基于表达式的计算器 EXPR-Eval

20337167 阿依那西

一、实验环境

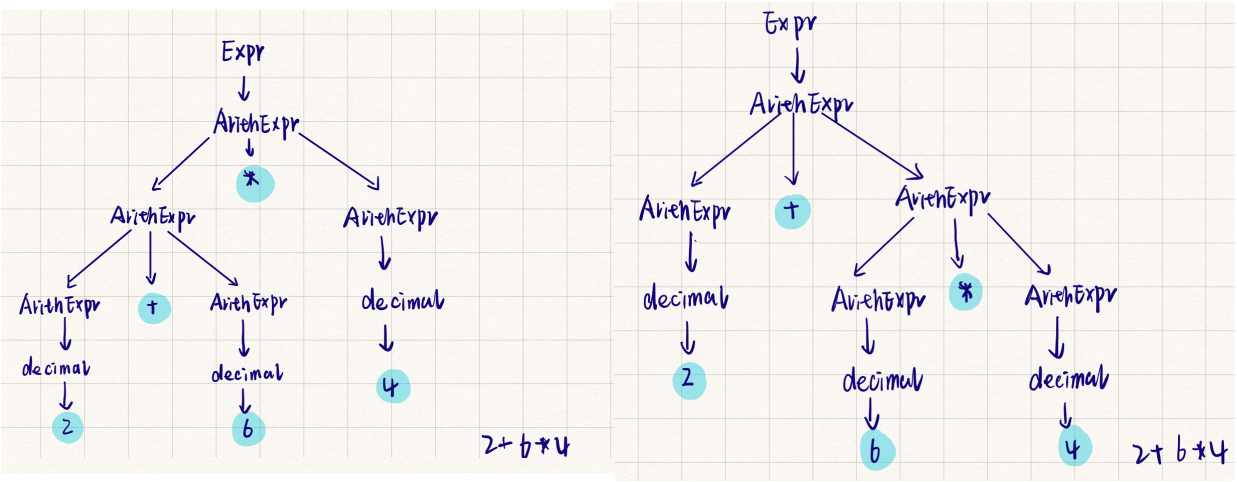
- 编程语言：Java JDK1.8
- 开发工具：Eclipse
- 实验软装置

二、实验内容

2.1 讨论语法定义的二义性

根据实验提供的BNF，举个报告中$2+6*4$的例子

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



显然语法是有二义性的,这是因为decimal间没有定义结合的优先级，这样就会导致生成不同的两颗语法树。

2.2 设计并实现词法分析程序

在本实验中，我定义了一个主类Token，用来做为其他词法单元的父类，再此基础上，将词法单元细分为以下五大类，分别是：

> 1. *数值类型的常量[包含小数以及科学技术法]*
> 2. *运算符*
>
> *Sign := [+-*/^()!&|?:>=<]*

> 3. *布尔类型的常量*
>   *Bool := [true|false]*
> 4. *标点符号*
>   *Sign := [,]*
> 5. *预定义运算符*
>   *Func := max|min|sin|cos*

- **如何对单词进行分类?**

以上分类运算符和预定义运算符属于同一类外，其余的符号各自为类。

- **如何处理对预定义函数名和布尔常量的识别?**

在词法分析器scanner中，当程序扫描到字符s，c或者m时，便会向后预读两个字符，以此来判断是否是预定义函数名，如果是，将该字符划分为OperatorToken()并返回该词法单元，否则，抛出错误。

同理，当程序扫描到字符t（或者f）时，便会向后预读三个（或者四个）字符，以此来判断是否是布尔常量。由此就能区分开预定义函数名和布尔常量。

- **如何处理科学计数法表示的数值常量?**

当程序读取到数字时，就会向后扫描，通过判断小数点以及E等的关系，直至判断到读完整个数值字符串，就将其新建DecimalToken的词法单元。

在DecimalTokenl类的构造函数中，根据E的位置，将科学计数法常量划分为数值和指数部分，并将其转化为对应的普通数值常量后存储起来。

- **如何处理字符串的边界?**

当程序读取完整个字符串后，再读取下一个词法单元时，就会返回一个DollarToken类的词法单元，用来标记以及读完整个表达式字符串。

2.3 构造算符优先关系表

由于本实验提供的BNF存在二义性，所以必须使用一种算法来使得文法满足无二义性的条件，而按照实验要求，通过定义算符（即终结符)的优先级和结合性来解决二义性冲突。故我们构造算符优先关系表OOP。

