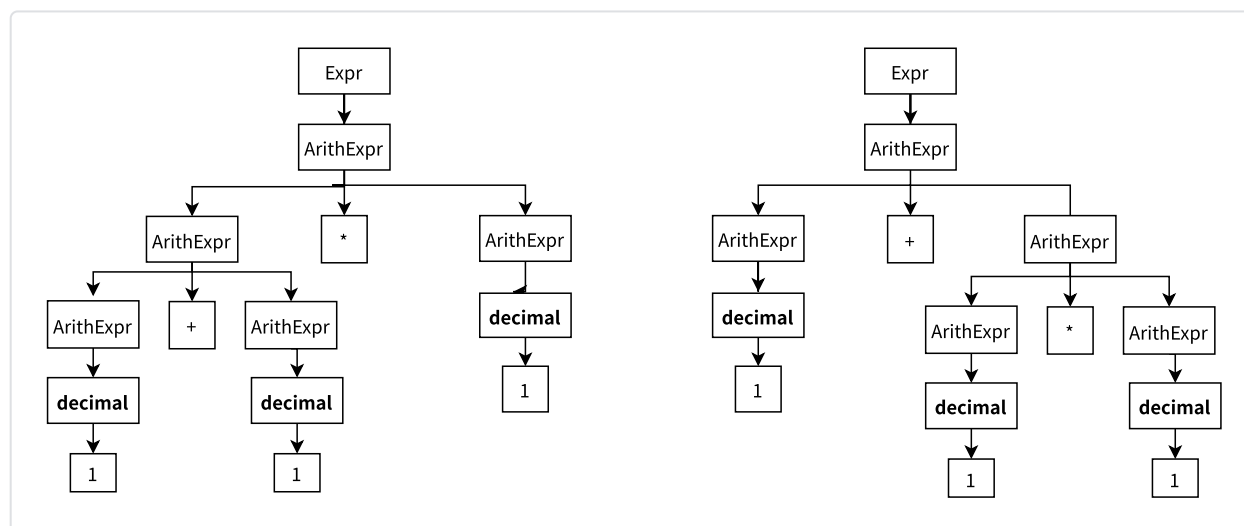


design

1 讨论语法定义的二义性

```
Expr      → ArithExpr
ArithExpr  → decimal | ( ArithExpr )
           | ArithExpr + ArithExpr | ArithExpr - ArithExpr
           | ArithExpr * ArithExpr | ArithExpr / ArithExpr
           | ArithExpr ^ ArithExpr
           | - ArithExpr
           | BoolExpr ? ArithExpr : ArithExpr
           | UnaryFunc | VariablFunc
```

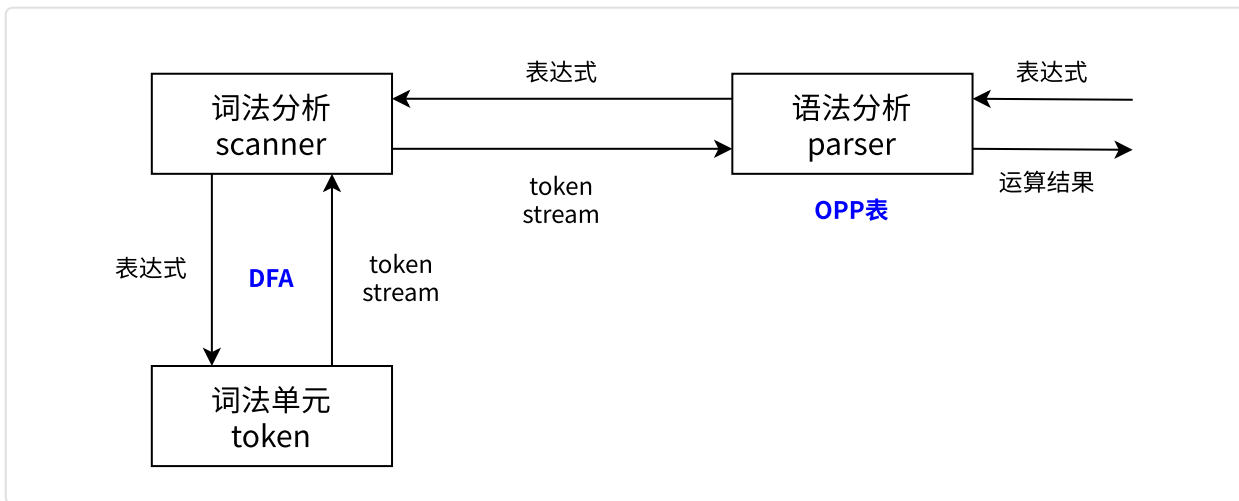
该实验中的语法定义具有二义性，以 `1+1*1` 为例，可以构造出两棵不同的语法树。



