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

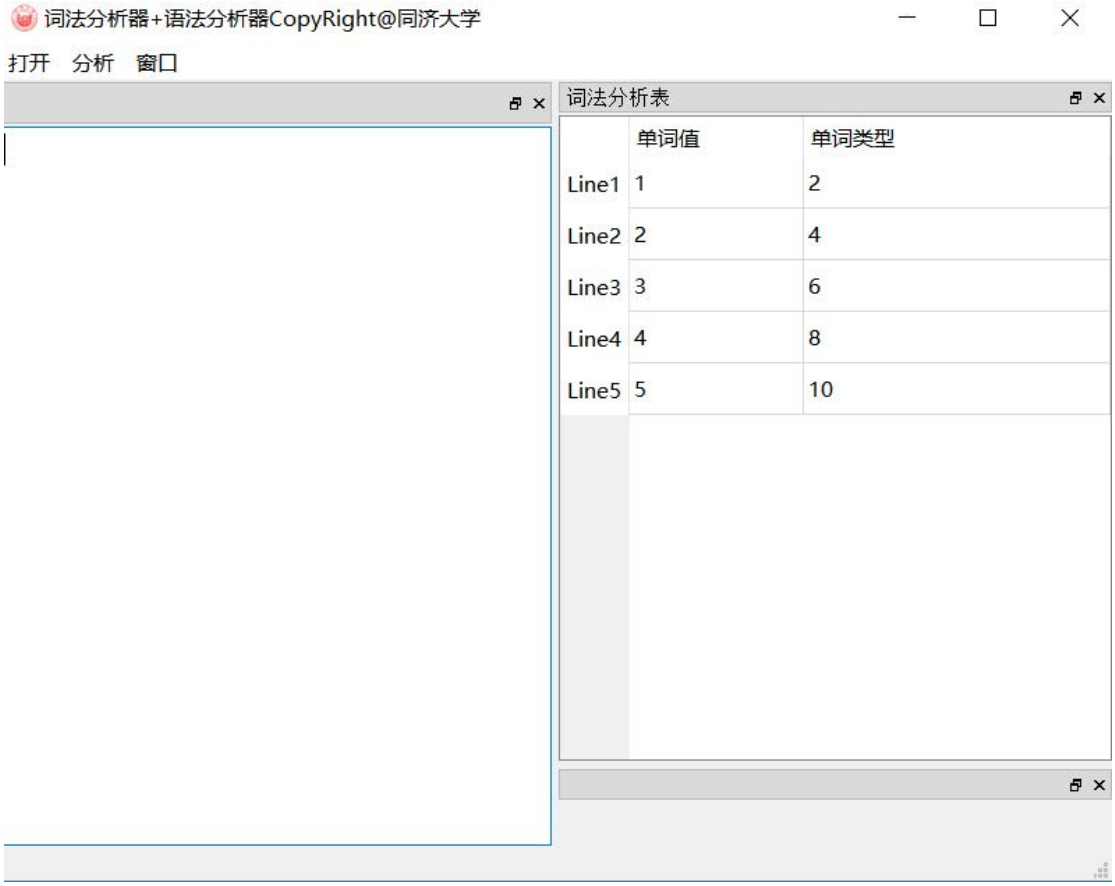
(2) 输出类 C 程序的语法分析树结构:

(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)