优先级和结合性质的定义如下:

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

OPP表可以理解为就是一张移入-归约表，通过比较两个运算符之间的优先级，来决定是执行移入，还是归约。规则为：维护一个操作符堆栈:如果栈顶符号优先级大于读入符号，或者两者优先级相同但符号是左结合的，那么就对栈顶符号进行归约;如果栈顶符号优先级小于读入符号，或者两者优先级相同但符号是右结合的，那么就将读入符号进行移入,为了使函数功能高效，本程序通过tag的值将归约分为以下几类：

- 一元取负和二元减号

对于减号和负号，在scanner阶段获取token就进行判断，记录下前一个获取到token的类型，若前一个token为`decimal`或`)`，则判定-为减号，否则判定为负号。

- 三元运算符和其他操作符

由于三元运算符优先级最低，所以，只有在当‘：’遇到‘$’时，才会进行归约，其他时候分情况移入或者抛出异常。

- 预定义函数

预定义函数名之后必须紧跟着括号，所以只有遇到‘（’才是合法的，将其移入。而遇到其他运算符都应该抛出相应的异常。

2.4 设计并实现语法分析和语义处理程序

关于OPP的运作，需要以下：

- 栈：存放当前已经读取但未处理的token
- 输入队列：存放未读取的token
- 算符优先关系表：每次操作需要取出栈顶的元素和输入队列首的元素，根据算符优先关系表，进行相应操作。

栈中用于存放当前已经读取但未处理的token，输入队列中为未读取的token，每一次output为输出的最简短语，即规约的需要处理的极简表达式。

下面为parser的主程序，根据输入的表达式的token类型查表进行shift、reduce、报错等操作

```
public double parse() throws ExpressionException {
        lookahead = new Terminal(scanner.getNextToken());
        while (true) {
            Terminal topMostTerminal = getTopMostTerminal();
            switch (OPPTable.table[topMostTerminal.getTag()]
[lookahead.getTag()]) {
            case 0:
                shift();
                break;
            case 1:
                reduce();
                break;
            case 2:
                return accept();
            case -1:
                throw new MissingOperatorException();
            case -2:
                throw new MissingRightParenthesisException();
            case -3:
                throw new MissingLeftParenthesisException();
            case -4:
                throw new FunctionCallException();
            case -5:
                throw new TypeMismatchedException();
            case -6:
                throw new MissingOperandException();
            case -7:
                throw new TrinaryOperationException();
            }
        }
    }
}
```

2.4.1shift 操作

shift操作表现为：将输入队列首的元素放置于栈顶，并删除队列首的元素。表示当前的token目前不需要处理，等待后续元素放入

```
private void shift() throws ExpressionException{
    stack.add(lookahead);
    lookahead = new Terminal(scanner.getNextToken());
}
```

2.4.2accept操作

```
private double accept() throws TypeMismatchedException {
    Expr k = stack.get(stack.size() - 1);
    if (k.getTag() == Scanner.kingOfChar.BoolExpr) {
        System.out.println(((BoolExpr) k).getValue());
        throw new TypeMismatchedException();
    }
    return ((ArithExpr) k).getValue();
}*
```

2.4.3reduce操作

reduce根据tag情况调用不同类型的reduce，实现如下：

```
private void reduce() throws ExpressionException {
    Terminal topMostTerminal = getTopMostTerminal();
    switch (topMostTerminal.getTag()) {
        case Scanner.kindOfChar.NUM:
            reducer.numReducer(stack);
            break;
        case Scanner.kindOfChar.BOOL:
            reducer.boolReducer(stack);
            break;
        case Scanner.kindOfChar.ADDSUB:
            reducer.add_subReducer(stack);
            break;
       ...
        default:
            System.out.println(topMostTerminal.getTag());
            break;
    }
}
```