2 架构设计

设计思路

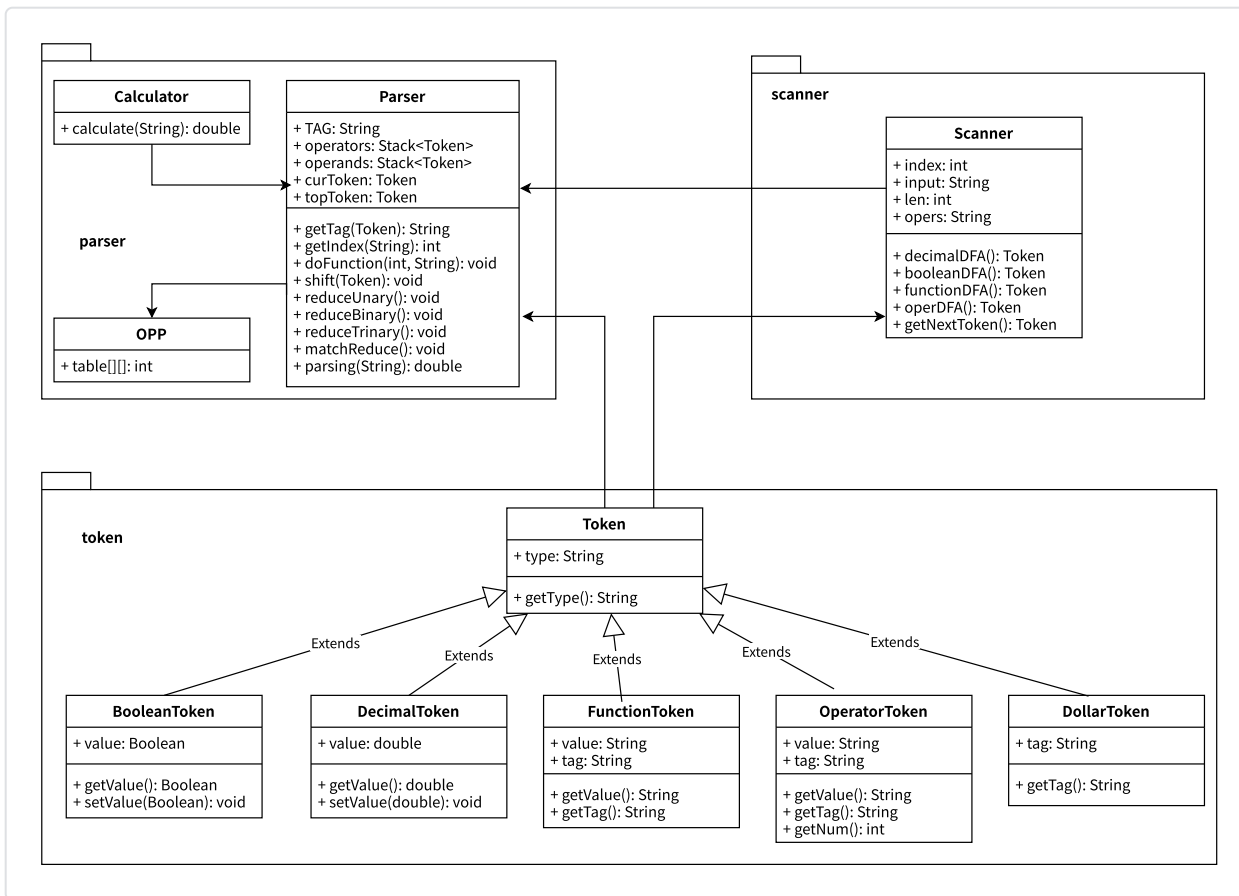
ExprEval在进行基于表达式的计算时，大概流程如下：语法分析部分把输入表达式传送到词法分析部分，词法分析部分根据状态转换图生成词法单元流，然后再把词法单元流返回到语法分析中根据语法规则和运算符优先级表进行相应的处理，最后输出运算结果。



