**北京邮电大学**

**计算机学院（国家示范性软件学院）**

**2020-2021 学年第 1 学期项目总结报告**

课程名称： 编译原理

项目名称： 语法分析器

项目完成人：

负责人姓名： 李兴诚 学号： 2018211936

姓名： 刘邦辰 学号： 2018211939

姓名： 胡鼎新 学号： 2018211932

姓名： 赵宸阳 学号： 2018211814

指导教师： 修佳鹏

日 期： 2020 年 12 月 23 日

# 项目目的和要求

编写语法分析程序，实现对算数表达式的语法分析，要求所分析的算数表达式由如下的文法产生：

E -> E + T | E – T | T

T -> T \* F | T / F | F

T -> ( E ) | num

1. 编写递归调用程序实现自顶向下的分析
2. 编写LL（1）语法分析程序
3. 编写语法分析程序实现自底向上的分析

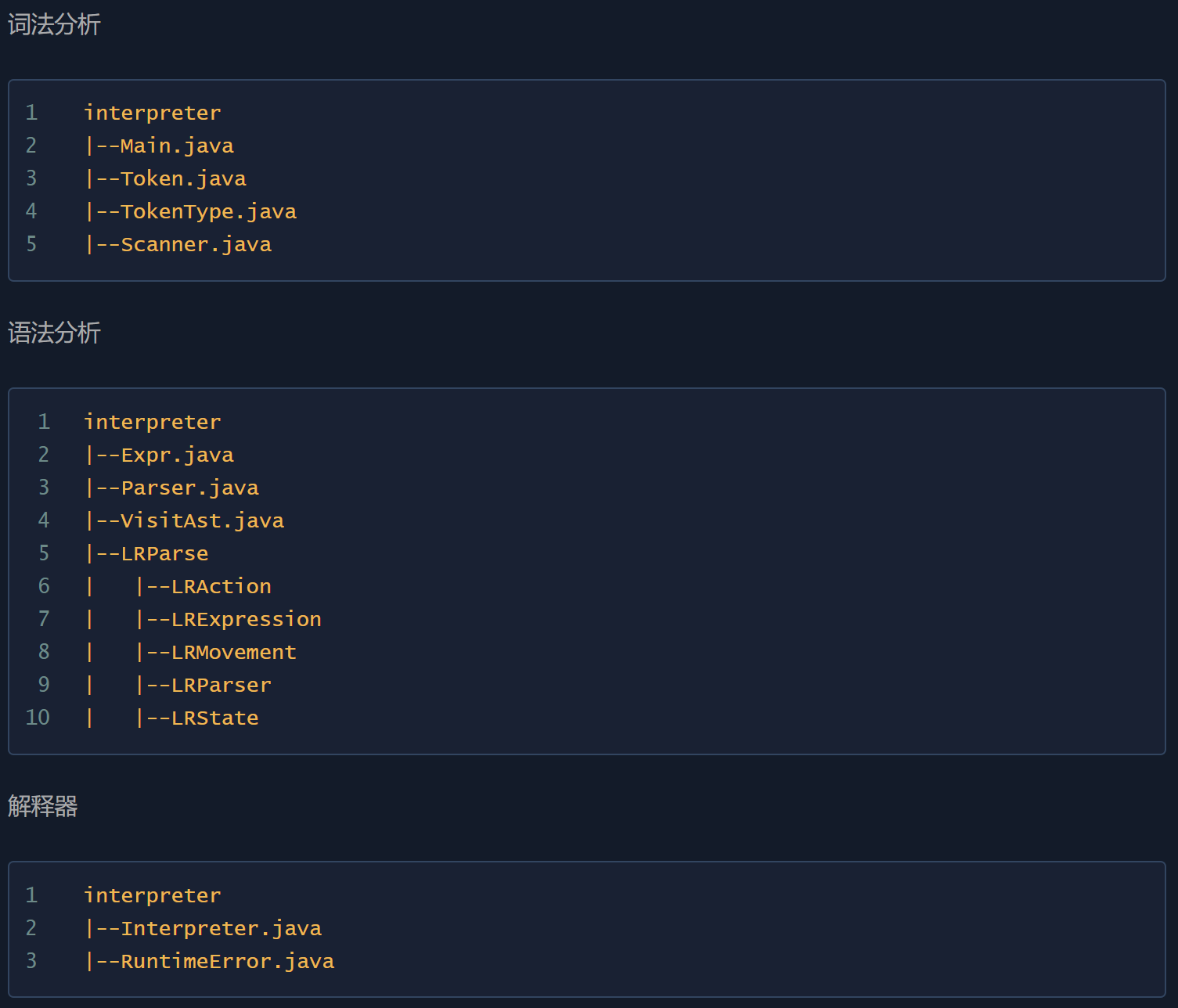
利用YACC自动生成语法分析程序，调用LEX自动生成的词法分析程序

# 二、项目实验环境

1. JDK12.0
2. IDEA Ultimate 2020.2
3. Python
4. GNU

# 项目内容

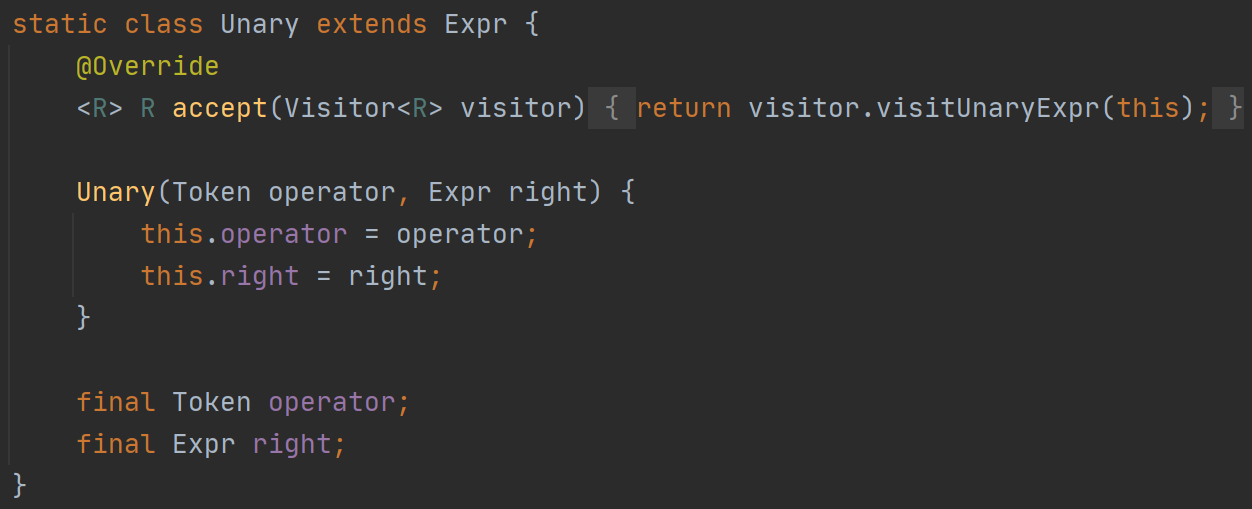
## 程序结构



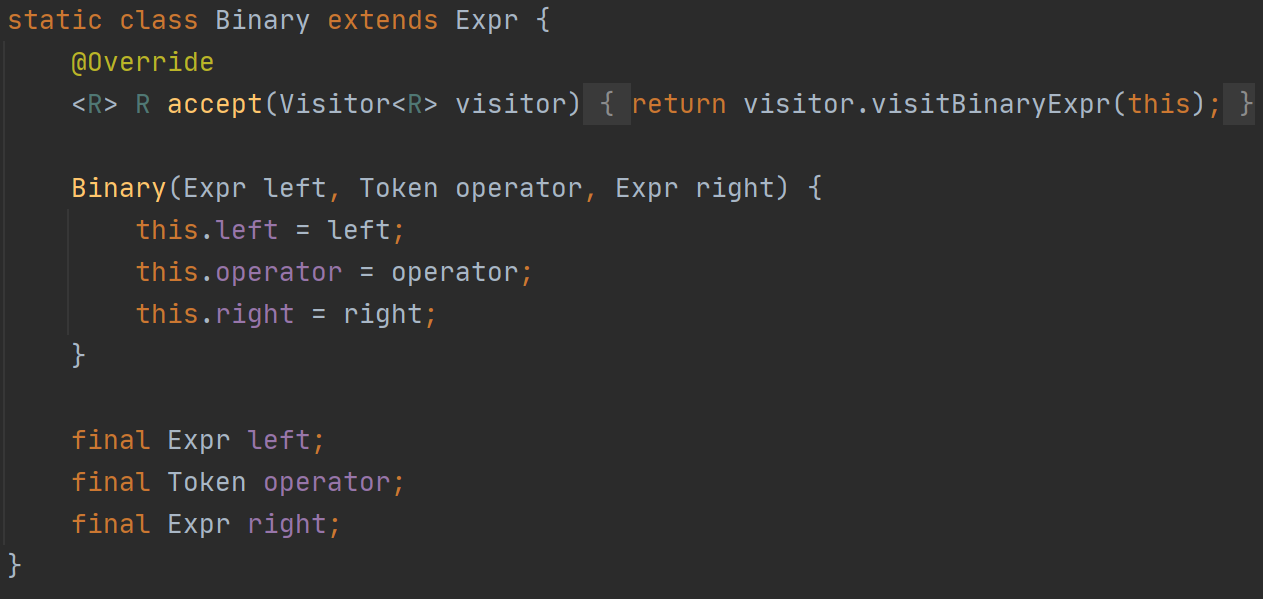
## 3.1递归下降分析

### 3.1.1语法树

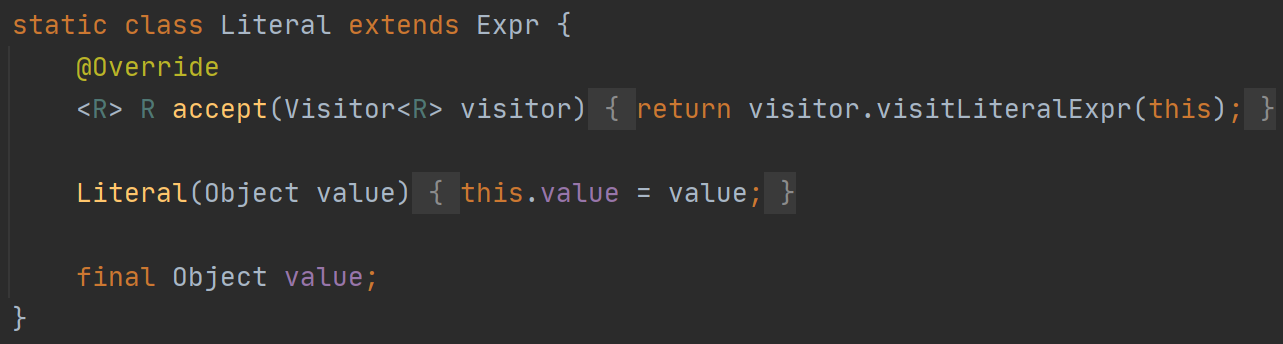
#### 一元表达式



#### 3.1.1.2 二元表达式

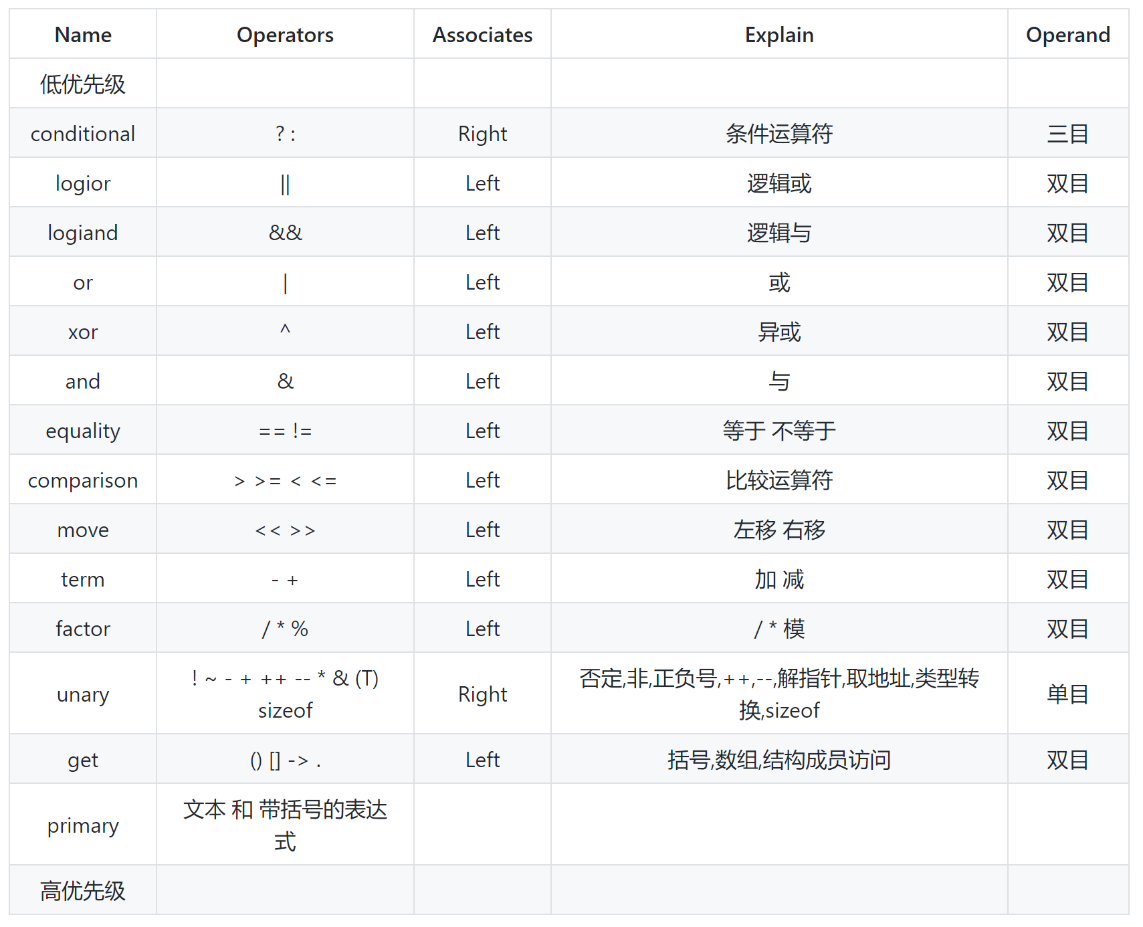


* 值表达式

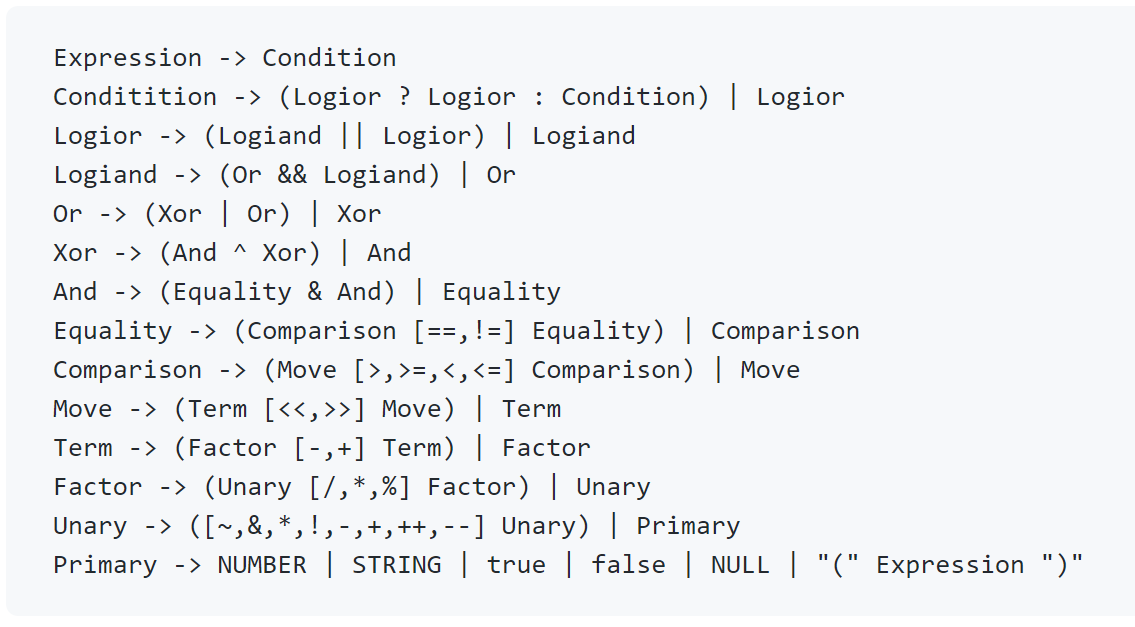


### Parser部分

#### 优先级区分

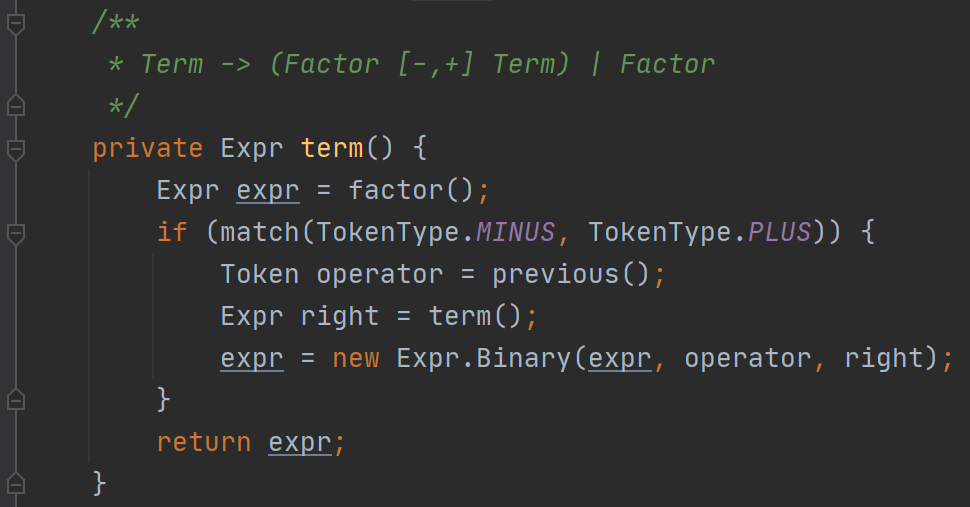


#### 上下文无关文法

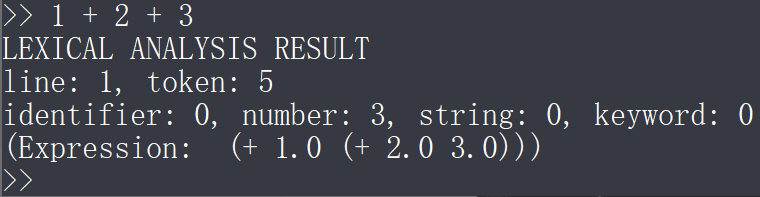


#### 递归下降

https://docimg6.docs.qq.com/image/RCnaTYy-1R98Y5bhXdMQMQ?w=545&h=34



### 结果



## LR语法分析

### 3.2.1 原理

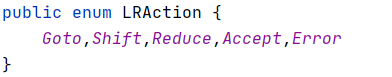
使用哈希表模拟LR分析表

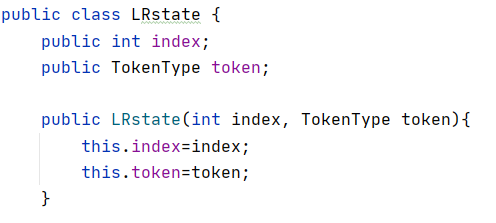
LRState->LRMovement

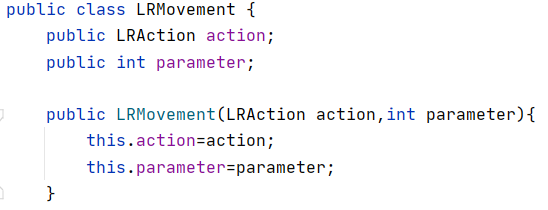
对这两个类重写hashcode和equals方法。

根据当前的符号栈顶元素和看到的下一个标识符，从哈希表中获取对应的动作。

在shell下使用LRparse命令，即可对后方的表达式进行LR语法分析。

