基于表达式的计算器 EXPR-Eval

一、词法分析

1. 单词划分

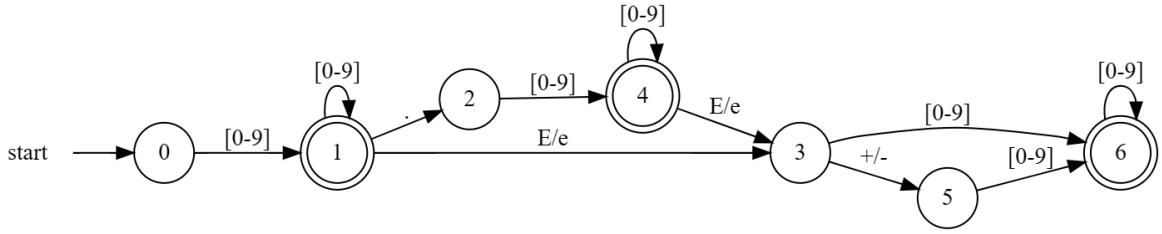
根据单词进行计算需要的语义区分进行划分单词，划分为如下9类

类型	单词
数值常量（Decimal）	比如10、10.1、10.1e+2
布尔常量（Bool）	true、false
算数运算符（Operator）	+、-、*、/、^
关系运算符（Relation）	<、<=、>、>=、<>、=、&、|
括号（Parenthesis）	（、）
预定义函数（Function）	sin、cos、max、min
逗号（Comma）	,
一元运算符（Unary）	- 、！
三元运算符（Triple）	？、：