UML图

为了实现设计模式中代码间的职责分离，本实验在架构上分为了三个模块：语法分析parser模块，词法分析scanner模块和词法单元token模块。

- 词法单元token模块：包含父类 `Token` 与其子类 `BooleanToken`、`DecimalToken`、`FunctionToken`、`OperatorToken` 和 `DollarToken`，负责建立不同类型词法单元的标记与保存词法单元的值
- 词法分析scanner模块：包含 `Scanner` 类，负责根据不同词法单元的状态转换图把表达式转换成词法单元流，需要借助 `Token` 类与其子类实现
- 语法分析parser模块：`Calculator` 类负责把输入表达式传送到 `Parser` 类与接受最终的计算返回结果；`OPP` 类负责保存运算符优先级表；`Parser` 类负责根据语法规则对词法单元流进行移入-规约处理，计算表达式运算结果，需要借助 `Scanner` 类和 `Token` 类及其子类实现



3 设计并实现词法分析程序

词法分析部分的核心是如何对词法单元分类和如何构建状态转换图。

词法单元分类

- **BooleanToken** 类：布尔类型的词法单元，类型为 **Boolean**，值为 **true** 或者 **false**
- **DecimalToken** 类：十进制数值的词法单元，类型为 **Decimal**，值为 **double** 类型的数值
- **FunctionToken** 类：预定义函数的词法单元，类型为 **Function**，标志为 **func**，具体包含了 **max**、**min**、**sin** 和 **cos** 这四个预定义函数
- **OperatorToken** 类：操作符类型的词法单元，类型为 **Operator**，具体包含了预定义函数和括号以外的运算符，并按下表的级别定义相应的标志；OperatorToken中还有一个 **getNum()** 方法用于标记运算符是一元运算符（级别3,8）、二元运算符（级别4,5,6,7,9,10）还是三元运算符（级别11）

级别	描述	算符	结合性质
1	括号	()	
2	预定义函数	sin cos max min	
3	取负运算（一元运算符）	-	右结合
4	求幂运算	^	右结合
5	乘除运算	* /	
6	加减运算	+ -	
7	关系运算	= <> < <= > >=	
8	非运算	!	右结合
9	与运算	&	
10	或运算		
11	选择运算（三元运算符）	? :	右结合