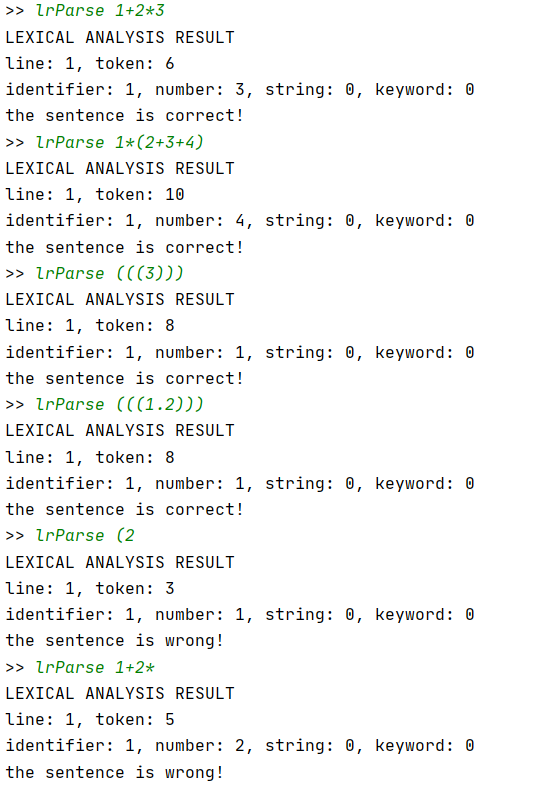






https://docimg3.docs.qq.com/image/K5967DdqMUftLQVG1VpleA?w=570&h=50

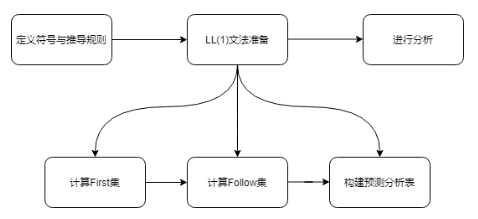
### 结果



## LL(1)语法分析

### 综述

LL(1) 语法使用python语言实现，本质上是使用python语言模拟LL(1)文法的构建与执行过程，依次实现了符号及推导规则的定义、First集与Follow集的计算、预测分析表的构建，以及最终对输入字符串进行分析的过程。具体流程如下所示：



*1*

接下来将依次介绍各部分的实现与代码。

### 确定终止符与非终止符

*# 定义终止符与非终止符*  
**class** ele:  
 **def** \_\_init\_\_(self, char, type):  
 self.char = char  
 self.type = type *# 1代表非终止符，0代表终止符*  
  
*#终止符与非终止符*  
E = ele('E', 1)  
T = ele('T', 1)  
F = ele('F', 1)  
E\_1 = ele('E\'', 1)  
T\_1 = ele('T\'', 1)  
num = ele('num', 0)  
add = ele('+', 0)  
minus = ele('-', 0)  
multi = ele('\*', 0)  
divis = ele('/', 0)  
leftbrackets = ele('(', 0)  
rightbrackets = ele(')', 0)  
end = ele('$',0)  
terminator=[num,add,minus,multi,divis,leftbrackets,rightbrackets]  
nonterminator=[E,T,E\_1,T\_1,F]

该部分首先定义了 ele类 储存终止符与非终止符的相关信息，类中包含了符号的字符表示以及所属类型（终止符或非终止符）。

在将所有终止符与非终止符加入后，新建了两个数组，分别储存所有的终止符与非终止符，方便后续进行遍历。

### 确定推导规则

*#语法推导规则的定义*  
rule\_1 = [E,T,E\_1]  
rule\_2 = [E\_1,add,T,E\_1]  
rule\_3 = [E\_1,minus,T,E\_1]  
rule\_4 = [E\_1]*#表示推导到空的情况*  
rule\_5 = [T,F,T\_1]  
rule\_6 = [T\_1,multi,F,T\_1]  
rule\_7 = [T\_1,divis,F,T\_1]  
rule\_8 = [T\_1]  
rule\_9 = [F,leftbrackets,E,rightbrackets]  
rule\_10 = [F,num]  
rule\_all=[rule\_1,rule\_2,rule\_3,rule\_4,rule\_5,  