2．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。

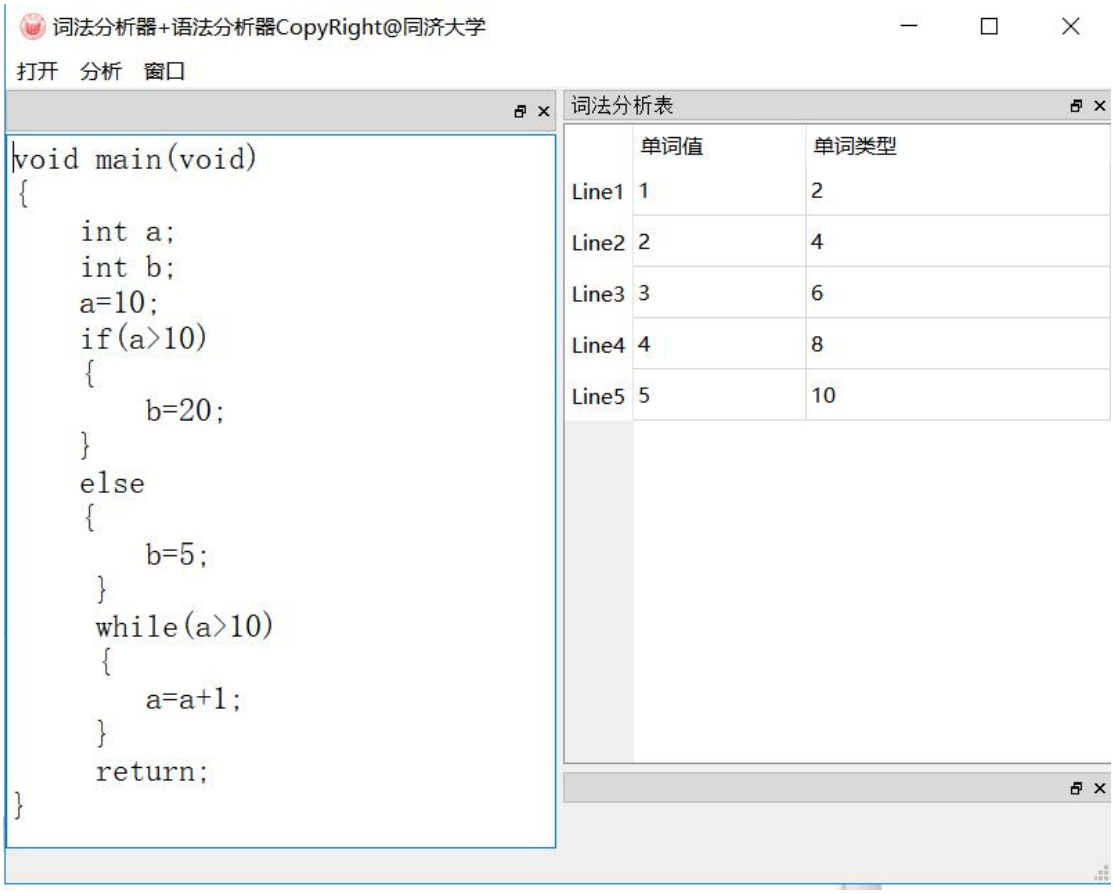
2.程序使用说明

2.1 主界面

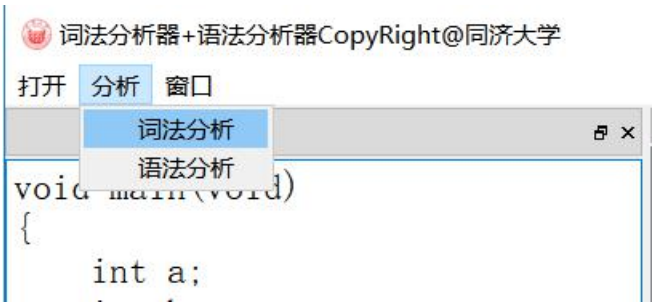


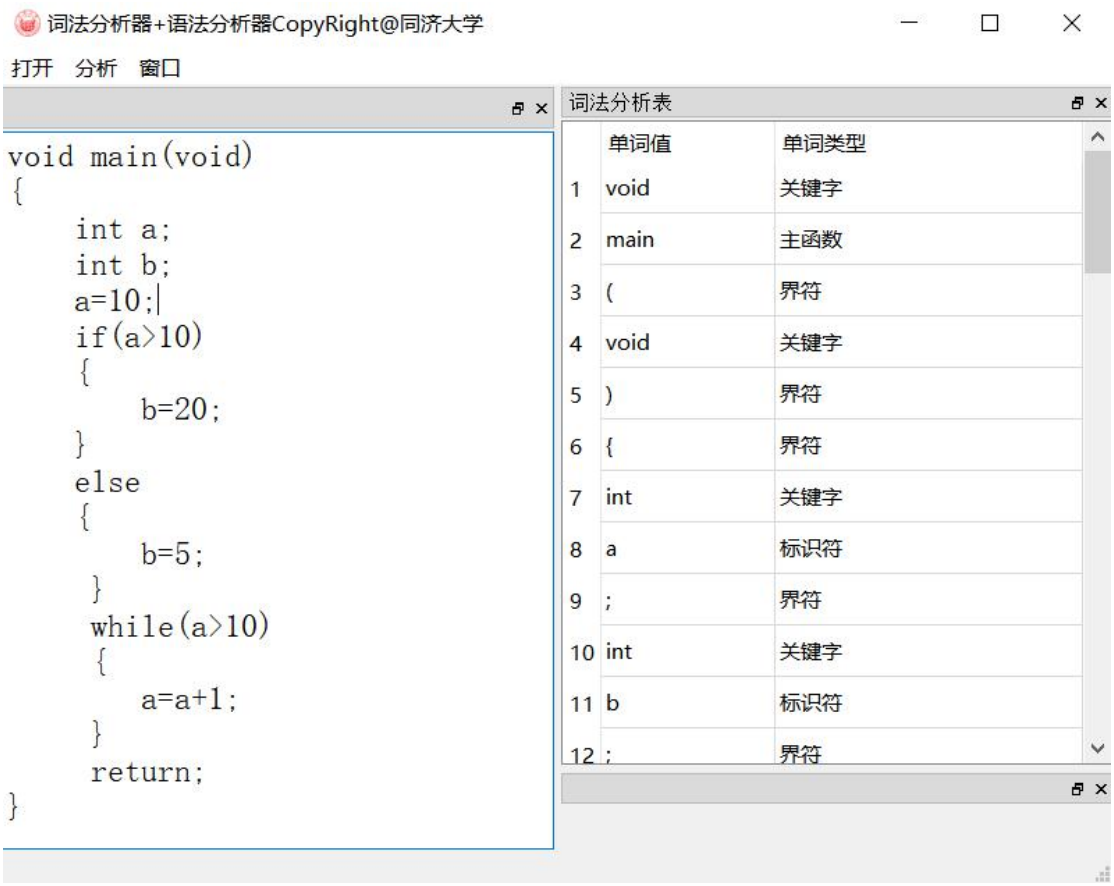
2.2 软件功能

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：



2) 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:





3) 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:

Grammar规则

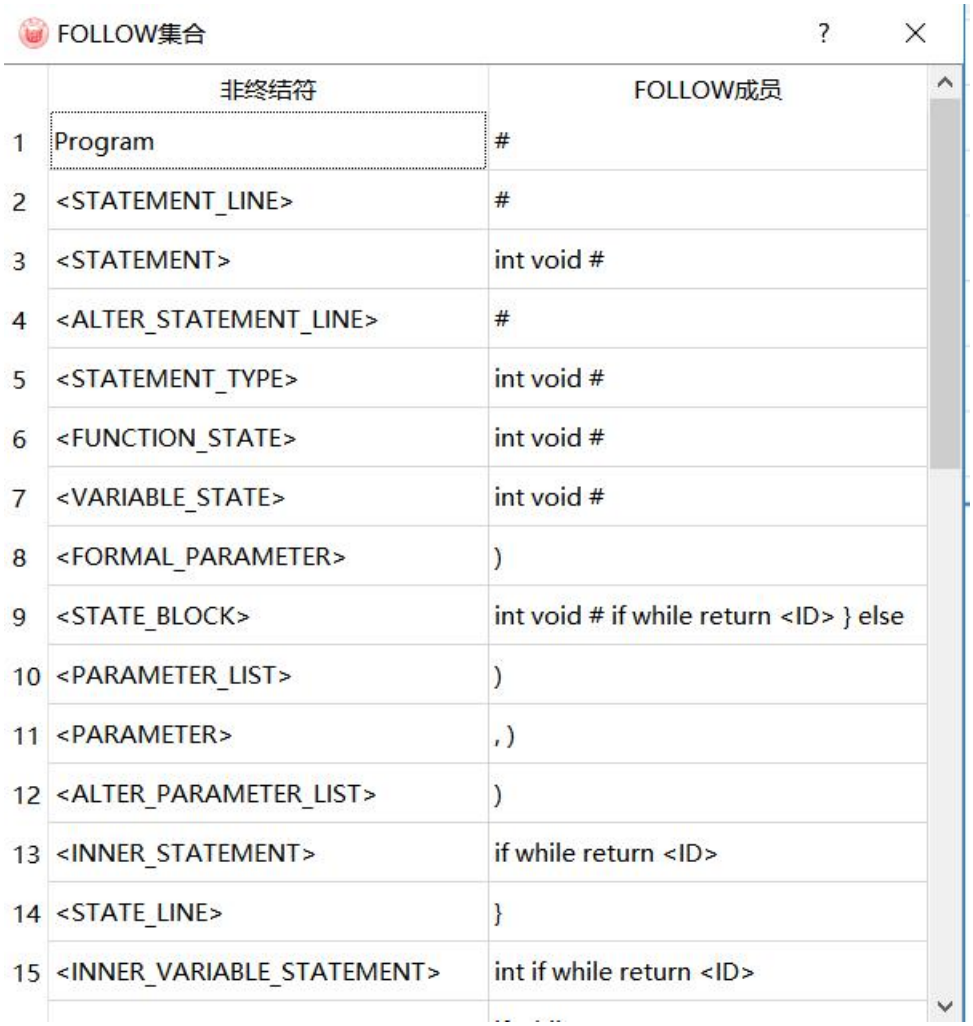
	产生式左部	产生式右部
1		
2	Program	<STATEMENT_LINE>
3	<STATEMENT_LINE>	<STATEMENT><ALTER_STATEME...
4	<ALTER_STATEMENT_LINE>	<STATEMENT><ALTER_STATEME...
5	<ALTER_STATEMENT_LINE>	NONE
6	<STATEMENT>	int<ID><STATEMENT_TYPE>
7	<STATEMENT>	void<ID><FUNCTION_STATE>
8	<STATEMENT_TYPE>	<VARIABLE_STATE>
9	<STATEMENT_TYPE>	<FUNCTION_STATE>
10	<VARIABLE_STATE>	;
11	<FUNCTION_STATE>	(<FORMAL_PARAMETER>)<STATE...
12	<FORMAL_PARAMETER>	void
13	<FORMAL_PARAMETER>	<PARAMETER_LIST>
14	<FORMAL_PARAMETER>	NONE
15	<PARAMETER_LIST>	<PARAMETER><ALTER_PARAMET...

该文法的 FIRST 集合显示如下：

FIRST集合

	非终结符	FIRST成员
1	Program	int void
2	<STATEMENT_LINE>	int void
3	<STATEMENT>	int void
4	<ALTER_STATEMENT_LINE>	NONE int void
5	<STATEMENT_TYPE>	; (
6	<FUNCTION_STATE>	(
7	<VARIABLE_STATE>	;
8	<FORMAL_PARAMETER>	void NONE int
9	<STATE_BLOCK>	{
10	<PARAMETER_LIST>	int
11	<PARAMETER>	int
12	<ALTER_PARAMETER_LIST>	, NONE
13	<INNER_STATEMENT>	NONE int
14	<STATE_LINE>	if while return <ID>
15	<INNER_VARIABLE_STATEMENT>	int

该文法的 FOLLOW 集合显示如下：



	非终结符	FOLLOW成员
1	Program	#
2	<STATEMENT_LINE>	#
3	<STATEMENT>	int void #
4	<ALTER_STATEMENT_LINE>	#
5	<STATEMENT_TYPE>	int void #
6	<FUNCTION_STATE>	int void #
7	<VARIABLE_STATE>	int void #
8	<FORMAL_PARAMETER>	)
9	<STATE_BLOCK>	int void # if while return <ID> } else
10	<PARAMETER_LIST>	)
11	<PARAMETER>	,)
12	<ALTER_PARAMETER_LIST>	)
13	<INNER_STATEMENT>	if while return <ID>
14	<STATE_LINE>	}
15	<INNER_VARIABLE_STATEMENT>	int if while return <ID>

