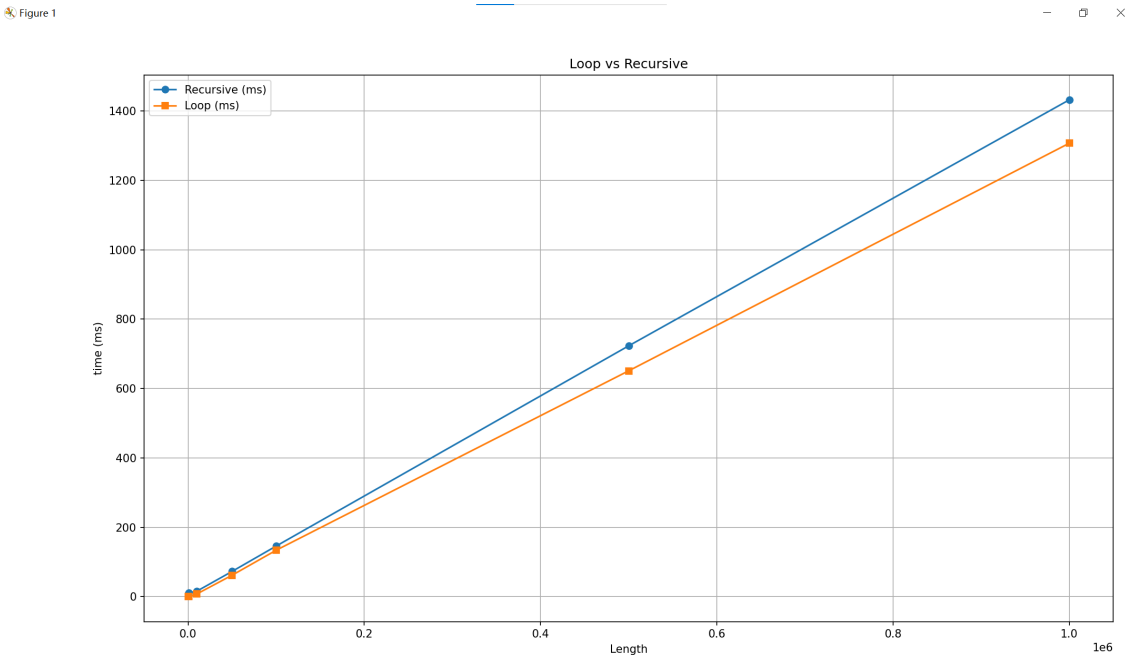
- **进行测试**：

1. 运行`Data_Generate.bat`生成随机表达式，表达式生成`.txt`文件存储在同一目录下
2. 运行`run_cpmpare.bat`读取生成的表达式`.txt`文件进行测试

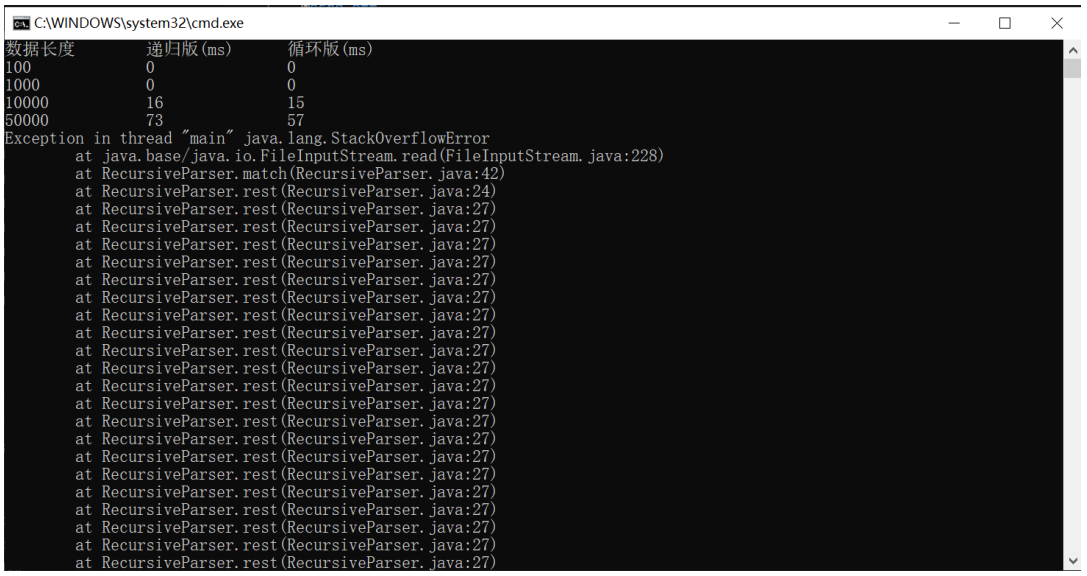
3. 得到运行测试数据，其中主要需要的数据应该为不同输入表达式对应的不同版本的运行时间，方便比较不同情况下两者计算性能的区别。输出如下图：

