同济大学编译原理课程

词法分析暨

LR(1)语法分析实验报告

同济大学
TONGJI UNIVERSITY

院　　系　电子与信息工程学院

专　　业　计算机科学与技术

组　　长　1852024 李兵磊

组　　员　1853790 庄镇华

组　　员　1854127 周家旋

指导老师　卫志华

完成日期　2020 年 12 月 9 日

目录

概述

目的与意义

更深入的理解、掌握、运用编译原理的相关知识，完整实现一个类 C 语言的编译器。现阶段，要根据已经学过的词法分析和 LR1 语法分析，通过设计、编制和调试一个典型的 LR（1）语法分析界面，进一步掌握语法分析方法。

主要任务

(1)根据 LR(1)分析法编写一个语法分析程序，可选择以下一项作为分析算法的输入：

a.直接输入根据已知文法构造的 LR(1)分析表；

b.输入已知文法的项目集规范族和转换函数，由程序自动生成 LR(1)分析表；

c.输入已知文法，由程序自动构造识别该文法活前缀 DFA 并生成 LR(1)分析表。

(2)所开发的程序可适用于不同的文法和任意输入串，且能判断该文法是否为 LR(1)文法。

(3)对输入的任意符号串，所编制的语法分析程序应能正确判断此串是否为文法的句子（句型分析），并要求输出分析过程。

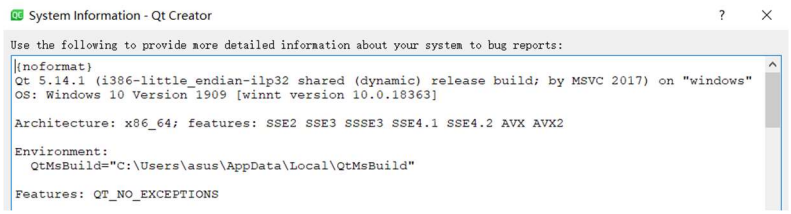
主要的开发工具

QT5.14.1

Qt 是桌面，嵌入式和移动的跨平台应用开发框架。支持的平台包括 Linux，OS X，Windows，VxWorks，QNX，Android，iOS，BlackBerry，Sailfish OS 等。Qt 本身不是一种编程语言。它是一个用 C ++编写的框架。但它不仅仅只是一个 GUI 工具包，它提供了在网络，数据库，OpenGL，Web 技术，传感器，通信协议（蓝牙，串行端口，NFC），XML 和 JSON 处理，打印，

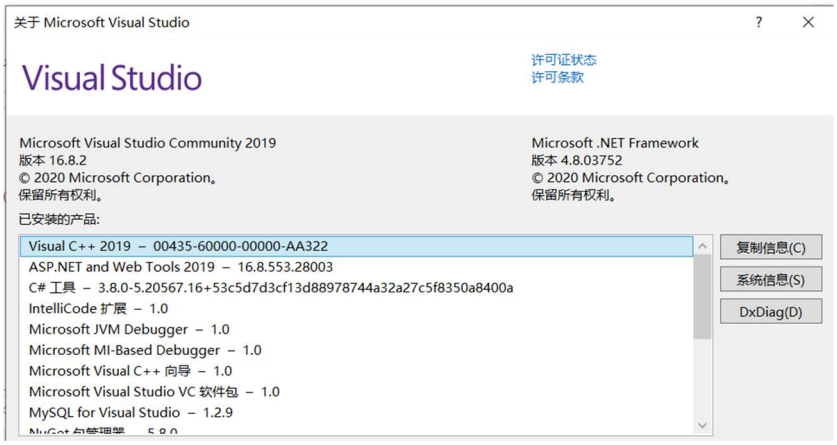
PDF 生成等领域的跨平台开发的模块。

Qt 拥有自己的集成开发环境（IDE），名为 Qt Creator。它运行在 Linux，OS X 和 Windows 上，提供智能代码完成，语法高亮，集成帮助系统，调试器和分析器集成以及所有主要版本控制系统（例如 git，Bazaar）的集成。在本次语法分析器课程设计，我们使用 QT5 进行程序的可视化，其优良的跨平台性使得生成的程序代码可以在任何支持的平台上编译与运行，而不需要修改源代码，会自动依平台的不同，表现平台特有的图形界面风格；其良好的封装机制使得 Qt 的模块化程度非常高，可重用性较好，对于我们实验开发来说是非常方便的；其丰富的 API 导致包括多达 250 个以上的 C＋＋类，还提供基于模板的 collections、serialization、file、I/O device、directory management 和 date/time 类。



VS2019

VS 是 Microsoft Visual Studio 的简称，是美国微软公司的开发工具包系列产品，是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如 UML 工具、代码管控工具、集成开发环境(IDE)等等。

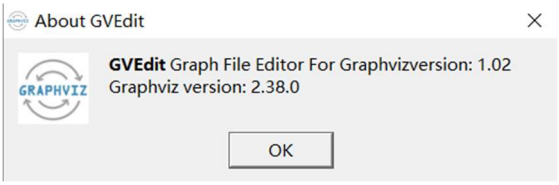


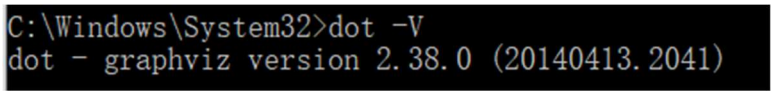
所写的目标代码适用于微软支持的所有平台，包括 Microsoft Windows、

Windows Mobile、Windows CE、.NET Framework、.NET Compact Framework 和 Microsoft Silverlight 及 Windows Phone。Visual Studio 是目前最流行的 Windows 平台应用程序的集成开发环境，最新版本为 Visual Studio 2019 版本，基于.NET Framework 4.7。我们使用 VS2019 主要进行目标代码编写与调试，利用 VS2019 友好的 IDE 开发环境、跨平台开发、高级调试和诊断功能与丰富的协作与测试工具等特性。

