**编译原理课程大作业报告一**

**语法分析器说明文档**



**成员： 涂远鹏-1652262**

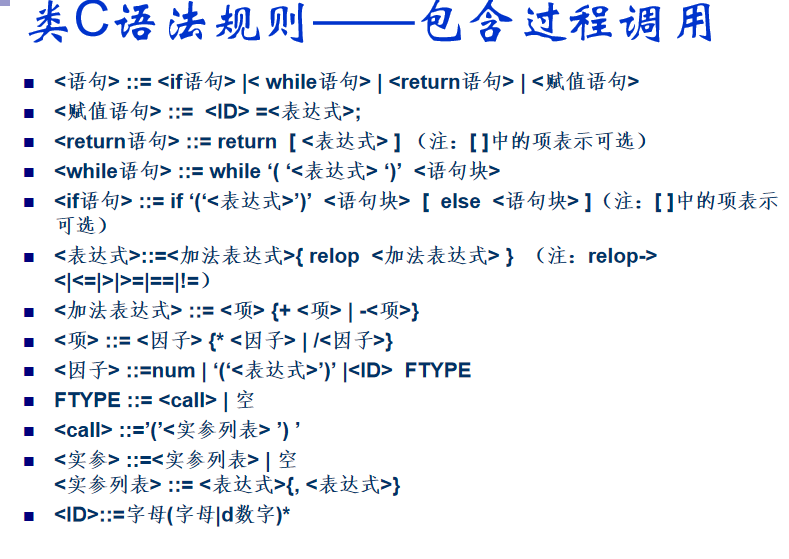
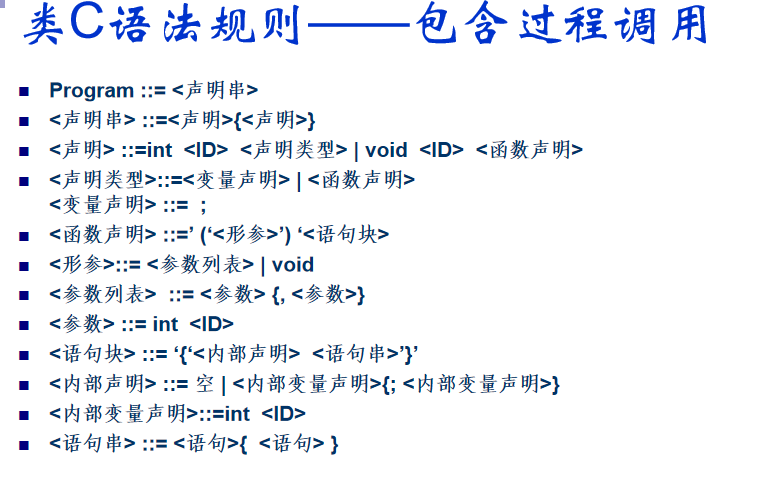
**成员： 黎盛烜-1652130**

**指导老师： 丁志军**

**日期： 2018年11月2日**

**1.语法分析器题目说明**

1.根据以下的类C语法规则分析给定的类C语言程序



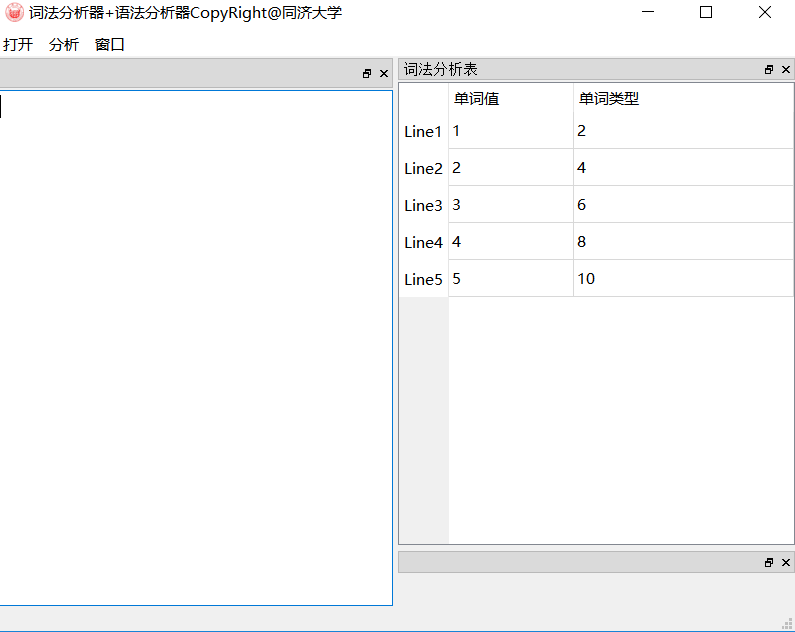
1. 输出文法的FIRST集合,FOLLOW集合
2. 输出类C程序的语法分析树结构：

(注由于类C语言程序分析出的LL(1)栈太大，所以此处没有做类C的LL(1)分析栈的输出)

