实验内容

1. 讨论 BNF 定义的文法是否存在二义性：

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 s 式中的二义性。

Answer：

（1）二义性文法：

所谓的二义性是指，在一个定义好的文法中，存在某个特定的输入，可以产生出两棵不同的语法分析树，这样的语法，我们称之为存在二义性的语法。

例如表达式串 34 - 3 * 42，可以有两种不同的分析树：
34 - 3 = 31, 31 * 42
3 * 42 = 126, 34 - 126

当我们定义语法制导时候，就会产生不同的结果，这是我们不希望得到的。

（2）巴科斯范式(BNF)
在双引号中的字("word")代表着这些字符本身。而 double_quote 用来代表双引号。
在双引号外的字（有可能有下划线）代表着语法部分。
尖括号(< >)内包含的为必选项。
方括号([ ])内包含的为可选项。
大括号({ })内包含的为可重复 0 至无数次的项。
竖线(|)表示在其左右两边任选一项，相当于"OR"的意思。
::= 是“被定义为”的意思。

Example:

FOR_STATEMENT ::=
"for" "(" (variable_declaration |
(expression ";") | ";")
[expression] ";"
[expression]
")" statement

（3）**如果单纯定义 BNF 文法，则依然存在二义性：**

例如表达式串 34 - 3 * 42，在未定义优先级和结合性的时候仍然存在二义性，如上述例子所示，存在两棵语法树。

（4）消除二义性的方法：

A) 强制将将输入转换为单一语法功能，如上述例子: 34 - 3 * 42 可用此强制方法 34 - (3*42)，这样对于本实验单独定义的文法，不会产生二义性。

B) 定义优先级和结合性（如实验说明 2.3.2）

对于语法的每一个运算（会产生二义性的运算），定义其优先级和结合性，使得产生式在选择语法规则的时候不具有多重选择，从而消除二义性。

（补充：在 flex 中，定义优先级巧妙用到”栈”的特点，也就是越往下的优先级越高的原因，因为在回溯的过程中会被先调用。）

（5）结论：

单纯的定义 BNF，可以在一定程度上使得语法更加规范，从而减少二义性。但在真实情况下，必须还要定义文法所对应的优先级和结合性才能够消除二义性。

（6）问题：实验文档中定义的不是 BNF

维基百科对于 BNF 形式上的定义：

Introduction [edit]

A BNF specification is a set of derivation rules, written as

```
<symbol> ::= __expression__
```

where <symbol>[6] is a *nonterminal*, and the __expression__ consists of one or more sequences of symbols; more sequences are separated by the vertical bar, '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are *terminals*. On the other hand, symbols that appear on a left side are *non-terminals* and are always enclosed between the pair <>.[6]

The '::=' means that the symbol on the left must be replaced with the expression on the right.

Example：

Example [edit]

As an example, consider this possible BNF for a U.S. postal address:

```
<postal-address> ::= <name-part> <street-address> <zip-part>

     <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                   | <personal-part> <name-part>

 <personal-part> ::= <initial> "." | <first-name>

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

      <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
   <opt-apt-num> ::= <apt-num> | ""
```

而本实验装置是纯粹的产生式定义文法：

2.3.1 表达式的 BNF 定义

ExprEval的表达式规格说明如下列BNF所示：

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

2.设计并实现词法分析程序

在进行词法分析的时候，使用的是正则表达式文法。通过自左向右逐个扫描字符串中的字符，匹配并且生成一个序列（token 对象序列）供文法分析器进行文法分析。

首先识别函数先识别当前词素单元的第一个字符，以此来判定词素的类型。我们通过一个 match 函数来实现。

然后，我们再继续识别后面的内容。

(1) 对于运算符，预定义函数，括号等：

这一类符号较为简单，都是通过连接组合而成的词素单元，我们通过一个一个识别得到。

如识别 sin:

```
private boolean matchFunction_sin() {
    if(inputString.charAt(lookahead) == 's') {
        lookahead++;
        if(inputString.charAt(lookahead) == 'i') lookahead++; else return false;
        if(inputString.charAt(lookahead) == 'n') lookahead++; else return false;
        return true;
    }
    return false;
}
```