GRAPHVIZ2.38.0

Graphviz 是一款图形绘制软件，与一般的"所见即所得"的普通画图工具主要使用鼠标拖拽不同，graphviz 使用一门名为 dot 的语言用来描述图表，用户使用 dot 写脚本，graphviz 根据脚本自动布局生成图表。graphviz 将这种方式称为"所思即所得"。我们主要使用 graphviz 进行 LR(1)语法树的绘制，利用其两个主要好处，一个是将用户从排版中解放出来，由工具自动处理这个过程，用户不必再关心如何布局，修改添加删除节点的时候也不用再对整个图的排版布局重新进行人工的整理；另一个好处是某些复杂的情况，比如代码的类图和调用图，dot 脚本可以使用其他工具自动生成。





课程设计计划

工作计划表		
时间	进度安排	完成情况
11.19-11.25	查阅与 LR(1)语法分析器相关的资料，组织小组会议，确定语法分析器的整体框架、重要的数据结构，并将总程序分成几个模块，确定模块之	基本完成了总体设计与成员分工

	间接口，最后进行模块设计任务的分配。	
11.26-12.2	具体的程序代码的编写，组织小组会议，确认各成员的模块的编写进度，交流解决代码的 bug，并对原先设定的框架结构进行部分调整。	各成员基本完成模块代码的编写
12.3-12.9	进行模块代码的整合，并进行调试，在调试过程中修正存在的 bug，优化语法分析器程序。之后，用 QT 对程序进行可视化，并撰写 LR(1)分析器课程设计报告。	完成了代码整合、代码调试、程序可视化与设计报告撰写

需求分析

程序任务输入：

• 选择一个类 C 的源程序文件输入。

• 分别进行语法分析，输出语法分析的结果。

• 修改或者既定地输入一个文法的基本信息，即文法的非终结符集合、文法的终结符集合与文法的产生式集合(文法的起始符号由产生式集合中第一个产生式的左边的文法符号），判断该文法是不是一个 LR（1）文法，并且产生 LR（1）分析表 action 表和 goto 表。

• 输入刚才词法分析的结果，测试该源程序文件是不是一个符合既定文法的源程序，错误的话给出错误提示。正确的话显示规约移进分析过程和最终的语法分析树。

输入形式

类 C 源程序输入

输入文件的格式不限制（*.cpp *.txt *.dat 均可），输入示例：

```
void main(void){
   int a;
   a=3;
   a=myfun(a));
   return;
}
```

文法输入形式

文法输出要转化为数字标号来输入，例如下列文法：

S→AB

A→aA| ε

B→bB| ε

设定 ε 为 0，a 为 1，b 为 2，变元 S 为-1，A 为-2，B 为-3。则输入文法的格式如下表所示：

```
2 3 5
1 2
-1 -2 -3
3 -1 -2 -3
3 -2 1 -2
2 -2 0
3 -3 2 -3
2 -3 0
```

其中第一行的三个数据分别为终结符的数量（不包含 ε ），非终结符的数量，产生式的数量。

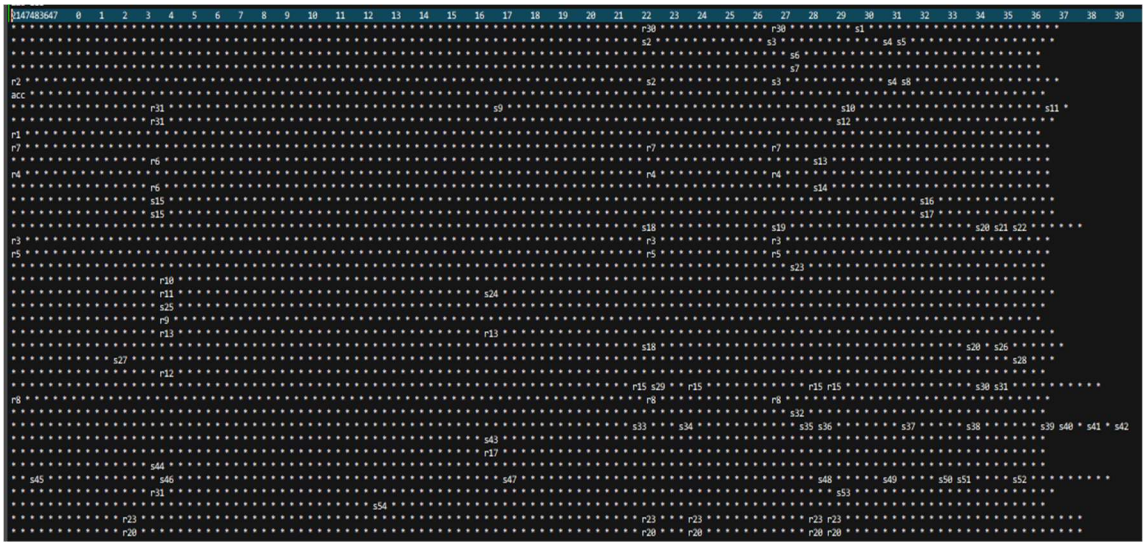
第二行的数据是终结符的标号，数量要和第一行的第一个数字相符合。

第三行的数据是非终结符的标号，数量要和第一行的第二个数字相符合。

剩余的数据行数要和第一行的第三个数字相符合，并且第一条产生式默认为开始的产生式，每条产生式的第一个数字为这条产生式中从左到右的终结符和非终结符的总和，第二个数字是左端非终结符的标号，剩余的数字是右边产生式的依次标号。注意如果同一个非终结符有多个产生式，要将他们分开，写成多行的形式，例如将 A→aA| ε 分成 A→aA 和 A→ ε 两行来输入。