1. 程序具有通用性，即所编制的语法分析程序能够适用于各类包含过程调用的类C程序。

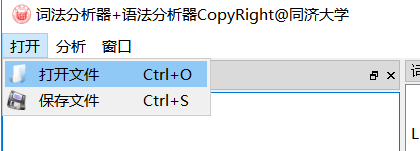
**2.程序使用说明**

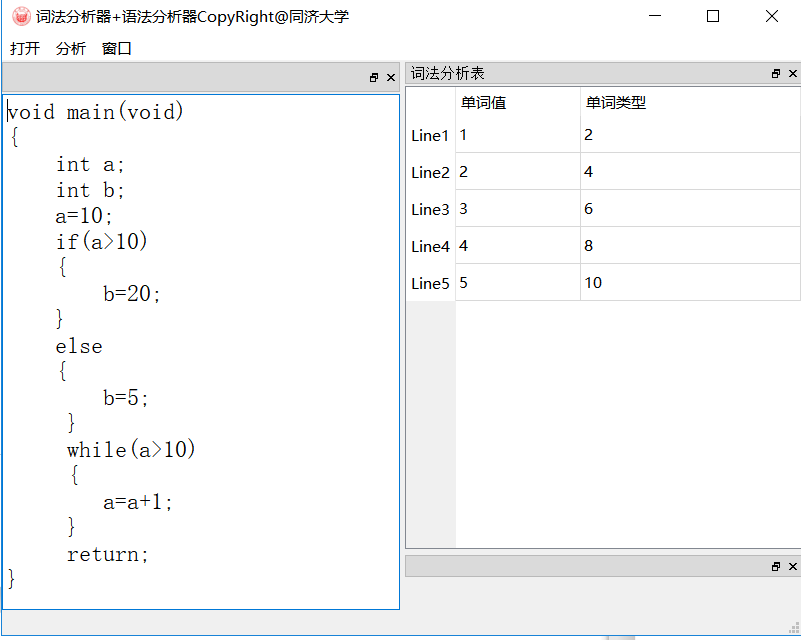
**2.1主界面**



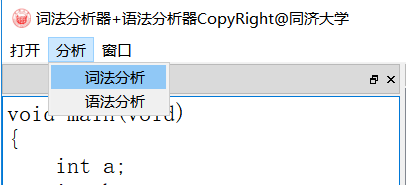
**2.2软件功能**

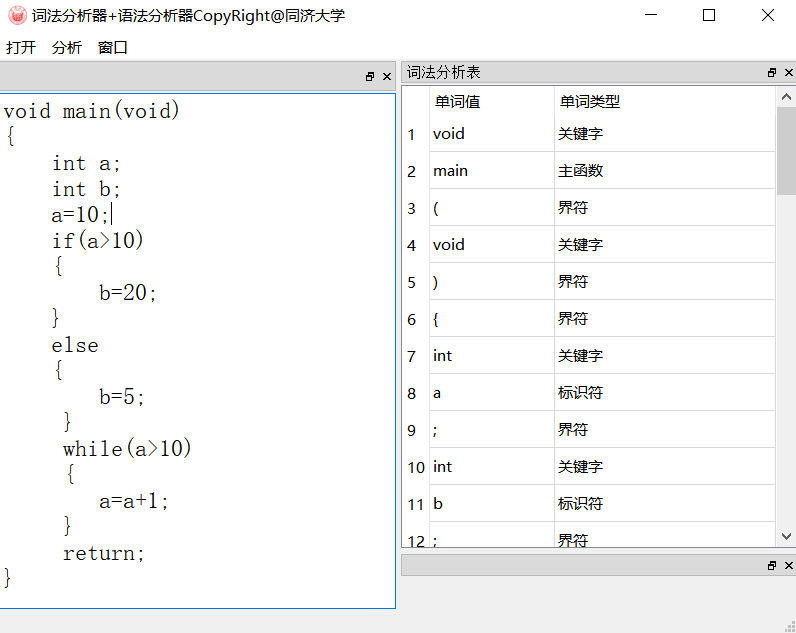
1. 首先选择输入的类C程序，选择的文件格式包含.txt，.c文件，选择成功后会在textedit文本框中显示该类C程序，显示如下：



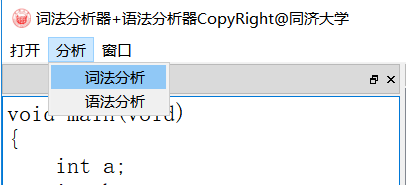


1. 由于语法分析需要使用到词法分析的结果所以新建了一个window添加了一个tableView控件显示语法分析的词法分析结果，点击词法分析可以得到第一步的词法分析结果，在右边的LexicalAnalysis的tableview表格控件中显示词法分析结果，左边一列为符号，右边列为该符号所属的类别，显示如下:

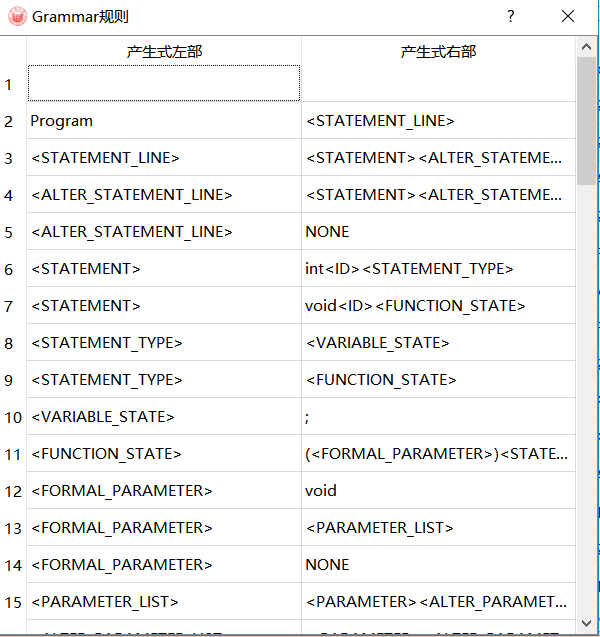




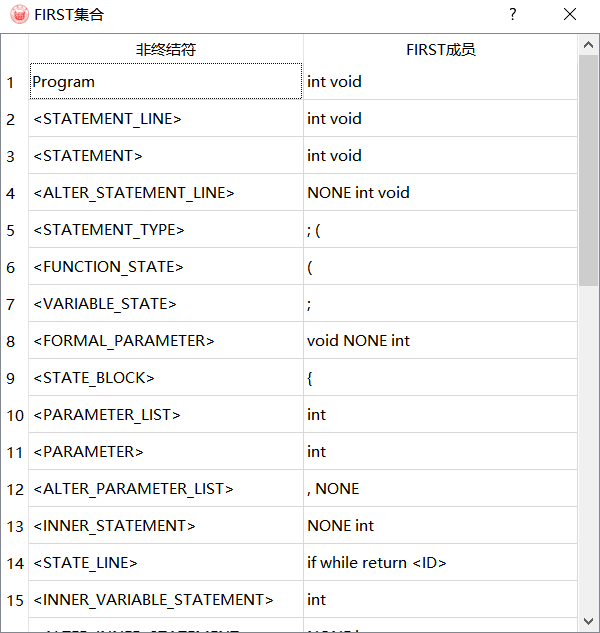
1. 点击分析中的语法分析,假如输入的类C分析程序语法正确的话，则会弹出三个Dialog以及在右下方的SyntaxAnalysis窗体中的QtreeWidget中显示该类C程序的语法分析树结构，三个Dialog分别为该类C文法的FIRST集合和FOLLOW集合以及该类C文法的所有产生式的内容，均使用tableView表格控件展示



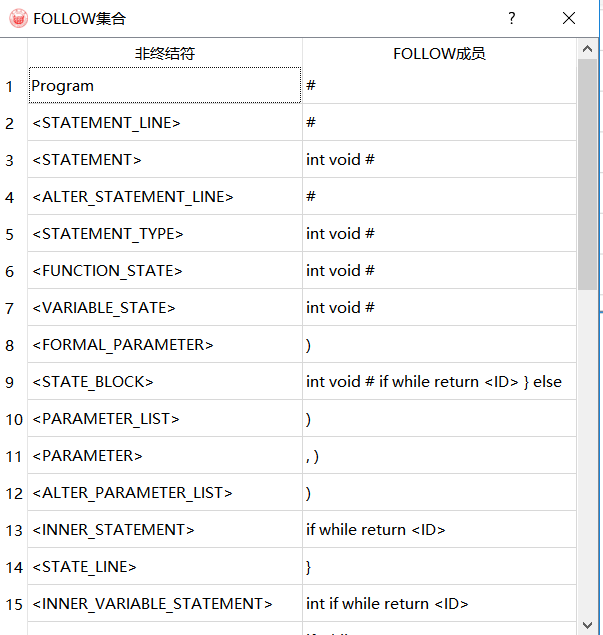
类C语法规则显示如下:



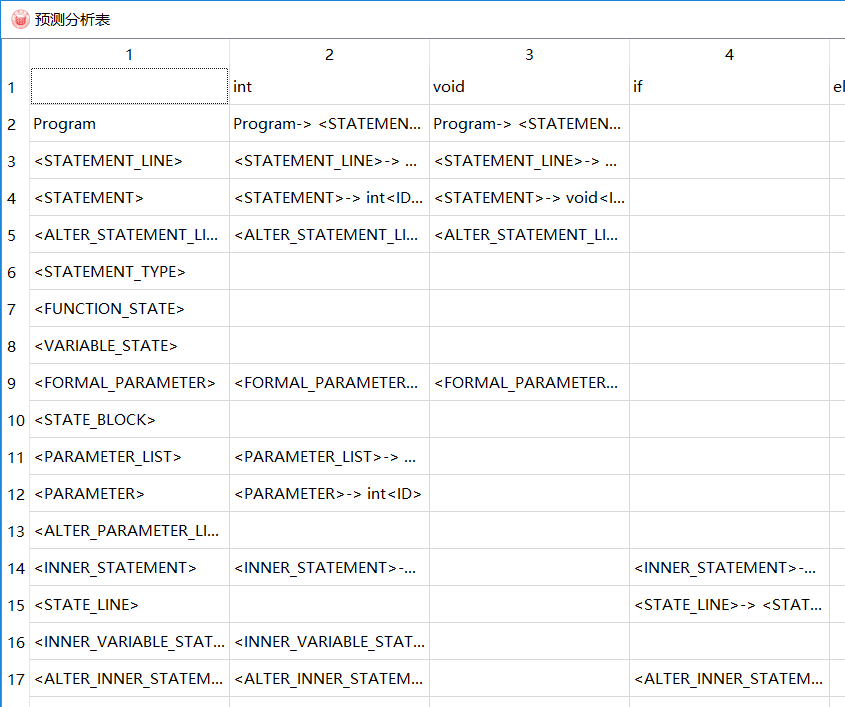
该文法的FIRST集合显示如下：



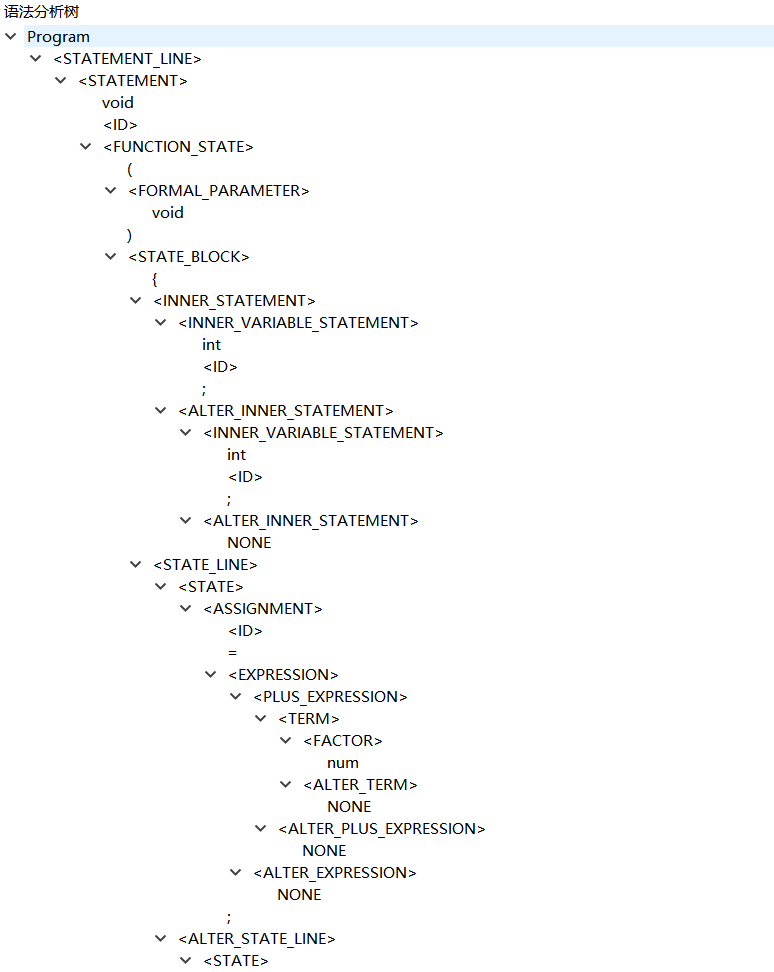
该文法的FOLLOW集合显示如下：



预测分析表显示如下：



显示的类C程序对应的语法分析树如下(由于整棵树太大，所以此处只截取一部分):



软件实现方法：

1. 类C程序的编辑与保存是使用textedit控件，通过getText()读取当前textedit中内容保存到指定文件中。
2. 使用QtreeWidget控件显示语法分析树
3. 使用LL(1)预测分析的方式得到整个语法分析的结果
4. FIRST集、FOLLOW集的显示使用TableView控件显示

3.设计思路

**3.1文法规则读入方式**

1. 由于处理之后的文法各个单词之间用空格划分开了，所以利用infile.get()函数即可读入一个完整的单词，每次处理一行，一行代表一个产生式
2. 判断读入的单词是否已经加入到当前的终结符表terminal\_sign中，表示已加入，假如已加入则直接建立该产生式左部与产生式右部的关系(具体做法为将proc[i].left赋值为该左部符号在非终结符表中的位置，proc[i].right数组从0到production\_maxlength(表示产生式右部最多该个数个单词)赋值为该产生式右部的单词在终结符或非终结符表中的编号并用terflag区分是在终结符表还是非终结符表)
3. 假如未加入，则将其加入终结符表中并置该符号的终结符表flag位为1，则继续判断非终结符表是否存有当前读入的符号，若存在则直接用编号建立关系，若不存在则动态生成新的非终结符并建立产生式对应关系



1. 读入并处理一个单词完毕置word\_flag位为0，重新读入一个单词直至读到换行符处理下一个文法产生式

**3.2 LL(1)消除左递归的方法**

此处没有用算法实现而是通过手动将原先的26条语法规则转成已经消除左递归的文法规则，消除后共有65个文法产生式，如下：

Program ::= <STATEMENT\_LINE>

<STATEMENT\_LINE> ::= <STATEMENT> <ALTER\_STATEMENT\_LINE>

<ALTER\_STATEMENT\_LINE> ::= <STATEMENT> <ALTER\_STATEMENT\_LINE>

<ALTER\_STATEMENT\_LINE> ::= NONE

<STATEMENT> ::= int <ID> <STATEMENT\_TYPE>

<STATEMENT> ::= void <ID> <FUNCTION\_STATE>

<STATEMENT\_TYPE> ::= <VARIABLE\_STATE>

<STATEMENT\_TYPE> ::= <FUNCTION\_STATE>

<VARIABLE\_STATE> ::= ;