将结果直接识别出来。

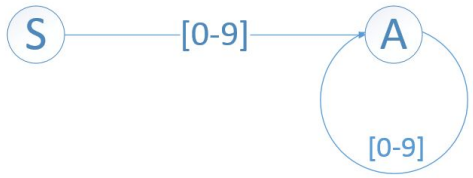
(2) 对于 Decimal 类型识别

根据下列文法规则构造 DFA（注，以下自动机为 NFA 化简到 DFA 的最终结果）

digit	→	**0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
integral	→	*digit* $^+$
fraction	→	**.** *integral*
exponent	→	**(E | e)** (+ | − | ε) *integral*
decimal	→	*integral* (*fraction* | ε) (*exponent* | ε)

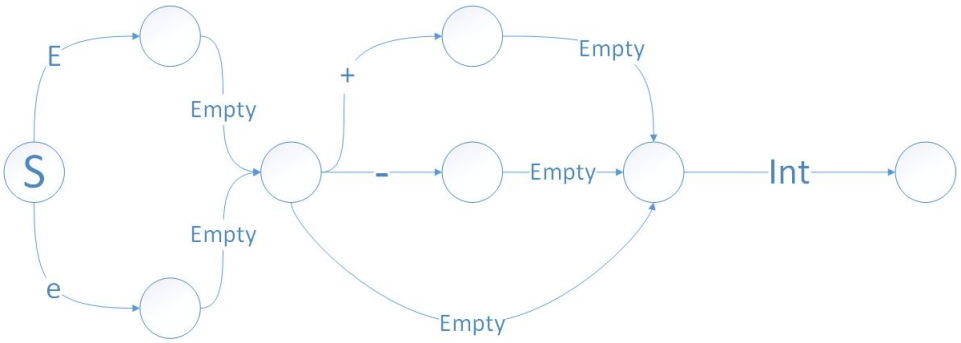
Digit 的自动机：

Intergral 的自动机：



Exponent 的自动机：

（NFA）



（DFA）

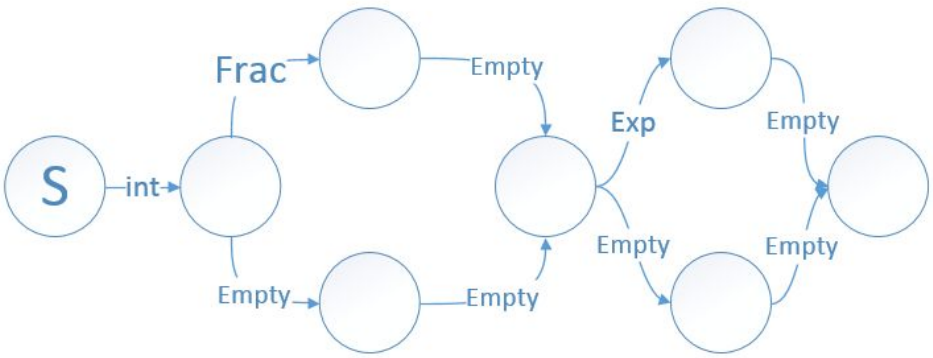
State	New S	E	e	+	-	i
A	{0}	B	C			
B	{1,3,6}			D	E	F
C	{2,3,6}			D	E	F
D	{4,6}					F
E	{5,6}					F
F	{7}					

得到的状态转换表：

```
static final int[][] exponentDFA = {
    { 1,  2, -1, -1, -1 },  // State A
    {-1, -1,  3,  4,  5 },
    {-1, -1,  3,  4,  5 },
    {-1, -1, -1, -1,  5 },
    {-1, -1, -1, -1,  5 },
    {-1, -1, -1, -1, -1 }
};
```

最终得到的 Decimal 的自动机：

（NFA）



（DFA）

State	E	e	+	-	i
0	1	2			
1			3	4	5
2			3	4	5
3					5
4					5
5					

（3）单词分类法：

使用面向对象的思想，首先构建一个 Token 的抽象类，表示所有的词素单元。然后具体的词素单元继承这个类。

```
public abstract class Token {
    public final int tag;
    public int operNum;
    public Token(int tag) {
        this.tag = tag;
    }
    public String ToString() {
        return new String("tag: " + this.tag);
    }
}
```

注意，其中的 operNum 代表运算符的元数

例如具体的词素单元类：

```
public class Decimal extends Token{
    public final double value;
    public Decimal(double value) {
        super(Tag.DECIMAL);
        this.value = value;
    }
    public String toString() {
        return new String("Decimal: " + this.value);
    }
}
```