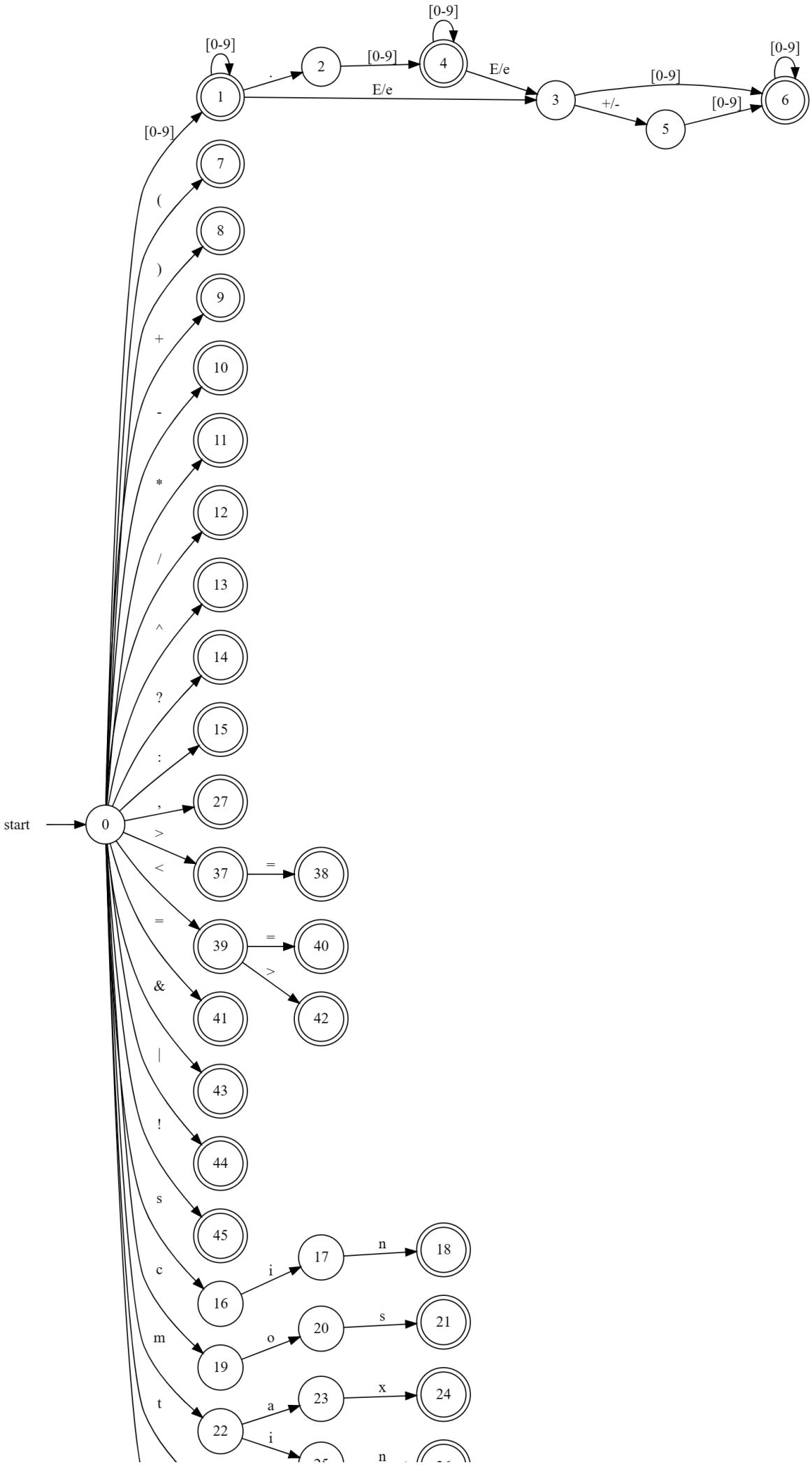
2. DFA

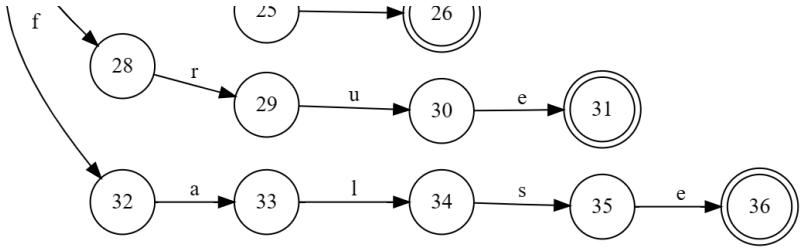
对于decimal数值常量，通过产生式得到的DFA如下

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)



其余的合法单词有：（、）、+、-、*、/、^、？、：、sin、cos、max、min、，、true、false、>、>=、<、<=、=、<>、&、|、！。综上得到以下有限自动机：


start
0
[0-9]
1
.
2
[0-9]
4
[0-9]
E/e
3
E/e
[0-9]
6
[0-9]
+/-
5
[0-9]
[0-9]
(
)
+
-
*
/
^
?
:
,
>
<
=
&
|
!
s
c
m
t
7
8
9
10
11
12
13
14
15
27
37
=
38
39
=
40
>
42
41
43
44
45
16
i
17
n
18
19
o
20
s
21
22
a
23
x
24
i
n



3. 程序设计

包括词法分析类`MyScanner`，有限状态机类`DFA`，状态机状态类`State`，扫描后传出对象的类`Token`

`MyScanner`为词法分析的主执行程序；`State`存储DFA每个状态的信息，包括是否为结束状态，结束状态的单词类型，状态的跳转边；`DFA`则用来执行自动机的状态跳转工作，返回是否可跳转以及合法的信息；`Token`则是词法分析的传出对象，用于语法和语义分析，根据单词类型定义多个子类，包括是否为终结符，token类型以及语义信息。

词法分析执行过程通过输入表达式expression到`MyScanner`，执行扫描操作，每扫入一个字符，则在`DFA`中进行状态跳转，状态转移过程如下：

每次读入两个字符，第一个字符判断是否可跳转到当前状态（一般是状态0才需要判断），第二个字符进行判断：

① 如果不存在转移边或下一个读入字符为$：如果当前状态是结束状态,则返回finished表示当前单词读取结束，可回到状态0读取下一个单词，否则返回error查找对应词法错误；