<FUNCTION\_STATE> ::= ( <FORMAL\_PARAMETER> ) <STATE\_BLOCK>

<FORMAL\_PARAMETER> ::= void

<FORMAL\_PARAMETER> ::= <PARAMETER\_LIST>

<FORMAL\_PARAMETER> ::= NONE

<PARAMETER\_LIST> ::= <PARAMETER> <ALTER\_PARAMETER\_LIST>

<ALTER\_PARAMETER\_LIST> ::= , <PARAMETER> <ALTER\_PARAMETER\_LIST>

<ALTER\_PARAMETER\_LIST> ::= NONE

<PARAMETER> ::= int <ID>

<STATE\_BLOCK> ::= { <INNER\_STATEMENT> <STATE\_LINE> }

<INNER\_STATEMENT> ::= <INNER\_VARIABLE\_STATEMENT> <ALTER\_INNER\_STATEMENT>

<INNER\_STATEMENT> ::= NONE

<ALTER\_INNER\_STATEMENT> ::= <INNER\_VARIABLE\_STATEMENT> <ALTER\_INNER\_STATEMENT>

<ALTER\_INNER\_STATEMENT> ::= NONE

<INNER\_VARIABLE\_STATEMENT> ::= int <ID> ;

<STATE\_LINE> ::= <STATE> <ALTER\_STATE\_LINE>

<ALTER\_STATE\_LINE> ::= <STATE> <ALTER\_STATE\_LINE>

<ALTER\_STATE\_LINE> ::= NONE

<STATE> ::= <if\_STATE>

<STATE> ::= <while\_STATE>

<STATE> ::= <return\_STATE>

<STATE> ::= <ASSIGNMENT>

<ASSIGNMENT> ::= <ID> = <EXPRESSION> ;

<return\_STATE> ::= return <EXPRESSION> ;

<EXPRESSION> ::= NONE

<while\_STATE> ::= while ( <EXPRESSION> ) <STATE\_BLOCK>

<if\_STATE> ::= if ( <EXPRESSION> ) <STATE\_BLOCK> <ALTER\_ELSE\_BLOCK>

<ALTER\_ELSE\_BLOCK> ::= else <STATE\_BLOCK>

<ALTER\_ELSE\_BLOCK> ::= NONE

<EXPRESSION> ::= <PLUS\_EXPRESSION> <ALTER\_EXPRESSION>

<ALTER\_EXPRESSION> ::= <relop> <PLUS\_EXPRESSION> <ALTER\_EXPRESSION>

<ALTER\_EXPRESSION> ::= NONE

<relop> ::= <

<relop> ::= <=

<relop> ::= >

<relop> ::= >=

<relop> ::= ==

<relop> ::= !=

<PLUS\_EXPRESSION> ::= <TERM> <ALTER\_PLUS\_EXPRESSION>

<ALTER\_PLUS\_EXPRESSION> ::= + <TERM> <ALTER\_PLUS\_EXPRESSION>

<ALTER\_PLUS\_EXPRESSION> ::= - <TERM> <ALTER\_PLUS\_EXPRESSION>

<ALTER\_PLUS\_EXPRESSION> ::= NONE

<TERM> ::= <FACTOR> <ALTER\_TERM>

<ALTER\_TERM> ::= \* <FACTOR> <ALTER\_TERM>

<ALTER\_TERM> ::= / <FACTOR> <ALTER\_TERM>

<ALTER\_TERM> ::= NONE

<FACTOR> ::= num

<FACTOR> ::= ( <EXPRESSION> )

<FACTOR> ::= <ID> FTYPE

FTYPE ::= <call>

FTYPE ::= NONE

<call> ::= ( <ACTUAL\_PARAMETER> )

<ACTUAL\_PARAMETER> ::= <ACTUAL\_PARAMETER\_LIST>

<ACTUAL\_PARAMETER> ::= NONE

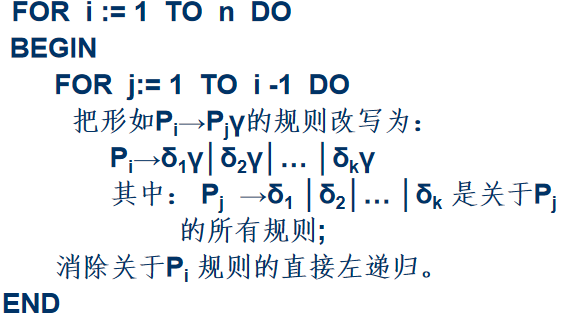
<ACTUAL\_PARAMETER\_LIST> ::= <EXPRESSION> <ALTER\_ACTUAL\_PARAMETER\_LIST>

<ALTER\_ACTUAL\_PARAMETER\_LIST>::=,<EXPRESSION><ALTER\_ACTUAL\_PARAMETER\_LIST>

<ALTER\_ACTUAL\_PARAMETER\_LIST> ::= NONE

左递归手动消除的算法思想如下：

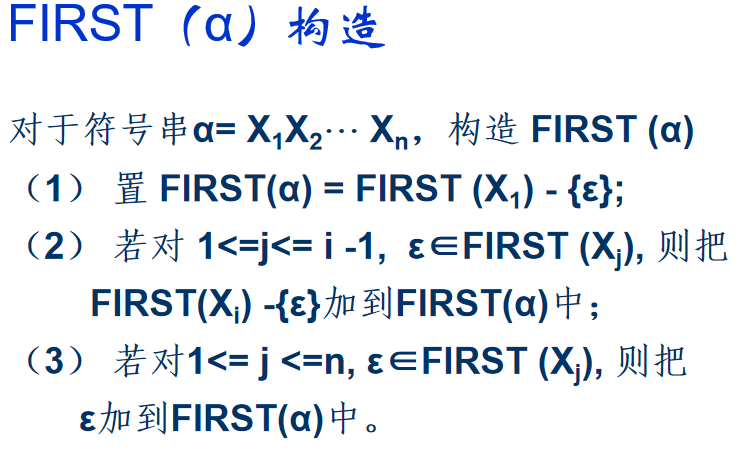
1. 首先将带有||号的文法产生式分解为生成确定的右部的产生式
2. 将剩余的文法规则按照如下结构进行产生式遍历