输出形式：

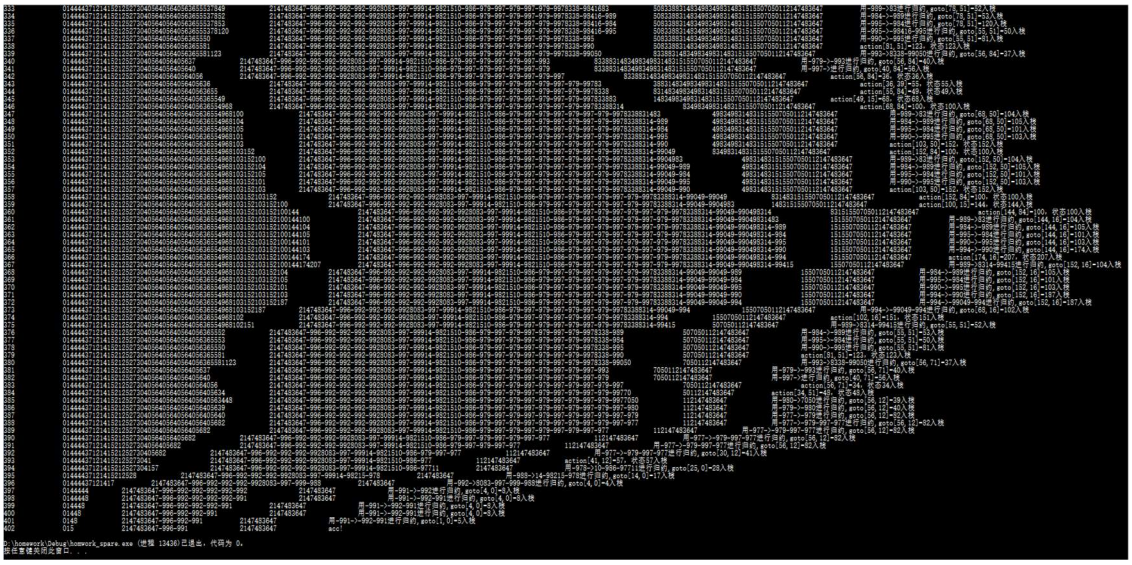
输出与输入的文法对应的 LR(1)分析表，即 ACTION 表与 GOTO 表。如下图所示是输出的文本文件：



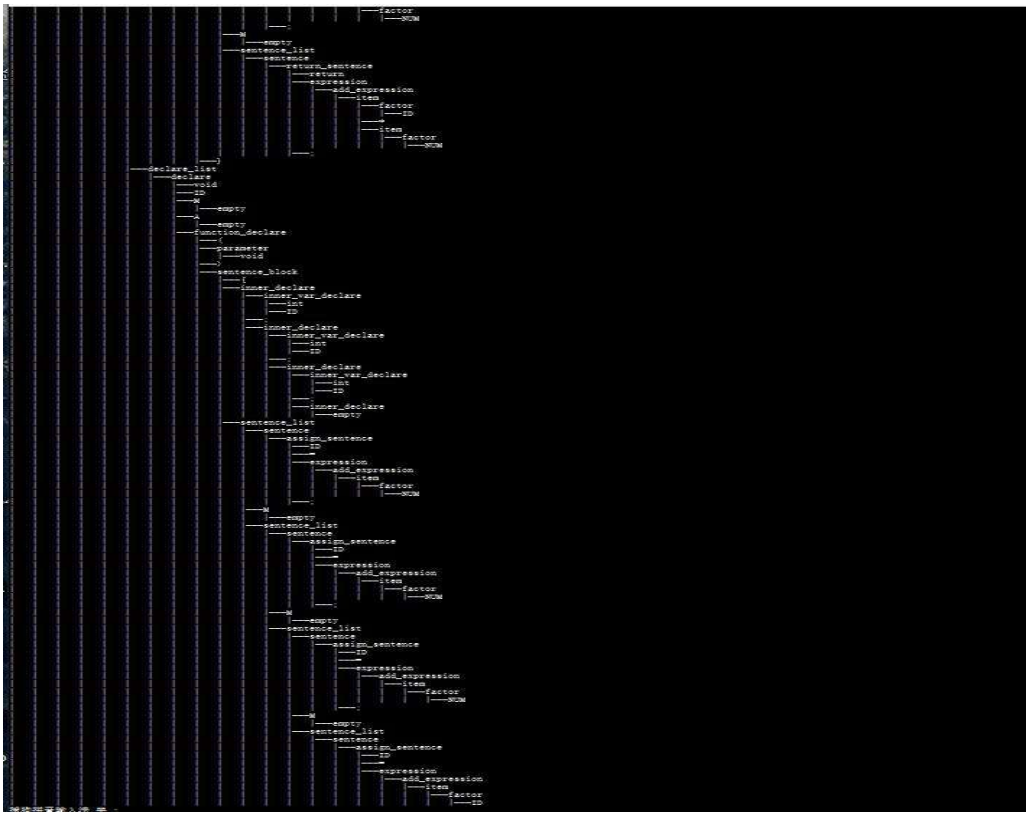
其中第一行是各个标识符（包括终结符和非终结符的标号），此后的每一行对应的是某一个状态对于遇到该终结符应作出的移进或者规约的动作简写，其中‘*’表示空白。

• 如果输入的字符串经过 LR(1)文法分析器分析后被认定为分析失败，则直接输出“不可分析”。如果输入的字符串可以分析，则输出 LR(1)文法分析器对该单词串的分析过程，包括步骤，符号栈，输入串，以及动作。

分析的输出文件如下图所示：



如果输入的字符串经过 LR(1)文法分析器分析后被认定为分析失败，则直接输出“不可分析”。如果输入的字符串可以分析，输出 LR(1)文法分析器对该单词构建的语法分析树。分析输出的树的形状为：