(4) 特别注意事项：

(a) ‘-’号运算符的重载。通过一元和二元运算符的区别，重载这个运算符。在词法分析的过程中，遇到‘-’需要检查其前后的运算符的关系，从而确定运算符的真正类型。

(b) 处理字符串右边界，增加一个结束符号‘#’表示字符串识别结束。

(c) 同时在字符串末尾添加一个‘Dolla’终结符号以方便语法分析的调用。

3.构造算符优先关系表

这是本次实验最为关键的一部分。

构建的关系表如下图所示：

```
public static final int table[][] = {
            /*(   )  func -   ^  md   pm cmp  !   &   |   ?   :   ,   $ */
    /*(*/    {S0, R1, S0, S0, S0, S0, S0, S0, S0, S0, S0, S0, E4, S0, E2},
    /*)*/    {R1, R1, R1, R1, R1, R1, R1, R1, R1, R1, R1, R1, R1, R1, R1},
    /*func*/ {S0, E6, E6, E6, E6, E6, E6, E6, E6, E6, E6, E6, E6, E6, E6},
    /*-*/    {S0, R2, S0, S0, S0, R2, R2, R2, E7, E5, E5, R2, R2, R2, R2},
    /*^*/    {S0, R1, S0, S0, S0, R3, R3, R3, E7, E5, E5, R3, R3, R3, R3},
    /*md*/   {S0, R1, S0, S0, S0, R3, R3, R3, E7, E5, E5, R3, R3, R3, R3},
    /*pm*/   {S0, R1, S0, S0, S0, S0, R3, R3, E7, E5, E5, R3, R3, R3, R3},
    /*cmp*/  {S0, R1, S0, S0, S0, S0, S0, E5, E7, R3, R3, R3, E4, E6, R3},
    /*!*/    {S0, R1, E5, E5, E5, E5, E5, S0, S0, R2, R2, R2, E4, E6, R2},
    /*&*/    {S0, R1, E5, E5, E5, E5, E5, S0, S0, R3, R3, R3, E4, E6, R3},
    /*|*/    {S0, R1, E5, E5, E5, E5, E5, S0, S0, S0, R3, R3, E4, E6, R3},
    /*?*/    {S0, E4, S0, S0, S0, S0, S0, S0, S0, S0, S0, S0, S0, E4, E4},
    /*:*/    {S0, R1, S0, S0, S0, S0, S0, E4, E4, E4, E4, S0, E4, E4, R4},
    /*,*/    {S0, R1, S0, S0, S0, S0, S0, E6, E6, E6, E6, S0, E4, S0, E6},
    /*$*/    {S0, E1, S0, S0, S0, S0, S0, S0, S0, S0, S0, S0, E4, E6, A0}
};
```

其中，+，-，*，/ 为同类型优先，合并在一起，预定义函数和逻辑运算的关系运算亦然。
说明，表格左侧为当前符号，表格的上方行代表将要读取的符号。
两者的优先关系由 OOPtable 定义：

```
public static final int S0 = 0; // shift
public static final int R1 = 1; // reduce bracket ()
public static final int R2 = 2; // reduce 1 operator
public static final int R3 = 3; // reduce 2 operator
public static final int R4 = 4; // reduce 3 operator
public static final int A0 = 5; // accept state

public static final int E1 = -1; // missing left bracket (
public static final int E2 = -2; // missing right bracket )
public static final int E3 = -3; // missing oprand
public static final int E4 = -4; // 3 operator error
public static final int E5 = -5; // type error
public static final int E6 = -6; // function error
public static final int E7 = -7; // syntax error
public static final int E8 = -8; // unknown error

public static final int SHIFT = 0;
public static final int RBRACKET = 1;
public static final int RUNARY = 2;
public static final int RBOPER = 3;
public static final int RTOPER = 4;
public static final int ACCEPT = 5;
public static final int ELBRACKET = -1;
public static final int ERBRACKET = -2;
public static final int EMOPRAND = -3;
public static final int ETOPER = -4;
public static final int ETYPE = -5;
public static final int EFUNC = -6;
public static final int ESYNT = -7;
public static final int EUNK = -8;
```

其中 S0 代表移入操作，
R1~R4 代表不同的规约法则，

A0 代表接受状态，
E1-E7 表示不可比的符号之间的出错定义规则。

如，当前符号为左括号时候，现在要读取的符号是‘）’则进行 R1（括号规约），读到‘（’进行移入，读到函数进行移入，若读到‘$’则抛出缺少右括号错误，读到其他符号则移入。

当前符号为‘）’时候，无论读到什么符号，都进行规约。

其他符号以此类推。

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?

