

## 三、《编译原理》实验

---

### 基于表达式的计算器 *EXPR-Eval*<sup>1</sup>

#### 1. 实验描述

##### 1.1 实验目的

本实验是一个基于编译原理课程核心教学内容的综合型、应用型实验。该实验借助一个基于表达式的计算器的设计与实现过程，帮助学生深入理解和牢固掌握编译原理中的词法分析、语法分析、语义分析等重要环节。本实验的核心知识点是算符优先分析（Operator Precedence Parsing，简称 OPP）技术，重点是算符优先关系表的构造。

本实验的主要目标包括：

- 掌握词法分析程序的工作原理与构造方法，包括较复杂的浮点数常量的词法规则定义及其识别程序的构造。学习如何根据词法规则定义（例如正则表达式或正则文法）来构造一个词法扫描程序的程序蓝图（即有限状态自动机），并利用高级程序设计语言实现词法分析过程。
- 掌握算符优先分析技术，除最基础的算术运算符外，还包括重载的（Overloading）一元运算符、三元运算符、关系运算符、逻辑运算符、预定义函数等运算符的处理。
- 掌握基本的语义处理技术，能够正确地处理表达式计算中的类型兼容检测和类型自动推导。
- 通过加强软件设计方面的交流与讨论，并在面向对象编程风格的大量实践，提高对面向对象设计的认识，养成良好的编程习惯，并了解大型工程文档的组织与提交。
- 加深了解软件测试的工作原理与使用方法，初步体会软件测试自动化的基本思路。

---

1. Credit: Prof. Wenjun Li @ SYSU. 若对实验内容有任何疑问或改进意见，请联系任课教师。

## 1.2 实验环境

本实验要求采用面向对象设计，并使用 Java 语言实现。在设计与实现中，注意面向对象方法中封装与信息隐藏、数据抽象、继承、多态性、异常处理、设计模式等技术的运用，并可借鉴实验软装置加强对图形用户界面（GUI）等类库 API 的掌握。

**编程语言：**Java JDK 1.5 或以上版本

**开发工具：**可自由选择 Eclipse、JBuilder 等 IDE 环境，也可直接采用 UltraEdit、EditPlus 等编辑器在命令行工作。但提交的实验结果必须独立于特定的 IDE，可直接运行在 JDK 上。

**实验软装置：**为让学生在实验过程中可集中精力完成对表达式语言处理的设计与实现，并便于学生和老师对实验结果进行测试，本实验提供了辅助的实验软装置。

所谓实验软装置，是在软件设计类课程实验中相当于电子技术、计算机硬件等专业实验中的示波器、实验箱之类的实验装置，只不过在软件设计类实验中这些装置全部以软件的形态出现。本质上，实验软装置是一个应用框架（Application Framework），它提供了整个实验中非核心部分的代码，学生自己完成的实验核心部分代码与这些代码结合在一起即构成了一个完整的实验结果。

此外，在实验软装置这个应用框架中通常还嵌入了负责软件测试的代码，这既可用于学生在实验过程自己对实验结果进行测试，也便于教师对学生提交的实验结果进行评价。

本文档第 3 部分详细介绍了本实验所用的实验软装置。

**编码规范：**要采用面向对象风格来设计实验中的所有类，并遵循一种良好的程序设计习惯。例如，如果你将一个程序的很多功能全部放在一个长长的 main() 方法中实现，这样的设计与编码风格会被扣分。

在实验过程中应注意培养规范的编码风格。本实验要求所有源代码严格遵循 Sun 公司（现为 Oracle 公司）关于 Java 程序设计语言的编码规范（Code Conventions for the Java Programming Language, Revised April 1999），详细内容请参见：[Code Conventions for the Java Programming Language: Contents \(oracle.com\)](http://codeconventions.oracle.com)。

完成项目代码后，应使用 JDK 附带的文档工具 javadoc，根据源程序中的文档化注释，自动生成相应的说明性文档。

## 1.3 实验内容

本实验要求使用 Java 语言设计并实现一个实际可用的计算器，其规格说明参阅本文档第 2 部分的详细描述。你必须按照软件工程的规范要求，编写本实验中设计与实现相关的文档；此外，你还必须在实验报告中撰写以下相关内容。

### 1.3.1 讨论语法定义的二义性

本文档第 2 部分以 BNF 描述了可接收的输入表达式的形式化语法定义。请问这一语法定义是否存在二义性？如果你回答不存在，请说明理由；如果你回答存在，请证明之，并说明如何解析表达式中的二义性。

上述 BNF 二义性的讨论有助于你更好地理解所处理的表达式语言。

### 1.3.2 设计并实现词法分析程序

在 EXPR-Eval 中，输入表达式支持布尔类型常量、数值类型常量（其中包括科学记数法）、各种算术运算、关系运算、逻辑运算、以及预定义函数等，请从本文档中提取支持的表达式语言的词法规则，并绘制识别其中所有合法单词的有限自动机（状态转换图），并以此作为程序蓝图编写你的词法分析程序。

建议你的词法分析程序的主类命名为 `Scanner`。为便于在此基础上开发语法分析和语义处理程序，建议你采用嵌入式测试（Embedded Testing）对你自己开发的词法扫描程序进行较为完善的测试。

在你的实验报告中，词法分析部分的重点是描述你在实验中：如何对单词进行分类，譬如算术运算符或关系运算符是分别作为一类单词，还是每一个运算符就是一类单词；如何处理对预定义函数名和布尔常量的识别，注意它们都属于标识符一类；如何处理科学记数法表示的数值常量；如何处理字符串的边界；等等。

### 1.3.3 构造算符优先关系表

本实验要求采用 OPP 作为语法分析技术，因而语法分析的核心问题是算符优先关系表的构造。请仔细构造你的算符优关系表，并在实验报告中说明你在表中如何处理一些较为敏感的关系；所谓敏感的关系，意指可能比较容易搞错的两个运算符之间的关系，譬如一元取负运

算符和二元减法运算符之间的关系、三元运算符与其他运算符之间的关系、预定义函数与其他运算符之间的关系等。

注意！请特别声明你是如何处理表达式中两个重载（Overloading）的运算符：一元取负运算符“-”和二元减法运算符“-”，例如  $2-3*-4$ 。

### 1.3.4 设计并实现语法分析和语义处理程序

以上述算符优先关系表为基础，编写 的语法分析程序和语义处理程序。建议你将语法分析程序的两个核心动作单独放在两个独立的子程序 `shift()`和 `reduce()`中，这符合结构化程序设计中每一子程序完成一个相对独立功能的基本原则。你的语义处理程序重点是完成类型推导与类型兼容性检查的工作；实际上，这些语义处理代码与语法分析代码通常是混合在一起的。