 rule\_6,rule\_7,rule\_8,rule\_9,rule\_10]

我们首先将文法转化成非左递归的文法。

使用list类型储存所有的推导规则，在加入规则之后，新建了一个数组用来储存所有的规则，方便后续进行遍历。

### 构建First集和Follow集

对于文法G的非终止符 ，其First集合表示所有可由 推导出的所有开头终结符号的集合，即：

$$\rm First(a\_i)=\{\alpha | a\_i\stackrel{\*}{\rightarrow}\alpha\beta,\alpha \in V\_T,a\_i 、\beta \in (V\_T \cup V\_N)^\*\}$$

采用不断迭代，更新集合，直到集合中的内容不再变化为止的策略，实现代码如下：

*#计算First集*  
**def** getFirst():  
 First={'E':[],'T':[],'F':[],'E\'':[],'T\'':[]}  
 *#先将所有直接终止符前缀加入*  
 **for** rule **in** rule\_all:  
 **if** (len(rule)!=1 **and** rule[1].type==0):  
 First[rule[0].char].append(rule[1])  
 *#接着一直迭代，直到集合不再变化*  
 count=countNum(First)*#先计算First集合种元素总数*  
 unchange=0  
 *#迭代，每次迭代完成后比较集合种总数是否有变化*  
 **while** unchange==0:  
 **for** rule **in** rule\_all:  
 **if** (len(rule) != 1 **and** rule[1].type == 1):  
 First[rule[0].char].extend(First[rule[1].char])  
 First[rule[0].char]=list(set(First[rule[0].char]))  
 count\_temp=count  
 count = countNum(First)  
 **if** (count\_temp==count):  
 unchange=1  
 *#集合不再变化后即计算完成*  
 **return** First

对于文法G的任何非终止符号 ，其Follow集是该文法的所有句型中紧跟在 中之后出现的非终止符或 $ 组成的集合，即：

$$\rm Follow(A)=\{\alpha | S \stackrel{\*}{\rightarrow}···Aa···,a\in V\_T\}$$

为了构建文法G的每个非终结符号 的Follow集合，我们采用如下策略，遍历所有的规则集合，不断将元素加入其Follow结合中，直到集合大小不再增大为止。

1. 对文法开始符号 ，置 $ 于 $\rm Follow(S)$ 中。
2. 若有产生式 ，则把$\rm Follow(\beta)$ 中所有非 元素加入到 $\rm Follow(B)$ 中。