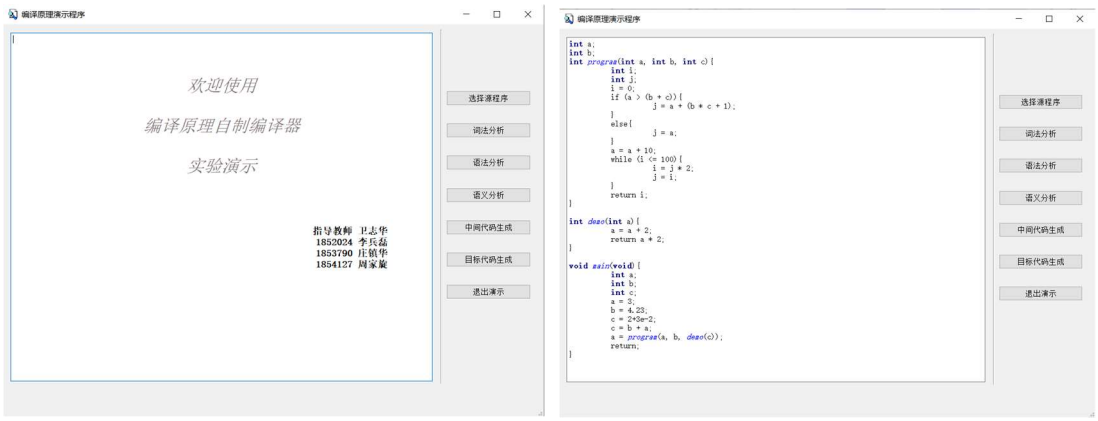


程序功能及用户界面

本程序可以利用词法分析器对符合条件的文法构造相应的 LR(1)语法分析器，其中分析栈用来存放状态，LR(1)分析表是 LR(1)语法分析器的核心部分，并且为后续调用语义分析模块做考虑。构建完该文法的 LR(1)语法分析器之后就可以对输入的单词串进行语法分析，如果分析成功则能够得到相应的语法分析树与具体语法分析过程。

为了使得用户界面更加友好，我们利用 QT5.14.1 来编写了整个的用户界面，下面来逐一的展示该界面的方法

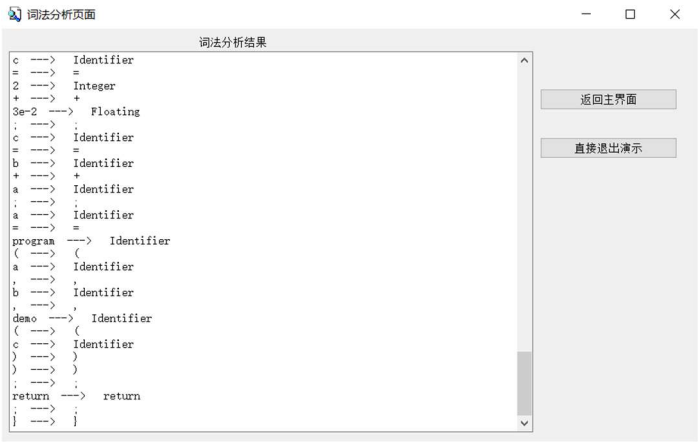
初始展示页面



右侧按钮分别是相对应环节或者功能的按钮，点击选择源程序可以选择要分析的类 C 源程序，选择相应的文件读入到文本框中。

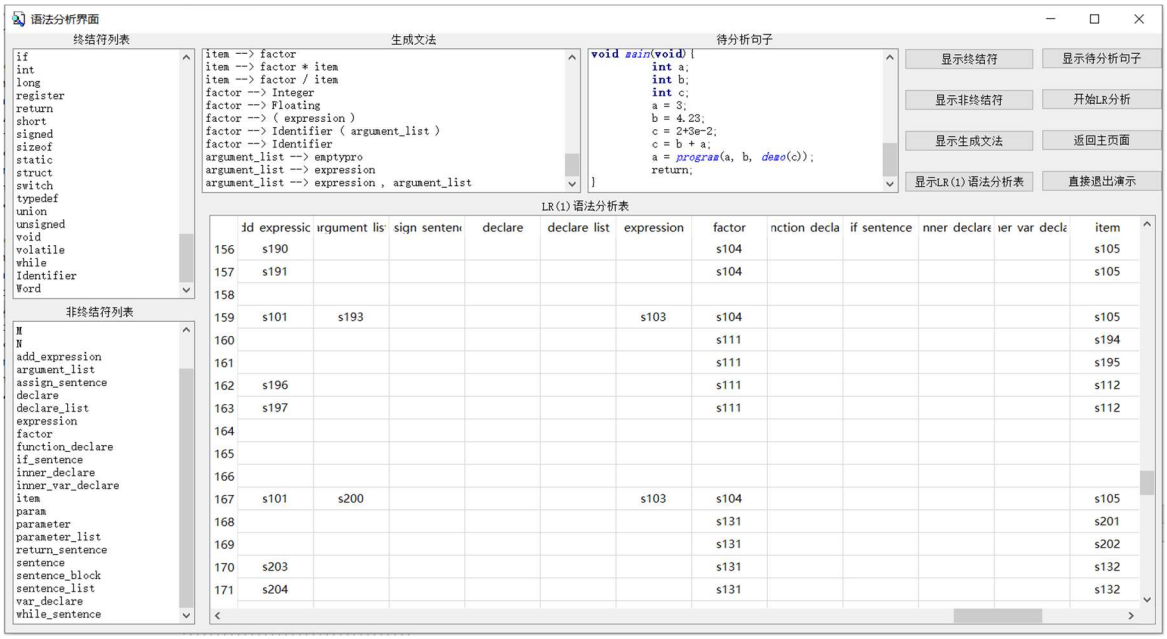
词法分析界面

词法分析界面显示将原文中的每个终结符分析结果显示在左边的文本框中。右边的按钮分别是返回主界面和直接退出演示程序，返回主界面之后可以继续进行语法分析等后续步骤，直接退出演示程序之后结束整个程序，关闭所有窗口，保留所有的分析结果，保留的文件在本目录中的 MyStoreFolder 的文件夹中。