- **实验数据**：





- **实验结论**：
  - 观察实验输出数据，可以观察得到循环实现相比尾递归实现有一些时间效率提升，两者都随着表达式规模增大呈线性增长，但循环版本比递归版本增长速率小一些。这可能是因为在 Java 中，每次递归调用都需要 JVM 为该方法创建新的栈帧，保存局部变量、返回地址等信息；而循环只是在同一栈帧内通过条件判断和跳转指令`goto`来重复执行，省去了大量的入栈和出栈开销。
  - 同时过长的递归、如测试1000000长度的输入串就很容易导致栈溢出，因为每一次递归都在调用栈上分配空间；而循环版本只占用固定的常量空间，不会因数据规模增大而导致栈溢出。实际上由于频繁发生栈溢出，实验中不得不使用参数将`JVM`默认1m栈大小更改为16倍大小，如下：




```
java -cp bin -Xss16m Benchmark
```

- 可以得到结论：在本实验中递归改为循环在时间性能上会带来部分提升，同时在空间性能上能带来较大的优化

3. 扩展错误处理功能

核心类：

将错误整合为一个类`ParseError`，面相错误对象进行处理，可以给出核心类和核心方法，并以此实现各种错误相关功能：

```java
/**
 * 表示解析过程中出现的语法错误，记录错误位置、描述和类型信息。
 * <p>
 * 本类用于在语法分析过程中收集错误信息，支持多错误记录机制。</p>
 */
public class ParseError {
    /**
     * 错误发生的位置1-based索引，面向用户可读的位置编号
     */
    public final int position;

    /**
     * 可读的错误描述信息
     */
    public final String message;

    /**
     * 错误分类标识
     */
    public final String errorType;

    /**
     * 构造新的错误记录对象
     * @param pos 错误在输入字符串中的位置1-based
     * @param msg 具体错误描述
```

```
     * @param type 错误类型标识
     * @throws IllegalArgumentException 当位置值小于1时抛出
     */
    public ParseError(int pos, String msg, String type) {
        if (pos < 1) throw new IllegalArgumentException("Position must be 1-based");
        this.position = pos;
        this.message = Objects.requireNonNull(msg);
        this.errorType = Objects.requireNonNull(type);
    }

    /**
     * 生成标准化的错误信息字符串
     * @return 格式化字符串，包含位置、类型和描述信息
     */
    @Override
    public String toString() {
        return String.format("Error at position %d: [%s] %s",
                             position, errorType, message);
    }
}
```

错误分类：

错误分类可以分为非严重错误、严重错误；非严重错误直接跳过处理即可，严重错误又可以分为词法错误和语法错误；语法错误又可以分为操作数后接操作数、运算符后接运算符两种情况。分别进行分类处理：

- 非严重错误：空白错误，不需要冻结输出
  - 空白错误：空格、换行符等。只需要跳过，不需要冻结输出。
- 严重错误：词法错误和语法错误，需要冻结输出
  - 词法错误：非法字符，如 `a`、`~`等非数字、非加减运算符的非法字符。需要冻结输出。
  - 语法错误：表达式结构错误，等违反加减运算语法规则的非法表达式。需要冻结输出。
    - 操作数后接操作数：如`11+2`
    - 运算符后接运算符：如`1++2`
    - 运算符后结束：如`1+`
    - 其余情况均认为词法错误