预测分析表显示如下：

预测分析表

	1	2	3	4	
1		int	void	if	el
2	Program	Program-> <STATEMEN...	Program-> <STATEMEN...		
3	<STATEMENT_LINE>	<STATEMENT_LINE>-> ...	<STATEMENT_LINE>-> ...		
4	<STATEMENT>	<STATEMENT>-> int<ID...	<STATEMENT>-> void<I...		
5	<ALTER_STATEMENT_LI...	<ALTER_STATEMENT_LI...	<ALTER_STATEMENT_LI...		
6	<STATEMENT_TYPE>				
7	<FUNCTION_STATE>				
8	<VARIABLE_STATE>				
9	<FORMAL_PARAMETER>	<FORMAL_PARAMETER...	<FORMAL_PARAMETER...		
10	<STATE_BLOCK>				
11	<PARAMETER_LIST>	<PARAMETER_LIST>-> ...			
12	<PARAMETER>	<PARAMETER>-> int<ID>			
13	<ALTER_PARAMETER_LI...				
14	<INNER_STATEMENT>	<INNER_STATEMENT>-...		<INNER_STATEMENT>-...	
15	<STATE_LINE>			<STATE_LINE>-> <STAT...	
16	<INNER_VARIABLE_STAT...	<INNER_VARIABLE_STAT...			
17	<ALTER_INNER_STATEM...	<ALTER_INNER_STATEM...		<ALTER_INNER_STATEM...	

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):

语法分析树

- Program
  - <STATEMENT_LINE>
    - <STATEMENT>
      - void
      - <ID>
      - <FUNCTION_STATE>
        - (
        - <FORMAL_PARAMETER>
          - void
        -)
        - <STATE_BLOCK>
          - {
          - <INNER_STATEMENT>
            - <INNER_VARIABLE_STATEMENT>
              - int
              - <ID>
              - ;
            - <ALTER_INNER_STATEMENT>
              - <INNER_VARIABLE_STATEMENT>
                - int
                - <ID>
                - ;
              - <ALTER_INNER_STATEMENT>
                - NONE
          - <STATE_LINE>
            - <STATE>
              - <ASSIGNMENT>
                - <ID>
                - =
                - <EXPRESSION>
                  - <PLUS_EXPRESSION>
                    - <TERM>
                      - <FACTOR>
                        - num
                      - <ALTER_TERM>
                        - NONE
                    - <ALTER_PLUS_EXPRESSION>
                      - NONE
                  - <ALTER_EXPRESSION>
                    - NONE
                - ;
            - <ALTER_STATE_LINE>
              - <STATE>

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

```
for (i = 1; i <= ntsign_num; i++)
{
    if (equal(temp, nonterminal_sign[i]))
    {
        nonterminal_existflag = 1;
        break;
    }
}
//已存有，直接用编号建立对应关系
if (nonterminal_existflag == 1)
{
    if (count == 1)
    {
        proc[line].left = i;
        proc[line].left_terflag = 0;
    }
    else
    {
        proc[line].right[count - 2] = i;
        proc[line].right_terflag[count - 2] = 0;
    }
}
//未存有，动态生成新的非终结符，并建立对应关系
```

4）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

```
FOR i := 1 TO n DO
BEGIN
    FOR j:= 1 TO i -1 DO
      把形如Pi→PjY的规则改写为：
         Pi→δ1Y|δ2Y|…|δkY
         其中： Pj →δ1 |δ2|…|δk 是关于Pj
              的所有规则;
    消除关于Pi 规则的直接左递归。
END
```

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：

FIRST（α）构造

对于符号串$\alpha= X_1X_2\cdots X_n$，构造 FIRST (α)

（1） 置 $FIRST(\alpha) = FIRST(X_1) - \{\varepsilon\}$;

（2） 若对 $1<=j<= i-1$，$\varepsilon \in FIRST(X_j)$, 则把 $FIRST(X_i) -\{\varepsilon\}$加到$FIRST(\alpha)$中；

（3） 若对$1<= j <=n$, $\varepsilon \in FIRST(X_j)$, 则把 ε加到$FIRST(\alpha)$中。

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:

```
//构造First集算法第（2）步
for (i = 0; i < proc_num; i++)
{
    //某个产生式右端第一个单词是终结符
    if (proc[i].right_terflag[0] == 1)
    {
        //判断First集中是否已存有该终结符，如果没有，加进去
        Add_etoB(proc[i].right[0], first_set[proc[i].left]);
    }
}
```