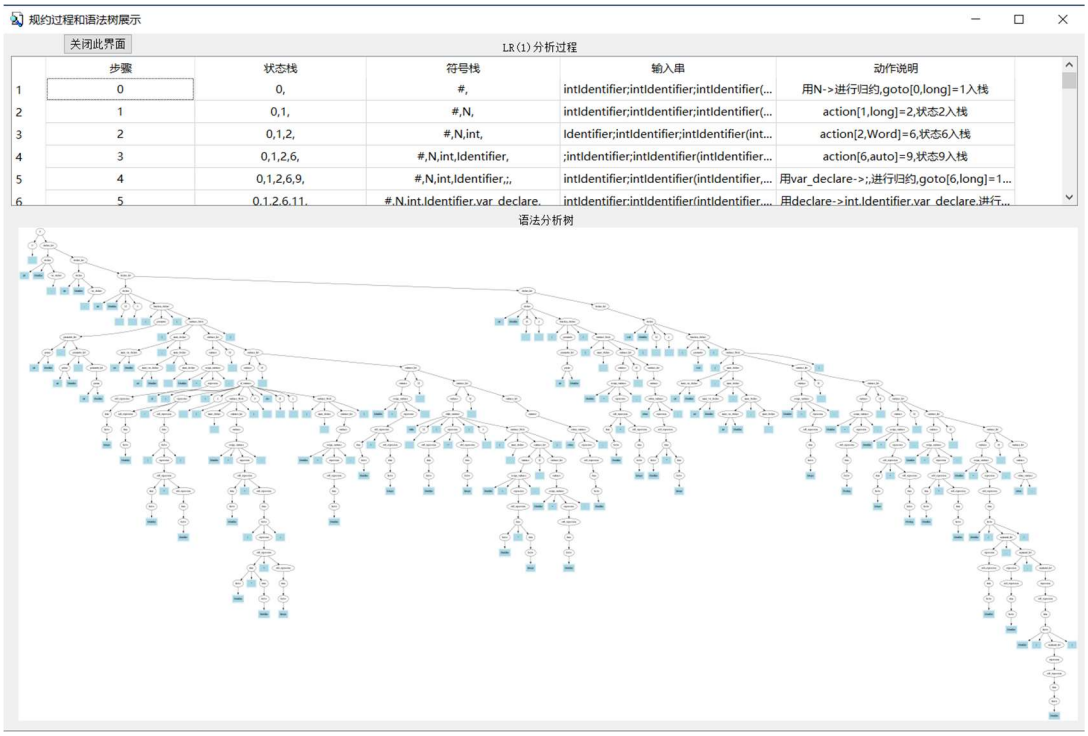
语法分析页面

打开此界面之后本来所有的文本框都是空白的，分别点击“显示终结符”、“显示非终结符”、“显示生成文法”、“显示 LR（1）语法分析表”、“显示待分析句子”的按钮分别在文本框内显示本次语法的部分展示以及待分析的句子和 LR（1）语法分析表。点击这些按钮之后显示的界面如下图所示：



点击开始 LR（1）分析按钮之后，对在主页面上选择的那个类 C 源文件进行分析。如果分析成功的话，在新的窗体上显示规约移进的过程和语法分析树；如果分析失败的话，弹出错误弹窗，给出分析错误的信息。

规约移进过程及语法分析树显示



可以在此页面上的规约移进的表格上下滑动条，查看整个的规约过程。下面显示整个的语法分析树，其中终结符用淡蓝色方框显示，非终结符变元用白色椭圆框显示。

查看完毕之后可以点击左上角“关闭此页面”按钮关闭此页面，返回上个页面。

概要设计

任务分解

词法分析

进行词法分析，将输入的字符序列转换为具有标号信息的终结符序列

语法分析

1. 求出每个产生式、终结符、非终结符的 First 集以及所有非终结符的 Follow 集。
2. 根据类 C 语言的文法，按照本文能接受的文法产生式输入文法，并且生成 LR(1)项集族。
3. 在求得 LR（1）项集族的过程中记录状态的转移信息，生成 Action 表和 Goto 表。
4. 根据生成的 Action 表和 Goto 表进行源文件中代码的语法分析，并输出 LR（1）的分析过程。
5. 在进行 LR（1）分析过程中存储语法分析树的结点信息，输出语法树。

界面显示

运行的结果部分存储在文件中，不便于修改输入文件的文件名，因此采用 Qt5.14.1 和 Qt Creator 来完成图形化界面的显示。整个前端界面将显示后端的运行过程，给与用户直观明了的执行信息。

