同济大学

TONGJI UNIVERSITY

词法分析器和语法分析器

实验报告

实验名称	词法分析器和语法分析器
实验成员	王耀辉（2052140） 刘峥（2053190） 王永庆（2052136）
日　　期	2022 年　11 月　8 日

目录

一、需求分析

1.1 程序任务输入及其范围

程序任务输入：

词法分析输入文件：code_in.txt，内部存储了要进行词法、语法分析的 c++源程序代码。

语法分析输入文件：grammar.txt，包括注释和正文两部分，注释部分介绍了文法规则的定义形式，正文部分是文法规则的具体定义，包括定义终结符和定义文法产生式两部分。

输入要求：

程序可以正确编译的文件是符合 C 语言语法的源程序文件，程序可以识别的文法文件是按照文法文件注释所规定格式的文法文件。

1.2 输出形式

词法分析输出文件：

1. Pre-Processed_Code.txt，存储经过预处理的源程序文件，文件中不包含程序注释和语句之间的多余空格。
2. Word_Lable.txt，存储词法分析程序生成的单词种别表，存储形式为：[单词种类]----[种别编码]----[单词符号]。
3. Lexical_Result.txt，存储单词分割判定后的结果，存储形式为：[单词种类]----[种别编码]----[单词符号]。
4. Lex_to_Parse.txt，记录词法分析程序送入语法分析的数据。

语法分析输出文件：

1. Tables.csv，记录 ACTION 表和 AOTO 表。
2. Zero_Closure.txt，记录 0 号初始闭包。
3. Analysis_Process.txt，输出对读入的 C 程序进行语法分析的规约过程，输出形式包括：规约步骤序号、状态栈和符号栈。
4. Parse_Tree.png，语法分析树图片，生成需安装 graphviz。

1.3 程序功能

读入一段源程序代码和一段文法定义文本，根据程序生成的自动机对源程序进行单词分割，对单个单词进行单词种别的识别，利用文法定义自动生成 ACTION 表和 GOTO 表，对经过词法分析后的源程序文件进行语法分析，判断源程序代码是否符合该文法，并输出判断结果和规约过程。

1.4 测试数据（正确，错误数据）

正确数据示例：（符合 C 语言规范的源程序）

```
int program(int a, int b, int c)
{
int i;
int j;
i = 0;
if (a > (b + c))
{
    j = a + (b * c + 1);
}
else
{
    j = a;
}
i = j * 2;
while (i <= 100)
{
    i = i * 2;
}
return i;
}

int demo(int a)
{
a = a + 2;
return a * 2;
}
//aaaaaaaaaaaaaaaa
/*aaaaa
qqq
*/  void main()
{
int a;
int b;
int c;
a = 3;
b = 4;
c = 2;
a = program(a, b, demo(c));
return;
```

```
}
```

错误数据示例：（非 C 语言程序，这里以 python 程序为例）

```
import numpy as np
from matplotlib import pyplot as plt

plt.figure(figsize=(9, 6))
n = 7
X=np.arange(n)
data= ['5', '10', '20', '40', '60','80','100']
width = 0.36
Y1 = [0.52, 0.639, 0.696, 0.831, 0.839,0.821,0.843]

plt.bar(data, Y1, width=width, facecolor='lightskyblue', edgecolor='white')
# width:柱的宽度

# 水平柱状图 plt.barh，属性中宽度 width 变成了高度 height
# 打两组数据时用+
# facecolor 柱状图里填充的颜色
# edgecolor 是边框的颜色
# 给图加 text
for x, y in zip(X, Y1):
    plt.text(x, y + 0.05, '%.2f' % y, ha='center', va='bottom')

plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.ylim(0,1)
plt.title("Accuracy-Iteration using mini-batch Gradient Descent")

plt.ylim(0, +1.25)
plt.show()
```

1.5 项目亮点

1. 在词法分析部分，程序共设置了 69 种词法类别，其中包括 28 种关键字和 41 种运算符，单词种类较为完备，能够满足大部分 C 语言的词法分析要求。
2. 在词法分析部分，程序能够准确的将普通浮点数和科学计数法浮点数进行单词划分并分类，程序也可以对不同进制的实整数进行单词分割和分类。
3. 在词法分析阶段，根据数据库相关专业知识，创建两个缓冲池用于源程序的词法分析，通过缓冲池的轮转调度，达到节约空间，清晰思路的目的。
4. 在语法分析部分，我们为项目闭包类设置类函数，求得可移进的字符及项目在闭

包中的位置和可归约的项目及对应产生式的序号，通过这两个类函数可以更加便捷的求得文法的 ACTION 和 GOTO 表。

5. 最终语法分析树以图片的形式直观可视化，LR(1)归约过程输出文档，能够更加清楚的表示语法分析的过程以及分析结果。

二、概要设计

2.1 任务的分解

本次实验任务主要分为词法分析部分和语法分析部分，词法分析部分包括

a. 对读入的源程序代码进行预处理；
b. 利用自动机将预处理后的代码分割为单词；
c. 建立单词符号及内部表示表；
d. 对单个单词进行类型判定；
e. 输出对源文件进行词法分析后的结果文件。

语法分析部分包括：

a. 读入文法定义文件，生成 LR(1)项目；
b. 计算所有符号的 FIRST 集；
c. 根据 FIRST 集，计算所有的 LR(1)项目集族，即闭包并建立状态转移 DFA；
d. 根据闭包建立 ACTION 表和 GOTO 表；
e. 根据 ACTION 表和 GOTO 表对词法分析结果进行 LR(1)语法分析。

2.2 数据类型的定义

- **unit 类**

功能：

存储词法分析后的单个单词的信息。

类成员变量：

string type 单词种别。

string value 单词符号。

类成员函数：

unit(string tp, string v) 初始化 unit 实例。

代码实现：

```
class unit
{
public:
    string type;
    string value;
    unit(string tp, string v);
};
```

- **Buffer 类**

功能：

读入输入文件时的缓冲区。

类成员变量：

buffer，字符数组

count，记录读入字符个数

类成员函数：

Buffer()，构造函数。

~Buffer()，析构函数。

代码实现：

```
class Buffer {
public:
    char* buffer;
    int count;

    Buffer() {
        count = 0;
        buffer = new char[BUFFER_SIZE];
    }
    ~Buffer() {
        delete buffer;
    }
};
```

- **LR1_item 类**

功能：

存储语法分析时的 LLR(1)项目。

类成员变量：

int left，左侧符号序号编号；

vector<int> right，右侧符号序号编号；

int dot_site，中心点的位置；

int forward，向前看的符号编号 ；

int grammar_index ，项目生成式的编号。

类成员函数：

void print()，输出函数；

LR1_item()，构造函数；

LR1_item(int l, vector<int>& r, int ds, int fw, int gi)，构造函数；

bool operator==(const LR1_item& item)，重载==，判断两个 LLR(1)项目是否相同；

void LR1_itemInit(int l, vector<int>& r, int ds, int fw, int gi) ，初始化 LR1_item 实例。

代码实现：

```
class LR1_item
{
public:
 int left;//左侧符号序号编号
 vector<int> right;//右侧符号序号编号
 int dot_site;//中心点的位置
 int forward;//向前看的符号编号
 int grammar_index;//这个LR1项是哪个产生式出来的,其实是有冗余，有这个index就已经有了left
 和right
public:
 void print();
 LR1_item() { left = 0; dot_site = 0; forward = 0; grammar_index = 0;
right.push_back(0); };
 LR1_item(int l, vector<int>& r, int ds, int fw, int gi);
 bool operator==(const LR1_item& item);
 void LR1_itemInit(int l, vector<int>& r, int ds, int fw, int gi);
};
```

- **LR1_closure 类**

功能：

存储语法分析时的项目闭包。

类成员变量：

vector<LR1_item> key_item，该闭包的关键项目；

vector<LR1_item> closure，存储项目闭包的动态数组。

类成员函数：

bool isIn(LR1_item item)，判断项目是否在该闭包中；

bool operator==(LR1_closure& clos)，重载==，判断两个闭包是否相同；

map<int, vector<int>> getShiftinSymbol()，得到可移进的字符以及项目在闭包中的位置；

vector<pair<int, int>> getReduceSymbol()，得到可以归约的符号和对应的产生式的序号；

void print(const vector<symbol>symbols)，输出函数。

代码实现：

```
class LR1_closure
{
public:
vector<LR1_item> key_item;//该闭包的关键项目，有没有用不知道
vector<LR1_item> closure;//项目闭包
public:
//TODO:这个需要考虑要不要保留
bool isIn(LR1_item item);//返回该项目是否在该闭包中
bool operator==(LR1_closure& clos);
map<int, vector<int>> getShiftinSymbol();//得到可移进的字符以及项目在闭包中的位置
vector<pair<int, int>> getReduceSymbol();//得到可以归约的符号和对应的产生式的序号
void print(const vector<symbol>symbols);
};
```