- 对tag为NEG的TopMostTerminal进行reduce

```
public void reduceNEG(ArrayList<Expr> stack) throws ExpressionException {
        int peekIndex = stack.size() - 1;
        Expr peek = stack.get(peekIndex);
        if (peek.getTag() != Tag.ArithExpr) throw new
TypeMismatchedException();
        double v = ((DecimalToken)peek.token).getValue();
        ArithExpr arithExpr = new ArithExpr(-v);
        stack.remove(peekIndex);
        stack.remove(peekIndex -1);
        stack.add(arithExpr);
        return;
    }
```

- 对tag为AND或OR的TopMostTerminal进行reduce

```
public void reduceANDOR(ArrayList<Expr> stack) throws ExpressionException {
        int peekIndex = stack.size()-1;
        Expr a = stack.get(peekIndex-2);
        Expr o = stack.get(peekIndex-1);
        Expr b = stack.get(peekIndex);
        if((b.getTag() == Tag.AND) || (b.getTag() == Tag.OR)) throw new
MissingOperandException();
        if(a.getTag() != Tag.BoolExpr|| b.getTag() != Tag.BoolExpr)
            throw new TypeMismatchedException();
        BoolExpr be;
        if(o.getTag() == Tag.AND)
            be=new BoolExpr(((BoolExpr)a).getValue() &&
((BoolExpr)b).getValue());
        else
            be=new BoolExpr(((BoolExpr)a).getValue() ||
((BoolExpr)b).getValue());
        stack.remove(peekIndex);
        stack.remove(peekIndex - 1);
        stack.remove(peekIndex - 2);
        stack.add(be);
        return;
    }
```

- 对tag为BOOL的TopMostTerminal进行reduce

```
public void reduceBOOL(ArrayList<Expr> stack) throws ExpressionException {
        int peekIndex = stack.size() - 1;
        Terminal t = (Terminal)stack.get(peekIndex);
        BoolExpr b = new BoolExpr((BooleanToken)t.token);
        stack.remove(peekIndex);
        stack.add(b);
        return;
    }
```

- 对tag为COLON的TopMostTerminal进行reduce,三元运算符则先判断布尔表达式，合法后根据布尔值删除其余的栈元素

```
public void reduceCOLON(ArrayList<Expr> stack) throws ExpressionException {
        int peekIndex = stack.size() - 1;    // be? ae1 : ae2
        Expr a = stack.get(peekIndex);  // ae2
        Expr b = stack.get(peekIndex - 1);  // :
        Expr c = stack.get(peekIndex - 2);  // ae1
        Expr d = stack.get(peekIndex - 3);  // ?
        Expr e = stack.get(peekIndex - 4);  //be

        if(a.getTag() != Tag.ArithExpr || c.getTag() != Tag.ArithExpr ||
e.getTag() != Tag.BoolExpr)
            throw new TypeMismatchedException();
        if(d.getTag() != Tag.QM)
            throw new exceptions.TrinaryOperationException();
        if(((BoolExpr)e).getValue() == true)
        {
            //  keep ae1
            stack.remove(peekIndex);
            stack.remove(peekIndex - 1);
            stack.remove(peekIndex - 3);
            stack.remove(peekIndex - 4);
            return;
        }
        else
        {
            //  keep ae2
            stack.remove(peekIndex - 1);
            stack.remove(peekIndex - 2);
            stack.remove(peekIndex - 3);
            stack.remove(peekIndex - 4);
            return;
        }
    }
```

- **对tag为NUM的TopMostTerminal进行reduce,数字的reduce较简单，只是转换格式后加入到栈中**

```
public void numReducer(ArrayList<Expr> stack) {
    int i = stack.size() - 1;
    Terminal k = (Terminal) stack.get(i);
    ArithExpr arithExpr = new ArithExpr((Decimal) k.token);
    stack.remove(i);
    stack.add(arithExpr);
}
```

- **对tag为NEG的TopMostTerminal进行reduce,取负则将下一个token的值取相反数入栈，取非同理，不赘述**

```
public void negReducer(ArrayList<Expr> stack) throws
ExpressionException {
    int i = stack.size() - 1;
    Expr k = stack.get(i);
    if (k.getTag() != Scanner.kindOfChar.ArithExpr)
        throw new TypeMismatchedException();
    double v = ((Decimal) k.token).getValue();
    ArithExpr arithExpr = new ArithExpr(-v);
    stack.remove(i);
    stack.remove(i - 1);
    stack.add(arithExpr);
}
```