数据类型的定义

词法分析器

词法分析器的实现头文件中有以下的几个结构体和类的定义，现在用表格

的形式将这些类名还有作用展现出来，并在表格之后附上结构体或类的定义。

标号	类或结构体名	类或结构体说明
1	LexSegment	用于分割源程序文件的分割组成类
2	_Res	存储词法分析结果的基本类
3	Trie	字典树类，便于快速查找终结符
4	LexAnalyzer	词法分析器类，词法分析器的实现函数以及结果存储
5	VT	终结符的枚举类

```
/*  分割组成类  */
class  LexSegment  {
public:
        int  left_index  =  -1;        //原文中的相对偏移位置左端
        int  right_index  =  -1;      //原文中的相对偏移位置右端
        VT  component_type  =  VT::VTNull;  //终结符的种类
public:
        LexSegment();                 //无参构造函数
        LexSegment(int  li,  int  ri,  VT  ctype); //三参数构造函数
};


/*  字典树结果结构体  */
struct  _Res  {
        VT  val;
        int  begin;
        int  end;
};


/*  字典树类  */
class  Trie  {//方便查找关键字和运算符
private:
        bool  isLeaf;       //是否为字典树叶子结点的标志
        Trie*  next[128];   //指向下一个字典树的指针数组
        VT  type;
public:
        Trie();
        void  insert(const  string&  word,  const  VT&  t);      //插入字典树元素
```

```
        vector<_Res>  match(const  string&  word);  //检查关键字或者运算符是否匹配
};


/*  词法分析类  */
class  LexAnalyzer  {
private:
        char*  original_code  =  NULL;    //原始字符串
        int  code_length  =  -1;          //字符串长度
        vector<LexSegment>  inner_result; //词法分析结果
        vector<VT>  component_type;//
        Trie  trie;                       //方便查找
private:
        void  init();                 //初始化，清空内存
        void  preAnalyze();           //分析宏指令、字符、字符串、单行注释、多行注释
        void  analyzeSpliters();      //分析分隔符，空格和回车
        void  analyzeKeywordsAndOperators();//分析关键字和运算符
        void  analyzeWords();                 //分析整数、浮点数、标识符和其他单词
        void  setComponent(int  si,  int  ei,  VT  com); //储存分析结果
public:
        LexAnalyzer();                //构造函数，初始化字典树
        void  Analyze(const  char*  code); //词法分析的总体过程
        void  Analyze(const  string&  code); // 构造函数，初始化字典树
        void  GetResult(vector<LexSegment>&  result); //传出结果
};
```

语法分析器

语法分析器的实现头文件中有以下的几个结构体和类的定义，现在用表格的形式将这些类名还有作用展现出来，并在表格之后附上结构体或类的定义。

标号	类或结构体名	类或结构体说明
1	Item	用于记录项目集的基本项目结构体
2	SyntaxAnalyzer	语法分析器类，内有成员函数以及实现过程
3	Trie	字典树类，便于快速查找终结符
4	LexAnalyzer	词法分析器类，词法分析器的实现函数以及结果存储
5	VT	终结符的枚举类

```
struct Item              //项目结构体
{
        int nump;          //产生式编号
        int ppos;          //.的位置
        string forward;
     //项目的向前搜索符集，比如 S -> .BB,    a/b 。则 forward=“ab”.
};


class SyntaxAnalyzer {
private:
    vector<pair<int, string>> VTs;
    //记录终结符的向量，其中元素为 pair<int,string>,第 1 个元素为该终结符的标号，第 2 个元素为该终结符的实际字符，例如，pair<52,"break">
    vector<pair<int, string>> VNs;
    //记录中间变元的向量，其中元素为 pair<int,string>,first:该变元的标号，second:该变元的实际字符，例如，pair<-200,"VN_Program">
    vector<vector<int>> PATs;
    //记录生成规则的向量,其中元素为 int,first:该变元的标号，second:该变元推出的表达式，例如，[ -200,-100,45,12],代表 -200 -> -100,45,12
    void InitVTs();
    void InitVNs();
    void InitPATs();
    vector<int> InitPATs_SingleLine(const char* p);
    //分析某一行内的数据信息，可能要增加多个生成式

public:
    int table[MAX_N][MAX_N];                //预测分析表 -1
    int tb_s_r[MAX_N][MAX_N];               //是移进项还是规约项,-1,-2.
    int numvt;                                          //终结符的数目
    int num;
    string srrc;

    vector<vector<struct Item>> itemSet;//项目集族
    int edge[MAX_N][3];
    int head[MAX_N];    //第 i 个项目集的头
    int nume = 0;    //边数
    vector<int> word;

    map<int, int> getNum;
    map<int, int> getString;
    vector<vector<int>> production;
```

```
        string  getProduce[MAX_N];
        string  first[MAX_N];
    public:
        SyntaxAnalyzer();
        ~SyntaxAnalyzer();
        void  VectorClear();
        void  PrintVectorVTs();
        void  PrintVectorVNs();
        void  PrintVectorPATS();
        void  LR1_Analyze_file(ofstream&  out);

        void  LR_init();   //  用 init()  初始化程序中一些变量。
        void  LR_inputGrammar(string  filePath); //使用 inputGrammar 函数，将输入文
件 input.txt 中的终结符，非终结符以及产生式存到相应的数据结构中。
        void  LR_getProduction();   //合并左式相同的产生式
        void  LR_dfsGetFirst(const  vector<vector<int>>&  production,
            map<int,  int>&  getNum,  string  getProduce[MAX_N],
            string  first[MAX_N],  int  nv,  int  nump,  bool  getfirst[MAX_N]);
        void  LR_getFirst();         //计算每个文法符号的 FIRST 集
        void  LR_change_first();  //FIRST 集去重，比如我们计算得到的
                        //FIRST(B)={a,b,c,a}，就需要清除掉一个 a.
        void  LR_change_production(vector<vector<int>>&  production);
        void  LR_getItemSet();
        bool  LR_getLR1Table();
        void  LR_addegde(int  from,  int  to,  int  w);
        bool  LR_totalControl();    //根据生成的 LR(1)分析表对测试程序进行分析。移
进或归约或接受或报错。
        void  LR_printCurState(int  count,  stack<int>  state,  stack<int>  wd,
        int  i,  map<int,  int>&  getString);
    //下面是一些用于记录文法分析结果的备份函数，再一次进行分析可直接从备份中获得，
不必再从头分析获得 LR（1）移进规约表格。
        void  LRProductionBackup(const  char*  filename,  vector<vector<int>>pro);
        void  LRProductionReturn(const  char*  filename,vector<vector<int>>&  pro);
        void  LRTableBackup1(const  char*fname,const  int  row,const  int  col);
        void  LRTableBackup2(const  char*fname,const  int  row,const  int  col);
        void  LRTableReturn1(const  char*  fname);
        void  LRTableReturn2(const  char*  fname);
    };
```