- **LR1_Grammar 类**

功能：

存储语法分析器所需要的数据和函数。

类成员变量：

vector<LR1_item> item_sum，存储所有的项目；

vector<LR1_closure> closure_sum，存储所有可能出现的闭包；

map<pair<int, int>, int> DFA，前面的 pair 是<closure 的编号，符号的编号>，对应的是能连接的目标 closure 编号；

map<pair<int, int>, ACTION_item> ACTION，存储 ACTION 表；

map<pair<int, int>, GOTO_item> GOTO，存储 GOTO 表；

LR1_item start_item，初始项目；

LR1_closure start_closure，初始项目闭包。

类成员函数：

int checkClosure()，初始化 start_item 和 start_closure；

LR1_closure computeClosure(vector<LR1_item>)，给定项目计算闭包；

int getClosureIndex(LR1_closure& clos)，判断闭包集合中是否有该闭包，若有返回序号，若没有返回-1；

void getClosureSum()，得到所有闭包，初始闭包是 0 号闭包，在处理过程中同时确定 DFA；

void computeACTION_GOTO()，计算 ACTION 表和 GOTO 表；

void printTables()，打印 ACTION 和 GOTO 表；

void analyze(vector<unit>& lexical_res)，进行归约，并打印中间过程。

代码实现：

```
class LR1_Grammar :public grammar
{
public:
 vector<LR1_item> item_sum;//存所有的项目，set没有编号
 vector<LR1_closure> closure_sum;//所有可能出现的闭包，相当于编个号
 map<pair<int, int>, int> DFA;//前面的pair是<closure的编号，符号的编号>，对应的是能连接的目标closure编号
 //相当于就是表示连接关系
 //ACTION表和DFA有区别，在于归约项，如何当该归约时表示归约产生式序号
 map<pair<int, int>, ACTION_item> ACTION;//ACTION表
 //GOTO表就是非终结符与状态之间，只有状态转移或空
 map<pair<int, int>, GOTO_item> GOTO;

 LR1_item start_item; //初始项目
 LR1_closure start_closure; //初始项目闭包
 LR1_Grammar(){};
 LR1_Grammar(const string file_path);

public:
 //初始化start_item和start_closure
 int checkClosure(); //从grammar继承的rules，从开始产生式开始，使得项目集中第一个是闭包
 LR1_closure computeClosure(vector<LR1_item>);//给定项目计算闭包
 //判断闭包集合中是否有该闭包，若有返回序号，若没有返回-1
 int getClosureIndex(LR1_closure& clos);
 //得到所有闭包，初始闭包是0号闭包，在过程中同时确定DFA
 void getClosureSum();
 //计算ACTION表和GOTO表
 void computeACTION_GOTO();
 //打印ACTION和GOTO表
 void printTables();
 //进行归约，在过程中进行打印
```

```
    void analyze(vector<unit>& lexical_res);
};
```

- **symbol 类**

功能：

存储单个符号的相关属性。

类成员变量：

symbol_class type，文法符号种类；

set<int> first_set，记录该 symbol 的 first 符号对应的 symbol 表下标；

set<int>follow_set，记录该 symbol 的 follow 符号对应的 symbol 表下标；

uniqstr tag，符号名。

类成员函数：

symbol(symbol_class type, const string tag)，构造函数。

代码实现：

```
class symbol {
public:
    symbol_class type;//文法符号种类
    set<int> first_set;//记录该symbol的first符号对应的symbol表下标
    set<int>follow_set;//记录该symbol的follow符号对应的symbol表下标
    uniqstr tag;//符号名
    symbol(symbol_class type, const string tag);
};
```

- **rule 类**

功能：

存储单条文法产生式。

类成员变量：

int left_symbol，存储产生式左部非终结符的标号；

vector<int> right_symbol，存储产生式右部的符号标号序列。

类成员函数：

rule(const int left, const vector<int>& right)，构造函数。

代码实现：

```
class rule {
public:
    int left_symbol;
    vector<int> right_symbol;
```

```
    rule(const int left, const vector<int>& right);
};
```

- **grammar 类**

功能：

语法分析类。

类成员变量：

vector<symbol>symbols，存储符号表；

set<int>terminals，存储终结符在 symbol 中的下标；

set<int>non_terminals，存储非终结符在 symbol 中的下标；

vector<rule>rules，存储所有的文法产生式；

int start_location，记录起始产生式在 rules 中的位置。

bool haveStartToken，是否定义起始符

bool haveAllTerminalToken，是否出现%token 标识符，%token 后是文法中所有的终结符号

bool haveExtendStartToken，是否定义拓展文法起始符

bool haveEndToken，是否定义 end 终结符

类成员函数：

grammar(){}，构造函数

grammar(const string file_path)，构造函数

void ReadGrammar(const string file_path)，从文件中读入文法

void print()，输出函数

int Find_Symbol_Index_By_Token(const string token)，

void InitFirst()，初始化 first 集合

void InitFirstTerm()，初始化终结符的 FIRST 集合

void InitFirstNonTerm()，初始化非终结符的 FIRST 集合

void PrintFirst()，输出函数

void ProcessFirst()，处理非终结符中 first 集合包含非终结符的情况

set<int>GetFirst(const vector<int>& str)，返回一个符号串的 first 集合

void initGrammar()，初始化文法

void checkGrammar()，判断输入的文法是否符合要求

代码实现：

```
class grammar {
public:
    vector<symbol>symbols;//所有的符号表

    set<int>terminals;//终结符在symbol中的下标
    set<int>non_terminals;//非终结符在symbol中的下标
    vector<rule>rules;//所有的文法
    int start_location;//起始产生式在rules中的位置
    grammar(){};
    grammar(const string file_path);
    //从file中读入grammar
    void ReadGrammar(const string file_path);
    void print();

    int Find_Symbol_Index_By_Token(const string token);

    //初始化first集合
    void InitFirst();
    void InitFirstTerm();
    void InitFirstNonTerm();
    void PrintFirst();
    void ProcessFirst();
    //返回一个符号串的first集合
    set<int>GetFirst(const vector<int>& str);
    //
public:
    bool haveStartToken;
    bool haveAllTerminalToken;
    bool haveExtendStartToken;
    bool haveEndToken;
    void initGrammar();
    void checkGrammar();

};
```

- **base 类**

功能：

完成基本的类型判断。

类成员变量：

无。

类成员函数：

int charKind(char c)，判断输入字符类型 是 数字 字母 还是 其他符号 状态机使

用；

int wordWrongAnalysis(char str[], int type)，错误判断；

int isDelimiter(char c)，界符；

int isDelimiter(char* c)，界符；

bool spaceCanDelete(char c)，判断空格能否删除。

…

代码实现：

```
class base {
public:
    virtual ~base();
public:
    int charKind(char c);//判断输入字符类型 是 数字 字母 还是 其他符号 状态机使用
    int wordWrongAnalysis(char str[], int type);//错误判断
    int isDelimiter(char c);//界符
    int isDelimiter(char* c);//界符
    bool spaceCanDelete(char c);//判断空格能否删除
protected:
    int isSeparator(char c);//分隔符
    int isBracketsLeft(char c);//左括号
    int isBracketsRight(char c);//右括号
    int isBracketsLeftBig(char c);//左大括号
    int isBracketsRightBig(char c);//右大括号

    int isPoint(char c);//.
    int isBracketsLeftSquare(char c);//[
    int isBracketsRightSquare(char c);//]
    int isPointArrow(char str[]);//->
    int isRegion(char str[]);//::
    int isRegionXigou(char str[]);//::~
    int isColon(char c);//:

    int isEnd(char c);//结束符
    int isStr(char str[]);//字符串
    int isChar(char str[]);//字串是字符

    int isInt(char str[]);//整型
    int isFloat(char str[]);//float 型 +-xx.xx e +-xx.xx
    int isFloatTool(char str[]);//float型 +-xx.xx

    int isSignWord(char str[]);// 标识符
    int isKeyWord(char str[]);//保留字  关键字

    int isNum(char c);//是不是数字
    int isLetter(char c);//大小写字母

    int isSpecialSign(char c);//看标识符命名是否正确
```

```
    int isBinocularOperator(char str[]);//判断双目运算符
    int isMonocularOperator(char str[]);//判断单目运算符

    int isBlank(char str[]);//判断空格
};
```

- **定义枚举数据类型：**

  定义文法符号的种类

```
typedef enum
{
    unknown_sym,           /* Undefined.  */
    token_sym,          /* Terminal. */
    nterm_sym,          /* Non-terminal */
    epsilon, /* null */
    end /* end terminal*/
}symbol_class;
```

- **定义 ACTION 表中的枚举常量**

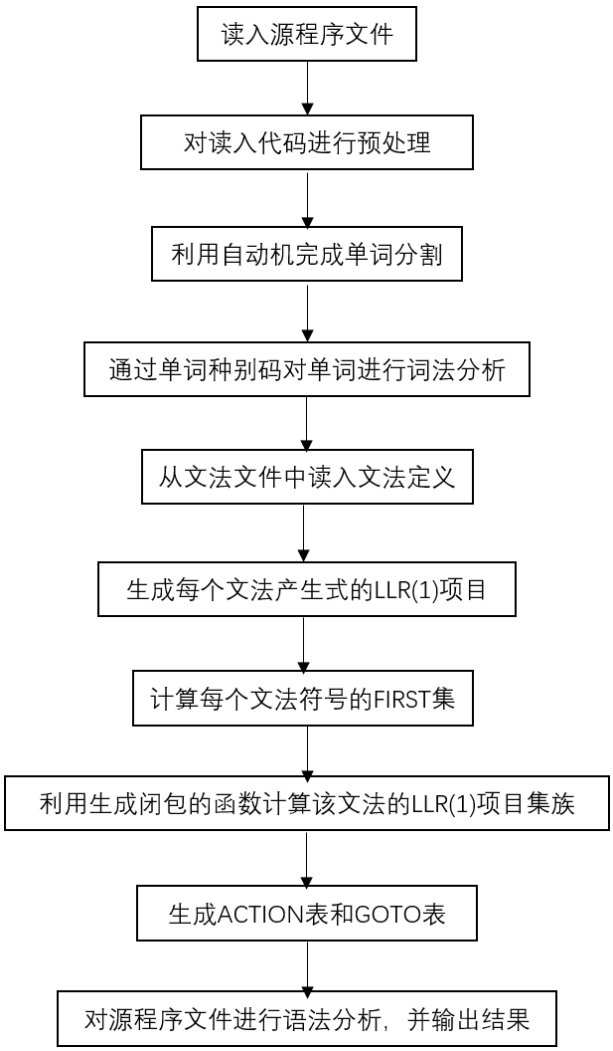
```
//ACTION表中可以存在的动作
enum ACTION_Option
{
SHIFT_IN,//移进
REDUCE,//归约
ACCEPT,//接受
REJECT//拒绝
};
```