在语法分析和语义处理过程中发现任何错误，均以异常（Exception）的形式对外报告；本实验的实验软装置已预定义了各种出错情况对应的异常类型，详见本文档第 3 部分。

建议你的语法分析和语义处理程序的主类命名为 `Parser`。为便于对语法分析和语义处理程序进行调试，你既可以采用嵌入式测试，也可以使用实验软装置中提供的测试工具。

在你的实验报告中，语法分析与语义处理部分的重点是描述你在实验中：如何实现 `OPP` 的核心控制程序，请以伪码或 `Java` 代码给出核心的算法；如何实现各种运算符的归约（Reduce）动作；如何对语义进行处理，主要是类型的兼容性检查与类型的推导；等等。

### 1.3.5 测试你的实验结果

请使用本实验的实验软装置中提供的测试用例对自己的实验结果进行测试。实验软装置提供了 `simple` 和 `standard` 两个级别的测试；如果时间许可，请在通过了所有 `standard` 测试用例后，再提交你的最终实验结果。

注意，任课教师在评价你的实验结果时，会使用比 `standard` 测试多出数倍的测试用例来测试你的实验结果。因而，如果时间许可，你应该编写比 `standard` 测试更多的测试用例，以减少自己实验结果中可能存在的设计错误或实现错误。本文档第 3 部分介绍了如何以 `XML` 文档书写自己设计的测试用例，这些测试用例可借助实验软装置自动完成回归测试（Regression Testing）。

## 1.4 提交结果

请将实验的所有结果存放在一个目录 “学号\_姓名\_lab2” 中。

例如，学号为“00184163”、姓名为“赵德柱”的同学应将其完成的实验全部结果存放在“00184163\_赵德柱\_lab2”的目录中。

最后将该目录压缩成单个文件后，在指定截止时间前提交到本课程的[指定链接](#)（参阅本课程网站上的指引）。

注意，实验最终提交的实验结果应包括：

- 所用的实验软装置的所有内容。
- 完成的所有源程序代码，全部存放在 `src\parser` 文件夹中。
- 生成的 `javadoc` 文档，全部存放在 `doc` 文件夹中。
- 自定义的测试用例 XML 文档 `mytest.xml`，存放在 `testcases` 文件夹中。
- 撰写的实验报告 `design.doc` 或 `design.pdf`，直接存放在根目录。
- 自述文件 `readme.txt`，给出你的姓名、学号、电子邮件以及其他补充说明。

实验报告中应包括：二义性分析、词法分析的设计与实现、算符优先关系定义、语法分析与语义处理的设计与实现、程序运行的屏幕截图、实验的心得体会等内容，并利用 Word 的样式自动生成实验报告的目录。

请注意，你在实验报告中的屏幕截图应包括运行你的嵌入式测试，以及运行实验软装置中的各种测试用例等屏幕画面。如果你的实验报告中未包含任何程序运行的屏幕截图，任课教师在评价你的实验结果时会倾向于理解为你的程序不可运行。

## 2. 计算器规格说明

EXPR-Eval 是一个基于表达式的计算器，用户输入一个表达式后，对表达式进行分析并显示计算结果，如图 1 所示。



图 1. EXPR-Eval 计算器

本章描述了本实验中待开发的计算器的功能需求、计算器所支持表达式的语法与语义等，实验结果必须符合本章指定的规格说明。

## 2.1 功能需求

不同于 Windows 系统自带的计算器（参见“开始→程序→附件→计算器”的功能），是一个基于表达式的计算器，用户可以直接输入一个符合平时习惯的表达式。将检查输入表达式是否正确，如果正确则给出计算结果，否则显示出错提示信息，如图 2 所示。



图 2. 报告输入表达式中存在的错误

### 2.1.1 基本功能

EXPR-Eval 提供了一种图形用户界面，支持如下功能：

**Calculate**

对输入表达式进行分析和计算，如果未发现错误则显示计算结果，否则报告相应的错误信息。你在本实验中的主要任务就是设计并实现该按钮提供的功能，计算器与用户交互的表示层（Presentation Tier）以及测试框架等其他功能均已由实验软装置完整地提供。

**Clear**

清除输入框（上部）和输出框（下部）中的已有内容。

**Copy**



将输出框的内容复制到系统剪贴板。

Paste

将系统剪贴板中的内容复制到输入框。

Help

显示帮助信息，包括实验软装置的版权信息、版本号、当前版本计算器支持的输入表达式的主要特性等，如图 3 所示。

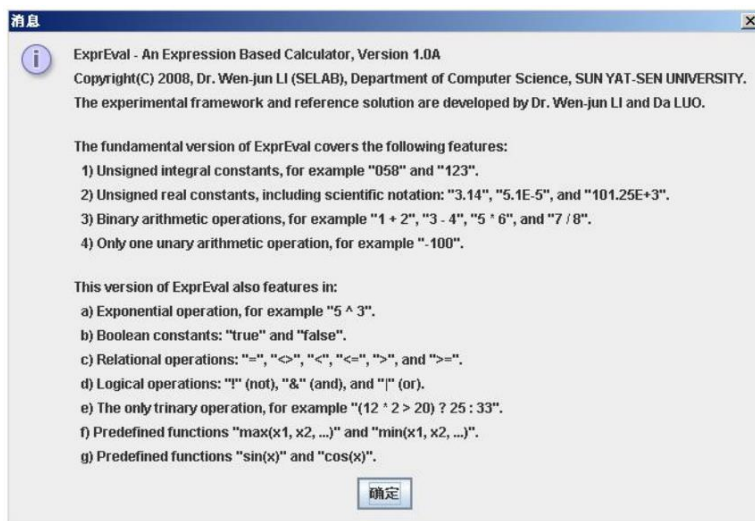


图 3. 显示 EXPR-Eval 帮助信息

Exit

退出计算器，同时退出 JVM 进程。

## 2.1.2 用户界面

EXPR-Eval 计算器的控制按钮被设计为一个浮动的工具条（Tool Bar），因而这些按钮可以被拖动到计算器的上下或左右的任意位置，甚至浮出作为单独的工具窗口，如图 4~6 所示。