- 对于加减、乘除、与或操作则取前一个和后一个token进行相应计算，代码形式相同，下面以乘除为例例举代码:

  对tag为MULDIV的TopMostTerminal进行reduce，取栈中三个token，判断合法后获得结果后删除原token并将结果压栈

```
public void mul_divReducer(ArrayList<Expr> stack) throws
ExpressionException {
    int i = stack.size() - 1;
    Expr a = stack.get(i - 2);
    Expr o = stack.get(i - 1);
    Expr b = stack.get(i);
    if ((b.getTag() == Scanner.kindOfChar.MULDIV))
        throw new MissingOperandException();
    if (a.getTag() != Scanner.kindOfChar.ArithExpr || b.getTag() !=
Scanner.kindOfChar.ArithExpr)
        throw new TypeMismatchedException();
    ArithExpr ae;
    if (o.token.getString().equals("*"))
```

```
            ae = new ArithExpr(((ArithExpr) a).getValue() * ((ArithExpr)
b).getValue());
        else if (((ArithExpr) b).getValue() == 0)
            throw new DividedByZeroException();
        else
            ae = new ArithExpr(((ArithExpr) a).getValue() / ((ArithExpr)
b).getValue());
        stack.remove(i);
        stack.remove(i - 1);
        stack.remove(i - 2);
        stack.add(ae);
    }
```

- 对tag为RE的TopMostTerminal进行reduce，即关系运算，则判断合法后根据关系运算调用其他函数判断最后将结果压栈

```
public void reduceRE(ArrayList<Expr> stack) throws ExpressionException
{
            int peekIndex = stack.size() - 1;
            Expr a = stack.get(peekIndex - 2);
            Expr o = stack.get(peekIndex - 1);
            Expr b = stack.get(peekIndex);
            if(b.getTag() == Tag.RE) throw new MissingOperandException();
            if(a.getTag() != Tag.ArithExpr || b.getTag() != Tag.ArithExpr)
                throw new TypeMismatchedException();
            BoolExpr be = null;
            switch(((OperatorToken)o.token).getValueofString())
            {
                case "=":
                    be = new BoolExpr(((ArithExpr)a).getValue() ==
((ArithExpr)b).getValue());
                    break;
                case "<>":
                    be=new BoolExpr(((ArithExpr)a).getValue() !=
((ArithExpr)b).getValue());
                    break;
                case "<=":
                    be=new BoolExpr(((ArithExpr)a).getValue() <=
((ArithExpr)b).getValue());
                    break;
                case "<":
                    be=new BoolExpr(((ArithExpr)a).getValue() <
((ArithExpr)b).getValue());
                    break;
                case ">=":
```

```
                be=new BoolExpr(((ArithExpr)a).getValue() >=
((ArithExpr)b).getValue());
                break;
            case ">":
                be=new BoolExpr(((ArithExpr)a).getValue() >
((ArithExpr)b).getValue());
                break;
        }
        stack.remove(peekIndex);
        stack.remove(peekIndex - 1);
        stack.remove(peekIndex - 2);
        stack.add(be);
        return;
    }
```

- 对tag为RP的TopMostTerminal进行reduce，即右括号的reduce相对复杂些，若是为了保证计算优先级的括号则检查后删除即可，若是功能函数的括号则需要根据功能函数的种类取相应token进行计算后将结果压栈

```
public void reduceRP(ArrayList<Expr> stack) throws ExpressionException
{
        int peekIndex = stack.size() - 1;

        Expr a= stack.get(peekIndex);  // )
        Expr o= stack.get(peekIndex - 1); //bool or ae or ael, error
if operand
        Expr b= stack.get(peekIndex - 2); // ( or ,
        Expr c= stack.get(peekIndex - 3);//$ or func or ArithExpr

        if(o.getTag() ==Tag.RE || o.getTag() ==Tag.COMMA)
        {
            throw new MissingOperandException();
        }
        else if(o.getTag() == Tag.BoolExpr) // (be) to be
        {
            stack.remove(peekIndex);
            stack.remove(peekIndex - 2);
            return;
        }
        else if(o.getTag() == Tag.ArithExprList)
        {
            if(c.getTag() != Tag.ArithExpr) throw new
TypeMismatchedException();
            ArithExprList ael=new ArithExprList((ArithExpr)c,
(ArithExprList)o);
            Expr d= stack.get(peekIndex - 4);
```