3. 若有产生式 ,或者有产生式 ，但是 $\epsilon \in \rm First(\beta)$ ，则把 $\rm First(A)$ 中的所有元素加入到 $\rm Follow(B)$ 中

* *#计算Follow集*  
  **def** getFollow():  
   Follow = {'E': [], 'T': [], 'F': [], 'E\'': [], 'T\'': []}  
   *#先将直接出现在后面的非终止符的Fist集合与终止符加入*  
   **for** rule **in** rule\_all:  
   **if**(len(rule)!=1):  
   **for** index **in** range(1, len(rule) - 1):  
   **if** (rule[index].type == 1 **and** rule[index + 1].type == 0):  
   Follow[rule[index].char].append(rule[index + 1])  
   Follow[rule[index].char] = list(set(Follow[rule[index].char]))  
   **elif** (rule[index].type == 1 **and** rule[index + 1].type == 1):  
   Follow[rule[index].char].extend(FirstSet[rule[index + 1].char])  
   Follow[rule[index].char] = list(set(Follow[rule[index].char]))  
   *#接着将推导式左侧非终止符的Follow集合加入*  
   **for** rule **in** rule\_all:  
   **if**(len(rule)!=1):  
   **for** index **in** range(1, len(rule) - 1):  
   **if** (rule[index].type == 1 **and** rule[index + 1].type == 1 **and** isNullInFirst(rule[index + 1])):  
   Follow[rule[index].char].extend(Follow[rule[0].char])  
   Follow[rule[index].char] = list(set(Follow[rule[index].char]))  
   **if** (rule[len(rule) - 1].type == 1):  
   Follow[rule[len(rule) - 1].char].extend(Follow[rule[0].char])  
   Follow[rule[len(rule) - 1].char] = list(set(Follow[rule[len(rule) - 1].char]))  
   **return** Follow