② 如果存在转移边且下一个字符不是$: 返回unfinished表示可转移到下一个状态。

`MyScanner`每读取一个新单词同时调用`DFA`中的该新单词对应结束状态的`State`对象，来创建相应类型的`Token`对象加入列表，至读取所有新单词结束，词法分析过程完成。

其他注意事项：

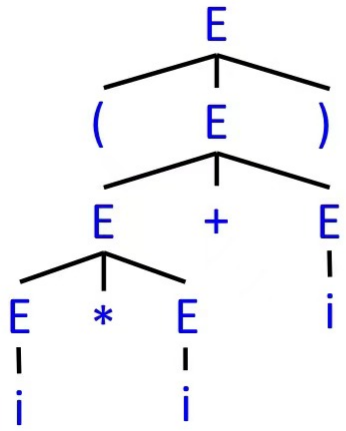
- 其中关于`-`符号是一元还是二元操作符的判定，通过记录上一个单词，当上一个单词是数值常量或右括号时，判定为二元运算符，否则认定为一元运算符。
- 如果输入为空格，空格相当于空串；输入true和false时一律转化为小写。

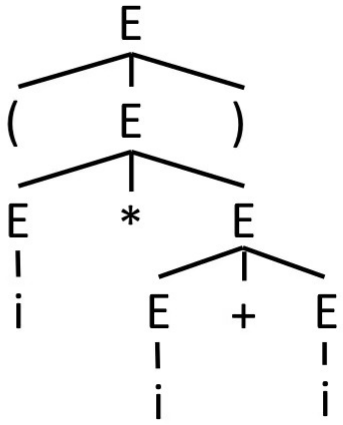
二、语法语义分析

1. 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
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
```

该BNF存在二义性，以理论课课件中相同例子可证明，对于同一个表达式`(i*i+i)`存在至少以下两种语法树，且通过这个例子可看出二义性也会带来计算结果的不同，影响语义。





对此可以通过规定算符优先级和结合性质来避免二义性的发生，如下表所示

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. 算符优先关系表

根据产生式和优先级结合性质表构造出算符优先关系表

D表示decimal，F表示function，B表示boolean，R表示relation

	D	(	)	+-	*/	^	?	:	-	F	,	B	R	&	|	!	$
D	-1	-1	>	>	>	>	>	>	-1	-1	>	-1	>	>	>	-1	>
(	<	<	=	<	<	<	<	-5	<	<	=	<	<	<	<	<	-4
)	-1	-1	>	>	>	>	>	>	<	<	>	<	>	>	>	-1	>
+-	<	<	>	>	<	<	>	>	<	<	>	<	>	>	>	<	>
*/	<	<	>	>	>	<	>	>	<	<	>	<	>	>	>	<	>
^	<	<	>	>	>	<	>	>	<	<	>	<	>	>	>	<	>
?	<	<	-5	<	<	<	<	=	<	<	-5	<	<	<	<	<	-5
:	<	<	>	<	<	<	<	>	<	<	>	<	<	<	<	<	>
-	<	<	>	>	>	>	>	>	<	<	>	<	>	>	>	<	>
F	-6	=	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6
,	<	<	=	<	<	<	<	0	<	<	<	<	<	<	<	<	-2
B	-1	-1	>	-7	-7	-7	>	-7	-7	-1	-7	-1	-7	>	>	-1	-7
R	<	<	>	<	<	<	>	-7	<	<	-7	<	>	>	>	<	1
&	<	<	>	<	<	<	>	-7	<	<	-7	<	<	>	>	<	1
|	<	<	>	<	<	<	>	-7	<	<	-7	<	<	<	>	<	1
!	<	<	>	<	<	<	>	-7	<	<	-7	<	<	>	>	<	1
$	<	<	-3	<	<	<	<	-5	<	<	-7	<	<	<	<	<	>

其中>表示归约reduce操作，<和=表示移入shift操作，空白位置(不是>、<、=的位置)存在错误，根据经验填入具体的报错信息。使用0表示移入（即<和=），使用1表示移入（即>），2表示acc得到最终结果，错误信息不确定则先移入或归约，在之后的归约过程能检测出具体的错误类型

-1：缺少操作符 MissingOperatorException

-2：缺少运算量 MissingOperandException

-3：缺少左括号 MissingLeftParenthesisException

-4：缺少右括号 MissingRightParenthesisException

-5：三元运算符错误 TrinaryOperationException

-6：函数错误 FunctionCallException

-7：类型不匹配 TypeMismatchedException

如何处理敏感关系：在词法分析中对单词进行了较为精细的划分，比如将`-`直接分为两个类别，在词法分析中就由了明确的区分，并将二者建立不同的Token类对象，使得在语法和语义分析中不会出现混淆的问题。三元运算符同样单独划分为一类，因为`?`、`:`都只出现在三元运算中，将其划分为单独一类而不是笼统的字符类能够更容易在语法判断中定位具体错误。

3. 语法分析和语义处理程序

语法分析和语义处理能够放在同一个过程中，在语法归约同时就能进行语义计算，发现语义错误。

包括主执行类`Parser`，归约操作类`MyReducer`，分析的传入参数类`Token`。

`MyReducer`根据不同的单词类型进行不同的归约操作，因此为reduce操作定义了多个类的方法，适用于不同的归约操作。`Token`包括多个类型的子类是为了服务于语法分析过程代码可读性，实际Comma、Function、Operator、Parenthesis、Sign、Triple、Unary能够融合为一个类加入逻辑判断，但为了使用Token类时方便辨别，定义多个类进行区分。`Parser`中存储了上文定义的算符优先关系表，以及缓冲区buffer和栈stack，是存放Token类对象的列表，通过读入buffer的首个Token以及stack最顶部的终结符Token，查找算符优先关系表得到下一步操作，下面是该过程的伪代码：