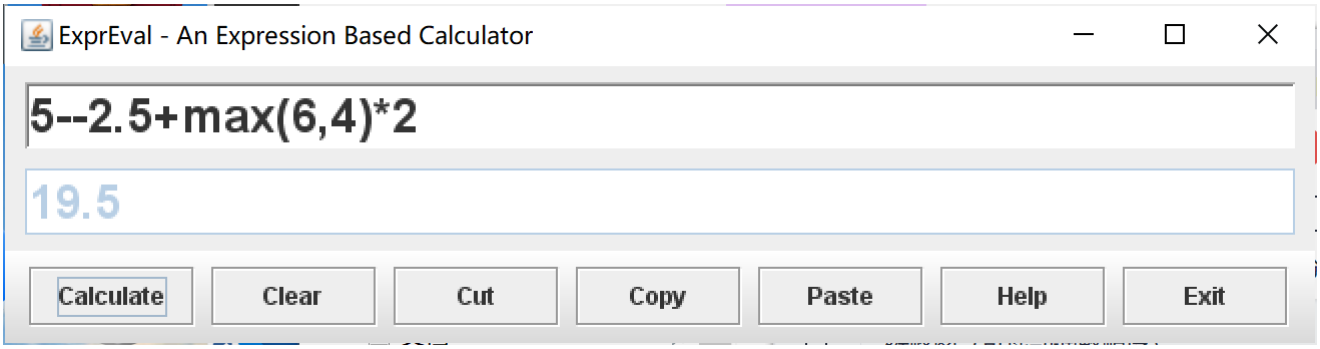
```
            Expr f = stack.get(peekIndex - 5);
            ArithExpr ae=null;
            if(d.getTag() == Tag.LP && f.getTag() == Tag.FUNC) //
max(ae,ael) to ae
            {
                switch(((OperatorToken)(f.token)).getValueofString())
                {
                    case "max":
                        ae=new ArithExpr(ael.max); break;
                    case "min":
                        ae=new ArithExpr(ael.min); break;
                    default:
                        throw new FunctionCallException();
                }
                stack.remove(peekIndex);  //)
                stack.remove(peekIndex - 1);    //  ael
                stack.remove(peekIndex - 2);    //  ,
                stack.remove(peekIndex - 3);    //  ae
                stack.remove(peekIndex - 4);    //  (
                stack.remove(peekIndex - 5);    //  max min
                stack.add(ae);
                return;
            }
            else                            // ae,ael) to ael)
            {
                stack.remove(peekIndex - 1);
                stack.remove(peekIndex - 2);
                stack.set(peekIndex - 3, ael);
                return;
            }
        }
        else if(o.getTag() == Tag.ArithExpr)
        {
            if(b.getTag() == Tag.COMMA) // ae,ae) to ae,ael)
            {
                ArithExprList ael = new ArithExprList((ArithExpr)o);
                stack.set(peekIndex - 1, ael);
                return;
            }

            else if(c.getTag() == Tag.FUNC) //sin(ae)
            {
                if(o.getTag() != Tag.ArithExpr) throw new
TypeMismatchedException();
                ArithExpr ae=null;
                switch(((OperatorToken)(c.token)).getValueofString())
```