### 构建预测分析表

预测分析表采用如下策略进行构造：

如果有产生式 ，当 呈现在分析栈栈顶时

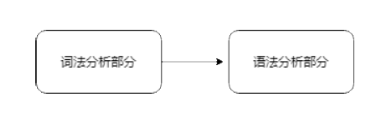
1. 如果当前输入符号 $a\in \rm First(\alpha)$ 时， 应被选作 的唯一合法代表去执行分析任务，即表项 M[A,a] 中应放入产生式
2. 如果 $\epsilon \in \rm First(\alpha)$ 并且当前输入符号 $b \in \rm FOLLOW(A)$ ，则应把产生式 放入表项 M[A,b] 中

实现代码如下所示

**def** createPredictionTabel():  
 PredictionTabel = {'E': {}, 'T': {}, 'F': {}, 'E\'': {}, 'T\'': {}}  
 **for** rule **in** rule\_all:  
 *# 先处理First集的情况*  
 **if**(len(rule)!=1):  
 *#推导式右侧第一个字符是终止符*  
 **if**(rule[1].type==0):  
 PredictionTabel[rule[0].char][rule[1].char]=rule  
 *#推导式右侧第一个字符是非终止符*  
 **if**(rule[1].type==1):  
 **for** ele **in** FirstSet[rule[1].char]:  
 PredictionTabel[rule[0].char][ele.char] = rule  
 *#接着处理Follow集的情况*  
 **else**:  
 **for** ele **in** FollowSet[rule[0].char]:  
 PredictionTabel[rule[0].char][ele.char] = rule  
 *#最后再处理推导至空的情况*  
 PredictionTabel[rule[0].char]['$'] = rule  
 **return** PredictionTabel

### 进行语法分析

具体分析部分分为词法分析与语法分析两个部分。词法分析部分依次读入输入的字符，将其转化成词素，此处特别注意的是数字恶识别。语法分析部分则模拟分析栈与分析表的运行，并在分析的过程中将中间过程输出。若分析过程出错，则也会有提示信息。



*image-20201212232557301*

该部分的实现代码如下：

**def** analysis(str):  
 *#定义输入栈*  
 status=1  
 inputStack=[]  
 **for** index **in** range(len(str)):  
 **if**(str[index].isdigit()):  
 **if**(index == (len(str)-1) **or** (1 - str[index+1].isdigit())):  
 inputStack.append('num')*#对于数字的识别并进行特殊处理*  
 **else**:  
 inputStack.append(str[index])  
 inputStack.append('$')  
 *#定义分析栈*  
 analysisStack=[end,E]  
 *#开始进行分析*  
 **while** (len(analysisStack)!=1):  
 **if**(len(inputStack)==1 **and** len(analysisStack)==1):*#当输入栈为空时，结束分析*  
 **break**  
 *#进行分析，分别考虑分析栈栈尾为终止符与非终止符的情况*  
 **if**(analysisStack[len(analysisStack)-1].type==0):  
 **if**(analysisStack.pop().char!=inputStack.pop(0)):  
 error()  
 status=-1  
 **break**  
 **else**:  
 printList(analysisStack)  
 printStr(inputStack)  
 print()  
 **else**:  
 **if**(inputStack[0] **in** PredictionTable[analysisStack[len(analysisStack)-1].char].keys()):  
 temp=PredictionTable[analysisStack.pop().char][inputStack[0]]  
 **for** index **in** range(1,len(temp)):  
 analysisStack.append(temp[len(temp)-index])  
 printList(analysisStack)  
 printStr(inputStack)  
 printRule(temp)  
 print()  
 **else**:  
 error()  
 status=-1  
 **break**  
 **if**(len(inputStack)!=1):  
 **if**(status==1):  
 error()  
 **else**:  
 print("分析成功")

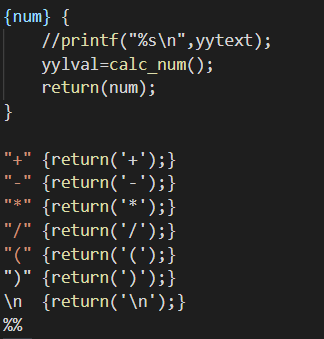
## YACC语法分析

### YACC简介

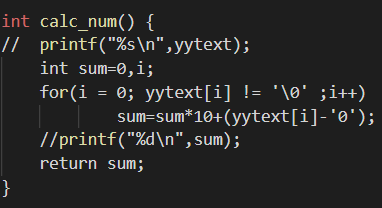
YACC是一款语法分析器的自动生成工具。用户按照YACC规定的规则写出文法说明文件，该文件一般以.y为扩展名，YACC编译器将此文法说明文件转换成用C编写的语法分析器文件filename.tab.c以及对应的头文件 filename.tab.h。

在使用YACC编写语法分析器之前，我们需要在该文件夹中使用Lex完成文法分析的编写，这个文件里应该包含语法分析驱动函数yyparse()以及LALR分析表。语法分析驱动程序则调用yylex()这个函数获取输入记号，每次调用yylex()都能获取一个输入记号。

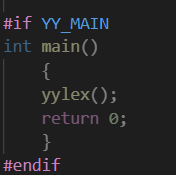
### Lex程序内容分析



Lex程序中主要进行数字以及计算符号的定义。



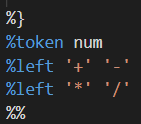
同时，Lex中定义了calc\_num()函数，负责将识别到的数字转换成其对应的数值。



在有lex程序编译成的c语言程序中，含有yylex()函数被语法分析程序的驱动程序调用。

### YACC程序内容分析

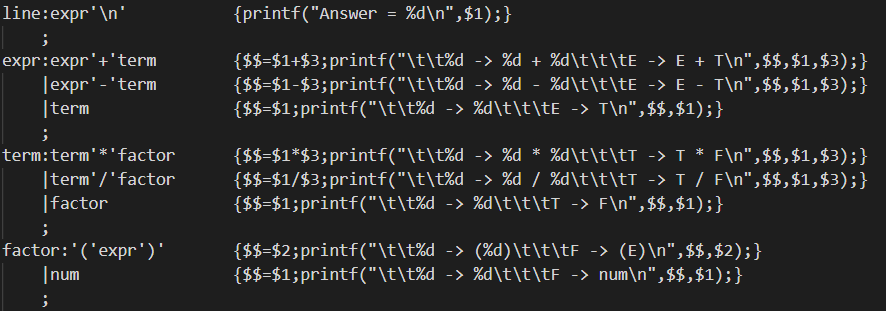
在YACC程序中首先定义了文法中使用的终结符。以%taken开头定义的终结符不具有优先级关系，以%left开头定义的终结符具有优先级关系，先定义的终结符优先级较高，后定义的终结符优先级较低，left表示符号满足左结合规律。