3)遍历上面产生式集合结果并删除永不使用的产生式

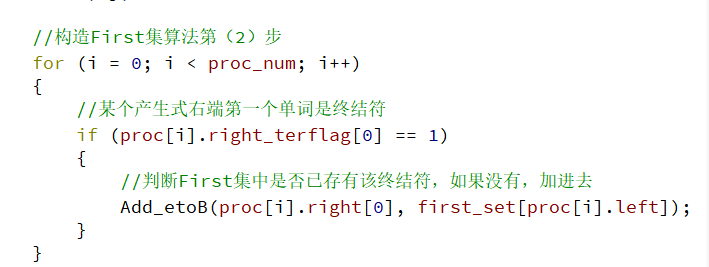
**3.3构造FIRST集合算法步骤**

算法步骤：



具体实现：

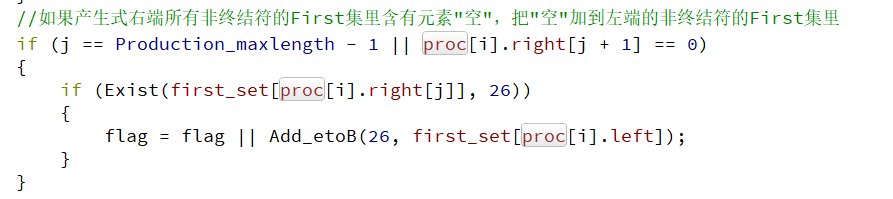
1)首先查找右部第一个单词即为终结符的产生式，然后把该终结符加入到该产生式右部非终结符的first集合中并判断是否之前是否已经将该终结符加入:



1. 对于产生式第一个符号为非终结符的,把该产生式的右部的第一个非终结符的FIRST集加入到左端的非终结符的FIRST集中,除此之外，如果产生式右端的某个非终结符的First集里含有元素"空"，把下一个符号的First集加到左端的非终结符的First集里:



3)处理极端情况:该产生式的右部的所有非终结符均包含”空”元素,那么也将”空”加入到该产生式左部的FIRST集合中:



4)返回步骤一，遍历所有产生式，直至判断结束为止

**3.4构造FOLLOW集合算法步骤**

算法步骤：

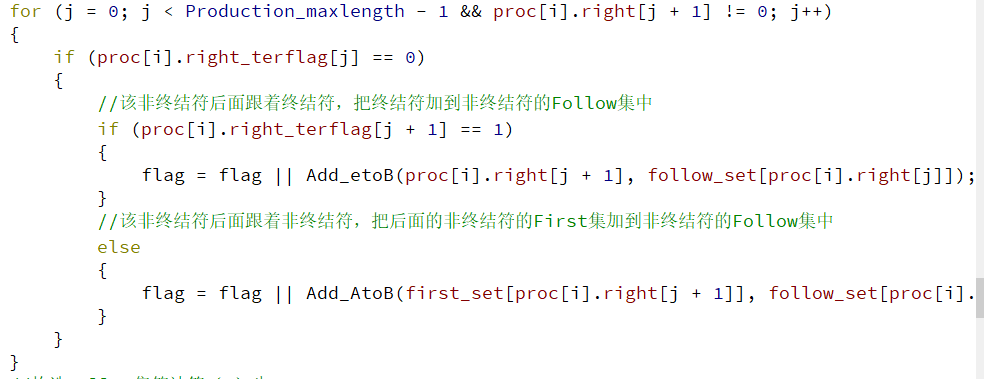


具体实现：

1)若A为文法开始符号，置#于FOLLOW(A）中



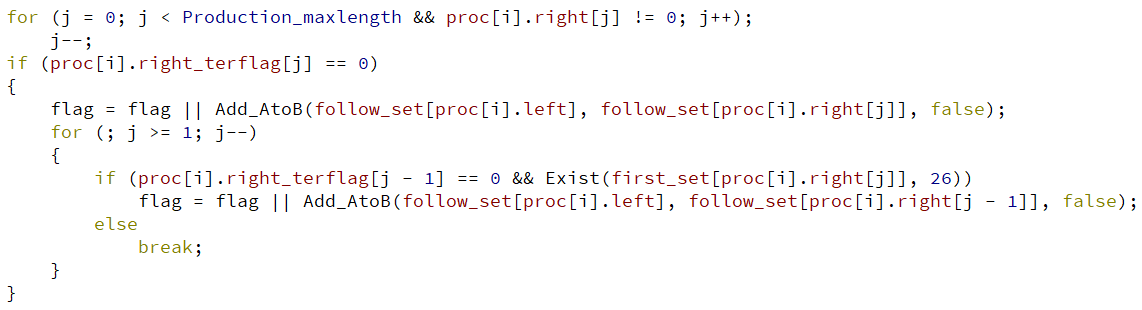
2)若有产生式B→αAβ,则将FIRST(β) - {ε}加到FOLLOW (A)中,此处需要分两种情况，一种为A后面跟的为终结符那么就只需要将该终结符加入到FOLLOW表中，一种为后面跟的为非终结符那么就需要将该后面跟的非终结符的FIRST集加入到FOLLOW中，因为A后面紧跟的元素就是该非终结符的FIRST成员：



1. 若有B→αA或B →αAβ, 且β->ε则将FOLLOW(B)加到FOLLOW(A)中，此处实现为：

首先找到该产生式右部的最后一个符号，判断该符号是否为终结符，假如不是，那么就将产生式左部的FOLLOW集合加入到右部最后一个符号的FOLLOW集中，并循环判断该产生式右部最后一个符号产生是否为空(此处通过判断该符号的FIRST集为空达到)，假如为空，那么也向倒数第二个非终结符的FOLLOW集中加入该产生式左部的FIRST集，直至右部符号全部判断结束为止。

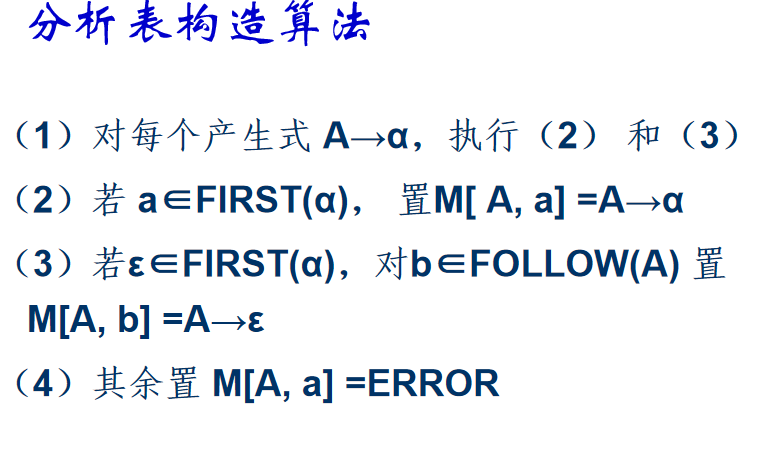
通过Add\_AtoB(follow\_set[proc[i].left], follow\_set[proc[i].right[j - 1]], false);步骤实现加入follow操作



4)反复使用以上规则, 直至 FOLLOW(A)不再增大为止.