```
while(true){
    stackTop = stack.getTopTerminal;    // 最顶部终结符
    lookahead = buffer.getFirstToken;
    action = table[stackTop.index][lookahead.index];
    switch(action){
        case 0 ->shift();
        case 1 -> reduce();
        case 2 -> return accept();
        case -1 -> MissingOperatorException();
        ...
        case -7 -> TypeMismatchedException();
    }
}
```

Shift通过复制buffer的首个Token移入stack并输出buffer首个Token实现

Reduce操作伪代码如下：需要不断归约直到查询的下一步操作为移入或报错退出

```
stackTop = stack.getTopTerminal;
lookahead = buffer.getFirstToken;
action = table[stackTop.index][lookahead.index];
while(action == 1){
    stack = MyReducer(stack).getResult();   // 得到归约后的stack
    stackTop = stack.getTopTerminal;        // 下一个最顶部的终结符
    action = table[stackTop.index][lookahead.index];
}
```

4. 不同运算符的语法归约和语义处理

在`MyReducer`执行归约操作中，对以下7中运算符进行了归约处理，由decimal、boolean、operator、unary、parenthesis、relation、triple。其中不包括Comma逗号，以及Function函数，因为在算符优先关系表中，并不存在对这两类符号的归约操作，同样不存在归约操作的符号还有`?`、`(`，`?`会随`:`一起归约，`(`会随`)`一起归约，Function和Comma会随`(`，`)`一起归约。

归约操作实际上是还是得到一个Token，与归约前不同在于Token的`terminal`改为了false表示为非终结符，归约之后的Token都为decimal或boolean类型，下面对不同运算符类型的归约操作进行说明

① decimal：复制当前Token，设置为非终结符Token替换掉原先Token（或直接将`terminal`改为false)

② boolean：与decimal同理，复制为非终结符并替换原先Token

③ operator

语法分析：查看该操作符前后是否为已经归约的非终结符，否则报错缺少操作数

语义处理：查看该操作符前后Token类型是否为decimal，否则报错类型不匹配；提取前后decimal的语义值，即double类型数值，根据operator类型，进行+ - * / 操作得到计算值value，得到值后新建decimal类型Token，赋值value，设`terminal`为false。用新建的Token替换被归约的3个Token

④ unary

语法分析：判断该操作符后是否存在Token，否则报错缺少操作数

语义处理：查看unary类型值为`-`还是`!`,`-`则判断后面Token是否为decimal类型，`!`则判断是否为boolean类型，否则返回类型不匹配，对decimal类型取负或对boolean类型取反后，使用该值新建非终结符Token，替换被归约的2个Token

⑤ parenthesis

语法分析：寻找判断是否存在左括号，不存在则返回缺少左括号错误；查看括号之间是否存在参数，不存在则返回缺少操作数；如果左括号前是function类型Token，则使用函数归约方法，否则使用括号归约方法。

1. 函数归约时，括号内有效内容为`(decimal,decimal,...,decimal)`格式，奇数位置必须为decimal类型Token，偶数位必须为Comma类型Token，按顺序进行检查，如遇奇数位置不满足条件，若Token为boolean类型，返回类型不匹配错误（语义错误），若Token为，返回缺少操作数，其余错误返回函数使用错误；如遇偶数位置不满足条件，则返回函数使用错误。进而通过通过sin/cos和max/min分类计算