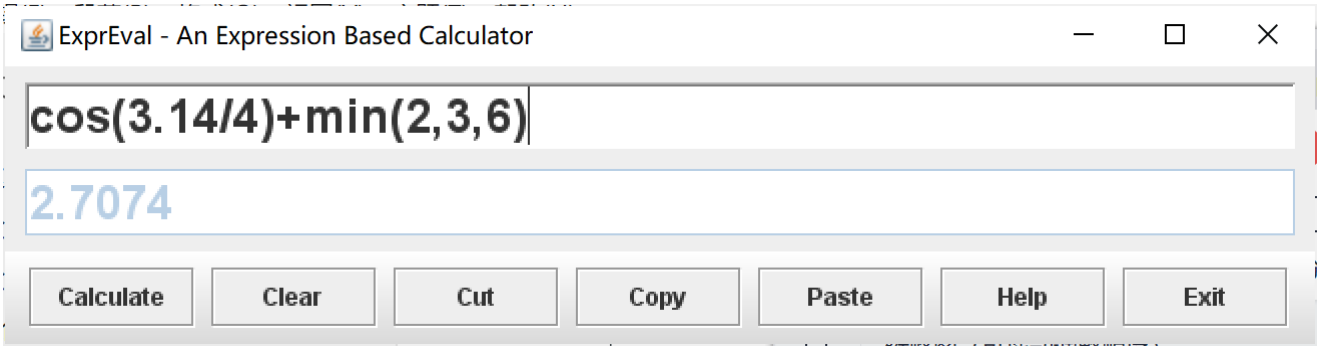
```
                {
                    case "sin":
                        ae=new
ArithExpr(Math.sin(((ArithExpr)o).getValue()));
                        break;
                    case "cos":
                        ae=new
ArithExpr(Math.cos(((ArithExpr)o).getValue()));
                        break;
                    default:
                        throw new MissingOperandException();
                }
                stack.remove(peekIndex);
                stack.remove(peekIndex - 1);
                stack.remove(peekIndex - 2);
                stack.remove(peekIndex - 3);
                stack.add(ae);
                return;
            }
            else //(ae) to ae
            {
                System.out.println(1);
                stack.remove(peekIndex);
                stack.remove(peekIndex - 2);
                return;
            }
        }
        throw new SyntacticException();
    }
}
```