- **定义 GOTO 表中的枚举常量**

```
//GOTO表中可以存在的动作
enum GOTO_Option
{
GO,
REJECT_
};
```

装 订 线

2.3 主程序流程



2.4 模块间的调用关系

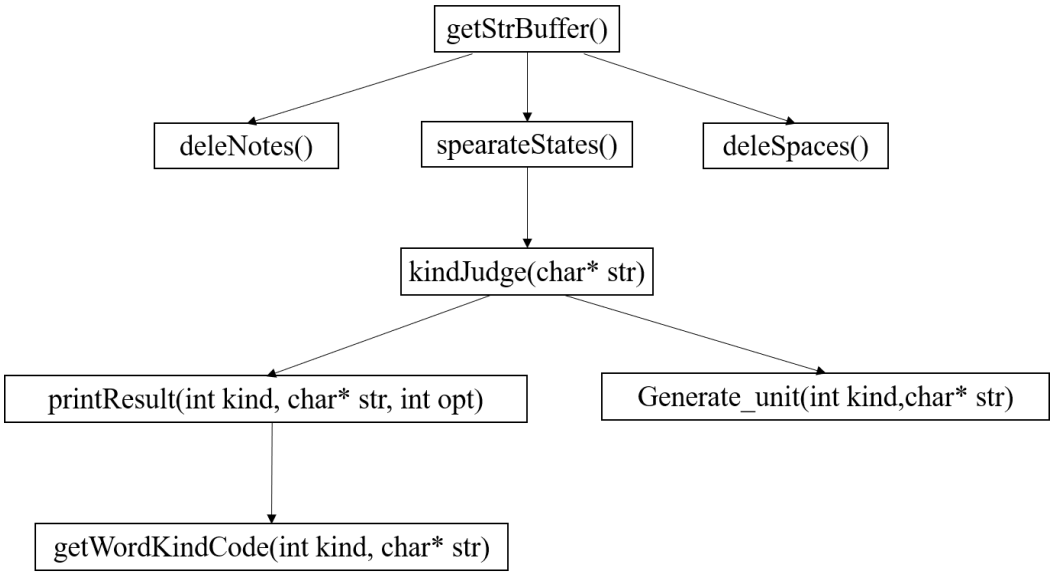
- **analysis 类中各功能模块的调用关系**

功能模块列表：

```
void getStrBuffer()
void deleNotes();
void deleSpaces();
void spearateStates();
void kindJudge(char* str);
void printResult(int kind, char* str, int opt);
int getWordKindCode(int kind, char* str);
```

unit Generate_unit(int kind,char* str);

模块调用关系图(图中箭头由调用模块指向被调用模块，下同)



- **LR1_Grammar 类中各功能模块的调用关系**

功能模块列表：

int checkClosure();

LR1_closure computeClosure(vector<LR1_item>);

int getClosureIndex(LR1_closure& clos);

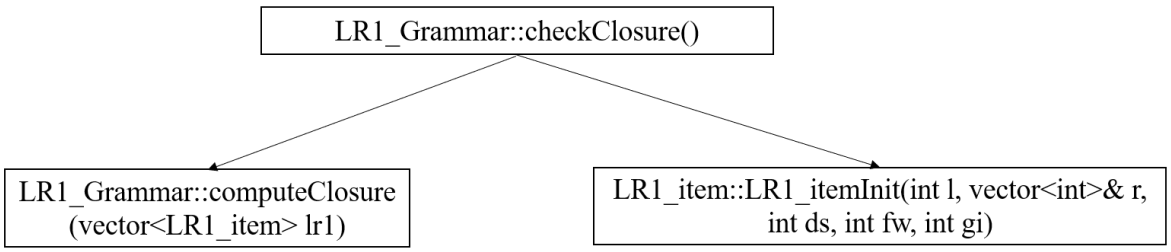
void getClosureSum();

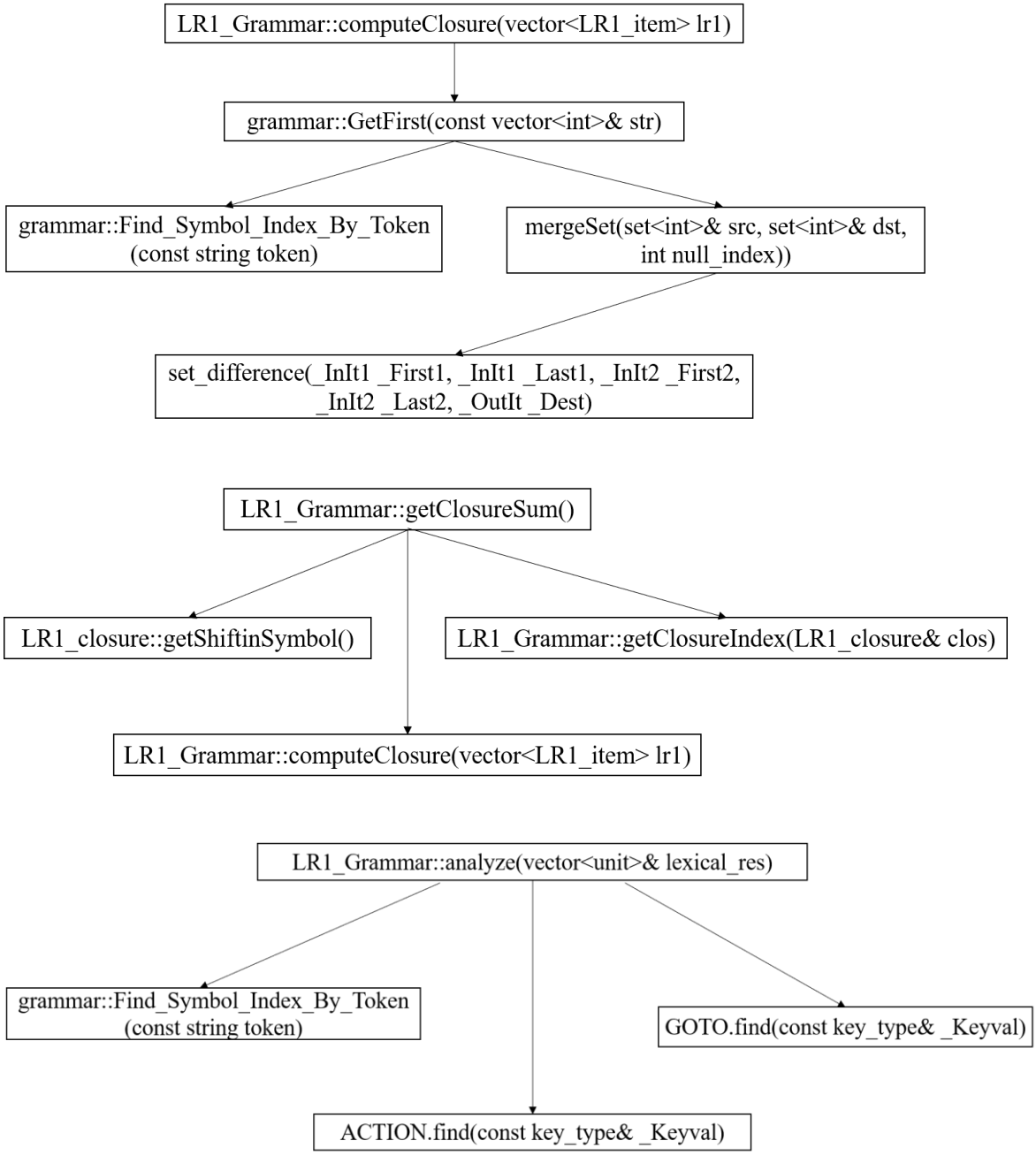
void computeACTION_GOTO();

void printTables();

void analyze(vector<unit>& lexical_res);

模块调用关系图




LR1_Grammar::computeClosure(vector<LR1_item> lr1)
grammar::GetFirst(const vector<int>& str)
grammar::Find_Symbol_Index_By_Token (const string token)
mergeSet(set<int>& src, set<int>& dst, int null_index))
set_difference(_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _InIt2 _Last2, _OutIt _Dest)
LR1_Grammar::getClosureSum()
LR1_closure::getShiftinSymbol()
LR1_Grammar::getClosureIndex(LR1_closure& clos)
LR1_Grammar::computeClosure(vector<LR1_item> lr1)
LR1_Grammar::analyze(vector<unit>& lexical_res)
grammar::Find_Symbol_Index_By_Token (const string token)
GOTO.find(const key_type& _Keyval)
ACTION.find(const key_type& _Keyval)

装 订 线

三、详细设计

本部分着重讲解程序实现过程中的若干重要函数。

- `void analysis::getStrBuffer()`

重点变量：

int buffer_flag 标识当前读入缓冲区是否进行了轮转

int buffer_choose 表示当前选择的读入缓冲区的标号

Buffer buffer_read[2]创建两个读入缓冲区

Buffer buffer_end 存储已经读入的要送入状态机进行单词分割的字符串

vector<unit> analysis_res 存储词法分析得到的，要送入语法分析器的中间结果

重点功能：

将源程序读入缓冲区，然后对缓冲区中的内容依次进行清理注释、清理空格、分割单词、判断类型和结果输出。其中清除注释由 deleNotes()实现，清理空格由 deleSpaces()实现，判断类型由 spearateStates()和 kindJudge()实现，输出结果由 printResult 函数实现。

getStrBuffer 的具体实现过程为：程序从输入文件中读到非结束符之外的字符，就将该字符加入到缓冲区中。如果当前读入的缓冲区满，进行缓冲区的轮换，然后调用 deleNotes()和 deleSpaces()清除轮换后缓冲区中的注释和多余空格；如果函数读到换行符，直接调用 deleNotes()和 deleSpaces()清除当前缓冲区中的注释和多余空格；其他情况下，函数继续从文件中读取下一个字符。当清除过缓冲区中的注释和空格后，将读入缓冲区 buffer_read[buffer_choose]的内容保存到结果缓冲区 buffer_end 中，然后调用 spearateStates()和 kindJudge()实现单词分割和种类判断，最后，根据 buffer_flag 的值决定之后将读入的内容存到哪一个缓冲区，并将 buffer_flag 恢复零值。