2) 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:

```
//把右端第一个非终结符的First集加到左端的非终结符的First集里
flag = flag || Add_AtoB(first_set[proc[i].right[0]], first_set[proc[i].left], true);
//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
for (j = 0; j < Production_maxlength - 1 && proc[i].right[j + 1] != 0 && proc[i].right_terflag[j] == 0; j++)
{
    if (Exist(first_set[proc[i].right[j]], 26))
    {
        if (proc[i].right_terflag[j + 1] == 1)
        {
            flag = flag || Add_etoB(proc[i].right[j + 1], first_set[proc[i].left]);
        }
        else
            flag = flag || Add_AtoB(first_set[proc[i].right[j + 1]], first_set[proc[i].left], true);
    }
    else
        break;
}
```

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:

```
//如果产生式右端所有非终结符的First集里含有元素"空"，把"空"加到左端的非终结符的First集里
if (j == Production_maxlength - 1 || proc[i].right[j + 1] == 0)
{
    if (Exist(first_set[proc[i].right[j]], 26))
    {
        flag = flag || Add_etoB(26, first_set[proc[i].left]);
    }
}
```

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：

FOLLOW(A)构造

对于文法G的每个非终结符，构造FOLLOW(A)的方法是：

（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；

（3）若有B→αA或B →αAβ，且β $\stackrel{*}{\Rightarrow}$ ε

则将FOLLOW(B)加到FOLLOW(A)中；

（4）反复使用以上规则，直至 FOLLOW(A)不再增大为止。

具体实现：

1)若 A 为文法开始符号，置#于 FOLLOW(A）中

```
//构造Follow集算法第（1）步
follow_set[1][0] = 27;//首先置FOLLOW表中开始符号的FOLLOW为#
```

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：

```
for (j = 0; j < Production_maxlength - 1 && proc[i].right[j + 1] != 0; j++)
{
    if (proc[i].right_terflag[j] == 0)
    {
        //该非终结符后面跟着终结符，把终结符加到非终结符的Follow集中
        if (proc[i].right_terflag[j + 1] == 1)
        {
            flag = flag || Add_etoB(proc[i].right[j + 1], follow_set[proc[i].right[j]]);
        }
        //该非终结符后面跟着非终结符，把后面的非终结符的First集加到非终结符的Follow集中
        else
        {
            flag = flag || Add_AtoB(first_set[proc[i].right[j + 1]], follow_set[proc[i].
        }
    }
}
```

3) 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

```
for (j = 0; j < Production_maxlength && proc[i].right[j] != 0; j++);
    j--;
if (proc[i].right_terflag[j] == 0)
{
    flag = flag || Add_AtoB(follow_set[proc[i].left], follow_set[proc[i].right[j]], false);
    for (; j >= 1; j--)
    {
        if (proc[i].right_terflag[j - 1] == 0 && Exist(first_set[proc[i].right[j]], 26))
            flag = flag || Add_AtoB(follow_set[proc[i].left], follow_set[proc[i].right[j - 1]], false);
        else
            break;
    }
}
```

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""：

```
QString  MTableview[Nonterminal_Sign_num + 1][Terminal_Sign_num + 1];
```

```
void Grammar::MTable()
{
    int i, j, k;
    //置空值
    for (i = 0; i <= Nonterminal_Sign_num; i++)
    {
        for (j = 0; j <= Terminal_Sign_num; j++)
        {
            table[i][j].pro.left = 0;
            for (k = 0; k < Production_maxlength; k++)
            {
                table[i][j].pro.right[k] = 0;
            }
            table[i][j].ProNo = -1;
        }
    }
```

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：

对于G中的每一个产生式， A -> α ,执行以下2步：

1.for ∀ a ∈ FIRST(α)， 将 A -> α 填入 M [A, a];

1. if （ε ∈ FIRST(α)）

∀ a ∈ FOLLOW (A)， 将 A -> ε 填入 M [A, a];