图 4. 重新布置 EXPR-Eval 的工具按钮（上端）



图 5. 重新布置 EXPR-Eval 的工具按钮（右端）



图 6. 重新布置 EXPR-Eval 的工具按钮（浮出窗口）

### 2.1.3 计算功能

EXPR-Eval 计算器支持如下计算功能：

- 可计算的输入表达式是一个常量表达式，其中的常量允许使用 2 种类型：数值类型和布尔类型。数值类型的常量可以是 Pascal 等程序设计语言中的整数或浮点数；布尔类型的常量只允许有 true 和 false。
- 支持 5 种基本算术运算，包括加法、减法、乘法、除法、幂运算；这些都是二元运算。
- 可使用负号 “-” 作为一元运算符，表示取负。注意，取负运算符与减法运算符是两个重载的运算符，一个是一元运算符，一个是二元运算符。
- 支持类似 C/C++ 语言的三元运算符 “?:”。例如，输入表达式  $5 > 2 * 2.4 ? 12 : 6.5 * 2$  的求值结果为 12。
- 在输入表达式中可使用关系运算和逻辑运算写出布尔表达式，这些布尔表达式只能作为三元运算 “?:” 中的第一个子表达式，不可直接作为计算结果。可使用圆括号确定算术运算、关系运算和逻辑运算等运算的计算次序。
- 提供了 4 个预定义函数，包括 sin、cos、max 和 min。其中，函数 sin(x) 和 cos(x) 只能有一个参数；而函数 max(x1, x2, ...) 和 min(x1, x2, ...) 支持可变数目的参数，但它们至少要有 2 个或以上的参数。
- 如果用户输入的表达式存在错误，应准确地报告其中的错误，并指明是词法错误、语法错误还是语义错误。



## 2.2 语法规则

在 EXPR-Eval 的输入表达式中只可使用空格（Space）作为分隔各个单词的空白符号（Blank）。多个连续的空白符号其含义与单个空白符号相同；在表达式中的运算符、运算量、括号之间，也允许不使用空白符号进行分隔，意即空白符号是可选的。

在许多程序设计语言中常见的空白符号，如制表符（Tab）、回车、换行、注释等，在 EXPR-Eval 的输入表达式中禁止使用。

### 2.2.1 布尔类型的常量

逻辑表达式支持使用两个布尔常量：**true** 和 **false**。

注意，EXPR-Eval 的输入表达式是大小写无关的；例如，**true**、**True** 和 **TRUE** 都是合法的布尔常量。

### 2.2.2 数值类型的常量

数值常量仅支持十进制，其书写形式类似 Pascal 语言中的整数和浮点数常量，但仅支持无符号的数值类型。它们必须符合以下正规定义式的描述：

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

例如，以下是正确的数值常量写法：

4.7   5.00   7E3   1.7e7   9.334e-2   5.1e+3   3.14E-0

其中，科学记数法的含义同一般的程序设计语言。譬如，9.334e-2 相当于 0.09334，5.1e+3 相当于 5100，3.14E0 或 3.14E-0 均相当于 3.14。又如，以下是错误的数值常量写法：

.11   e4   4.e6   3.   7e

因为 .11 和 e4 缺少整数部分，4.e6 和 3. 缺少小数部分，7e 缺少指数部分。

无论是整数类型还是浮点类型的常量，均会将它们转换为双精度浮点数据在程序内部进行运算。在本实验中不必考虑比双精度浮点数更高精度的相关问题。

注意，据上述定义，在 中不支持负数作为常量。虽然 **-3.14** 是合法的，但它应被理解为数值常量 **3.14** 的一元取负运算 “-”，意即词法分析的结果这应该是两个单词而不是一个单词。

## 2.3 语法规则

EXPR-Eval 所支持的表达式采用与普通表达式类似的语法形式，本小节采用标准 BNF（而不是扩展的 EBNF）定义表达式的语法规格说明，并规定了表达式中各运算符的优先级和结合性质，最后用自然语言 和例子描述了各种运算的语义。

### 2.3.1 表达式的 BNF 定义

EXPR-Eval 的表达式规格说明如下列 BNF 所示：

<i>Expr</i>	→ <i>ArithExpr</i>
<i>ArithExpr</i>	→ <b>decimal</b>   ( <i>ArithExpr</i> )   <i>ArithExpr</i> + <i>ArithExpr</i>   <i>ArithExpr</i> - <i>ArithExpr</i>   <i>ArithExpr</i> * <i>ArithExpr</i>   <i>ArithExpr</i> / <i>ArithExpr</i>   <i>ArithExpr</i> ^ <i>ArithExpr</i>   - <i>ArithExpr</i>   <i>BoolExpr</i> ? <i>ArithExpr</i> : <i>ArithExpr</i>   <i>UnaryFunc</i>   <i>VariablFunc</i>
<i>UnaryFunc</i>	→ <b>sin</b> ( <i>ArithExpr</i> )   <b>cos</b> ( <i>ArithExpr</i> )
<i>VariablFunc</i>	→ <b>max</b> ( <i>ArithExpr</i> , <i>ArithExprList</i> )   <b>min</b> ( <i>ArithExpr</i> , <i>ArithExprList</i> )
<i>ArithExprList</i>	→ <i>ArithExpr</i>   <i>ArithExpr</i> , <i>ArithExprList</i>
<i>BoolExpr</i>	→ <b>true</b>   <b>false</b>   ( <i>BoolExpr</i> )   <i>ArithExpr</i> > <i>ArithExpr</i>   <i>ArithExpr</i> >= <i>ArithExpr</i>   <i>ArithExpr</i> < <i>ArithExpr</i>   <i>ArithExpr</i> <= <i>ArithExpr</i>   <i>ArithExpr</i> = <i>ArithExpr</i>   <i>ArithExpr</i> <> <i>ArithExpr</i>   <i>BoolExpr</i> & <i>BoolExpr</i>   <i>BoolExpr</i>   <i>BoolExpr</i>   ! <i>BoolExpr</i>

其中，**decimal** 的定义如第 2.2.3 小节所示。

### 2.3.2 优先级和结合性质

在 EXPR-Eval 支持的表达式中，各运算符的优先级与结合性质定义如下（其中未说明的结合性质默认为左结合）：

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

## 2.4 语义描述

在 EXPR-Eval 表达式中，各种运算符均有其严格的语义，因而可能出现有语义错误的情况（例如类型不匹配、除数为 0 等）或容易误解的情况（例如右结合性质等）。在 EXPR-Eval 的设计与实现中须小心处理这些语义。

### 2.4.1 括号与算术运算

加法、减法、乘法、除法、乘方（求幂）、括号等运算的语义与日常所见算术表达式中的含义相同，不再赘述。参加这些运算的操作数（或运算量，Operand）均须为算术表达式类型，不可是一个布尔类型的子表达式，否则会产生一个类型不匹配错误。

例如，表达式  $2 + 6 * 4$  的求值结果为 26，表达式  $20 ^ {((3 - 1) * 2 - 2)}$  的求值结果为 400，而表达式  $2 + 15 - (3 > 5) + 10$  虽然语法上是正确的，但会产生一个类型不匹配错误。