代码实现：

代码实现详见 附录 A1。

- `void analysis::spearateStates()`

重点变量：

char word[BUFFER_SIZE]保存从 buffer_end 分割出来的单词；

int count = 0 记录当前 word 中的字符个数；

bool finish = false 记录是否读完了一个单词；

int state 记录自动机当前所处的状态。

重点功能：

在词法分析器中，自动机的作用主要是将经过预处理后的扫描缓冲区内的语句分割成若干单词，划分的依据是：通过逐个读入语句字符，让自动机的状态进行切换直到到达一个结束状态，此时提取出该字符串作为一个单词，之后自动机回到初态继续读入字符进行状态转移。当读到扫描缓冲区提供的语句的结尾时该过程结束。

在自动机的状态方面，从初态开始，依据读入的第一个字符类型，共划分了 9 种状态对应 9 条路径，分别是：字母，数字，$和_，\,”,’，要结束的字符，空格，其他字符。进入到第一个状态后，不同的路径对不同的输入字符进行不同的状态转换。当该单词需要结束时进入一个统一的结束状态，在该状态，自动机将单词传给类别识别函数，并将状态重新转换为初始零状态，等待下一个字符的读入。在过程中需要注意字符串内的转义字符，数字的多种类型表示，浮点数，标识符

代码实现：

代码实现详见 附录 A2。

- `set<int>grammar::GetFirst(const vector<int>& str)`

重点变量：

`set<int>first_set`：记录传入符号串的 first 集合

`bool is_epsilon`：判断符号串的每个符号能否推导出空串

`int empty_location`：记录空串在 symbol 表的位置

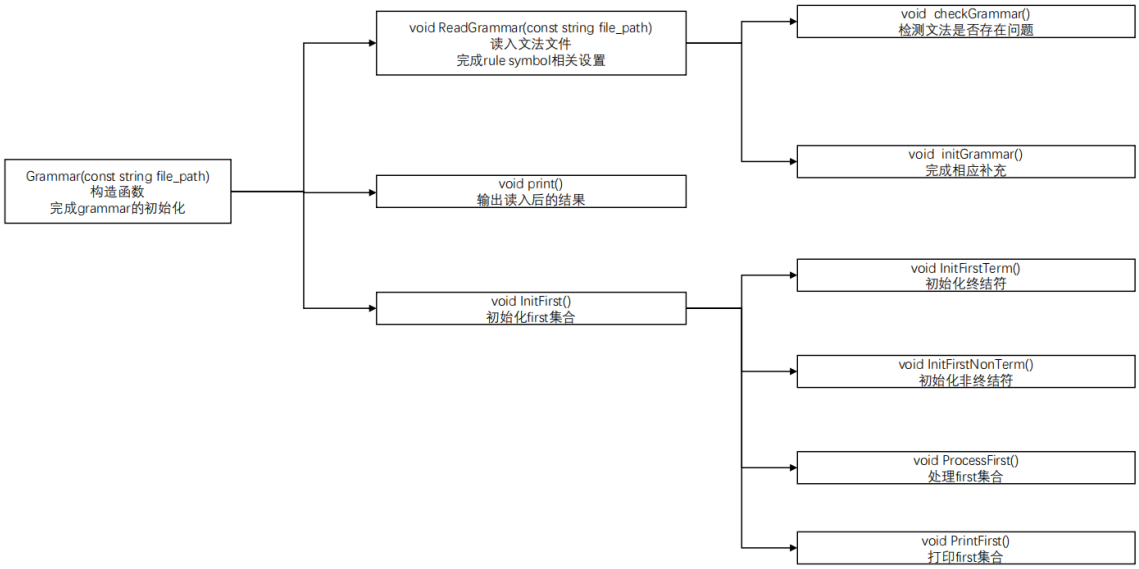
重点功能：

输入一串符号串，返回该符号串的 first 集合。

如果当前符号串的 first 集合是空串，要将空串加入符号串的 first 集合；如果符号串不是空串，那么需要对符号串进行遍历：

1. 当前符号是非终结符，要判断当前非终结符能否推导出空串，完成空串是否加入符号串 first 集合的判断。
2. 完成非终结符 first 集合的合并
3. 当前符号是终结符，意味着遍历结束，可以返回 first 集合

First 集合的函数调用图：



代码实现：

代码实现详见 附录 A3。

- `LR1_closure LR1_Grammar::computeClosure(vector<LR1_item> lr1)`

重点变量：

vector<LR1_item> lr1：关键项目，通过关键项目计算整个项目的完整闭包；

LR1_closure closure_now ：存储最后闭包运算结果的变量，里面有关键项目 key_item 和闭包 closure 两个存储变量；

LR1_item item_now：得到当前闭包的每个 rule 文法，完成当前 rule 的闭包计算；

vector<int>BetaA：存储字符串每个符号对应的下标，作为符号串传入 GetFirst 函数中计算其 first 集合；

bool have_exist：用于判断当前闭包中是否存在此项，如果有，那么不加入闭包中，避免重复；

bool is_epsilon ：用于判断当前rule文法是否为空串，若为空串，需要完成 A->* epsilon=A->epsilon *，即·dot下标的移动，利用此变量将空串和非空串联系在一个循环中完成，提高程序运行效率。

重点功能：

通过关键项目计算整个项目的完整闭包。传入关键项目完成项目族的闭包推导。

对项目族进行遍历，判断当前项目是否可以推导出后继项目并加入项目族中，需要考虑的情况：

1. •的位置
   a) 如果•在项目的最后一个位置，那么意味着这条项目没有拓展价值
   b) 如果•在项目的中间位置，需要判断•后的符号，若为终结符，那么后续无操作，若为非终结符，需要将•后从第二个开始的所有字符合并，求 first 集合，将其一个个加入项目集族中
   c) 如果•后为空串，直接处理为可归约状态，将•移向空串末位
2. 闭包项目族的数目不再增加，意味着闭包运算结束

代码实现：

代码实现详见 附录 A4。

- `void LR1_Grammar::computeACTION_GOTO()`

函数功能：

计算当前文法的 ACTION 和 GOTO 表。

重点变量：

ACTION，GOTO。

重点功能：

在计算完成 DFA 状态集和状态转移关系后，调用该函数，构造当前文法的 ACTION 表和 GOTO 表。

首先根据 DFA 的转移关系得到移进动作，通过遍历 DFA 的状态集，将转移符号是终结符的转移关系以当前状态序号、转移终结符序号和达到状态序号的形式记录在 ACTION 表中，并将该项标记为 SHIFT_IN 移进动作，若转移符号是非终结符，则以相同的形式记录在 GOTO 表中。

之后根据闭包集的归约项得到 ACTION 表中的归约和接受动作。遍历闭包集合，使用 `LR1_closure::getReduceSymbol()`函数求出每个闭包的归约项，对每一个归约项，以当前闭包序号、归约的字符序号、使用的归约产生式序号的形式将其记录在

ACTION 表中，若该项目是初始归约项目，则将该项标记为 ACCEPT 接受动作，若不是，则将该项标记为 REDUCE 归约动作。

代码实现：

代码实现详见 附录 A5。

- `int LR1_Grammar::analyze(vector<unit>& lexical_res)`

函数功能：

使用当前文法对测试程序词法分析结果进行归约分析并将打印归约过程，返回状态码以说明是否成功归约。

重点变量：

1. `vector<int> status_stack`：归约过程中的状态栈，保存状态序列。
2. `vector<int> symbol_stack`：归约过程中的符号栈，保存符号序列。
3. `vector<unit>& lexical_res`：词法分析结果，以 vector 向量形式输入给语法分析部分，其中保存每一个单词的原始值与分类值。

重点功能：

首先初始化符号栈、状态栈和输入字符串，向符号栈中压入结束符号序号，向状态栈中压入 0 号初始状态，向输入串的末尾添加结束符号。

之后开始归约过程，每次从输入串中读入一个终结符并放入符号栈中，以当前符号栈的顶部状态作为当前状态，以该终结符作为转移字符，在 ACTION 表中查找对应的转移动作。若没有对应动作，则报错。若查找到的动作是移进，则继续向状态栈中加入转移后的状态序号；若查找到的动作时归约，则不读入当前输入字符，使用归约规则，将符号栈中涉及归约的字符清除，替换成产生式左侧的非终结符，并在状态栈中删去相同数量的状态序号，之后以当前状态栈和符号栈的栈顶元素为索引在 GOTO 表中查找，若未找到，则报错，否则向状态栈中加入要转移到的状态序号；若查找到的状态是接受状态，则归约过程结束，归约成功。

代码实现：

代码实现详见 附录 A6。

四、调试分析

4.1 测试数据，测试输出的结果

正确数据输出结果：（内容较多的文件只展示首尾部分）

- 词法分析预处理文件 Pre-Processed_Code.txt

```
int program(int a,int b,int c)
{
int i;
int j;
i=0;
if(a>(b+c))
{
j=a+(b*c+1);
}
else
{
j=a;
}
i=j*2;
while(i<=100)
{
i=i*2;
}
return i;
}
int demo(int a)
{
a=a+2;
return a*2;
}
void main()
{
int a;
int b;
int c;
a=3;
b=4;
c=2;
a=program(a,b,demo(c));
return;
}return;
```