1. `STATE`、`parse`

   `STATE`枚举类型标识两个状态，`EXPECT_OPERAND`表示当前应出现操作数（数字），`EXPECT_OPERATOR`表示当前应出现运算符（+或-），便于后续`parse`创建有限状态机：

```
    private enum State {
        /** 当前应出现操作数（数字） */
        EXPECT_OPERAND,
        /** 当前应出现运算符（+或-） */
        EXPECT_OPERATOR
    }
```

`parse`是错误处理和表达式遍历的核心方法。

- 当状态为`EXPECT_OPERAND`调用`handleOperandState`处理操作数
- 当状态为`EXPECT_OPERATOR`调用`handleOperatorState`处理运算符
- 每次状态切换都调用`skipWhitespace`以跳过空格，同时检测非严重错误
- 每处理完一个状态，切换到另一个状态，直到表达式结尾应该为操作数，否则产生**严重错误-语法错误-运算符后结束**类型的错误。调用`recordError`记录：对应错误信息、设置`isSyntaxError`为true、设置`isFreeze`为true

```
/**
* 执行完整的解析过程
* <p>
* 解析流程：
* <ol>
*   <li>初始化状态机</li>
*   <li>循环处理每个字符</li>
*   <li>根据当前状态处理输入</li>
*   <li>错误发生时冻结输出</li>
* </ol>
* 注意：此方法可能抛出运行时异常
*/
public void parse() {
    while (lookahead < input.length()) {
        skipWhitespace();
        if (lookahead >= input.length()) break;

        final char ch = input.charAt(lookahead);
        pos = lookahead + 1; // 更新用户可见位置

        if (state == State.EXPECT_OPERAND) {
            handleOperandState(ch);
        } else {
            handleOperatorState(ch);
        }
    }

    // 处理输入结束后的悬挂运算符
    if (state == State.EXPECT_OPERAND && pendingOperator != 0) {
        recordError("Missing operand after operator '" +
pendingOperator + "'", true, true);
    }
}
```

2. `skipWhitespace`、`handleOperandState`、`handleOperatorState`：

`skipWhitespace`处理空格。

- 若当前输入字符为空格，产生**非严重错误**类型的错误。调用`recordError`记录：空格的错误信息、设置`isSyntaxError`为true、设置`isFreeze`为false。最后直接让当前字符索引`lookahead`+1，跳过当前错误字符

```java
/**
* 跳过连续空白字符并记录错误
* <p>
* 每跳过一个空白字符记录一个非严重错误</p>
*/
private void skipWhitespace() {
    while (lookahead < input.length() &&
Character.isWhitespace(input.charAt(lookahead))) {
        pos = lookahead + 1; // 更新用户可见位置
        lookahead++;
        recordError("Whitespace detected", false, false); // 记录非严重错
误
    }
}
```

`handleOperandState`处理操作数。

- 若当前输入字符为操作数，不产生错误，直接调用后缀表达式处理方法
- 若当前输入字符为操作数，产生**严重错误-语法错误-操作数后接操作数**类型的错误。调用`recordError`记录：对应错误信息、设置`isSyntaxError`为true、设置`isFreeze`为true。最后直接让当前字符索引`lookahead`+1，跳过当前错误字符。（特别地，初始状态为`EXPECT_OPERAND`，即要求表达式第一个符号为操作数，否则也产生上述错误）
- 若当前输入字符为其他字符，产生**严重错误-词法错误**类型的错误。调用`recordError`记录：对应错误信息、设置`isSyntaxError`为false、设置`isFreeze`为true。最后直接让当前字符索引`lookahead`+1，跳过当前错误字符

```java
/**
* 处理期待操作数的状态
* @param ch 当前字符
*/
private void handleOperandState(char ch) {
    if (Character.isDigit(ch)) {
        processDigit(ch);
    } else if (isOperator(ch)) {
        recordError("Missing digit before operator '" + ch + "'", true,
true);
        lookahead++;
    } else {
        recordError("Unexpected character '" + ch + "'", false, true);
        lookahead++;
    }
}
```

`handleOperatorState`处理运算符。

- 若当前输入字符为运算符，不产生错误，直接调用后缀表达式处理方法
- 若当前输入字符为运算符，产生**严重错误-语法错误-运算符后接运算符**类型的错误。调用`recordError`记录：对应错误信息、设置`isSyntaxError`为true、设置`isFreeze`为true。最后直接让当前字符索引`lookahead`+1，跳过当前错误字符