    1. sin/cos函数要求只存在一个decimal类型Token，否则返回函数使用错误
    2. max/min函数要去函数至少存在2个decimal类型Token，否则返回缺少操作数

2. 括号归约时，括号内有效内容为`(decimal)`、`(boolean)`格式，如果内部超过一个Token，一般不会发生， 因为会在前面过程的归约中就发现错误，如果发生返回语法错误

语义处理：函数归约时，查询括号内部参数的语义信息，在根据function类型，采取相应的计算，使用计算结果来新建非终结符Token，替换被归约的`function`~`)`部分的Token；括号归约时，使用decimal或是boolean的Token替换被归约到`(`~`)`部分的Token

⑥ relation

语法分析：与operator类似，查看该操作符前后是否为已经归约的非终结符，否则报错缺少操作数

语义处理：如果relation值为&、|，查看该操作符前后Token类型是否为boolean，如果是<、<=、>、>=、=、<>，查看前后Token是否为decimal类型，否则报错类型不匹配；提取前后decimal或boolean的语义值，即double类型数值，根据operator类型，进行<、<=、>、>=、=、<>、&、|操作得到布尔值，使用布尔值新建boolean类型Token，设`terminal`为false。用新建的Token替换被归约的3个Token

⑦ triple

语法分析：已知`:`的位置，查找`?`的位置，判断是否符合`nonterminal_0 ? nonterminal_1 : nonterminal_2`的格式，否则返回缺少操作符

语义处理：判断`nonterminal_0`是否为boolean类型，`nonterminal_1`和`nonterminal_2`是否同为decimal或boolean类型，否则返回类型不匹配，不存在错误则取三个Token的语义值进行三目运算，得到结果新建非终结符Token，替换被归约的5个Token

三、结果测试

使用实验软装置的测试用例simple和standard，结果通过

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

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

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

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

编写测试用例，使用实验指导的用例，发现存在一例错误

```
------------------------------------------------------------
Statistics Report (44 test cases):

        Passed case(s): 43 (97.72727272727273%)
        Warning case(s): 1 (2.2727272727273%)
        Failed case(s): 0 (0.0%)
============================================================
```

错误如下，Expected要求的缺少操作数无法理解，个人看法还是应该判定在归约`:`符号是发生的三元运算错误

```
E123 An operand is expected.
Input: 3.14 * 2 >= 2.5 * 3 ? (6 : 7) + 8
Expected output: MissingOperandException
Warning: exception is not an expected type (exceptions.TrinaryOperationException: Syntactic error in trinary operation.
```

四、总结心得

实验结果能够基本进行正确的词法语法语义分析，但在发生错误时对错误类型的划分还有待更精细的划分，比如单独一个`,`或`:`是应该划分到哪种类型的语法错误，暂且判定为MissingOperandException和TrinaryOperationException错误，但也已经能够满足基本的错误判别。本次实验整体地把词法分析、语法分析到语义分析过了一遍，对理论课程的学习内容有了一个更清晰的认识，同时也对代码编写的规范性有更熟练的掌握。