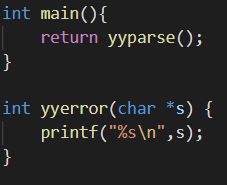
接下来是规则段，记录了文法的产生式以及规约整个产生式时应该采取的动作。

产生式中的每一个 | 分隔的部分都单独占一行，整个产生式写完后，结尾处应加上分号。

由下图的代码可见，在产生式规约时，程序分别执行了数值的计算以及将推导过程输出两个动作。



最终在YACC代码中定义了程序入口函数以及遇到错误时的错误显示函数。



### 编译及运行过程

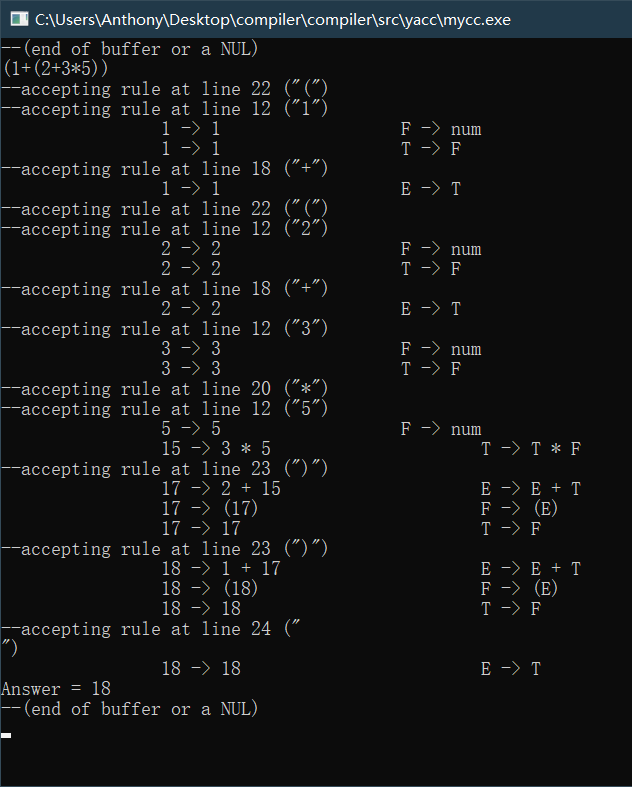
在windows环境下，lex程序由flex程序替代，YACC程序由Bison程序替代，分别使用两个程序将源代码编译成c语言代码。

|  |
| --- |
| bison -d C:\Users\Anthony\Desktop\compiler\compiler\src\yyac\second.y  flex -d C:\Users\Anthony\Desktop\compiler\compiler\src\yyac\first.l |

接着将c语言代码进行联合编译

|  |
| --- |
| gcc -o mycc C:\Users\Anthony\Desktop\compiler\compiler\src\yyac\second.tab.c C:\Users\Anthony\Desktop\compiler\compiler\src\yyac\lex.yy.c |