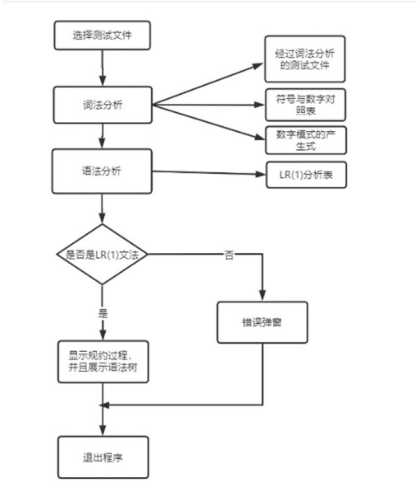
主程序流程及模块调用关系

整个程序主要可以分为三个部分：词法分析相关模块、产生式相关模块和语法分析相关模块。

主程序流程

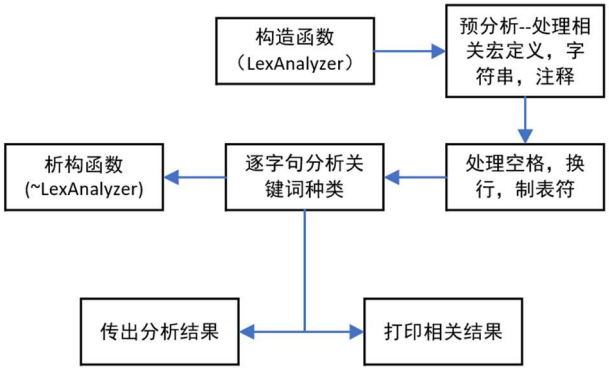
整个程序的整体流程包括源程序输入、词法分析、语法分析、结果展示等几个部分，

其基本流程如下图所示：



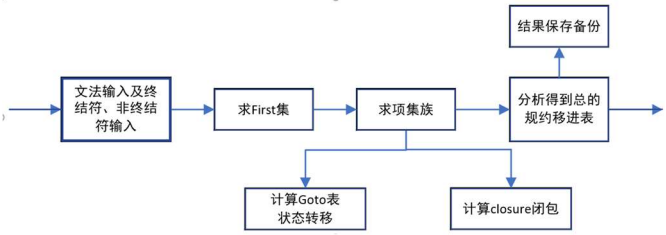
词法分析模块

词法分析相关模块完成词法分析并输出标识符流。



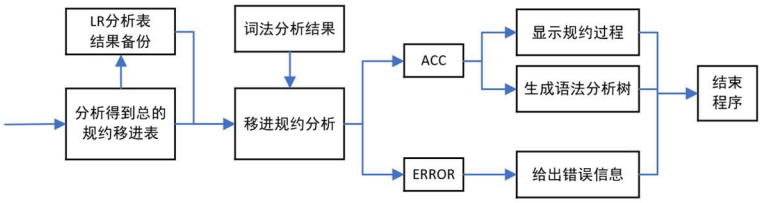
文法处理模块

产生式相关模块完成产生式的读入和分析并给出 First 集。



语法分析模块

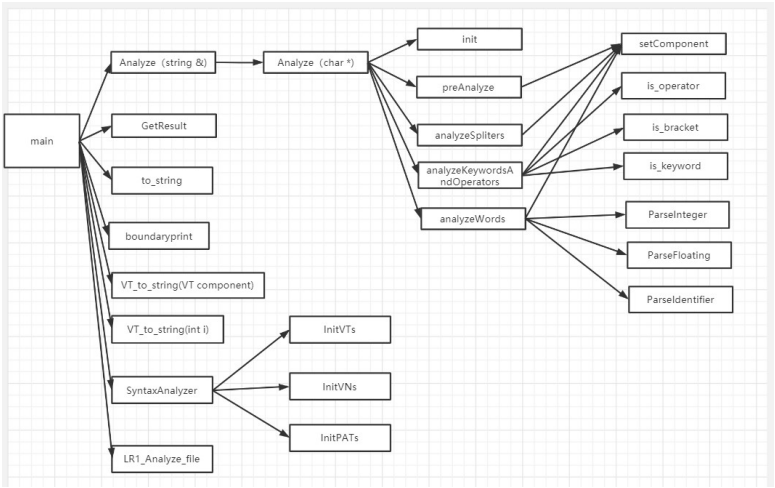
语法分析相关模块完成语法分析并给出语法分析过程和语法分析树。



详细设计

词法分析

函数调用图



重点函数

词法分析总体过程

词法分析分为 4 步，首先分析宏指令、字符、字符串、单行注释、多行注释，然后在上一步基础上分析分割符（包含空格和回车），然后分析关键字与运算符，最后分析整数、浮点数、标识符和其他单词。

```
    /* 词法分析总体过程 */
    void LexAnalyzer::Analyze(const char* code) {
        init();
        original_code = new char[code_length = strlen(code) + 1];
        memcpy(original_code, code, code_length);
        component_type.clear();
        component_type.resize(strlen(code) + 1,
LexComponent::LexComponentNull);
        preAnalyze();     //分析宏指令、字符、字符串、单行注释、多行注释
        analyzeSpliters();//分析分割符（空格和回车）
        analyzeKeywordsAndOperators();//分析关键字与运算符
        analyzeWords();              //分析整数、浮点数、标识符与其他单词
    }
```