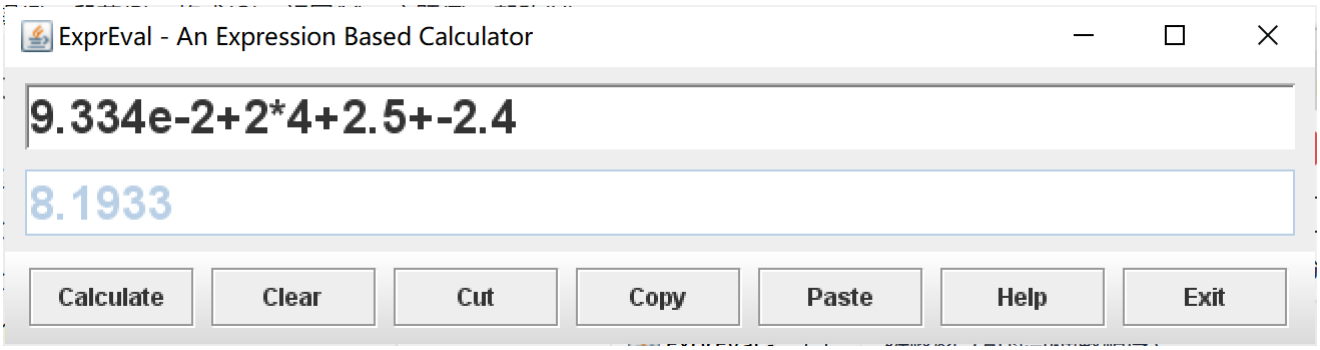
> *...具体代码请看Reducer.java*

2.5 实验测试

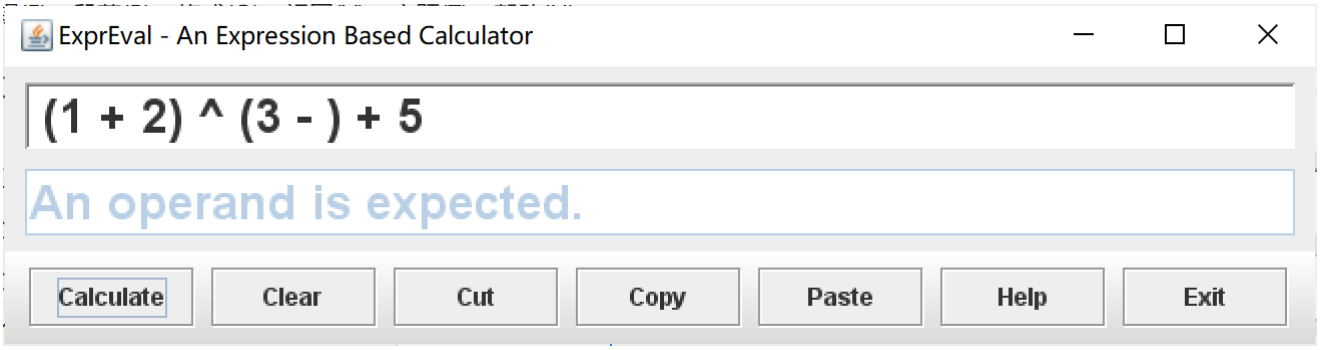
- 手工测试

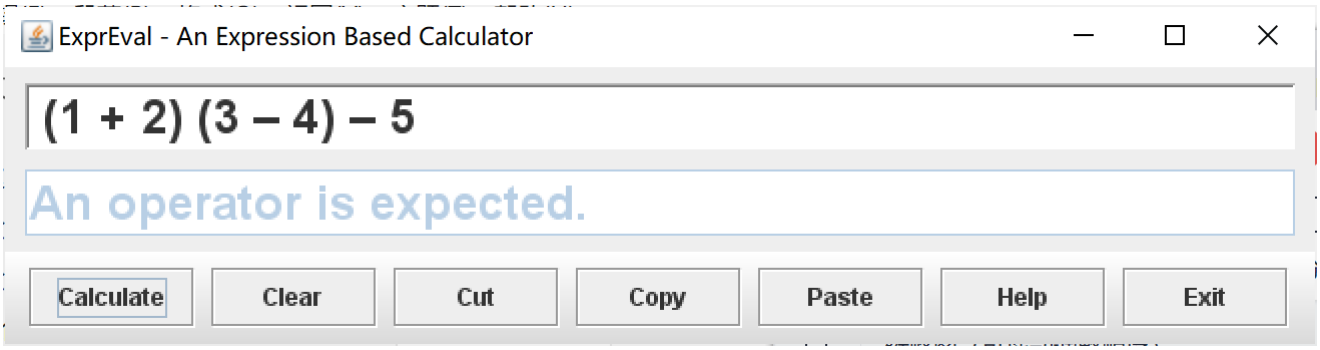


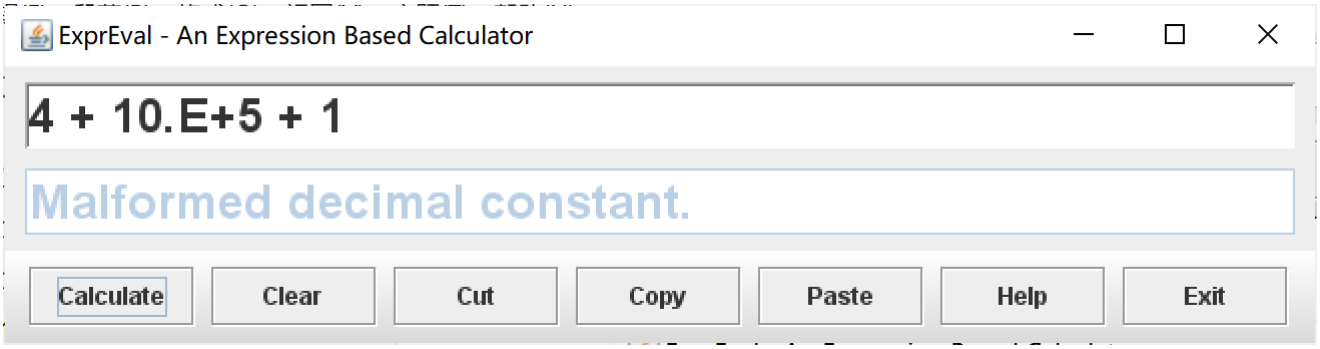




以上结果均正确，下面看异常







- 简单测试

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

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

- 标准测试

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

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

请按任意键继续
```

- 个人回归测试

```
------------------------------------------------------
Statistics Report (13 test cases):

        Passed case(s): 13 (100.0%)
        Warning case(s): 0 (0.0%)
        Failed case(s): 0 (0.0%)
======================================================

请按任意键继续. . .
```

2.6 实验心得

通过这一次的实验，我对编译原理中的词法分析、语法分析、语义分析等重要环节有了更深层次的理解，我认为本次实验的难点在于优先级关系表的构建，scanner的处理，以及parser的规约等，以及debug花费了大量的时间，但总体来说收获满满，对于Java语言面向对象的设计有了更加深刻的理解，希望在之后的实验中再接再厉！