- `DollarToken` 类：结束符号的词法单元，类型为 `Dollar`，标志为 `$`

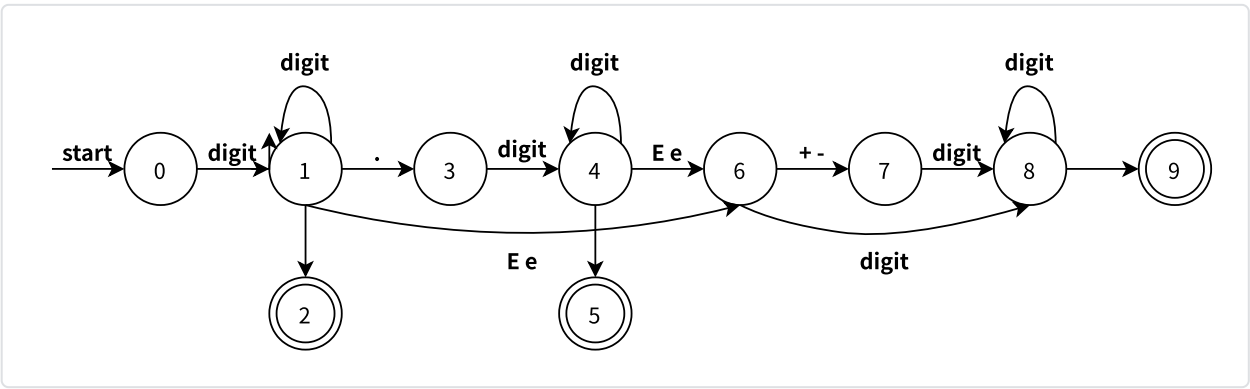
状态转换图

以下相关DFA的源代码可在Scanner类中查看。

- `decimalDFA()`：十进制数值的状态转换函数

<i>digit</i>	→	0 1 2 3 4 5 6 7 8 9
<i>integral</i>	→	<i>digit</i> ⁺
<i>fraction</i>	→	. <i>integral</i>
<i>exponent</i>	→	(E e) (+ - ε) <i>integral</i>
<i>decimal</i>	→	<i>integral</i> (<i>fraction</i> ε) (<i>exponent</i> ε)

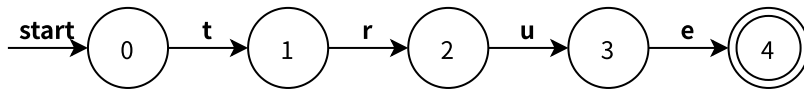
根据以上正则表达式可以画出下面的状态转换图：



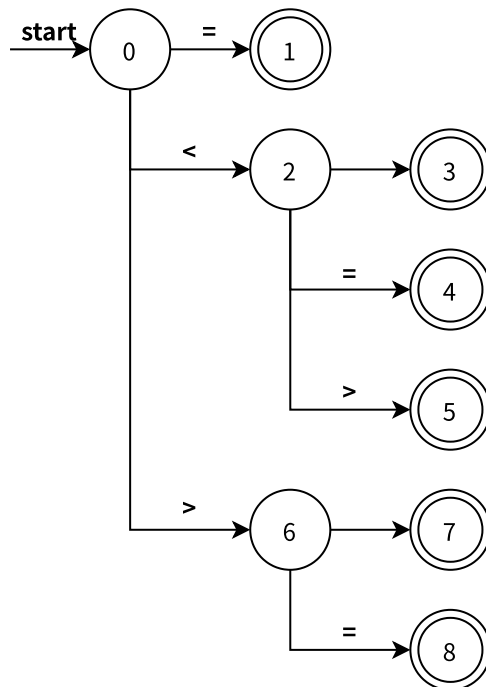
- 如果在中间哪一步发生了错误会抛出 `IllegalDecimalException` 异常
- 如果成功执行到终止符，返回一个新建的 `DecimalToken` 对象
- 在 `DecimalToken` 对象中，`fraction` 和 `integral` 的数值转换直接由java中的 `parseDouble()` 方法完成，`exponent` 的数值转换需要借助 `parseDouble()` 方法和

`Math.pow()` 方法完成

- `booleanDFA(Character curChar)`：布尔类型的状态转换图（字符匹配，以 `true` 为例）



- `functionDFA()`：预定义函数的状态转换图（也是字符匹配，类似于上面的 `true`）
- `operDFA(Character curChar)`
 - 关系运算符需要使用状态转换图处理，其余运算符不需要