- 若当前输入字符为其他字符，产生**严重错误-词法错误**类型的错误。调用`recordError`记录：对应错误信息、设置`isSyntaxError`为false、设置`isFreeze`为true。最后直接让当前字符索引`lookahead`+1，跳过当前错误字符

```java
/**
* 处理期待运算符的状态
* @param ch 当前字符
*/
private void handleOperatorState(char ch) {
    if (isOperator(ch)) {
        processOperator(ch);
    } else if (Character.isDigit(ch)) {
        recordError("Missing operator before digit '" + ch + "'", true, true);
        lookahead++;
    } else {
        recordError("Unexpected character '" + ch + "'", false, true);
        lookahead++;
    }
}
```

3. `recordError`

`recordError`是与类`ParseError`的主要接口，将分类后的错误信息传递给`ParseError`，并创造一个实例加入`errors`列表以记录所有错误信息

- 参数`msg`记录错误描述信息
- 参数`isSyntaxError`标识是否属于语法错误
- 参数`isFreeze`标识是否触发输出冻结

```java
/**
* 记录错误并更新冻结状态
* @param msg 错误描述信息
* @param isSyntaxError 是否属于语法错误
* @param isFreeze 是否触发输出冻结
* @throws NullPointerException 当msg为null时抛出
*/
private void recordError(String msg, boolean isSyntaxError, boolean isFreeze) {
    if (isSyntaxError) {
        errors.add(new ParseError(pos, msg, "Syntax Error"));
    }
    else{
        errors.add(new ParseError(pos, msg, "Lexical Error"));
    }
    if (isFreeze) {
        freezeOutput = true;
    }
}
```

错误定位：

错误定位主要通过字符索引`lookahead`扫描并计算错误位置：

1. `Parser`成员变量中给出`lookahead`、`pos`

   `position = lookahead + 1`记录当前扫描到的位置，而`lookahead`在上面错误分类具体实现中已经给出、即在扫描表达式过程中不断自增，实现字符索引定位

```java
/**
* 当前字符索引0-based，用于内部处理
*/
private int lookahead = 0;

/**
* 当前字符位置1-based，用于错误报告
*/
private int pos = 0;
```

2. `ParseError`类成员变量`position`记录错误的位置

```java
/**
* 构造新的错误记录对象
* @param pos 错误在输入字符串中的位置1-based
* @param msg 具体错误描述
* @param type 错误类型标识
* @throws IllegalArgumentException 当位置值小于1时抛出
*/
public ParseError(int pos, String msg, String type) {
    if (pos < 1) throw new IllegalArgumentException("Position must be 1-based");
    this.position = pos;
    this.message = Objects.requireNonNull(msg);
    this.errorType = Objects.requireNonNull(type);
}
```

3. `toString`生成标准化的错误信息字符串

   程序在错误发生时终止输出表达式、但继续扫描，以收集所有错误的位置，同时输出信息包含错误定位信息`"Error at position x"`

```java
/**
* 生成标准化的错误信息字符串
* @return 格式化字符串，包含位置、类型和描述信息
*/
@Override
public String toString() {
    return String.format("Error at position %d: [%s] %s",
                         position, errorType, message);
}
```

出错恢复：

解析器的错误恢复策略：

1. `skipWhitespace`检测到空格后，跳过该字符继续解析。
2. `processDigit`、`processOperator`、`flushPendingOperator`后缀表达式解析核心方法

   一旦检测到严重错误，`isFreeze`置`true`、`freezeOutput`置`true`，后续后缀表达式解析只记录错误，不再修改`fullOutput`，防止生成错误的后缀表达式

```
/**
* 处理数字字符
* @param ch 当前数字字符
*/
private void processDigit(char ch) {
    if (!freezeOutput) {
        fullOutput.append(ch);
    }
    state = State.EXPECT_OPERATOR;
    flushPendingOperator();
    lookahead++;
}

/**
* 处理运算符字符
* @param ch 当前运算符
*/
private void processOperator(char ch) {
    pendingOperator = ch;
    state = State.EXPECT_OPERAND;
    lookahead++;
}

/**
* 输出缓存的运算符
*/
private void flushPendingOperator() {
    if (pendingOperator != 0 && !freezeOutput) {
        fullOutput.append(pendingOperator);
        pendingOperator = 0;
    }
}
```