此工作在词法分析阶段已经完成，在语法分析阶段只需要判断其操作符的元数即可。

```
case '-':
    if(t.operNum == 1) {
        return minus;
    } else {
        return plumin;
    }
```

4.设计并实现语法分析和语义处理程序（详情见代码）

按照之前的规约法则，分别设计 R1,R2,R3 对应的规约函数。

规约括号时候，逐个往前规约，遇到一元运算符则调用一元规约，遇到二元则进行二元规约，遇到三元则进行三元规约。最终知道规约到‘（’出现为止，否则抛出缺少左括号错误。规约完毕后判断当前词素单元是否为函数词素，若是，继续进行函数规约。

规约一元、二元、三元运算符时候，将一个操作数取出进行运算，放回栈中。

Shift:

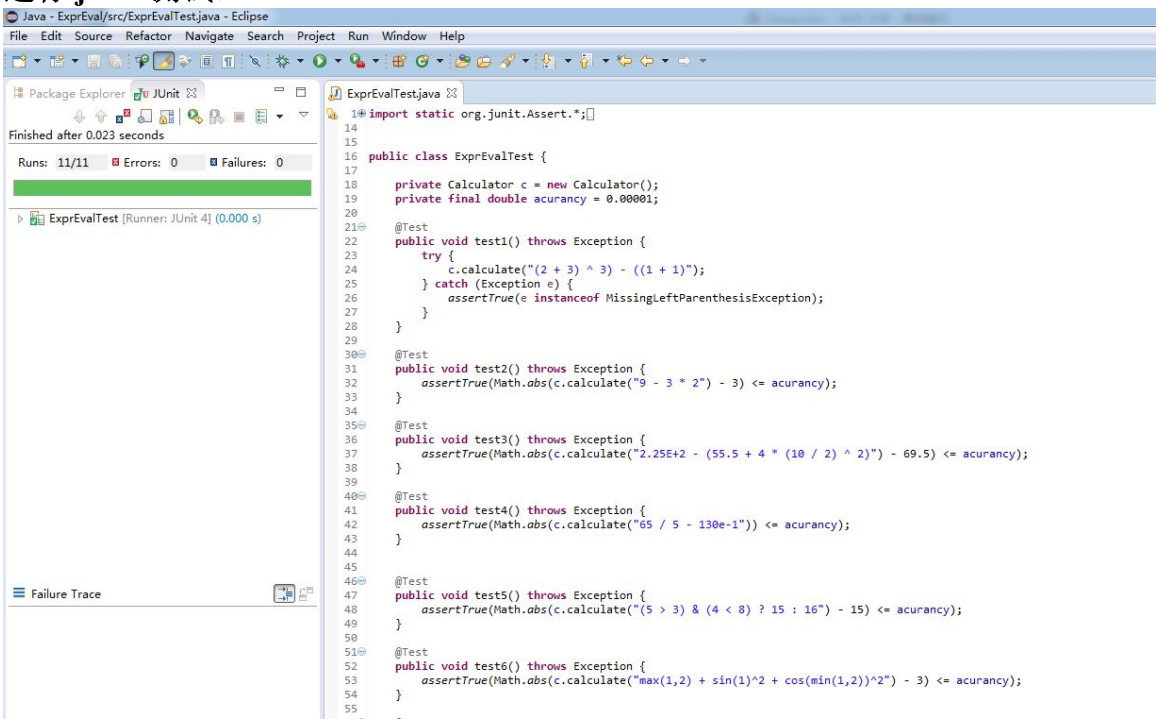
```
private void shift(Token t) {
    operators.push(t);
}
```

Reduce:

```
private void reduce(int reducer) throws DividedByZeroException,
                                        MissingLeftParenthesisException,
                                        MissingOperandException,
                                        FunctionCallException,
                                        TypeMismatchedException,
                                        MissingOperatorException,
                                        TrinaryOperationException {
    switch (reducer) {
        case OOPTable.RBRACKET: reduceBracket(); break;
        case OOPTable.RUNARY: reduceUnary(); break;
        case OOPTable.RBOPER: reduceBoper(); break;
        case OOPTable.RTOPER: reduceToper(); break;
        default: break;
    }
}
```

5. 测试

进行 junit 测试：



测试结果，实验成功。

实验感想：

不是一般的 project 的难度，非常困难，非常有挑战性，连续作战数天后征服！