- 识别取负运算符：只有当 `-` 前为 `)` 或者数字时，它才代表减号，否则就是负号
- 识别括号运算符的一个语法错误：如果 `curChar` 为 `)` 且前一个元素是 `(`，则抛出 `MissingOperandException` 异常。因为操作符和操作数不在同一个栈，所以需要在词法分析特别处理这一类异常情况
- 识别三元运算符的一个语法错误：如果 `curChar` 为 `:` 且前一个元素是 `?`，则抛出 `MissingOperandException` 异常。处理理由同上

总结

词法分析的流程是先在构造函数对输入字符串进行去除空格的预处理，然后在 `getNextToken()` 方法里扫描字符串，并调用相应的DFA方法新建不同类型的 `Token` 对象

4 构造运算符优先关系表

Java

```
1 public class OPP {
2     public static final int MISSINGLEFTPARENTHESIS = -7; // 缺少左括号
3     public static final int SYNTACTICEXCEPTION = -6;      // 语法错误
4     public static final int MISSINGOPERAND = -5;          // 缺少操作数
5     public static final int TYPEMISMATCH = -4;           // 类型错误
6     public static final int FUNCTIONCALL = -3;           // 函数语法错误
7     public static final int MISSINGRIGHTPARENTHESIS = -2; // 缺少右括号
8     public static final int TRINARYOPERATION = -1;       // 三元运算符异常
9     public static final int SHIFT = 0;                   // 移入
10    public static final int RDUNAOPER = 1;                // 单元运算
11    public static final int RDBINAOPER = 2;                // 二元运算
12    public static final int RDTRINAOPER = 3;                // 三元运算
13    public static final int RDPARENTHESIS = 4;             // 括号运算
14    public static final int ACCEPT = 5;                    // 接受
15    public static final int table[][] = {
16        /*栈顶*/ /*( ) func - ^ md pm cmp ! & | ? : , $ 读入字符*/
17        /*(*)*/ {0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, -2},
18        /**)*/ {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4},
19        /*func*/ {0, -3, -3, -3, -3, -3, -3, -3, -3, -3, -3, -3, -3, -3, -3},
20        /*-* */ {0, 4, 0, 0, 0, 1, 1, 1, 1, -6, -4, -4, 1, 1, 1, 1},
21        /*^*/ {0, 4, 0, 0, 0, 0, 2, 2, 2, -6, -4, -4, 2, 2, 2, 2},
22        /*md*/ {0, 4, 0, 0, 0, 0, 2, 2, 2, -6, -4, -4, 2, 2, 2, 2},
23        /*pm*/ {0, 4, 0, 0, 0, 0, 2, 2, 2, -6, -4, -4, 2, 2, 2, 2},
24        /*cmp*/ {0, 4, 0, 0, 0, 0, 0, 0, -4, -6, 2, 2, 2, -1, -3, 2},
25        /*!*/ {0, 4, -4, -4, -4, -4, -4, 0, 0, 1, 1, 1, -1, -3, 1},
26        /*&*/ {0, 4, -4, -4, -4, -4, -4, 0, 0, 2, 2, 2, -1, -3, 2},
27        /*|*/ {0, 4, -4, -4, -4, -4, -4, 0, 0, 0, 2, 2, -1, -3, 2},
28        /*?*/ {0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1},
29        /*:*/ {0, 4, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, -1, -1, 3},
30        /*,*/ {0, 4, 0, 0, 0, 0, 0, 0, -3, -3, -3, -3, 0, -1, 0, -3},
31        /*$*/ {0, -7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -3, 5}
32    };
33 }
```

5 设计并实现语法分析和语义处理程序

`Parser` 类的参数：

- `String[] TAG` 数组：负责辅助运算符优先关系表的位置索引
- `Stack<Token> operators`：存储运算符词法单元的栈
- `Stack<Token> operands`：存储运算量词法单元的栈
- `Token curToken`：当前扫描获取到的词法单元
- `Token topToken`：运算符词法单元栈的栈顶词法单元