3. `recordError`创建`ParseError`类实例，将其加入`errors`列表

   对任何错误，程序把发现的错误加入`ParseError`类实例的`errors`列表，不终止程序并继续执行扫描，最后`printColoredErrors`集中输出表达式中的所有错误。

```
/**
* 收集到的错误列表
*/
private final List<ParseError> errors = new ArrayList<>();
```

```java
/**
* 获取解析过程中发现的所有错误
* @return 不可修改的错误列表
*/
public List<ParseError> getErrors() {
    return Collections.unmodifiableList(errors);
}

/**
* @param errors 错误列表
*/
private static void printColoredErrors(List<ParseError> errors) {
    System.err.println("Parsing errors were found:");
    for (ParseError err : errors) {
        System.err.println(err);
    }
}
```

正确示例：

运行`testcase-001.bat`：

```
Running Testcase 001: a correct input from DBv2.
================================================
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+
================================================
请按任意键继续. . .
```

运行`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:
12-3+4-5+6-7+8-9+0-

End of program.
------------------------------------------------
The output should be:
12-3+4-5+6-7+8-9+0-
================================================
请按任意键继续. . .
```

错误示例：

严重错误-语法错误-操作数后接操作数：

运行`testcase-003.bat`：

如图，错误定位在第2个字符，类型为语法错误，错误信息为`Missing operator before digit '5'`

```
Running Testcase 003: missing an operator.
================================================
The input is:
95+2
------------------------------------------------
Input an infix expression and output its postfix notation:
9 (error)
Parsing errors were found:
Error at position 2: [Syntax Error] Missing operator before digit '5'

End of program.
------------------------------------------------
The output should be:
9 (error)
================================================
请按任意键继续. . .
```

严重错误-语法错误-运算符后接运算符:

运行`testcase-004.bat`:

如图，错误定位在第5个字符，类型为语法错误，错误信息为`Missing digit before operator '-'`

```
Running Testcase 004: missing an operand.
================================================
The input is:
9-5+-2
------------------------------------------------
Input an infix expression and output its postfix notation:
95- (error)
Parsing errors were found:
Error at position 5: [Syntax Error] Missing digit before operator '-'

End of program.
------------------------------------------------
The output should be:
95- (error)
================================================
请按任意键继续. . .
```

非严重错误:

输入`9 -5+2`:

如图，错误定位在第2个字符，类型为词法错误，错误信息为`Whitespace detected`

```
Input an infix expression and output its postfix notation:
9 +5-2
95+2- (error)
Parsing errors were found:
Error at position 2: [Lexical Error] Whitespace detected

End of program.
请按任意键继续. . .
```

严重错误-词法错误:

输入`9+5-2a`:

如图，错误定位在第6个字符，类型为词法错误，错误信息为`Unexpected character 'a'`

```
Input an infix expression and output its postfix notation:
9+5-2a
95+2- (error)
Parsing errors were found:
Error at position 6: [Lexical Error] Unexpected character 'a'

End of program.
请按任意键继续. . .
```

严重错误-语法错误-运算符后结束：

输入`9+5-`：

如图，错误定位在第4个字符，类型为语法错误，错误信息为`Missing operand after operator '-'`

```
Input an infix expression and output its postfix notation:
9+5-
95+ (error)
Parsing errors were found:
Error at position 4: [Syntax Error] Missing operand after operator '-'

End of program.
请按任意键继续. . .
```

所有错误类型(展示出错恢复)：

输入`9 +55+-a`：

如图：

1. 扫描`9`、`_`，此时检测为**非严重错误**，直接跳过。错误定位在第2个字符，类型为词法错误，错误信息为`Whitespace detected`
2. 扫描`+`、`5`，此时后缀表达式出栈输出为`95+`
3. 扫描`5`，此时检测为**严重错误-语法错误-操作数后接操作数**，冻结输出。错误定位在第5个字符，类型为语法错误，错误信息为`Missing operator before digit '5'`
4. 扫描`+`、`-`，此时检测为**严重错误-语法错误-运算符后接运算符**。错误定位在第7个字符，类型为语法错误，错误信息为`Missing digit before operator '-'`
5. 扫描`a`，此时检测为**严重错误-词法错误**。错误定位在第8个字符，类型为词法错误，错误信息为`Unexpected character 'a'`
6. 扫描串尾结束符，此时检测为**严重错误-语法错误-运算符后结束**。错误定位在第8个字符，类型为语法错误，错误信息为`Missing operand after operator '+'`