最终得到可执行程序，能够正常运行：

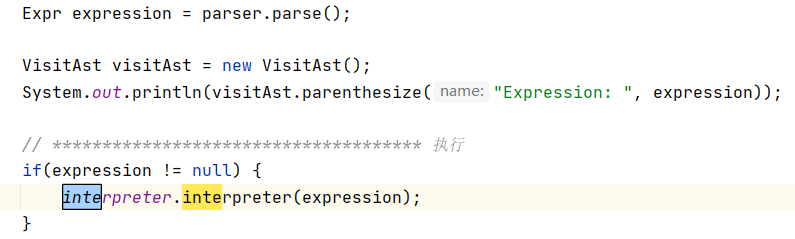


## 解释器

这是一个计算算数表达式的解释器

**main函数调用解释器：**

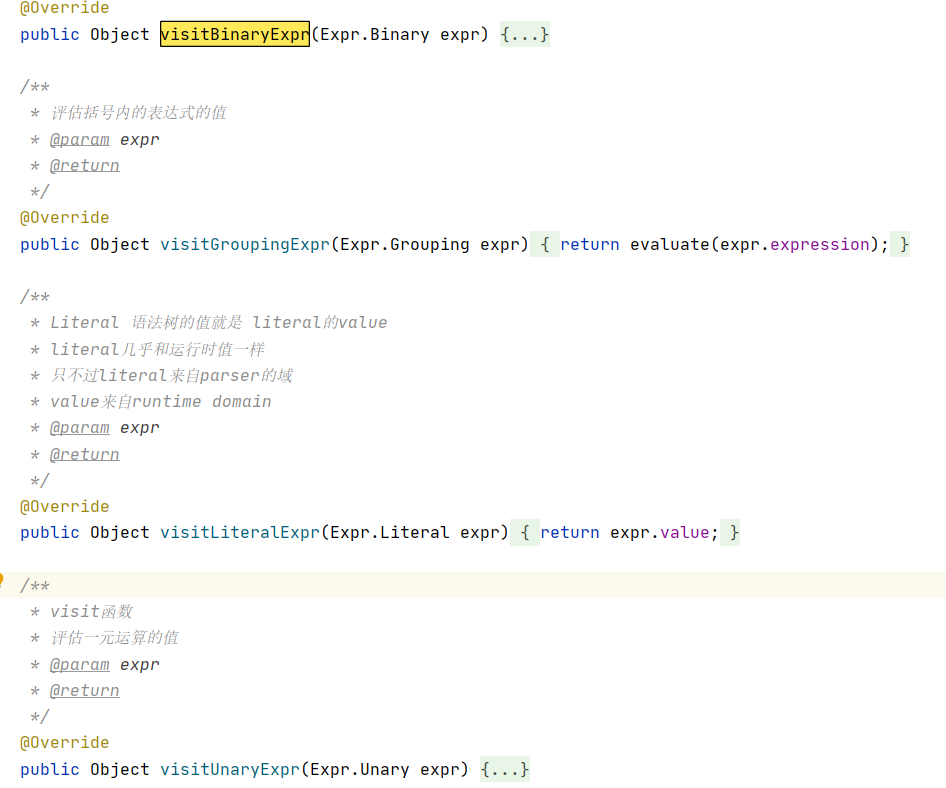
**将parser得到的Expression传入interpreter**



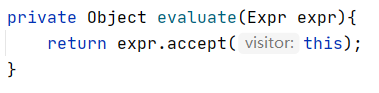
Interpreter理所应当是**Expr类的访问者**

https://docimg4.docs.qq.com/image/M7w_0WTMN81VA8uRSbrMvQ?w=423&h=63

我们只需要**复写4个访问者函数**



接着在evaluate函数内执行expr的accept进行**递归调用访问者函数进行分析**

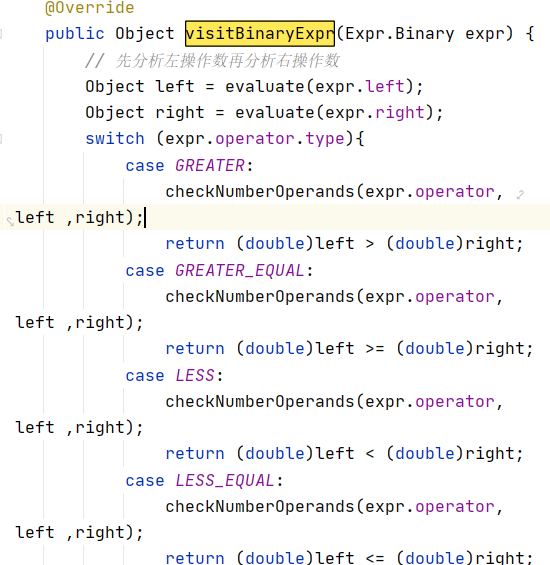


**由于是解释器，因此不用生成中间代码**，visit函数结构如下：

对于非二元运算，直接通过java直接运算得到值，返回



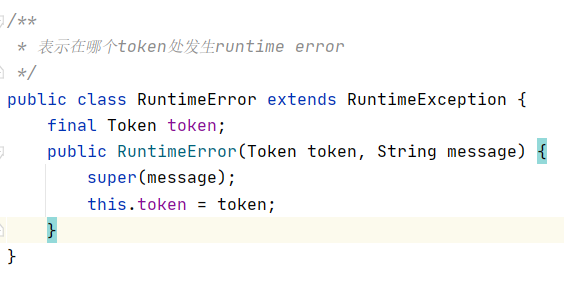
对于二元运算，先判断二元运算符，再通过java直接计算



### RuntimeError

解释器在运行过程中，可能会由于语义错误产生runtime error

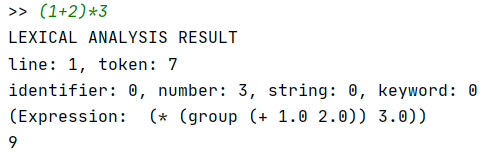
构造新的Error类：

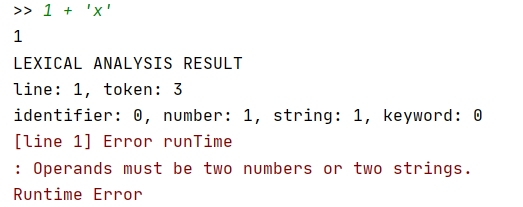


当产生如数字＋字符等语义错误时，就会通过Java的异常机制抛出RuntimeError

### 运行结果

最终的运行结果如下：



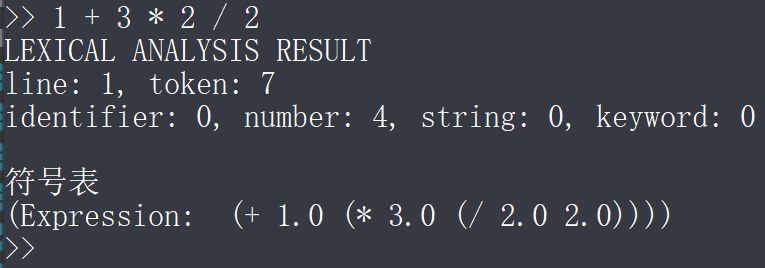