注意，乘方运算是右结合的。例如，表达式  $2 ^ 4 ^ 3$  等价于  $2 ^ (4 ^ 3)$ 。

### 2.4.2 取负运算

取负运算符具有较高的优先级，同时它与减法运算符是重载的运算符。例如，表达式  $2.5 + -2.4$  的求值结果是 0.1，表达式  $5 - -2.5$  的求值结果为 7.5。

### 2.4.3 预定义函数

提供了 4 个预定义函数：sin()、cos()、max()、min()，用户可在输入表达式中使用这些预定义函数。

$\sin()$  和  $\cos()$  函数用于计算三角函数值, 均需有且仅有一个数值类型的参数 (即不可使用布尔类型的值作为其参数), 且参数的值参照弧度制计算。例如, 表达式  $\sin(3.14/2)$  的求值结果为  $0.999999\cdots$ , 而表达式  $\cos(3.14/4)$  的求值结果是  $0.707388\cdots$ 。

$\max()$  和  $\min()$  函数分别计算多个参数中的最大值和最小值, 多个参数之间使用逗号分隔。注意, 这两个函数支持可变数目的参数, 即参数的个数允许是任意的, 但它们至少需要 2 个参数。例如, 表达式  $\max(3, 2, 6)$  的求值结果为 6, 表达式  $\min(3, 2, 6)$  的求值结果为 2, 表达式  $\max(\min(32, 7), \max(6, 4))$  的求值结果为 7。

#### 2.4.4 三元运算

EXPR-Eval 支持惟一的三元运算与 C/C++ 程序设计语言中的三元运算类似, 其形式如下:

$$Expr1 ? Expr2 : Expr3$$

其中,  $Expr1$  必须为布尔类型的子表达式,  $Expr2$  和  $Expr3$  必须为数值类型的子表达式。若  $Expr1$  的求值结果为 **true**, 则三元表达式的求值结果为表达式  $Expr2$  的值; 若  $Expr1$  的求值结果为 **false**, 则三元表达式的求值结果为表达式  $Expr3$  的值。

例子, 表达式  $2.25 < 4/2 ? 5 : 6$  的求值结果为 6, 表达式  $2 \geq 2 ? 1.5 : 2.5$  的求值结果为 1.5, 表达式  $\max(-1, 0) > 0 ? 5 : 3 \geq 0 ? 4 : 5$  的求值结果为 4。

#### 2.4.5 关系运算与逻辑运算

EXPR-Eval 中的关系运算和逻辑运算与通常程序设计语言中的关系运算和逻辑运算完全相同。例如, 表达式  $true | false$  的求值结果为 **true**, 表达式  $!(True \& FALSE)$  的求值结果为 **true**, 表达式  $5 < 4$  的求值结果为 **false**, 表达式  $true ? 5 : false ? 1 : 0$  的求值结果为 5。

注意, 在 EXPR-Eval 中 “等于” 关系的运算符是 “=”, “不等于” 关系的运算符是 “ $\neq$ ”, 这与众多程序员熟悉的 C/C++ 和 Java 语言的语法不同。此外, 布尔类型的值不可作为计算器最后的求值结果, 否则会产生一个类型不匹配错误。

### 3. 软实验装置

本章详细描述了你在本实验中使用的实验软装置。

## 3.1 文件组织

在为 [ExprEval](#) 提供的实验软装置中，包括以下目录结构和文件：

目录结构			描述
bin\	exceptions\	<b>DividedByZeroException.class</b>	实验软装置中预定义的 16 种异常，你的程序检测到
		... (共 16 个.class 文件)	输入表达式中的错误时，应抛出相应的异常。
	gui\	<b>MainWindow.class</b>	GUI 界面的主程序。
		<b>MainWindow\$1.class</b>	GUI 界面的内部类。
		<b>MainWindow\$2.class</b>	
		... (共 8 个.class 文件)	
	test\	<b>ExprEvalTest.class</b>	自动回归测试的主程序。
		<b>TestCase.class</b>	一个测试用例的抽象。
<b>ExprEval.class</b>		启动 <b>ExprEval</b> 计算器的主程序。	
doc\	<b>*.html</b>		运行 <b>doc.bat</b> 后根据你的源程序中的注释生成的 HTML 文档。
ref\	<b>*.html</b>		实验软装置所有源程序的 HTML 文档。
src\	parser\	<b>Calculator.java</b>	是你在本实验中需要设计并实现的类，实验软装置仅 为该类提供了一个简单的框架。
testcases\	<b>simple.xml</b>		几个简单的测试用例。
	<b>standard.xml</b>		标准的测试用例，覆盖了所有简单的用例。
<b>build.bat</b>		将你的程序与实验软装置合并在一起，构建一个可运行程序的脚本。 正常运行该脚本后， <b>\bin</b> 下 会创建一个 <b>parser</b> 目录，并存放你的程序的字节码。	
<b>doc.bat</b>		生成你的程序的 javadoc 文档的脚本。	
<b>run.bat</b>		运行你的可运行程序的脚本，需要在 <b>build.bat</b> 成功后才可正常执行。	
<b>test_simple.bat</b>		运行\testcases\simple.xml 中定义的所有简单测试用例的脚本。	
<b>test_standard.bat</b>		运行\testcases\standard.xml 中定义的所有标准测试用例的脚本。	

## 3.2 工作原理

[ExprEval](#) 实验软装置中各源程序的文档参见 [refindex.html](#)。其工作原理如下：

计算器的入口程序是 **ExprEval.class**，该类只有一个 **main()** 方法，不使用任何命令行参数；其方法体中的惟一功能是创建一个 **gui.MainWindow** 对象实例（该对象也是一个 **JFrame** 类型的实例），并调用其 **show()** 方法使得 GUI 界面开始工作。

在 `gui.MainWindow` 对象实例的构造方法中创建了屏幕上所见的 `ExprEval` 用户界面中的各个图形组件。当用户点击 `Calculate` 按钮时，它将创建一个 `parser.Calculator` 的对象实例，并以输入框中的当前字符串为参数，调用该对象实例的 `calculate()` 方法：

**public double calculate(String expression) throws ExpressionException:**

如果调用正常返回，则将计算结果格式化后显示在计算器的输出框，否则将异常信息作为错误消息也显示在输出框。程序包 `exceptions` 中的各种异常类分别代表了不同的出错类别（如后详述），错误消息记录在这些异常类的对象中。