- 词法分析单词种别表 Word_Label.txt

```
1  ! : 29
2  != : 43
3  # : 61
4  % : 30
5  & : 32
6  && : 40
7  ( : 57
8  ) : 58
9  * : 27
10 *= : 48
11 + : 25
12 ++ : 38
13 += : 46
14 , : 56
15 - : 26
```

```
<= : 42
= : 35
== : 44
> : 36
>= : 45
>> : 51
[ : 63
] : 64
^ : 34
blank : 54
break : 1
case : 2
char : 3
class : 4
continue : 5
default : 7
define : 9
do : 6
double : 8
else : 10
float : 70
for : 12
if : 13
include : 15
int : 14
integer : 69
iostream : 24
long : 16
main : 17
return : 18
signword : 52
switch : 19
typedef : 20
unsigned : 22
void : 21
while : 23
wrongword : 53
{ : 59
| : 33
|| : 41
} : 60
~ : 31
```

- 词法分析结果 Lexical_Result.txt

```
[关键字]----[14]----[int]
[空格]----[54]----[ ]
[标识符]----[52]----[program]
[左括号]----[57]----[(]
[关键字]----[14]----[int]
[空格]----[54]----[ ]
[标识符]----[52]----[a]
[分隔符]----[56]----[,]
[关键字]----[14]----[int]
[空格]----[54]----[ ]
[标识符]----[52]----[b]
[分隔符]----[56]----[,]
[关键字]----[14]----[int]
[空格]----[54]----[ ]
[标识符]----[52]----[c]
[右括号]----[58]----[)]
[左大括号]----[59]----[{]
[关键字]----[14]----[int]
[空格]----[54]----[ ]
[标识符]----[52]----[i]
```

```
[标识符]----[52]----[a]
[界符]----[55]----[;]
[关键字]----[14]----[int]
[空格]----[54]----[ ]
[标识符]----[52]----[b]
[界符]----[55]----[;]
[关键字]----[14]----[int]
[空格]----[54]----[ ]
[标识符]----[52]----[c]
[界符]----[55]----[;]
[标识符]----[52]----[a]
[单目运算符]----[35]----[=]
[整数]----[69]----[3]
[界符]----[55]----[;]
[标识符]----[52]----[b]
[单目运算符]----[35]----[=]
[整数]----[69]----[4]
[界符]----[55]----[;]
[标识符]----[52]----[c]
[单目运算符]----[35]----[=]
[整数]----[69]----[2]
[界符]----[55]----[;]
[标识符]----[52]----[a]
[单目运算符]----[35]----[=]
[标识符]----[52]----[program]
[左括号]----[57]----[(]
[标识符]----[52]----[a]
[分隔符]----[56]----[,]
[标识符]----[52]----[b]
[分隔符]----[56]----[,]
[标识符]----[52]----[demo]
[左括号]----[57]----[(]
[标识符]----[52]----[c]
[右括号]----[58]----[)]
[右括号]----[58]----[)]
[界符]----[55]----[;]
[关键字]----[18]----[return]
[界符]----[55]----[;]
[右大括号]----[60]----[}]
```

装 订 线

- 语法分析所用到的词法分析流 Lex_to_Parse.txt

```
int----int
<ID>----program
(----(
int----int
<ID>----a
,----,
int----int
<ID>----b
,----,
int----int
<ID>----c
)----)
{----{
int----int
<ID>----i
;----;
int----int
<ID>----j
;----;
<ID>----i
=----=
<INT>----0
;----;
if----if
(----(
<ID>----a
>----> 
(----(
<ID>----b
+----+
<ID>----c
```

```
<ID>----c
;----;
<ID>----a
=----=
<INT>----3
;----;
<ID>----b
=----=
<INT>----4
;----;
<ID>----c
=----=
<INT>----2
;----;
<ID>----a
=----=
<ID>----program
(----(
<ID>----a
,----,
<ID>----b
,----,
<ID>----demo
(----(
<ID>----c
)----)
)----)
;----;
return----return
;----;
}----}
```

- 处理过后的文法产生式集合 Grammar_Rules.txt

```
@ 终结符
return if else while void int <ID> <INT> ; , ( ) { } + - * / = > < >= <= != == #
非终结符
S  Program  ExtDefList  ExtDef  VarSpecifier  FunSpecifier  FunDec  Block  CreateFunTable_m  VarList  ParamDec
rule0  left: S  right: Program
rule1  left: Program  right: ExtDefList
rule2  left: ExtDefList  right: ExtDef ExtDefList
rule3  left: ExtDefList  right: @
rule4  left: ExtDef  right: VarSpecifier <ID> ;
rule5  left: ExtDef  right: FunSpecifier FunDec Block
rule6  left: VarSpecifier  right: int
rule7  left: FunSpecifier  right: void
rule8  left: FunSpecifier  right: int
rule9  left: FunDec  right: <ID> CreateFunTable_m ( VarList )
rule10  left: CreateFunTable_m  right: @
rule11  left: VarList  right: ParamDec , VarList
rule12  left: VarList  right: ParamDec
rule13  left: VarList  right: @
rule14  left: ParamDec  right: VarSpecifier <ID>
rule15  left: Block  right: { DefList StmtList }
rule16  left: DefList  right: Def DefList
rule17  left: DefList  right: @
rule18  left: Def  right: VarSpecifier <ID> ;
rule19  left: StmtList  right: Stmt StmtList
rule20  left: StmtList  right: @
rule21  left: Stmt  right: AssignStmt ;
rule22  left: Stmt  right: ReturnStmt ;
rule23  left: Stmt  right: IfStmt
rule24  left: Stmt  right: WhileStmt
rule25  left: Stmt  right: CallStmt ;
rule26  left: AssignStmt  right: <ID> = Exp
rule27  left: Exp  right: AddSubExp
rule28  left: Exp  right: Exp Relop AddSubExp
rule29  left: AddSubExp  right: Item
rule30  left: AddSubExp  right: Item + Item
rule31  left: AddSubExp  right: Item - Item
rule32  left: Item  right: Factor
rule33  left: Item  right: Factor * Factor
rule34  left: Item  right: Factor / Factor
rule35  left: Factor  right: <INT>
rule36  left: Factor  right: ( Exp )
rule37  left: Factor  right: <ID>
rule38  left: Factor  right: CallStmt
rule39  left: CallStmt  right: <ID> ( CallFunCheck Args )
rule40  left: CallFunCheck  right: @
rule41  left: Args  right: Exp , Args
rule42  left: Args  right: Exp
rule43  left: Args  right: @
rule44  left: ReturnStmt  right: return Exp
rule45  left: ReturnStmt  right: return
rule46  left: Relop  right: >
rule47  left: Relop  right: <
rule48  left: Relop  right: >=
rule49  left: Relop  right: <=
rule50  left: Relop  right: ==
rule51  left: Relop  right: !=
rule52  left: IfStmt  right: if IfStmt_m1 ( Exp ) IfStmt_m2 Block IfNext
rule53  left: IfStmt_m1  right: @
rule54  left: IfStmt_m2  right: @
rule55  left: IfNext  right: @
rule56  left: IfNext  right: IfStmt_next else Block
rule57  left: IfStmt_next  right: @
rule58  left: WhileStmt  right: while WhileStmt_m1 ( Exp ) WhileStmt_m2 Block
rule59  left: WhileStmt_m1  right: @
rule60  left: WhileStmt_m2  right: @
```

装 订 线

- ACTION 表和 GOTO 表(表格较大，只展示部分)

STATE	ACTION						GOTO										
	return	if	else	while	void	int	<ID>	<INT>	;			(	)	{	}	+	-
0					s1	s2											
1							r7										
2							r8										
3																	
4																	
5					s1	s2											
6							s9										
7							s10										
8																	
9									s12								
10											r10						
11													s14				
12					r4	r4											
13											s16						
14	r17	r17		r17		s17	r17							r17			
15					r5	r5											
16						s17						r13					
17							r6										
18							s24										
19	s25	s26		s27			s28							r20			
20	r17	r17		r17		s17	r17							r17			
21							s37										
22												s38					
23										s39		r12					
24									s40								
25							s41	s42	r45		s43						
26											r53						
27											r59						
28											s51						
29														s53			
30	s25	s26		s27			s28							r20			
31									s55								