A) 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。

B) 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。

C) 遍历所有产生式得出完整分析表

```
for (i = 1; i <= proc_num; i++)
{
    //构造分析表算法第（2）步
    if (proc[i].right_terflag[0] == 1)
    {
        if (proc[i].right[0] == 26)
        {
            for (k = 0; k < tsign_num && follow_set[proc[i].left][k] != 0; k++)
                fill_proc(proc[i].left, follow_set[proc[i].left][k], i);
        }
        else
            fill_proc(proc[i].left, proc[i].right[0], i);
    }
    else
    {
        for (j = 0; j < tsign_num && first_set[proc[i].right[0]][j] != 0; j++)
        {
            if (first_set[proc[i].right[0]][j] == 26)
            {
                for (k = 0; k < tsign_num && follow_set[proc[i].left][k] != 0; k++)
                    fill_proc(proc[i].left, follow_set[proc[i].left][k], i);
            }
            else
                fill_proc(proc[i].left, first_set[proc[i].right[0]][j], i);
        }
    }

}
```

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：

```
private:
        const char* data;
        int child_num;
        STree* child[MAX_CHILD_NUM];
        //变量名
        char name[NAME_LENGTH];
        //各种Label
        char* True;
        char* False;
        char* next;
        char* begin;

        bool flag;      //访问标志，区分语法树的遍历是进去还是回来的时候
        Quadruples_List* QLparent;
        Quadruples_List* QLchild[MAX_CHILD_NUM];

public:
        int ProNo;
        STree();
        STree(const char* str);
        void setData(const char *str);
        void setName(const char *str);
        const char* getData();
        char* getName();
        int getchild_num();
        STree* getChild(int position);
        void Add_Child(STree T,int position);
        friend class Semantic;
        friend class MainWindow;
        friend bool Analyze(STree* &root,int *rowline,Grammar *temp,QString path);
        void QuadOut(STree* T);
};
```

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：

```
//如果节点是<ID>（标识符）或者num（数值），记录下它的值
if (strcmp(node->getData(), G.terminal_sign[7]) == 0 || strcmp(node->getData(), G.terminal_sign[8]) == 0)
    node->setName(next_word.value);
```

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：

```
for (i = ChildNo; i >= 0; i--)//A->空这个产生式由于其childNo为-1所以此循环会直接跳过
{
    e.code = G.table[left.code][next_word.code].pro.right[i];
    e.ter_flag= G.table[left.code][next_word.code].pro.right_terflag[i];
    if (!(e.code == 26 && e.ter_flag == 1))
        S.push(e);

    if (G.table[left.code][next_word.code].pro.right_terflag[i] == 1)
        temp.setData(G.terminal_sign[G.table[left.code][next_word.code].pro.right[i]]);
    else
        temp.setData(G.nonterminal_sign[G.table[left.code][next_word.code].pro.right[i]]);

    node->Add_Child(temp, i);
    if (strcmp(temp.getData() ,"NONE")!=0)
        ST.push(node->getChild(i));
}
```

3.7 预测分析算法步骤

算法步骤：

（1）若x＝a＝“#”，即STACK 栈顶符号为“#”，当前输入符号为“#”，则分析成功。

（2）若x＝a≠“#”，即栈顶符号x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。

具体实现：

1)首先将#和起始符号压入分析栈：

```
e.code = 27;
e.ter_flag = 1;
S.push(e);//#入栈
e.code = 1;
e.ter_flag = 0;
S.push(e);//起始符号入栈
```

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。

```
//栈顶是终结符
if (S.top().ter_flag == 1)
{
    if (next_word.code == S.top().code)
    {
        S.pop();
        if (S.empty())
            break;
        node = ST.top();
        ST.pop();
        //如果节点是<ID>（标识符）或者num（数值），记录下它的值
        if (strcmp(node->getData(), G.terminal_sign[7]) == 0 || strcmp(node->getData(
            node->setName(next_word.value);

        next_word = L.Result(fp);
    }
    else
        return false;

}
```

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:

```
//判断M[A,a]是否为一产生式
if (G.table[S.top().code][next_word.code].pro.left != 0)
{
    left = S.top();
    node = ST.top();
    node->ProNo = G.table[S.top().code][next_word.code].ProNo;
    S.pop();
    ST.pop();
    for (ChildNo = 0; ChildNo < Production_maxlength && G.table[left.code][next_word.code].pro.right[Chi
       ChildNo--;
    for (i = ChildNo; i >= 0; i--)//A->空这个产生式由于其childNo为-1所以此循环会直接跳过
    {
        e.code = G.table[left.code][next_word.code].pro.right[i];
        e.ter_flag= G.table[left.code][next_word.code].pro.right_terflag[i];
        if (!(e.code == 26 && e.ter_flag == 1))
            S.push(e);

        if (G.table[left.code][next_word.code].pro.right_terflag[i] == 1)
            temp.setData(G.terminal_sign[G.table[left.code][next_word.code].pro.right[i]]);
        else
            temp.setData(G.nonterminal_sign[G.table[left.code][next_word.code].pro.right[i]]);

        node->Add_Child(temp, i);
        if (strcmp(temp.getData() ,"NONE")!=0)
            ST.push(node->getChild(i));
    }
```

```
else
    return false;//M[A,a]为空，发现语法错误
```