在实验软装置中，仅提供了 `parser.Calculator` 的一个简单框架，每次调用 `calculate()` 方法都返回一个伪随机数，而不管输入参数是什么表达式。你在本实验中的主要任务就是重新设计 `parser.Calculator` 类的 `calculate()` 方法，在方法体中调用你自己编写的词法分析、语法分析、语义处理等程序，返回参数指定的表达式的正确求值结果，或准确地抛出一个异常对象以指明表达式中错误的类型。

### 3.3 预定义异常

如前所述，`parser.Calculator` 类的 `calculate()` 方法未发现输入表达式有错则返回 `double` 类型的计算结果；但如若发现输入表达式有词法错误、语法错误、或语义错误，则应抛出一个合适的异常对象，以表示输入表达式中错误的类型。

#### 3.3.1 异常的分类

实验软装置在 `exceptions` 程序包中预定义了 16 种用户自定义的异常类型，其根类设计为 `ExpressionException`。这些异常的具体分类结构如下所示：

```
ExpressionException （表达式中的所有错误）
|
|-- LexicalException （表达式中的词法错误）
|   |
|   |-- IllegalDecimalException （非法的数值常量）
|   |-- IllegalIdentifierException （非法的标识符）
|   |-- IllegalSymbolException （非法的符号）
|
|-- SyntacticException （表达式中的语法错误）
|   |
|   |-- MissingOperatorException （缺少运算符）
|   |-- MissingOperandException （缺少运算量）
|   |-- MissingLeftParenthesisException （遗漏了左括号）
|   |-- MissingRightParenthesisException （遗漏了右括号）
|   |-- FunctionCallException （预定义函数调用的语法形式错误）
|   |-- TrinaryOperationException （三元运算的语法形式错误）
|   |-- EmptyExpressionException （输入表达式为空）
|
|-- SemanticException （表达式中的语义错误）
|   |
|   |-- DividedByZeroException （除数为 0 错误）
|   |-- TypeMismatchedException （类型不匹配错误）
```



注意，上述异常分类是根据日常对表达式中错误的理解而划分的，并不是严格地按照异常产生的地点（词法分析程序、语法分析程序、语义处理程序等）来划分。譬如，类型不匹配错误实际上在输入表达式的 BNF 定义中已经约束，照一般处理应该划分为语法错误，但本实验中却划分为语义错误。

### 3.3.2 异常的使用

如果你的程序抛出了 **ExpressoinException** 之外的异常，则意味着你的程序本身产生了错误，而不是找到输入表达式的一个错误。

在你的程序中，应准确判断输入表达式中错误的类型，并抛出一个指明该类型错误的最具体异常类对象实例，而不是一律抛出 **ExpressoinException** 异常。找到错误后抛出异常的最简单方法，是在你的程序中写入类似如下的代码（例如发现一个缺少左括号的错误）：

```
throw new MissingLeftParenthesisException();
```

为此，最好在你的程序开头导入预定义异常的程序包：

```
import exceptions.*;
```

在一种良好的面向对象程序设计风格中，尽量将程序中抛出的异常限定在合理的最小范围。例如，你的词法扫描程序接口如果设计为：

```
public Token getNextToken() throws Exception;
```

就不如设计为：

```
public Token getNextToken() throws LexicalException;
```

因为词法分析的过程中找到的所有错误能够引发的异常不会超出 **LexicalException** 的范围，故前者显然声明了过多的异常。

## 3.4 异常类型详解

在编写代码前，请仔细阅读本小节关于预定义异常的使用方法。如果你认为这些预定义异常尚不足以表达更细致的异常分类，请尽管在已有预定义异常之下定义更明细的异常分类；例如，从异常类 **IllegalDecimalException** 派生出一个 **MissingExponentException** 类，表明科学记数法表示的数值在 E 之后缺少指数部分。

### 3.4.1 ExpressionException 异常

只有其他具体异常类无法表达该错误的类别时，才抛出 ExpressionException 异常。通常你的程序不应抛出这一异常。

### 3.4.2 LexicalException 异常

所有词法错误的通用类型；你的词法扫描程序抛出的异常不应超出此类的范围，即所抛出的异常必须都是此类的派生类。

仅当你的程序以预定义的 3 个 LexicalException 派生类不足以准确描述所找到的错误类别时，才直接抛出 LexicalException 异常。

### 3.4.3 IllegalDecimalException 异常

在数值类型常量的扫描过程中发现了单词错误则抛出此异常。

例如，4.、.7、和 4.E+3 会导致该异常抛出；但 E6、5@则不应抛出此异常，前者会被理解为错误的标识符，后者则含有不合法的符号。

运行以下测试用例应抛出 IllegalDecimalException 异常：

- 3.e3 + 1
- 4 + 10.E+5 + 1
- 3.3e3.3 + 1
- 1 + 3.3E.3 + 2
- 1 + 3.3E-(3 + 2)
- min(4., 7)
- 12.3Emax(4, 5, 6)

### 3.4.4 IllegalIdentifierException 异常

你的词法扫描程序遇到字母开头的单词时，应转入对一个标识符的识别。除预定义的函数名和布尔类型常量外，其他标识符均属非法的，此时抛出此异常。

例如，TAG 和 mix 会导致该异常抛出；但 sin5 则不抛出此异常，因为它会被解释为两个单词：

一个标识符和一个数值常量。注意，`ExprEval` 并未规定标识符与其他单词之间必须以空白符号分隔。

运行以下测试用例应抛出 `IllegalIdentifierException` 异常：

- `5 / v4 + 1`
- `4 + mix(5, 2) + 1`

### 3.4.5 `IllegalSymbolException` 异常

你的词法扫描程序若遇到不合法的字符则抛出此异常。例如 `5@34` 会导致该异常抛出。

运行以下测试用例应抛出 `IllegalSymbolException` 异常：

- `(5 @ 4) ? 7 : 8`

### 3.4.6 `SyntacticException` 异常

所有语法错误的通用类；你的语法分析程序抛出的异常不应超出此类的范围，即所抛出的异常必须都是此异常类的派生类。

仅当你的程序以预定义的 7 个 `SyntacticException` 派生类不足以准确描述找到的错误时，才直接抛出 `SyntacticException` 异常。

### 3.4.7 `MissingOperatorException` 异常

你的程序在检测到输入表达式中缺少运算符时，应抛出此异常。

注意，表达式 `23 + (43) + 5` 是一个正确的表达式，其中并不缺少运算符；而表达式 `2 + () + 5` 会被认为是缺少运算量，而不是缺少运算符。

运行以下测试用例应抛出 `MissingOperatorException` 异常：

- `(1 + 2) (3 - 4) - 5`
- `(1 + 2) ^ (3 - 4) 5`
- `cos(0.5)12.3E+4`

### 3.4.8 MissingOperandException 异常

你的程序在检测到输入表达式中缺少操作数（运算量）时，或表达式中调用预定义函数缺少相应的参数时，应抛出此异常。例如，运行以下测试用例应抛出 **MissingOperandException** 异常：