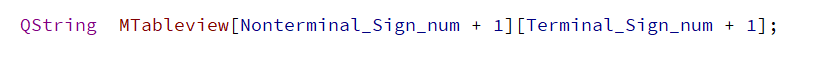
**3.5构造预测分析表算法步骤**

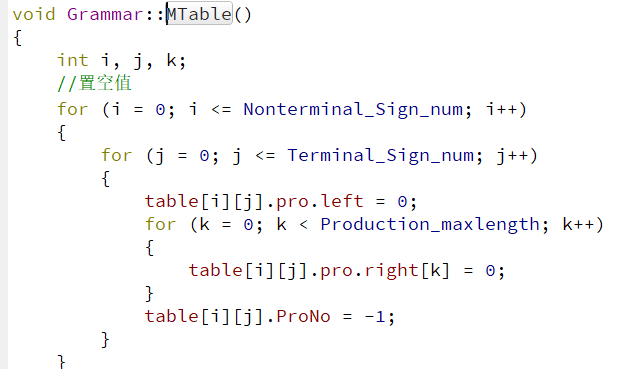
算法步骤：



具体实现：

1)LL(1)分析表用一个二维矩阵表示，其中每个非终结符对应一行，每个终结符对应一列，一个非终结符和一个终结符可以确定矩阵中的一个元素，元素的值表示该非终结符和该终结符对应的产生式。每个矩阵元素都是一个符号串，所有元素初始化为””：

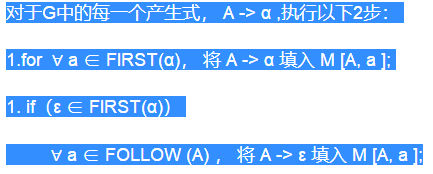




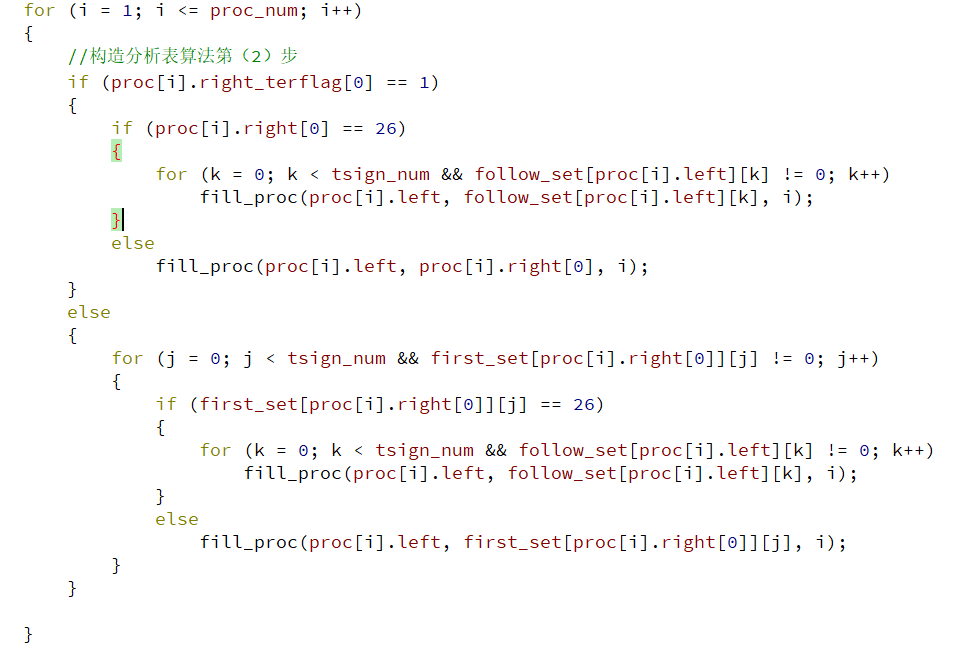
注：其中table[i][j]表示分析表第i行第j列的成员，ProNo用于记录该成员是产生式中的第几项，空的话为-1。pro记录该成员内容，pro.right[]数组记录该产生式的右部从左到右依次排列的终结符/非终结符在终结符表/非终结符表中的位置，pro.left为产生式左部在终结符表/非终结符表中的位置，pro.left\_terflag用于记录该符号是否为终结符，right\_terflag[]数组也是类似功能.

2)构造LL(1)表时，根据文法和各个产生式的First集，填写LL(1)分析表的内容。各产生式只要将右端填入到对应的表项中即可，用空串表示Error。在根据LL(1)分析表选择产生式进行推导时，若查到的产生式为空串表示无相应的产生式可选，不匹配错误；否则根据符号串得到相应的产生式进行推导，即进行压栈操作。

换句话说就是如下的算法：



1. 首先判断该产生式右部第一个符号是否为终结符号，若是，然后判断该符号是否为ε，若为ε就把该产生式左部和产生式左部的follow集对应的表项填入第i个产生式，若不是ε，就填入该产生式左部和该终结符对应的表项第i个产生式。
2. 假如该产生式右部第一个符号不是为终结符号，就遍历该产生式右部的所有单词的FIRST集合并判断该符号是否为ε，若为ε就把该产生式左部和产生式左部的follow集对应的表项填入第i个产生式，若不是ε，就将该产生式左部和该非终结符对应的FIRST集所有成员对应的表项填入第i个产生式。
3. 遍历所有产生式得出完整分析表



**3.6构造语法分析树算法步骤**

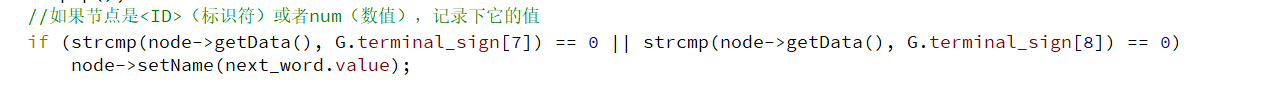
语法分析树的构造过程与语法分析的过程同时进行：

1)首先定义一颗树结构，STree其数据结构如下：

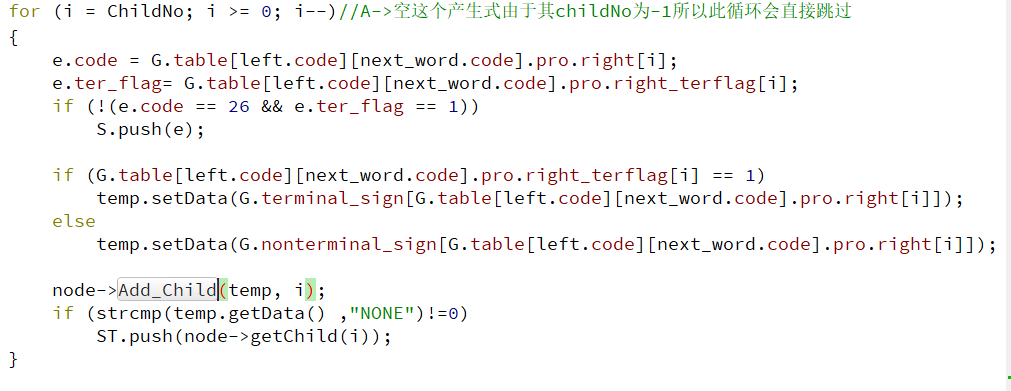


一个Stree包含该节点的数据Data成员，child\_num儿子成员个数，child为该节点的子树，name记录变量名，也包含了一些用于封装class STree的函数，方便获取节点信息