```
Input an infix expression and output its postfix notation:
9 +55+-a
95+ (error)
Parsing errors were found:
Error at position 2: [Lexical Error] Whitespace detected
Error at position 5: [Syntax Error] Missing operator before digit '5'
Error at position 7: [Syntax Error] Missing digit before operator '-'
Error at position 8: [Lexical Error] Unexpected character 'a'
Error at position 8: [Syntax Error] Missing operand after operator '+'

End of program.
请按任意键继续. . .
```

4. 程序的单元测试

测试方法：

由于其他单元实现的功能都比较简单，故只考虑对`Parser`类进行测试。而`Parser`核心功能就是实现后缀表达式的转化和错误检测，故考虑使用回归测试对这两个功能进行检测。

`ParserTest.java`采用JUnit编写了若干回归测试，测试内容与上述正确示例和错误示例中类似：

```
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.List;

public class ParserTest {

    @Test
    public void testValidSingleDigit() {
        Parser parser = new Parser("1");
        parser.parse();
        assertEquals("1", parser.getOutput());
        assertTrue(parser.getErrors().isEmpty());
    }

    @Test
    public void testValidExpression() {
        Parser parser = new Parser("9-5+2");
        parser.parse();
        assertEquals("95-2+", parser.getOutput());
        assertTrue(parser.getErrors().isEmpty());
    }

    @Test
    public void testMissingOperand() {
        Parser parser = new Parser("1+");
        parser.parse();
        List<ParseError> errors = parser.getErrors();
        assertEquals(1, errors.size());
        assertEquals("Missing operand after operator '+'",
errors.get(0).message);
    }

    @Test
    public void testUnexpectedOperator() {
        Parser parser = new Parser("+12");
        parser.parse();
        List<ParseError> errors = parser.getErrors();
        assertTrue(errors.size() >= 1);
        assertEquals("Missing digit before operator '+'",
errors.get(0).message);
    }

    @Test
    public void testMultipleErrors() {
        Parser parser = new Parser("1++2-");
        parser.parse();
```

```
        List<ParseError> errors = parser.getErrors();
        assertEquals(2, errors.size());
        assertEquals("Missing digit before operator '+'",
errors.get(0).message);
        assertEquals("Missing operand after operator '-'",
errors.get(1).message);
    }

    @Test
    public void testWhitespaceHandling() {
        Parser parser = new Parser("1 + 2 ");
        parser.parse();
        assertEquals("12+", parser.getOutput());
        assertEquals(3, parser.getErrors().size()); // 检测到3个空格
    }
}
```

进行测试:

`test.bat`将JUnit库链接到编译测试类，同时链接JUnit和测试类可执行文件进行回归测试：

```
@echo off
setlocal enabledelayedexpansion

REM 清理旧的编译文件
rmdir /s /q bin 2>nul
mkdir bin
rmdir /s /q test-bin 2>nul
mkdir test-bin

REM 编译主代码
javac -encoding UTF-8 -d bin src\*.java
if %errorlevel% neq 0 (
    echo 主代码编译失败
    exit /b %errorlevel%
)

REM 编译测试代码
javac -encoding UTF-8 -d test-bin -cp "bin;lib\junit-
4.13.2.jar;lib\hamcrest-core-1.3.jar" test\ParserTest.java
if %errorlevel% neq 0 (
    echo 测试代码编译失败
    exit /b %errorlevel%
)

REM 运行单元测试
java -cp "test-bin;bin;lib\junit-4.13.2.jar;lib\hamcrest-core-1.3.jar"
org.junit.runner.JUnitCore ParserTest

pause
endlocal
```

运行`test.bat`进行回归测试，发生错误时给出具体错误信息，正确时输出如下：


JUnit version 4.13.2
.....
Time: 0.016
OK (6 tests)
请按任意键继续. . .