- $(1 + 2) ^ (3 - ) + 5$
- $3 > 2.5 * 1.5 ? 9 :$
- $3.14 * 2 >= 2.5 * 3 ? (6 : 7) + 8$
- $7 > 0 ? 7 <= 0 ? : 6 : 5$
- $\sin()$
- $\cos(3.14, )$
- $\min()$
- $\min(2.5)$
- $\min(, 1.8)$
- $\max(3.14, )$
- $\max(17, , 87)$
- $\text{true} ? : 5$
- $2 + ( ? 4 : 5)$

### 3.4.9 MissingLeftParenthesisException 异常

你的程序在检测到输入表达式中的括号不匹配，并且是缺少左括号时，应抛出此异常。例如，运行以下测试用例应抛出 **MissingLeftParenthesisException** 异常：

- $(2 + 3) ^ 3) - ((1 + 1)$

### 3.4.10 MissingRightParenthesisException 异常

你的程序在检测到输入表达式中的括号不匹配，并且是缺少右括号时，应抛出此异常。例如，运行以下测试用例应抛出 **MissingRightParenthesisException** 异常：

- $((2 + 3) ^ ((3 - 1) + 1)$

### 3.4.11 FunctionCallException 异常

在调用预定义函数时，若函数调用的语法形式错误，则应抛出此异常。例如，**sin()**和**cos()**函数的参数表中多于一个参数时应抛出此异常；所有函数调用缺少左括号时，亦抛出此异常（注意，此时并不当作缺少左括号来处理）。

还应注意，当所有预定义函数的参数个数不足（未达到该函数最低要求的参数数目）时，你的程序应抛出 **MissingOperandException**，而不是抛出此异常。

运行以下测试用例应抛出 **FunctionCallException** 异常：

- **sin(2, 1)**
- **max5, 6, 8)**

### 3.4.12 TrinaryOperationException 异常

当三元运算符的“?”和“:”出现不配对时，应抛出此异常，例如表达式 **6?7:7:9**；当三元运算符与括号的匹配出问题，亦应抛出此异常，例如表达式 **5?(8:8)**。

运行以下测试用例应抛出 **TrinaryOperationException** 异常：

- **false?9:true?1:3:5**

### 3.4.13 EmptyExpressionException 异常

输入表达式为空，亦或输入表达式只含空白符号，没有任何合法的或不合法的单词，仅在此时你的程序才抛出此异常。仅当运行以下这类测试用例时，才会抛出 **EmptyExpressionException** 异常：

- （全空的输入表达式）

- (含 3 个空格的输入表达式)

### 3.4.14 SemanticException 异常

所有语义错误的通用类，你的语义处理程序抛出的异常不应超出此类的范围，即所抛出的异常必须都是此类的派生类。

仅当你的程序以预定义的 2 个 **SemanticException** 派生类不足以准确描述找到的错误时，才直接抛出 **SemanticException** 异常。

### 3.4.15 DividedByZeroException 异常

表达式任何计算过程（包括其中某一子表达式的计算）中，出现除数为 0 的情况则应抛出此异常。注意，在 `ExprEval` 中并不支持逻辑运算的短路求值！

例如，运行以下测试用例应抛出 **DividedByZeroException** 异常：

- $4 / (12 - 3 * 4) + 1$

### 3.4.16 TypeMismatchedException 异常

当参与运算的运算量与相应的运算符所要求的类型不匹配时，你的程序应抛出此异常。

各类运算符均有各自严格的类型要求，譬如算术运算符和关系运算符不允许使用布尔类型的值作为运算量，而逻辑运算符则不允许使用数值类型的值作为运算量。因而，为准确地判断类型兼容错误类型，你的程序还必须正确地完成子表达式类型的推导。显然，在 `ExprEval` 表达式中不允许类型转换，譬如 `false` 不可当作数值类型的 0 使用。

类型不兼容的表达式例子很多，譬如表达式 `true + 5`、`5 & 76`、`max(54, true)` 均会导致抛出该异常。运行以下测试用例应抛出 **TypeMismatchedException** 异常：

- $(13 < 2 * 5) + 12$
- $12 ? 34 : 56$
- $true ? 42.5 > 5 * 8 : 15$
- $4 ^ (32.5 > 65)$
- $\sin(32.5 > 65)$



## ● 32.5 | 65

## 3.5 回归测试

为便于任课教师对学生的实验结果进行测试和评价，同时也便于各位同学自己对实验结果进行测试以减少可能存在的设计或编码错误，ExprEval 实验软装置还包括了支持自动回归测试的工具。正确理解这些工具有助于以极其简单的方式实现回归测试，也可使用更多的自定义测试用例进一步提高软件测试的质量。

### 3.5.1 工作原理

根据目录下的 **test\_simple.bat** 和 **test\_standard.bat** 两个脚本分别启动基于简单测试用例和基于标准测试用例的自动回归测试过程。例如，**test\_simple.bat** 脚本的内容如下：

---

```
@echo off
cd bin
java test.ExprEvalTest ..\testcases\simple.xml
cd ..
pause
@echo on
```

---

其中，**\testcases\simple.xml** 是存放测试用例的 XML 文档的文件名字（实验软装置将所有的测试用例都组织在子目录**\testcases** 中）。若要为自己设计的测试用例编写一个启动运行的脚本，只需替换该 XML 文档的文件名即可。例如，**test\_standard.bat** 脚本的内容如下：

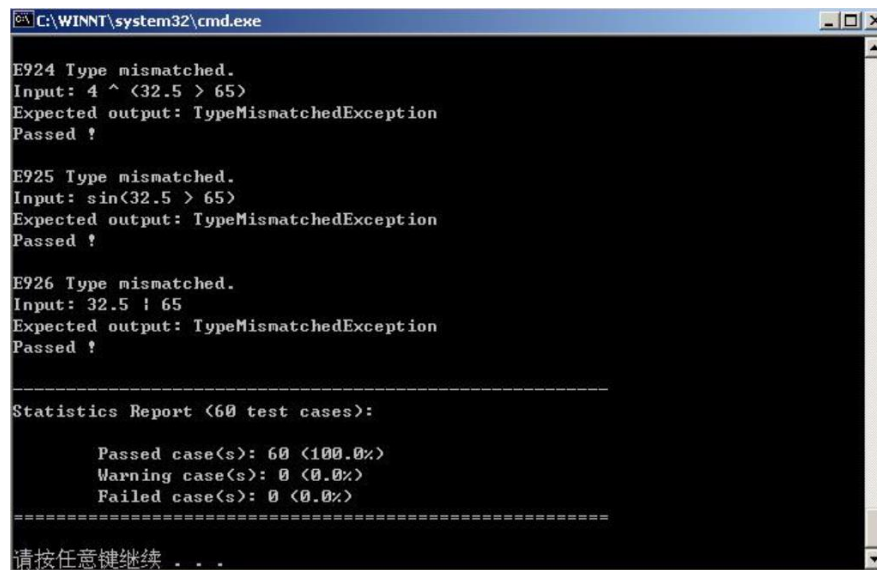
---

```
@echo off
cd bin
java test.ExprEvalTest ..\testcases\standard.xml > ..\testcases\report.txt
cd ..
type testcases\report.txt
pause
del testcases\report.txt
@echo on
```