# 项目结果及分析

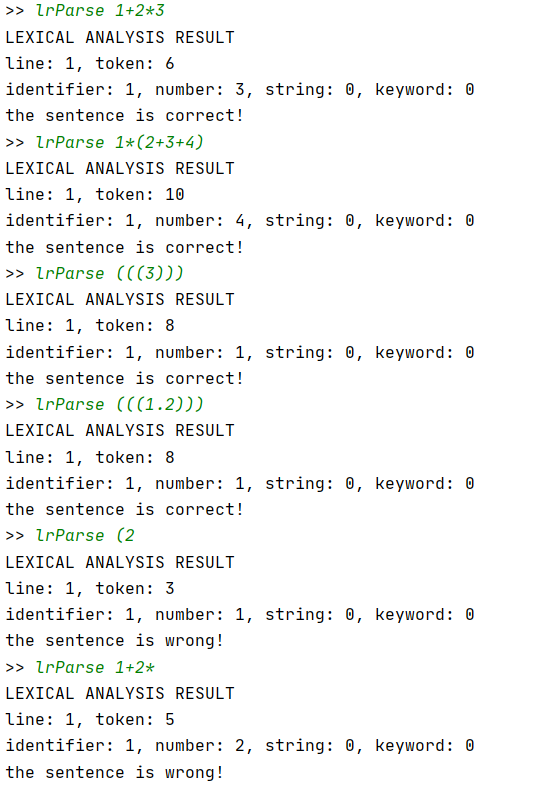
* 1. 使用了四种方式实现了语法分析器。

分别是yacc、LL(1)、LR、递归下降。

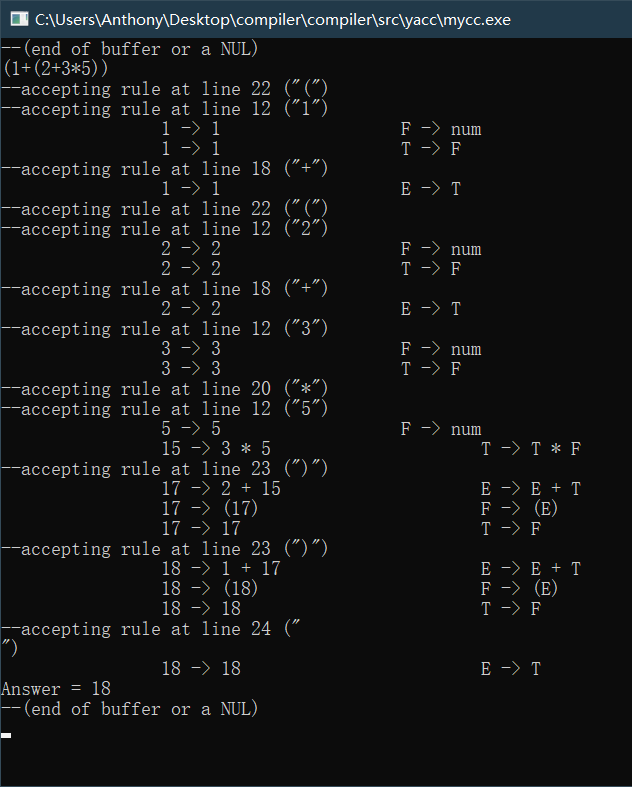
* + 1. **递归下降**



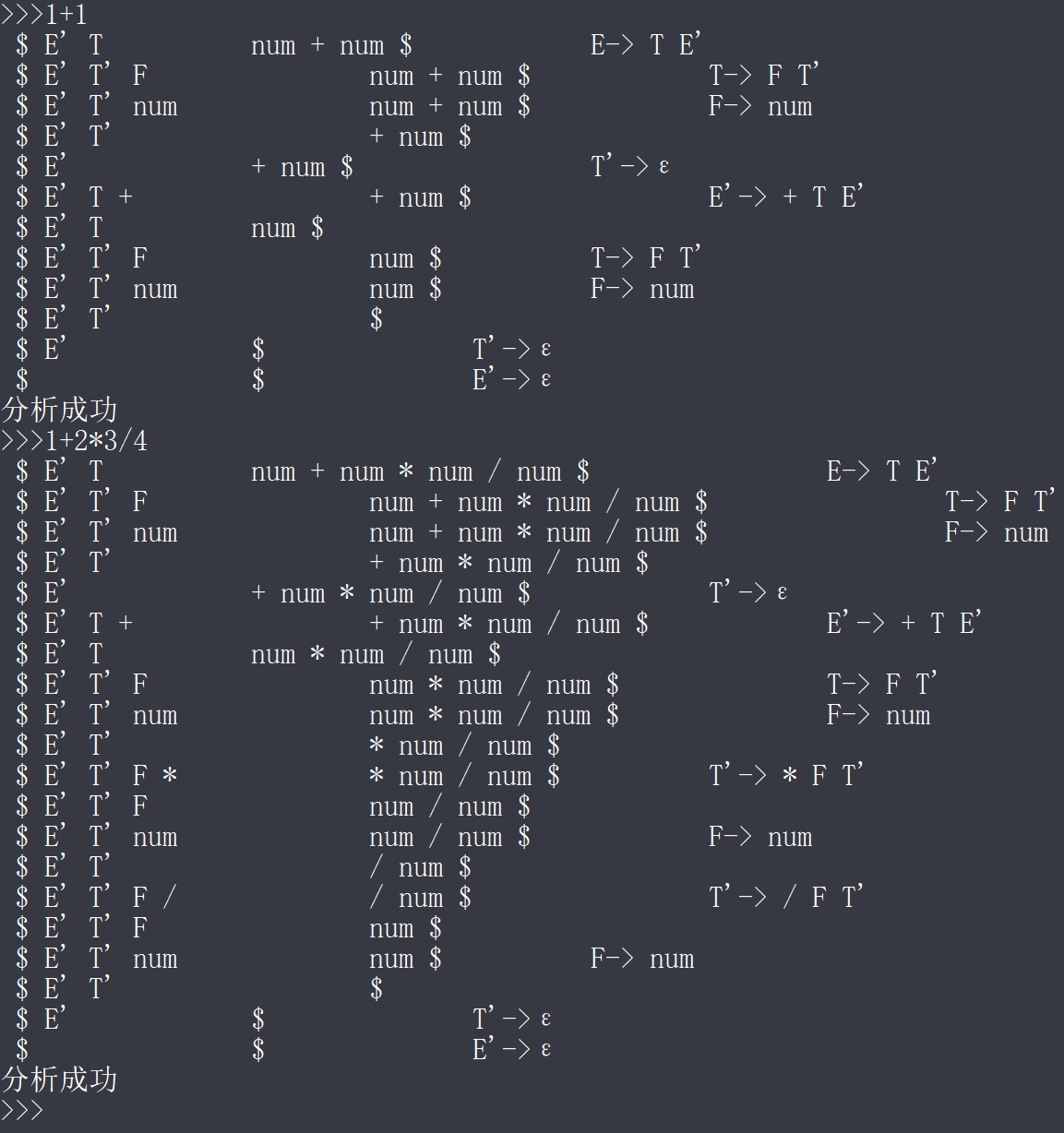
* + 1. **LR**



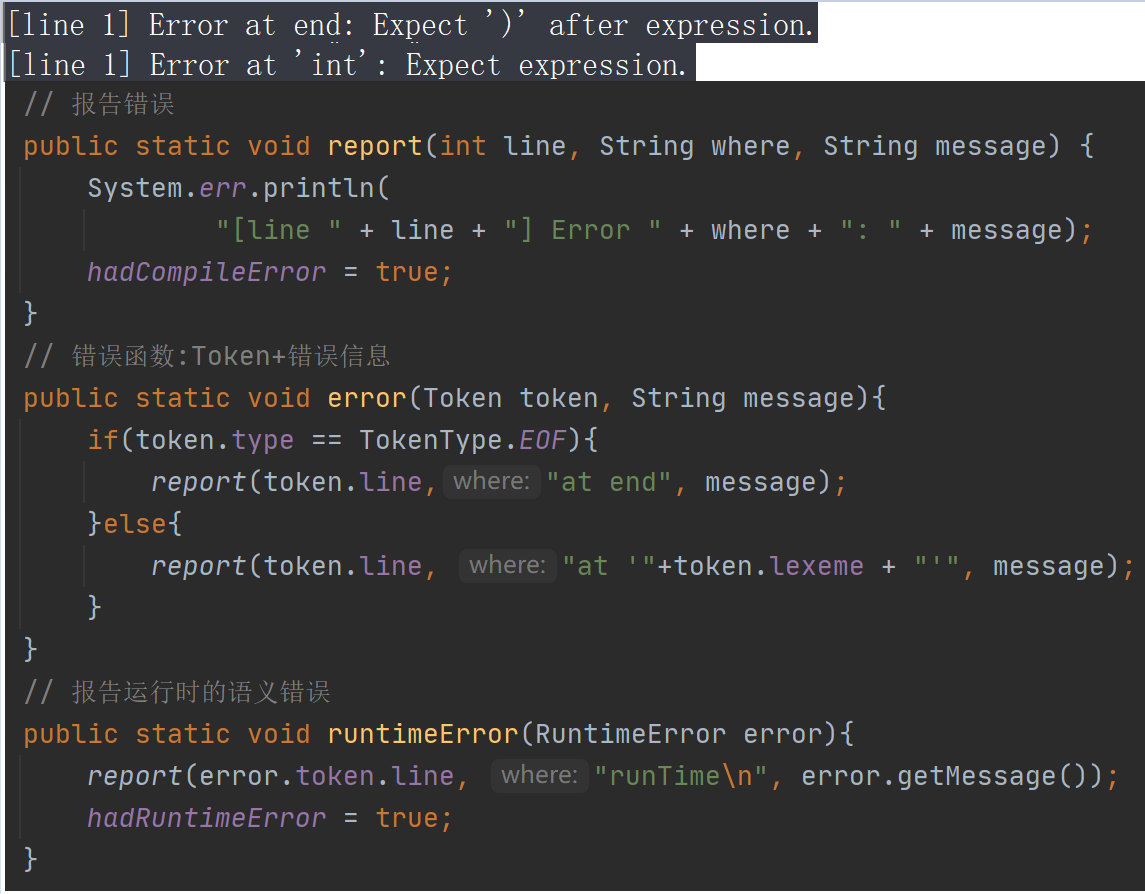
* + 1. **Yacc**



* + 1. **LL(1)**



* 1. 在递归下降语法分析器的基础上添加了语法制导功能，使得程序可以计算表达式的值并对运行时错误给出提示。



# 项目人员、进度安排及完成过程

## 5.1 任务分配

任务分配大致如下(文档编写与程序其他处理部分未计入)：

|  |  |  |
| --- | --- | --- |
|  | 任务 | 说明 |
| 李兴诚 | 递归下降分析 | Java手工编写的解释器 |
| 赵宸阳 | 解释器 |
| 刘邦辰 | LR语法分析 |
| 胡鼎新 | LL(1)语法分析，YYAC语法分析 | C与Python单独实现 |

## 进度安排

|  |  |
| --- | --- |
| 任务 | 说明 |
| 基本框架实现 | 对词法分析的记号流处理 |
| 递归下降分析与LR语法分析 | 语法分析 |
| LL(1)语法分析与YYAC语法分析 | 语法分析 |
| 解释器 | 扫描语法树，输出结果 |

## 完成过程

|  |  |
| --- | --- |
| 时间 | 进度 |
| 11月7日 | 解释器 |
| 11月14日 | yyac初稿 |
| 11月21日 | yyac语法分析与LL(1)语法分析 |
| 11月22日 | 递归下降分析 |
| 12月7日 | LR语法分析 |
| 12月16日 | Bug修复 |