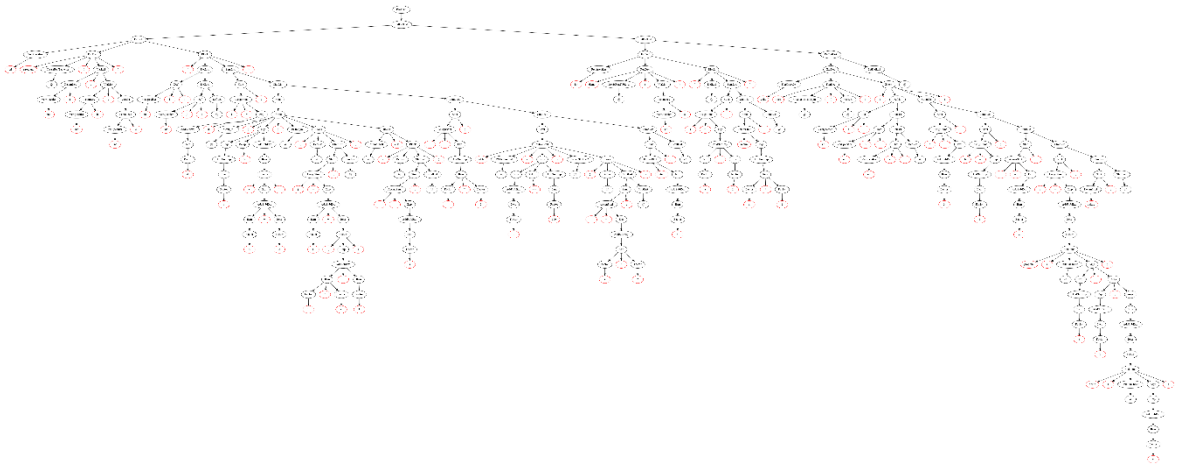
- LR(1)归约过程 Analysis_Procsee.txt

```
STEP        STATUS_STACK                SYMBOL_STACK
   0        0                           #
   1        0 2                           #int
   2        0 7                           #FunSpecifier
   3        0 7 10                           #FunSpecifier<ID>
   4        0 7 10 13                           #FunSpecifier<ID>CreateFunTable_m
   5        0 7 10 13 16                           #FunSpecifier<ID>CreateFunTable_m(
   6        0 7 10 13 16 17                           #FunSpecifier<ID>CreateFunTable_m(int
   7        0 7 10 13 16 21                           #FunSpecifier<ID>CreateFunTable_m(VarSpecifier
   8        0 7 10 13 16 21 37                           #FunSpecifier<ID>CreateFunTable_m(VarSpecifier<ID>
   9        0 7 10 13 16 23                           #FunSpecifier<ID>CreateFunTable_m(ParamDec
  10        0 7 10 13 16 23 39                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,
  11        0 7 10 13 16 23 39 17                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,int
  12        0 7 10 13 16 23 39 21                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,VarSpecifier
  13        0 7 10 13 16 23 39 21 37                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,VarSpecifier<ID>
  14        0 7 10 13 16 23 39 23                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,ParamDec
  15        0 7 10 13 16 23 39 23 39                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,ParamDec,
  16        0 7 10 13 16 23 39 23 39 17                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,ParamDec,int
  17        0 7 10 13 16 23 39 23 39 21                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,ParamDec,VarSpecifier
  18        0 7 10 13 16 23 39 23 39 21 37                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,ParamDec,VarSpecifier<ID>
  19        0 7 10 13 16 23 39 23 39 23                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,ParamDec,ParamDec
  20        0 7 10 13 16 23 39 23 39 58                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,ParamDec,VarList
  21        0 7 10 13 16 23 39 58                           #FunSpecifier<ID>CreateFunTable_m(ParamDec,VarList
  22        0 7 10 13 16 22                           #FunSpecifier<ID>CreateFunTable_m(VarList
  23        0 7 10 13 16 22 38                           #FunSpecifier<ID>CreateFunTable_m(VarList)
  24        0 7 11                        #FunSpecifierFunDec
  25        0 7 11 14                        #FunSpecifierFunDec{
  26        0 7 11 14 17                        #FunSpecifierFunDec{int
  27        0 7 11 14 18                        #FunSpecifierFunDec{VarSpecifier
  28        0 7 11 14 18 24                        #FunSpecifierFunDec{VarSpecifier<ID>
  29        0 7 11 14 18 24 40                        #FunSpecifierFunDec{VarSpecifier<ID>;
  30        0 7 11 14 20                        #FunSpecifierFunDec{Def

 340        0 5 5 7 11 14 19 30 30 30 28 52 48                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmt<ID>=Factor
 341        0 5 5 7 11 14 19 30 30 30 28 52 47                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmt<ID>=Item
 342        0 5 5 7 11 14 19 30 30 30 28 52 46                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmt<ID>=AddSubExp
 343        0 5 5 7 11 14 19 30 30 30 28 52 82                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmt<ID>=Exp
 344        0 5 5 7 11 14 19 30 30 30 31                        #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtAssignStmt
 345        0 5 5 7 11 14 19 30 30 30 31 55                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtAssignStmt;
 346        0 5 5 7 11 14 19 30 30 30 30                        #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmt
 347        0 5 5 7 11 14 19 30 30 30 30 25                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmtreturn
 348        0 5 5 7 11 14 19 30 30 30 30 32                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmtReturnStmt
 349        0 5 5 7 11 14 19 30 30 30 30 32 56                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmtReturnStmt;
 350        0 5 5 7 11 14 19 30 30 30 30 30                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmtStmt
 351        0 5 5 7 11 14 19 30 30 30 30 30 54                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmtStmtStmtList
 352        0 5 5 7 11 14 19 30 30 30 30 54                           #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmtStmtList
 353        0 5 5 7 11 14 19 30 30 30 54                        #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtStmtList
 354        0 5 5 7 11 14 19 30 30 54                     #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtStmtList
 355        0 5 5 7 11 14 19 30 54                  #ExtDefExtDefFunSpecifierFunDec{DefListStmtStmtList
 356        0 5 5 7 11 14 19 29                  #ExtDefExtDefFunSpecifierFunDec{DefListStmtList
 357        0 5 5 7 11 14 19 29 53                     #ExtDefExtDefFunSpecifierFunDec{DefListStmtList}
 358        0 5 5 7 11 15                        #ExtDefExtDefFunSpecifierFunDecBlock
 359        0 5 5 5                           #ExtDefExtDefExtDef
 360        0 5 5 5 8                           #ExtDefExtDefExtDefExtDefList
 361        0 5 5 8                           #ExtDefExtDefExtDefList
 362        0 5 8                           #ExtDefExtDefList
 363        0 4                           #ExtDefList
 364        0 3                           #Program

Parse successfully!
```

- 语法分析过程的语法树图片 ParseTree.png



- 错误数据输出结果 Lexical_Result.txt

```
[标识符]----[52]----[from]
[空格]----[54]----[ ]
[标识符]----[52]----[asyncore]
[空格]----[54]----[ ]
[标识符]----[52]----[import]
[空格]----[54]----[ ]
[标识符]----[52]----[read]
[标识符]----[52]----[from]
[空格]----[54]----[ ]
[标识符]----[52]----[cProfile]
[空格]----[54]----[ ]
[标识符]----[52]----[import]
[空格]----[54]----[ ]
[标识符]----[52]----[label]
[标识符]----[52]----[import]
[空格]----[54]----[ ]
[标识符]----[52]----[csv]
[标识符]----[52]----[from]
[空格]----[54]----[ ]
[标识符]----[52]----[urllib]
[点符]----[62]----[.]
[标识符]----[52]----[request]
[空格]----[54]----[ ]
[标识符]----[52]----[import]
[空格]----[54]----[ ]
[标识符]----[52]----[DataHandler]
[结束符]----[61]----[#]
[标识符]----[52]----[TODO]
```

```
[标识符]----[52]----[writer]
[点符]----[62]----[.]
[标识符]----[52]----[writerows]
[左括号]----[57]----[(]
[标识符]----[52]----[datalist_not_same]
[右括号]----[58]----[)]
```

- 错误数据归约过程报错 Analysis_Process.txt

```
    STEP        STATUS_STACK                SYMBOL_STACK
       0          0                           #

Parse Error:Non-existed action!
Present Status: 0
Present Terminal Type: <ID>
```

4.2 时间复杂度分析

1. 文法读入处理

文法读入处理里，时间复杂度主要体现在 first 集合的计算还有闭包的计算。

First 集合需要对每个非终结符进行遍历，然后需要遍历文法，找到对应的文法，其次完成右部 symbol 的判断，综上，共进行三次遍历， n 个非终结符，m 个文法，1~4 个 symbol，时间复杂度在$O(nm*4)->O(n^2)$。

Closure 集合的计算需要遍历核心项，然后遍历文法，找到对应的文法，对后继非终结符的 first 集合遍历，完成闭包的添加，综上，共有 n 个核心项，m 个文法，first 集合有 h 个成员，时间复杂度在$O(nmh)->O(n^3)$。

2. 闭包集合及 DFA 计算

在计算闭包集合过程中，外围循环计算次数取决于最终 DFA 状态集合的状态数目，假设该数目为N，内层循环的计算次数取决于每一状态中可转移的项目的数目，假设该数目最大为K，则闭包集合计算的时间复杂度为$O(NK)$。

3. 归约过程

在归约过程中，主要的计算次数在于遍历输入终结符串，以及在其中归约动作的次数，若词法分析输入串的大小为L，归约过程中使用到归约动作的次数是R，则归约的时间复杂度为$O(L+R)$。

4.3 每个模块设计和调试时存在问题的思考

1. 归约过程中 epsilon 的影响

在归约分析过程中，若查到 ACTION 表中是归约动作，且用于归约的产生式右侧的长度为 1 时，需要对 epsilon 进行特判。因为在本程序的存储结构中，epsilon 作为一个单独的符号存储，当产生式右侧没有任何符号时存储为 epsilon，占据一位，而实际上是没有符号的，如果按照一位计算，之后的归约过程应当从状态栈和符号栈中各弹出一个元素，就会导致归约错误。故此时应将右侧产生式的长度特殊修改为 0，之后才可以正确归约。

2. First 集合的求法

First 集合的计算，由于函数的传参设计不够清晰，导致出现了 first 的 set 集合合

并出现错位的情况，进而导致 first 集合中包含了符号表中所有的符号集合。后续的函数设计在函数解释阶段进行了清晰的接口说明，有效避免了后续错误的发生

五、总结和收获

（包括收获、遇到的问题、解决问题过程的思考，在实验过程中对课程的认识等内容）

在本次编译原理的词法、语法分析器实验中，我们对于编译原理学科有了进一步的了解，编译原理是一门关于编程语言的学科，重点在于如何设计一门机器可以识别的、对使用者友好的编程语言。通过自己动手实现一个类 C 语言的编译器，我们不仅对于 C 语言的语法更加熟悉，还了解了我们使用的编译器背后的工作机制，从词法分析到语法分析，逐步的将源程序向着等价机器语言的方向进行转换；同时，在实践过程中，也增强了我们对于课堂上学到的关于词法、语法分析的相关理论的理解和掌握。