---

注意，该脚本中使用了输出重定向机制，将测试结果存放在**\testcases\report.txt** 文本文件中，如果想重用测试结果，则不按任一键结束命令行窗口，而是直接用鼠标关闭窗口或按 **Ctrl+C** 中止，则该文本文件会被保留以备后用。

脚本命令中运行的 Java 主程序是 **test.ExprEvalTest**，该程序从指定的 XML 文档读取测试用例数据，将测试用例中的输入表达式作为参数传递给 **parser.Calculator** 类的 **calculate()** 方法，取得表达式求值的返回结果或可能抛出的异常后，与测试用例中的预期输出进行比较，并完成测试统计报告，如图 7 所示。



```
C:\WINNT\system32\cmd.exe

E924 Type mismatched.
Input: 4 ^ (32.5 > 65)
Expected output: TypeMismatchedException
Passed !

E925 Type mismatched.
Input: sin(32.5 > 65)
Expected output: TypeMismatchedException
Passed !

E926 Type mismatched.
Input: 32.5 ! 65
Expected output: TypeMismatchedException
Passed !

-----
Statistics Report (60 test cases):
    Passed case(s): 60 (100.0%)
    Warning case(s): 0 (0.0%)
    Failed case(s): 0 (0.0%)
=====
请按任意键继续 . . .
```

图 7. 实验软装置的自动回归测试报告

### 3.5.2 编写测试用例

实验软装置运行一次回归测试的所有测试用例均存放在一个 XML 文档中，该 XML 文档的根节点使用标记<test-case-definitions>和</test-case-definitions>标识所有测试用例，在根节点下每一测试用例的结点使用标记<test-case>和</test-case>括住该测试用例的所有数据。

每一测试用例可由以下标记描述：

- 标记<id>和</id>之间的内容描述该测试用例的编号。
- 标记<description>和</description>之间的内容是对该测试用例的文本描述和说明。
- 标记<input>和</input>之间的内容是指定的输入表达式。

● 当输入表达式不存在词法、语法或语义错误时，标记<output>和</output>中的内容指明输入该表达式后的预期输出；若输入表达式存在错误，则标记<exception>和</exception>之间的内容指明预期抛出的异常类名。

注意，在每一测试用例中标记<output>和<exception>只能出现其中的一种，因为一个测试用例不会出现两种不同的预期结果（包括异常亦只会抛出一种类型的异常）。此外，上述所有标记之间的内容不要用换行符分隔；如果将输入表达式写在多行中，测试程序将不能正确地识别测试用例中的输入数据和预期输出结果，因为回车、换行都不是 ExprEval 中合法的分隔符。

例如，以下是一个书写格式正确的测试用例定义：

---

```
<test-case>
  <id>C001</id>
  <description>A simple expression.</description>
  <input>9 - 3 * 2</input>
  <output>3</output>
</test-case>
```

---

其中，测试用例的编号是 **C001**（实验软装置中以前缀 C 的编号表示预期输出为正常的求值结果,以前缀 E 的编号表示预期输出为某一异常），其描述为 *A simple expression.*，输入表达式为 **9-3\*2**，预期输出结果为 **3**。

又如，以下也是一个书写格式正确的测试用例定义：

---

```
<test-case>
  <id>E012</id>
  <description>Right parenthesis expected.</description>
  <input>((2 + 3) ^ ((3 - 1) + 1)</input>
  <exception>MissingRightParenthesisException</exception>
</test-case>
```

---

其中，测试用例的编号是 **E012**，输入表达式为 **((2 + 3) ^ ((3 - 1) + 1)**，预期的输出结果是抛出一个异常，其类型为 **MissingRightParenthesisException**。

特别注意，由于 <、>、/、?、&、**false** 等输入表达式中的运算符在 XML 文档中有特殊的含义，因而如果输入表达式中包含这些符号则可能需要特殊处理。一种简单的方法是 **<![CDATA[** 和 **]]>** 将输入表达式括起来(欲进一步了解 XML 格式,可上网查找 XML 规范)，并与 **<input>** 和 **</input>** 标记写在同一行内。例如，以下是一个书写格式正确的测试用例：

---

```
<test-case>
  <id>C004</id>
  <description>Expression with relational and logical operations.</description>
  <input><![CDATA[(5 > 3) & (4 < 8) ? 15 : 16]]></input>
  <output>15</output>
</test-case>
```

---

如果上例中不使用 **<![CDATA[** 和 **]]>** 将输入表达式括起来，则在运行测试脚本时可能看到 XML 解析器抛出的如下异常信息：

---

```
[Fatal Error] ..%5Ctestcases%5Ccomplete.xml:338:20: The content of elements must
  consist of well-formed character data or markup.
...
XML parser encounters an error: org.xml.sax.SAXParseException: The content of el
ements must consist of well-formed character data or markup.
```

---

其中，338 指明 XML 文档中可能存在问题的行号。

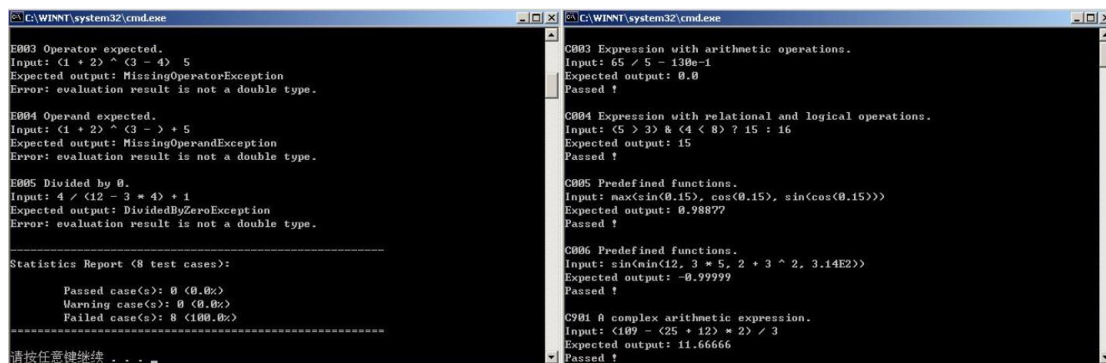
### 3.5.3 测试结果分析

如果输入表达式是一个正确的表达式,则测试输出结果应与预期输出完全相同;对于计算结果为实数的情况,测试程序仅比较到小数点后 4 位的精度,意即你编写设计用例时只须在预期输出中写到小数点后 4 位即可。如果此时测试输出结果与预期输出不同,则测试分析会报告一个严重错误 (Error)。