2)栈顶符号x为终结符且与当前输入符a匹配，由于该终结符已无法产生其他的子树了，所以返回父节点，弹出STree中的栈顶元素即该终结符，并设置节点的name为该单词的value值，此时该节点还未加入到语法分析树中：

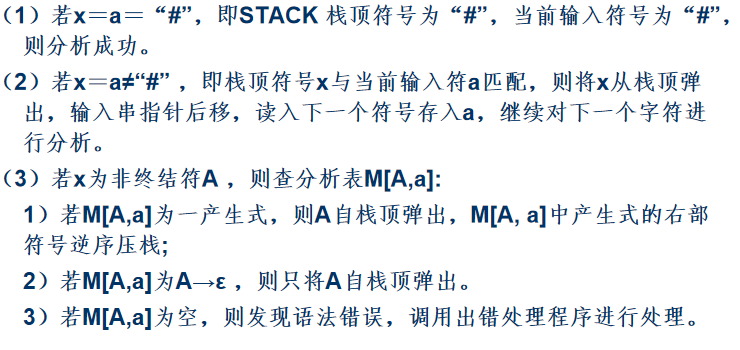


3)假如x为非终结符且该非终结符与当前输入符a在分析表中的位置为一产生式，则构造一颗子树，将该产生式右部的所有终结符以及非终结符均作为该节点的孩子放入，最后将该子树压入STree栈中，相当于将该子树与大的语法分析树连接：



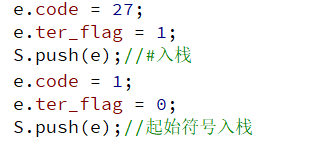
**3.7 预测分析算法步骤**

算法步骤：

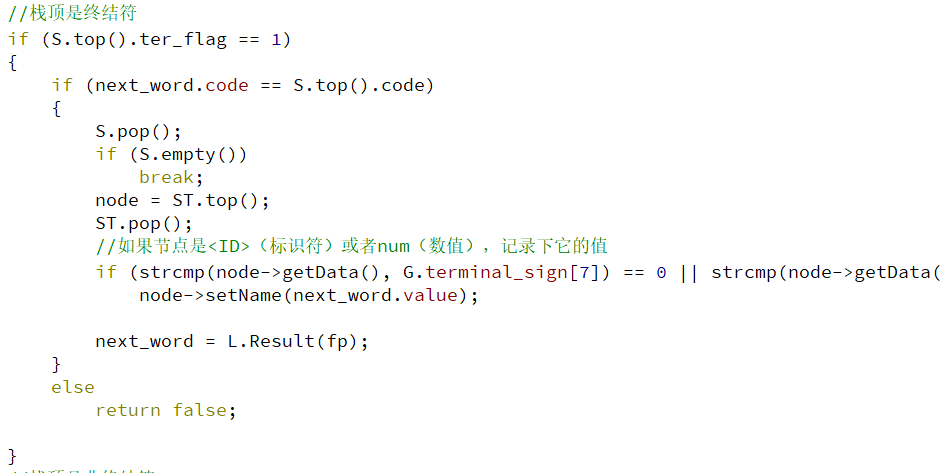


具体实现：

1)首先将#和起始符号压入分析栈：



2)栈顶符号x为终结符且与当前输入符a匹配，则将x从栈顶弹出，输入串指针后移，读入下一个符号存入a，继续对下一个字符进行分析。



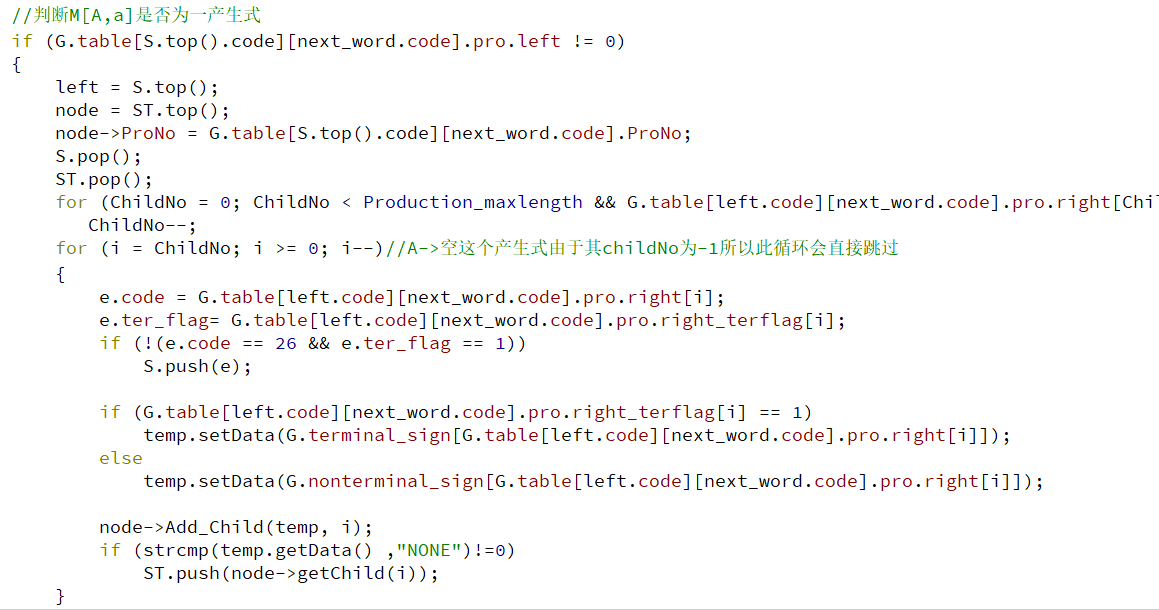
3)若x为非终结符A ，则查分析表M[A,a]:

1）若M[A,a]为一产生式，则A自栈顶弹出，M[A, a]中产生式的右部符号逆序压栈;