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:

```
QTreeWidget *STT=new QTreeWidget(ui->SyntaxAnalysisWin);
QList<QTreeWidgetItem *> rootList;
stack <STree*> ST;
stack<QTreeWidgetItem *>QItem;
STree *current;
ST.push(STREE);
QTreeWidgetItem *imageItem1 = new QTreeWidgetItem;    //添加第一个父节点
imageItem1->setText(0,QString(STREE->getData()) );
QItem.push(imageItem1);
rootList.append(imageItem1);
while (!ST.empty())
{
    current = ST.top();
    ST.pop();
    QTreeWidgetItem * tempItem=QItem.top();
    QItem.pop();
    for (int j=0; j < current->getchild_num(); j++)
    {
        ST.push(current->getChild(j));
        QTreeWidgetItem *newItem = new QTreeWidgetItem;    //添加第一个父节点
        newItem->setText(0,tr(current->getChild(j)->getData()));
        tempItem->addChild(newItem);
        QItem.push(newItem);
    }
}
QScrollArea *scrollArea = new QScrollArea;
```

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，

```
int row_first=0;
for(vector<pair<char*,QString>>::const_iterator iter = RESULT_FIRST.cbegin(); iter != RESULT_FIRST.cend(); iter++,row_first++)
{
    QTableWidgetItem *item0, *item1;
    item0 = new QTableWidgetItem;
    item1 = new QTableWidgetItem;
    QString txt = QString((*iter).first);
    item0->setText(txt);
    tableView->setItem(row_first, 0, item0);
    txt = QString((*iter).second);
    item1->setText(txt);
    tableView->setItem(row_first, 1, item1);
}
```

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/

Name	Last modified	Size	Metadata
Parent Directory		-	
submodules/	20-Jan-2017 13:19	-	
single/	20-Jan-2017 13:14	-	
qt-opensource-windows-x86-winrt-msvc2015-5.8.0.exe	20-Jan-2017 12:54	1.2G	Details
qt-opensource-windows-x86-winrt-msvc2013-5.8.0.exe	20-Jan-2017 12:53	1.2G	Details
qt-opensource-windows-x86-msvc2015_64-5.8.0.exe	20-Jan-2017 12:52	1.0G	Details
qt-opensource-windows-x86-msvc2015-5.8.0.exe	20-Jan-2017 12:59	1.0G	Details
qt-opensource-windows-x86-msvc2013_64-5.8.0.exe	20-Jan-2017 12:51	958M	Details
qt-opensource-windows-x86-msvc2013-5.8.0.exe	20-Jan-2017 12:50	947M	Details
qt-opensource-windows-x86-mingw530-5.8.0.exe	20-Jan-2017 12:49	1.2G	Details
qt-opensource-windows-x86-android-5.8.0.exe	20-Jan-2017 12:48	1.3G	Details
qt-opensource-mac-x64-clang-5.8.0.dmg	20-Jan-2017 12:45	1.3G	Details
qt-opensource-mac-x64-android-ios-5.8.0.dmg	20-Jan-2017 12:44	3.4G	Details
qt-opensource-mac-x64-android-5.8.0.dmg	20-Jan-2017 12:40	1.4G	Details
qt-opensource-linux-x64-android-5.8.0.run	20-Jan-2017 12:34	817M	Details
qt-opensource-linux-x64-5.8.0.run	20-Jan-2017 12:34	766M	Details
md5sums.txt	22-Nov-2017 13:16	1.1K	Details
android-patches-5.8-2017_11_16.tar.gz	22-Nov-2017 10:40	4.8K	Details

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