如果输入表达式中存在词法、语法、或语义错误,则测试输出结果抛出的异常应与预期输出的异常相同类型。否则,实验软装置将根据以下原则判定测试结果:

- 如果测试输出结果未抛出任何异常,而是给出某一个计算结果,则测试程序会报告一个严重错误 (Error)。
- 如果测试输出结果抛出了一个异常,但该异常不属于 **ExpressionException** 的派生类,则测试分析也会报告一个严重错误 (Error)。
- 如果测试输出结果抛出了一个异常,但异常类型与预期输出的异常类型不相同,则测试分析会报告一个警告错误 (Warning),因为对某一类型错误的判定可能会持不同观点。

如果测试分析未报告任何严重错误或警告错误,则显示 Passed 表示通过该测试用例的测试。测试程序报告每一测试用例的测试结果的运行画面如图 8 所示。



```
E003 Operator expected.
Input: (1 + 2) ^ (3 - 4) 5
Expected output: MissingOperatorException
Error: evaluation result is not a double type.

E004 Operand expected.
Input: (1 + 2) ^ (3 - ) + 5
Expected output: MissingOperandException
Error: evaluation result is not a double type.

E005 Divided by 0.
Input: 4 / (12 - 3 * 4) + 1
Expected output: DividedByZeroException
Error: evaluation result is not a double type.

-----
Statistics Report (8 test cases):
    Passed case(s): 0 (0.0%)
    Warning case(s): 0 (0.0%)
    Failed case(s): 8 (100.0%)
-----
请按任意键继续 . . .
```

```
C003 Expression with arithmetic operations.
Input: 65 / 5 - 130e-1
Expected output: 0.0
Passed !

C004 Expression with relational and logical operations.
Input: (5 > 3) & (4 < 8) ? 15 : 16
Expected output: 15
Passed !

C005 Predefined functions.
Input: max(sin(0.15), cos(0.15), sin(cos(0.15)))
Expected output: 0.98877
Passed !

C006 Predefined functions.
Input: sin(min(12, 3 * 5, 2 + 3 ^ 2, 3.14E2))
Expected output: -0.99999
Passed !

C901 A complex arithmetic expression.
Input: (189 - (25 + 12) * 2) / 3
Expected output: 11.66666
Passed !
```

图 8. 实验软装置的自动测试画面

### 3.5.4 一个完整的测试用例 XML 文档例子

一个具体的测试用例 XML 文档例子可参见\testcases\simple.xml, 其内容如下所示:

```
<?xml version="1.0"?>
<!-- Standard test cases for EvalExpr -->
<test-case-definitions>
```

<test-case>

<id>C001</id>

<description>A simple expression.</description>

<input>9 - 3 \* 2</input>

<output>3</output>

</test-case>

<test-case>

<id>C002</id>

<description>Expression with arithmetic operations.</description>

<input>2.25E+2 - (55.5 + 4 \* (10 / 2) ^ 2)</input>

<output>69.5</output>

</test-case>

<test-case>

<id>C003</id>

<description>Expression with arithmetic operations.</description>

<input>65 / 5 - 130e-1</input>

<output>0.0</output>

</test-case>

<test-case>

<id>C004</id>

<description>Expression with relational and logical operations.</description>

<input><![CDATA[(5 > 3) & (4 < 8) ? 15 : 16]]></input>

<output>15</output>

</test-case>

<test-case>

<id>C005</id>

<description>Predefined functions.</description>

<input>max(sin(0.15), cos(0.15), sin(cos(0.15)))</input>

<output>0.98877</output>

</test-case>

```
<test-case>
  <id>C006</id>
  <description>Predefined functions.</description>
  <input>sin(min(12, 3 * 5, 2 + 3 ^ 2, 3.14E2))</input>
  <output>-0.99999</output>
</test-case>

<test-case>
  <id>E001</id>
  <description>Left parenthesis expected.</description>
  <input>(2 + 3) ^ 3) - ((1 + 1)</input>
  <exception>MissingLeftParenthesisException</exception>
</test-case>

<test-case>
  <id>E002</id>
  <description>Right parenthesis expected.</description>
  <input>((2 + 3) ^ ((3 - 1) + 1)</input>
  <exception>MissingRightParenthesisException</exception>
</test-case>

<test-case>
  <id>E003</id>
  <description>Operator expected.</description>
  <input>(1 + 2) ^ (3 - 4) 5</input>
  <exception>MissingOperatorException</exception>
</test-case>

<test-case>
  <id>E004</id>
  <description>Operand expected.</description>
  <input>(1 + 2) ^ (3 - ) + 5</input>
  <exception>MissingOperandException</exception>
</test-case>
```



```
<test-case>
  <id>E005</id>
  <description>Divided by 0.</description>
  <input>4 / (12 - 3 * 4) + 1</input>
  <exception>DividedByZeroException</exception>
</test-case>
```

```
<test-case>
  <id>E006</id>
  <description>Type mismatched.</description>
  <input><![CDATA[(13 < 2 * 5) + 12]]></input>
  <exception>TypeMismatchedException</exception>
</test-case>
```

```
<test-case>
  <id>E007</id>
  <description>Scientific Notation Error.</description>
  <input>4 + 10.E+5 + 1</input>
  <exception>IllegalDecimalException</exception>
</test-case>
```

```
<test-case>
  <id>E008</id>
  <description>Not a predefined identifier.</description>
  <input>4 + mix(5, 2) + 1</input>
  <exception>IllegalIdentifierException</exception>
</test-case>
```

```
<test-case>
  <id>E009</id>
  <description>Function call error.</description>
  <input>sin(2, 1)</input>
  <exception>FunctionCallException</exception>
</test-case>
```

```
<test-case>
  <id>E010</id>
  <description>Function call error.</description>
  <input>min(2.5)</input>
  <exception>MissingOperandException</exception>
</test-case>

</test-case-definitions>
```

---

---