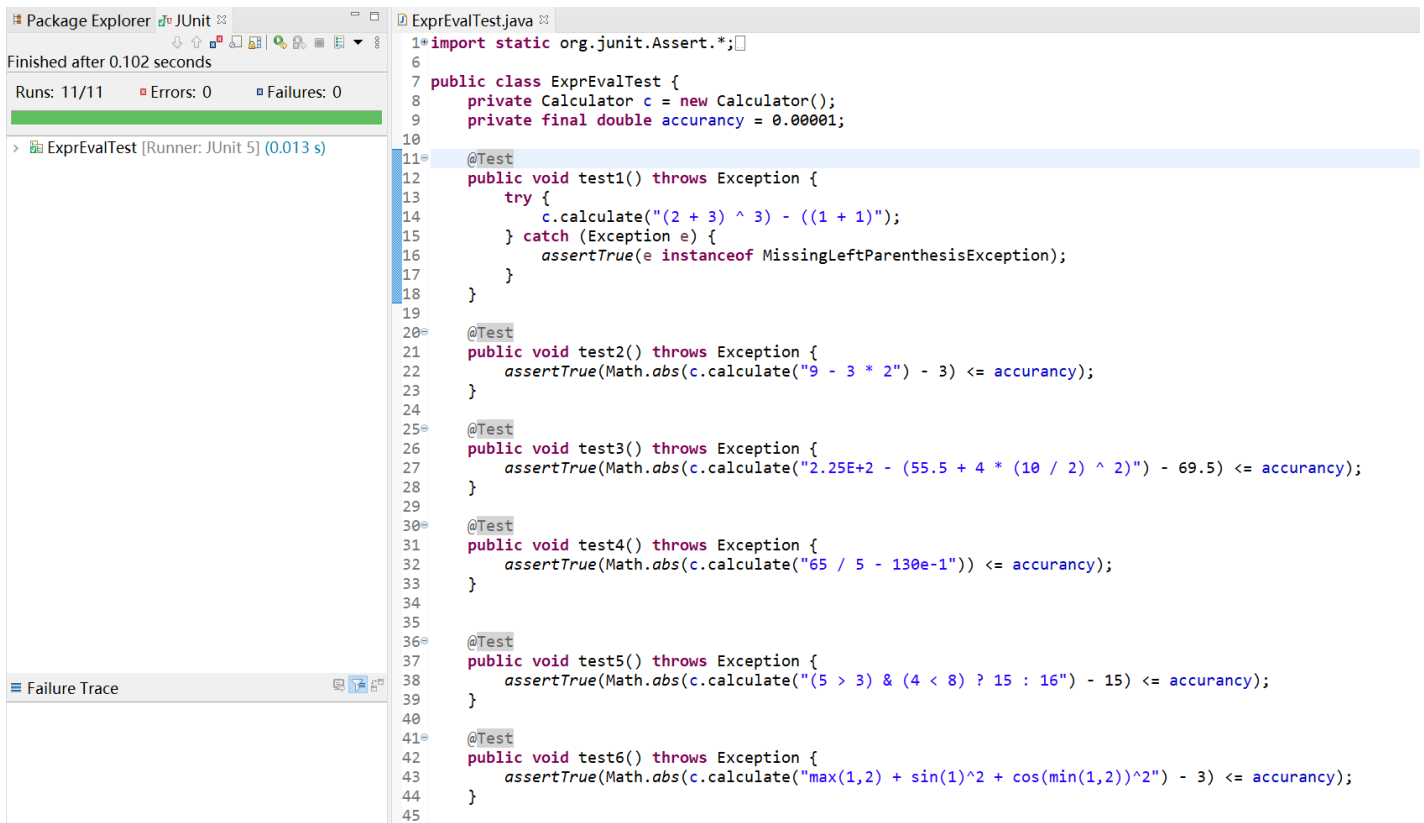
语法分析和语义处理主要是在 `Parser` 类中的 `parsing(String expression)` 方法完成：