此外为了对语法分析有更进一步的理解，我们查阅了 YACC 的迭代版 BISON 的编译源码，了解到了工程项目和课程作业的不同，同时通过对 bison 源码架构的学习，了解到我们的项目还有进一步提高时间效率和空间利用率的空间。比如记录符号表位置的下标可以使用 bit 位来表示，这样能极大的提高空间利用率，同时 bit 位的操作也能更好的提高性能。在时间效率方面，将 symbol 的 name 作为 hash 存储，能省去冗余的寻找过程，也能使时间效率得到极大的提升。

附录 A

A1 void analysis::getStrBuffer()

```
void analysis::getStrBuffer() {
    char c = '\0';
    int buffer_flag = 0;//缓冲区是否需要轮转
    while (1)
    {
        c = fgetc(fin);
        if (c == EOF)
        {
            //结束了
            deleNotes();
            deleSpaces();
            if (buffer_read[buffer_choose].count > 0)
            {
                strcpy(buffer_end.buffer, buffer_read[buffer_choose].buffer);
                buffer_end.count = buffer_read[buffer_choose].count;
                //进入状态机处理
                //注：给的缓冲区 有可能是不完整的字串 如果传入的太长了
                //eg: "111*n"超过300个了，就会分割开，
                buffer_read[buffer_choose].count = 0;
                fprintf(fout_pre, "%s\n", buffer_read[buffer_choose].buffer);
                spearateStates();
            }
            break;
        }

        //缓冲池满了
        if (buffer_read[buffer_choose].count == BUFFER_SIZE - 2)
        {
            buffer_read[buffer_choose].buffer[buffer_read[buffer_choose].count] = c;
            int i;
            for (i = 0; i < buffer_read[buffer_choose].count; i++)
            {
                if (isDelimiter(buffer_read[buffer_choose].buffer[i]))
                {
                    int j;//分界点
                    int k;
                    //把buffer_choose的转移到1-buffer_choose中，
                    for (j = 0, k = i + 1; k <= buffer_read[buffer_choose].count; k++,
j++)
                    {
                        buffer_read[1 - buffer_choose].buffer[j] =
buffer_read[buffer_choose].buffer[k];
                    }
                    //count大小重新设置
                    buffer_read[1 - buffer_choose].count = j;
                    buffer_read[buffer_choose].count = i;
```

```
                    //设置终结点
                    buffer_read[1 - buffer_choose].buffer[j] = '\0';
                    buffer_read[buffer_choose].buffer[i + 1] = '\0';

                    //缓冲区轮转
                    buffer_flag = 1;

                    break;
                }
            }

        }
        else if (c == '\n' && !note_flag)
        {
            buffer_read[buffer_choose].buffer[buffer_read[buffer_choose].count] = '\0';
        }
        else if (c == '\n')
        {
            buffer_read[buffer_choose].buffer[buffer_read[buffer_choose].count] = '\0';
        }
        else {
            buffer_read[buffer_choose].buffer[buffer_read[buffer_choose].count++] = c;
            continue;//继续吧
        }
        //继续处理换行后/缓冲池满后的处理
        deleNotes();
        deleSpaces();

        if (buffer_read[buffer_choose].count > 0)
        {
            strcpy(buffer_end.buffer, buffer_read[buffer_choose].buffer);
            buffer_end.count = buffer_read[buffer_choose].count;
            //进入状态机处理
            //注：给的缓冲区 有可能是不完整的字串 如果传入的太长了
            //eg: "111*n"超过300个了，就会分割开，
            buffer_read[buffer_choose].count = 0;
            fprintf(fout_pre, "%s\n", buffer_read[buffer_choose].buffer);
            spearateStates();
        }
        if (buffer_flag == 1)
        {
            //下一次 缓冲区轮转
            buffer_read[buffer_choose].count = 0;
            buffer_choose = 1 - buffer_choose;
            buffer_flag = 0;
        }
    }
    cout << "The result of lexical analysis has been saved in the res_out.txt file." <<
endl;
    cout << "The pre-processed code has been saved in the pre-processed_code.txt file."
<< endl;
    cout << "The word_lable has been saved in the word-lable.txt file." << endl;
    cout << "The analysis_res has been saved in the analysis_res.txt file." << endl;
}
```

A2 void analysis::spearateStates()

```
void analysis::spearateStates()
{
    char word[BUFFER_SIZE];
    int count = 0;//当前word中的字符个数
    bool finish = false;
    int state = 0;//初态，state为0就表示了在初态

    for (int i = 0; i <= buffer_end.count; i++)
    {
        switch (state)
        {
            //初态
        case 0:
            switch (charKind(buffer_end.buffer[i]))
            {
            case 1://字母
                word[count++] = buffer_end.buffer[i];
                state = 1;
                break;
            case 2://数字
                word[count++] = buffer_end.buffer[i];
                state = 2;
                break;
            case 3://$,_
                word[count++] = buffer_end.buffer[i];
                state = 3;
                break;
            case 4://转义符只会在字符串内部使用，否则作为一个字符单独出来
                word[count++] = buffer_end.buffer[i];
                state = 4;
                break;
            case 5:
                word[count++] = buffer_end.buffer[i];
                state = 5;
                break;
            case 6:
                word[count++] = buffer_end.buffer[i];
                state = 6;
                break;
            case 7:
                word[count++] = buffer_end.buffer[i];
                state = 7;
                break;
            case 8:
                word[count++] = buffer_end.buffer[i];
                state = 8;
                break;
            case 10:
                word[count++] = buffer_end.buffer[i];
                state = 10;
                break;
```

装 订 线

```
                default:
                    word[count++] = buffer_end.buffer[i];
                    break;
                }
                break;
            case 1:
                switch (charKind(buffer_end.buffer[i]))
                {
                case 1:case 2:case 3:
                    word[count++] = buffer_end.buffer[i];
                    break;
                default:
                    word[count] = '\0';
                    i--;
                    finish = 1;
                    state = 9;//结束状态
                }
                break;
            case 2:
                switch (charKind(buffer_end.buffer[i]))
                {
                case 1:
                case 2:
                    word[count++] = buffer_end.buffer[i];
                    break;
                case 7:
                    if (buffer_end.buffer[i] == '.')
                    {
                        word[count++] = buffer_end.buffer[i];
                        break;
                    }
                    else
                    {
                        word[count] = '\0';
                        i--;
                        finish = 1;
                        state = 9;//结束状态
                    }
                    break;
                case 8:
                    //现在是+-，前面是Ee
                    if ((buffer_end.buffer[i] == '+' || buffer_end.buffer[i] == '-') &&
(buffer_end.buffer[i - 1] == 'e' || buffer_end.buffer[i - 1] == 'E'))
                    {
                        word[count++] = buffer_end.buffer[i];
                        break;
                    }
                    else
                    {
                        word[count] = '\0';
                        i--;
                        finish = 1;
                        state = 9;//结束状态
                        break;
```

```
            }
        default:
            word[count] = '\0';
            i--;
            finish = 1;
            state = 9;//结束状态
            break;
        }
        break;
    case 3://好像$_和字母是一样的操作
        switch (charKind(buffer_end.buffer[i]))
        {
        case 1:case 2:case 3:
            word[count++] = buffer_end.buffer[i];
            break;
        default:
            word[count] = '\0';
            i--;
            finish = 1;
            state = 9;//结束状态
            break;
        }
        break;
    case 4:
        //字符串内转义符的情况在5态内部处理，这里处理单独的'\'
        word[count] = '\0';
        i--;
        finish = 1;
        state = 9;//结束状态
        break;
    case 5:
        word[count++] = buffer_end.buffer[i];
        if (buffer_end.buffer[i] == '"')
        {
            //此时一定不是初态，所以不需要判断i与1的关系
            if (buffer_end.buffer[i - 1] == '\\')
            {
            }
            else
            {
                word[count] = '\0';
                finish = 1;
                state = 9;
            }
        }
        break;
    case 6:
        word[count++] = buffer_end.buffer[i];
        if (buffer_end.buffer[i] == '\'')
        {
            //还有一种情况是'\''，还是得判断
            if (buffer_end.buffer[i - 1] == '\\')
            {
            }
```

装 订 线

```
            else
            {
                word[count] = '\0';
                finish = 1;
                state = 9;
            }
        }
        break;
    case 7:
        //要结束的字符，直接结束
        word[count] = '\0';
        i--;
        finish = 1;
        state = 9;
        break;
    case 8:
        switch (charKind(buffer_end.buffer[i]))
        {
        case 8:case 11:
            word[count++] = buffer_end.buffer[i];
            break;
        default:
            word[count] = '\0';
            i--;
            finish = 1;
            state = 9;
            break;
        }
        break;
    case 9://结束态
        //此时word已经得到，并且最后以\0结尾，故状态换成初始状态
        state = 0;
        count = 0;
        finish = 0;
        i--;
        kindJudge(word);
        break;
    case 10://空格另加
        switch (charKind(buffer_end.buffer[i]))
        {
        case 10:
            word[count++] = buffer_end.buffer[i];
            break;
        default:
            word[count] = '\0';
            i--;
            finish = 1;
            state = 9;
            break;
        }
        break;
    default:
        break;
    }
```

```
            if (buffer_end.buffer[i + 1] == '\0')
            {
                word[count] = '\0';
                kindJudge(word);
                break;
            }
        }
    }
```

A3 set<int>grammar::GetFirst(const vector<int>& str)

```
set<int>grammar::GetFirst(const vector<int>& str) {
set<int>first_set;
// above all 是不是空串
if (str.empty()) {
    return first_set;
}
//1.判断空串是否需要加入
//2.判断是终结符还是非终结符
// 3.判断非终结符能否推导出空串
bool is_epsilon = true;
int empty_location = Find_Symbol_Index_By_Token(EpsilonToken);
for (auto i = str.begin(); i != str.end(); i++)
{
    if (symbols[*i].type == symbol_class::token_sym)
    {
        is_epsilon = false;
        mergeSet(symbols[*i].first_set, first_set, empty_location);

        break;
    }
    if (symbols[*i].type == symbol_class::epsilon)
    {
        is_epsilon = false;
        first_set.insert(*i);
        break;
    }
    mergeSet(symbols[*i].first_set, first_set, empty_location);
    is_epsilon = symbols[*i].first_set.count(empty_location) && is_epsilon;
    if (!is_epsilon)
        break;

}
if (is_epsilon)
    first_set.insert(empty_location);
return first_set;
}
```