2）若M[A,a]为A→ε ，则只将A自栈顶弹出。

3）若M[A,a]为空，则发现语法错误，调用出错处理程序进行处理

实现如下：





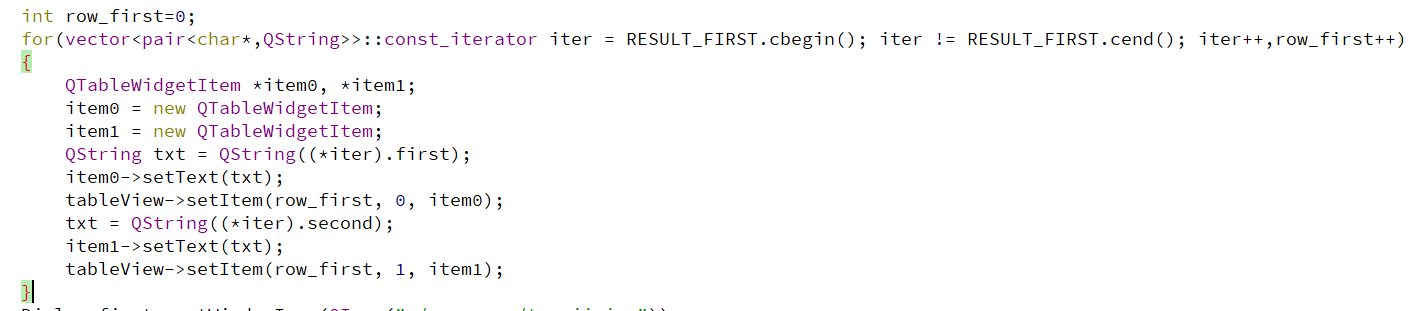
**3.8显示语法分析树的方式**

显示语法分析树是通过QT的QTreeWidget控件实现，根据之前语法分析时STree栈中的内容建立一颗语法分析树，每次都取栈顶元素并新建一个QTreeWidgetItem成员表示一个节点并循环取该节点的孩子，用QTreeWidgetItem的addChild函数新建孩子节点并设置该元素的显示内容为该栈顶元素的孩子的Data内容:



**3.9显示FIRST/FOLLOW集合/grammar规则/预测分析表的方式**

显示FIRST/FOLLOW集合/grammar规则/预测分析表所用的Qt控件为QtableWidget，



以FIRST集的表示为例，使用数据结构vector<pair<char\*,QString>>容器存储FIRST集的左侧的非终结符与右侧的对应该非终结符的终结符集合，然后在输出到表格时，每一行均分别新建两个QTableWidgetItem成员分别表示上面的两项即可。

**3.逻辑结构与物理结构**

逻辑结构：树形结构，栈结构，线性结构，图形结构

物理结构：栈，容器，数组，链表，自定义类结构，动态数组

自定义结构如下：

1.用于记录单词：

struct Word {

int code;

char value[50];

};

2.分析表结构：

struct Elem {

int code;

bool ter\_flag;//是否为终结符

};

struct Production {

int left;//左部符号在非终结符表数组中的位置

bool left\_terflag;

int right[Production\_maxlength];//右部符号在非终结符表/终结符表数组中的位置

bool right\_terflag[Production\_maxlength];//右部符号是否为终结符

};

struct TableElem {

Production pro;//预测分析表单个表项内容

int ProNo;//存入的是第几个产生式

};

其他元素成员使用的物理结构：

int proc\_num;//产生式数目

int tsign\_num;//终止符号数目

int ntsign\_num;//非终结符标志数目

Production proc[Production\_num];

int first\_set[Nonterminal\_Sign\_num + 1][Terminal\_Sign\_num + 1];

int follow\_set[Nonterminal\_Sign\_num + 1][Terminal\_Sign\_num + 1];

const char\* terminal\_sign[Terminal\_Sign\_num + 1];//终结符集合

char\* nonterminal\_sign[Nonterminal\_Sign\_num + 1];//非终结符集合

TableElem table[Nonterminal\_Sign\_num + 1][Terminal\_Sign\_num + 1];//预测分析表

vector <pair< char \*,QString>>FIRST\_VECTOR;//用于输出到tableview控件FIRST集存储vector

vector <pair< char \*,QString>>FOLLOW\_VECTOR;//用于输出到tableview控件FOLLOW集存储vector

vector <pair< QString,QString>>Grammar\_Analysis;//用于输出到tableview控件语法规则集存储vector

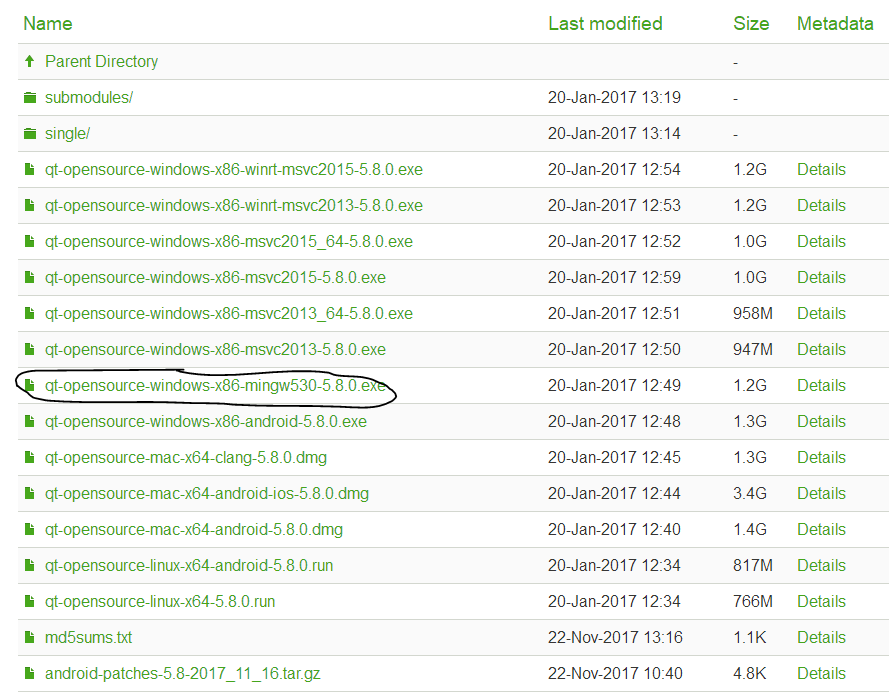
QString MTableview[Nonterminal\_Sign\_num + 1][Terminal\_Sign\_num + 1];//用于输出到tableview控件的预测分析表集存储数组

**4.调试运行环境**

编译环境：

Qt5+MingGW32位版本(需要MingGW编译器编译，假如使用MSVC2015编译器进行编译运行出来的exe会无法生成语法分析树中途crash，这是因为MSVC2015不支持中文编码的问题，MSVC2015编译器中,中文输出会提示”常量中有换行符错误”)

QT5+MingGW32下载64位版本地址为：http://download.qt.io/archive/qt/5.8/5.8.0/



运行环境：

Microsoft windows 10 专业版(64位)

内存 8GB（1600 Mhz）

注：由于文法规则grammar-en.txt是利用本地绝对路径读入的，所以移植到其他机器运行时需要修改syntax.cpp中init\_grammar()函数中grammar-en.txt的绝对路径才能正确运行生成语法分析树否则会提示语法分析错误