1. 以输入字符串为参数新建一个 `Scanner` 类对象
2. 利用 `getNextToken()` 方法获取 `Token` 对象
3. 如果 `Token` 对象的类型为 `Boolean` 或 `Decimal`，则压入 `operands` 栈中，利用 `getNextToken()` 方法获取下一个 `Token` 对象
4. 如果 `Token` 对象的类型不为 `Boolean` 或 `Decimal`，则根据 `curToken` 和 `topToken` 的标志索引到运算符优先关系表，执行相关动作：
 - `shift(Token oper)`：移入操作，把当前词法单元压入 `operators` 栈，利用 `getNextToken()` 方法获取下一个 `Token` 对象
 - `reduceUnary()`：一元运算符的规约操作，`operands` 栈和 `operators` 栈分别弹出一个元素，进行相应的计算后把计算结果压入 `operands` 栈
 - `reduceBinary()`：二元运算符的规约操作，`operands` 栈弹出两个元素，`operators` 栈弹出一个元素，进行相应的计算后把计算结果压入 `operands` 栈
 - `reduceTrinary()`：三元运算符的规约操作，`operands` 栈弹出三个元素，`operators` 栈弹出一个元素，进行相应的计算后把计算结果压入 `operands` 栈
 - `matchReduce()`：括号运算与预定义函数运算归约操作，利用布尔型变量 `matchCompleted` 记录括号的作用范围

6 测试实验结果

单元测试

`ExprEvalText.java`



The screenshot shows an IDE with two panes. The left pane displays the JUnit test runner output, indicating that the tests passed successfully. The right pane shows the source code of the `ExprEvalTest` class, which contains six test methods. The test runner output shows the following details:

- Package Explorer: JUnit
- Finished after 0.102 seconds
- Runs: 11/11
- Errors: 0
- Failures: 0
- ExprEvalTest [Runner: JUnit 5] (0.013 s)

The source code of `ExprEvalTest` is as follows:

```
1 *import static org.junit.Assert.*;
2
3 public class ExprEvalTest {
4     private Calculator c = new Calculator();
5     private final double accuracy = 0.00001;
6
7     @Test
8     public void test1() throws Exception {
9         try {
10             c.calculate("(2 + 3) ^ 3 - ((1 + 1))");
11         } catch (Exception e) {
12             assertTrue(e instanceof MissingLeftParenthesisException);
13         }
14     }
15
16     @Test
17     public void test2() throws Exception {
18         assertTrue(Math.abs(c.calculate("9 - 3 * 2") - 3) <= accuracy);
19     }
20
21     @Test
22     public void test3() throws Exception {
23         assertTrue(Math.abs(c.calculate("2.25E+2 - (55.5 + 4 * (10 / 2) ^ 2)") - 69.5) <= accuracy);
24     }
25
26     @Test
27     public void test4() throws Exception {
28         assertTrue(Math.abs(c.calculate("65 / 5 - 130e-1")) <= accuracy);
29     }
30
31     @Test
32     public void test5() throws Exception {
33         assertTrue(Math.abs(c.calculate("(5 > 3) & (4 < 8) ? 15 : 16") - 15) <= accuracy);
34     }
35
36     @Test
37     public void test6() throws Exception {
38         assertTrue(Math.abs(c.calculate("max(1,2) + sin(1)^2 + cos(min(1,2))^2") - 3) <= accuracy);
39     }
40 }
41
42
43
44
45
```

回归测试

test_simple.bat

```
=====
Statistics Report (8 test cases):
```

```
Passed case(s): 8 (100.0%)
Warning case(s): 0 (0.0%)
Failed case(s): 0 (0.0%)
=====
```

test_standard.bat

```
=====
Statistics Report (16 test cases):
```

```
Passed case(s): 16 (100.0%)
Warning case(s): 0 (0.0%)
Failed case(s): 0 (0.0%)
=====
```