A4 void analysis::spearateStates()

```
LR1_closure LR1_Grammar::computeClosure(vector<LR1_item> lr1)
{
//传入核心项
```

```
//计算其闭包
LR1_closure closure_now;
closure_now.key_item = lr1;
closure_now.closure = lr1;
//遍历核心项
for (int i = 0; i < closure_now.closure.size(); i++) {
    //处理当前rule
    LR1_item item_now = closure_now.closure[i];
    //如果*在最后一个位置
    if (item_now.dot_site >= item_now.right.size())
    {
        continue;
    }
    //当前rule下，dot后的符号对应的下标
    int dot_next_symbol_index = item_now.right[item_now.dot_site];
    symbol dot_next_symbol = symbols[dot_next_symbol_index];
    //开始符号判断
    // 如果这玩意后面是个空串，那么设置为后继
    if (dot_next_symbol.type == symbol_class::epsilon)
    {
        closure_now.closure[i].dot_site++;
        continue;
    }
    //如果这玩意后面是个终结符 那就不用加
    if (dot_next_symbol.type == symbol_class::token_sym)
    {
        continue;
    }
    //如果这玩意后面是个非终结符，把这个非终结符的first加进来
    //将dot后面从第二个开始所有的字符和加入的终结符合并，求一个first集合
    vector<int>BetaA(item_now.right.begin() + item_now.dot_site + 1,
item_now.right.end());
    BetaA.push_back(item_now.forward);
    //初始化完成
    set<int> BetaAset = GetFirst(BetaA);
    //A->α·Bβ,a
    //B->XX ,first(β,a)
    //完成此步的添加
    //遍历所有rule，找到对应的rule规则
    for (int j = 0; j < rules.size(); j++)
    {
        rule rule_now = rules[j];
        if (dot_next_symbol_index != rule_now.left_symbol)
            continue;
        //开始加入到closure里
        //此处仍需要判断右部产生式是不是空串
        //空串 dot位置在末端
        //遍历first
        for (auto it = BetaAset.begin(); it != BetaAset.end(); it++) {
            //closure里是否有这项?
            bool have_exist = false;
            bool is_epsilon = false;
            is_epsilon = (symbols[rule_now.right_symbol[0]].type ==
symbol_class::epsilon);
```

```
                for (auto temp = closure_now.closure.begin(); temp !=
    closure_now.closure.end(); temp++)
                {
                    if (* temp == LR1_item(rule_now.left_symbol, rule_now.right_symbol,
    have_exist, *it, j))
                    {
                        have_exist = true;
                        break;
                    }
                }
                //如果没有
                if (!have_exist)
                {
                    closure_now.closure.push_back(LR1_item(rule_now.left_symbol,
    rule_now.right_symbol, is_epsilon, *it, j));
                }

            }
        }

    }
    return closure_now;
}
```

A5 void LR1_Grammar::computeACTION_GOTO()

```
void LR1_Grammar::computeACTION_GOTO()
{
//计算完成DFA和闭包集后，构造ACTION表，<closure的编号，符号的编号>,动作
//记录的是所有的终结符，这里只记录存在的状态
//DFA中记录的都是转移，还需要记录归约状态，里面是产生式的序号
//所有的REJECT状态都没有记录

//根据DFA得到移进动作
for (auto it = this->DFA.begin(); it != this->DFA.end(); it++)
{
    //转移符号是终结符，记录在ACTION表中
    if (terminals.find(it->first.second) != terminals.end())
    {
        ACTION_item act_item(ACTION_Option::SHIFT_IN, it->second);
        this->ACTION[pair<int, int>(it->first.first, it->first.second)] = act_item;
    }
    else
    {
        //非终结符记录在GOTO表中
        GOTO_item goto_item(GOTO_Option::GO, it->second);
        this->GOTO[pair<int, int>(it->first.first, it->first.second)] = goto_item;
    }
}

//根据闭包的归约项得到归约/接受动作
for (int i = 0; i < this->closure_sum.size(); i++)
{
```

```
        vector<pair<int, int>> reduce_line = this->closure_sum[i].getReduceSymbol();
        for (int j = 0; j < reduce_line.size(); j++)
        {
            //在第i个闭包状态，遇到终结符reduce_line[j].first，要使用reduce_line[j].second
产生式归约
            //*******************************************
            //  (TODO:需不需要看前向符号是#？)
            //如果该归约项目是初始归约项目，则应设置为接受状态

            if (reduce_line[j].second == start_location)
            {
                ACTION_item act_item(ACTION_Option::ACCEPT, reduce_line[j].second);
                this->ACTION[pair<int, int>(i, reduce_line[j].first)] = act_item;
            }
            else
            {
                ACTION_item act_item(ACTION_Option::REDUCE, reduce_line[j].second);
                this->ACTION[pair<int, int>(i, reduce_line[j].first)] = act_item;
            }
        }

    }
    }
```

A6 int LR1_Grammar::analyze(vector<unit>& lexical_res)

```
    void LR1_Grammar::analyze(vector<unit>& lexical_res)
    {
    vector<int> status_stack;//状态栈
    vector<int> symbol_stack;//符号栈
    int step = 0;

    unit end(EndToken, EndToken);
    lexical_res.push_back(end);//在输入串的最后加上结束符号
    status_stack.push_back(0);//状态栈先压入状态0
    symbol_stack.push_back(Find_Symbol_Index_By_Token(EndToken));//在符号栈中先放入结束符号

    ofstream ofs(analysis_process_path, ios::out);

    const int width = 5;
    const int interval = 10;
    const int start_step = 10;
    const int step_status = 20;
    const int status_symbol = 30;
    const int symbol_lex = 150;

    ofs << setw(start_step) << "STEP" << setw(step_status) << "STATUS STACK" <<
setw(status_symbol) << "SYMBOL STACK" << setw(symbol_lex) << "INPUT" << endl;

    //开始进行语法分析
    for (int i = 0; i < lexical_res.size(); i++)
    {
        //每次从输入串读入一个非终结符，放入符号栈中，看当前状态下遇到该符号后ACTION表中的
```

```
操作
        //如果是转移，就状态栈中加入转移后的状态
        //如果是归约，就使用归约规则，将符号栈中涉及归约的项换成右侧表达式，状态栈中删去相同数量的状态，并从GOTO表中查此时状态遇到该非终结符应转移到哪里
        //并将转移后的状态压入状态栈
        //当遇到ACTION中为acc时，结束，或reject（即ACTION表中找不到转移），则结束（GOTO中找不到也是错误）
        string present_terminal = lexical_res[i].value;
        int present_terminal_serial = Find_Symbol_Index_By_Token(present_terminal);
        int present_status = status_stack.back();
        auto it = ACTION.find(pair<int, int>(present_status, present_terminal_serial));

        int error_code = 0;
        //不存在，即REJECT，错误退出
        if (it == ACTION.end())
        {
            error_code = 1;
        }
        else
        {
            switch (it->second.op)
            {
            case ACTION_Option::SHIFT_IN:
            {
                //移进
                status_stack.push_back(it->second.serial);//新状态入栈
                symbol_stack.push_back(present_terminal_serial);//读入的终结符压栈
                break;
            }
            case ACTION_Option::REDUCE:
            {
                //归约，要归约则当前输入串不加一！！
                i--;
                rule rule_need = rules[it->second.serial];//要使用的产生式
                int right_length = rule_need.right_symbol.size();//要归约掉的长度
                for (int k = 0; i < right_length; k++)
                {
                    status_stack.pop_back();//状态栈移出
                    symbol_stack.pop_back();//符号栈移出
                }
                symbol_stack.push_back(rule_need.left_symbol);//符号栈压入非终结符
                int temp_status = status_stack.back();

                //归约之后查看GOTO表
                auto goto_it = GOTO.find(pair<int, int>(temp_status,
rule_need.left_symbol));
                if (goto_it == GOTO.end())//不存在转移，则应退出GOTO，编译错误
                {
                    error_code = 2;
                    break;
                }
                else
                {
                    if (goto_it->second.op == GOTO_Option::GO)
```

```
                {
                    status_stack.push_back(goto_it->second.serial);//将新状态压栈
                }
                else//不会出现
                {
                    error_code = 2;
                    break;
                }
            }
            break;
        }
        case ACTION_Option::ACCEPT:
        {
            //接受状态，直接返回
            ofs << "Parse successfully!" << endl;
            ofs.close();
            return;
            break;
        }
        case ACTION_Option::REJECT:
        {
            error_code = 1;
            break;
        }
        default:
            break;
        }
    }
    //有error，直接退出
    if (error_code == 1)
    {
        ofs << "Parse Error:Non-existed action!" << endl;
        break;
    }
    else if (error_code == 2)
    {
        ofs << "Parse Error:Non-existed goto!" << endl;
        break;
    }

    //输出这一行
    ofs << setw(start_step) << step;
    ofs << setw(start_step);
    for (int t = 0; t < status_stack.size(); t++)
    {
        ofs << " " << status_stack[t];
    }
    ofs << setw(status_symbol);
    for (int t = 0; t < symbol_stack.size(); t++)
    {
        ofs << symbols[symbol_stack[t]].tag;
    }
    ofs << setw(symbol_lex);
    for (int t = i; t < lexical_res.size(); t++)
```

```
        {
            ofs << lexical_res[t].value;
        }

        step++;
    }
    ofs.close();
}
```

装

订

线