分析宏指令、字符、字符串、单行注释、多行注释

这是词法分析的第一步，首先从源程序中去除一些无用的注释以及需要替换的宏指令和一些具有干扰性的字符、字符串，其实现主要由几个状态判断 bool 变量来完成，如当 is_single_line_comment 变量为 true 的时候就代表当前字符是单行注释中的字符。

```
    /* 分析宏指令、字符、字符串、单行注释、多行注释 */
    void LexAnalyzer::preAnalyze() {
        bool is_single_line_comment = false;//单行注释
        bool is_multiline_comment = false;  //多行注释
        bool is_string = false;             //字符串
        bool is_char = false;               //字符
        bool is_macro = false;              //宏指令
```

```
        int start_index = 0;
        for (int i = 0; ; ++i) {
            char cch = original_code[i];
            if ((is_char || is_string) && cch == '\\') {//转义字符处理
                i += 2;
                cch = original_code[i];
            }
            if (is_macro) {                 //宏指令
                if (!cch || cch == '\n') {
                    setComponent(start_index, i,
LexComponent::LexComponentMacro);
                    is_macro = false;
                }
            }
            else if (is_single_line_comment) {//单行注释
                if (!cch || cch == '\n') {
                    setComponent(start_index, i,
LexComponent::LexComponentComment);
                    is_single_line_comment = false;
                }
            }
            else if (is_multiline_comment) {  //多行注释
                if (cch == '/' && original_code[i - 1] == '*') {
                    setComponent(start_index, i + 1,
LexComponent::LexComponentComment);
                    is_multiline_comment = false;
                }
            }
            else if (is_string) {            //字符串处理
                if (cch == '\"') {
                    setComponent(start_index, i + 1,
LexComponent::LexComponentString);
                    is_string = false;
```

```
            }
        }
        else if (is_char) {                //字符处理
            if (cch == '\'') {
                setComponent(start_index, i + 1,
LexComponent::LexComponentCharacter);
                is_char = false;
            }
        }
        else if (cch == '*' && i != 0 && original_code[i - 1] ==
'/') {
            is_multiline_comment = true;
            start_index = i - 1;
        }
        else if (cch == '\"') {
            is_string = true;
            start_index = i;
        }
        else if (cch == '\'') {
            is_char = true;
            start_index = i;
        }
        else if (cch == '/' && i != 0 && original_code[i - 1] ==
'/') {
            is_single_line_comment = true;
            start_index = i - 1;
        }
        else if (cch == '#') {
            is_macro = true;
            start_index = i;
        if (!cch) return;
      }
   }
```

分析分割符（空格和回车）

这一部分较为简单，主要功能是将源程序中的分割符统一，便于之后其他关键字、标识符等词语的分析。

```
    /* 分析分割符（空格和回车） */
    void LexAnalyzer::analyzeSpliters() {   //分割符
        int start_index = 0;
        bool is_spliter = false;
        for (int i = 0;; ++i) {
            if (component_type[i] && !is_spliter)
                continue;
            char cch = original_code[i];
            if (cch == ' ' || cch == '\t') {
                if (!is_spliter) {
                    is_spliter = true;
                    start_index = i;
                }
            }
            else {
                if (is_spliter) {
                    setComponent(start_index, i,
LexComponent::LexComponentWhiteSpace);
                    is_spliter = false;
                }
            }
            if (cch == '\n')
                setComponent(i, i + 1,
LexComponent::LexComponentEndLine);
            else if (!cch)
                break;
    }
```

分析关键字与运算符

主要采用字典树匹配关键字与运算符，这样可以避免回退，降低时间复杂

度，字典树匹配的具体实现见实现亮点这一小节。

```
    /* 分析关键字与运算符 */
    void LexAnalyzer::analyzeKeywordsAndOperators() {
        string tmp_code = original_code;
        for (int i = 0; i < code_length; ++i)
            if (component_type[i])
                tmp_code[i] = ' ';
        //字典树匹配关键字与运算符，时间复杂度降低
        //返回匹配得到的单词位置与类别
        auto res = trie.match(tmp_code);
        for (auto& seg : res)
            if (is_operator(seg.val))//运算符
                setComponent(seg.begin, seg.end, seg.val);
        for (auto& seg : res)
            if (is_bracket(seg.val)) {//括号
                setComponent(seg.begin, seg.end, seg.val);
            }
        for (auto& seg : res)
            if (is_keyword(seg.val))//关键字
                if ((seg.begin == 0 || component_type[seg.begin - 1])
&& (seg.end == code_length || component_type[seg.end]))
                    setComponent(seg.begin, seg.end, seg.val);
    }
```

分析整数、浮点数、标识符与其他单词

这一部分主要通过调用识别整数、浮点数、标识符的 DFA 子函数来实现，具体实现见实现亮点这一小节。

```
    /* 分析整数、浮点数、标识符与其他单词 */
    void LexAnalyzer::analyzeWords() {
        int start_index = -1;
        string ori = original_code;
        for (int i = 0; ; ++i) {
            if (start_index < 0 && (original_code[i]
```

```
&& !component_type[i]))
                start_index = i;//词语的开始位置
            else if (start_index >= 0 && (!original_code[i] ||
component_type[i])) {
                string code = ori.substr(start_index, i -
start_index) + ' ';
                if (code == "true " || code == "false ")//bool 型
                    setComponent(start_index, i,
LexComponent::LexComponentBoolean);
                else if (ParseInteger(code))//无符号整数
                    setComponent(start_index, i,
LexComponent::LexComponentInteger);
                else if (ParseFloating(code))//浮点数
                    setComponent(start_index, i,
LexComponent::LexComponentFloating);
                else if (ParseIdentifier(code))//标识符
                    setComponent(start_index, i,
LexComponent::LexComponentIdentifier);
                else //其他单词
                    setComponent(start_index, i,
LexComponent::LexComponentWord);
                start_index = -1;
            }
            if (!original_code[i])
                break;
        }
    }
```

实现亮点

字典树的使用

Trie 树，即字典树，也有的称为前缀树，是一种树形结构。广泛应用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是最大限度地减少无谓的字符串比较，查询效率比较高。Trie 的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

算法特征：

- 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
- 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
- 每个节点的所有子节点包含的字符都不相同。
- 它的 key 都为字符串，能做到高效查询和插入，时间复杂度为 O(k)，k 为字符串长度，缺点是如果大量字符串没有共同前缀时很耗内存。

分析关键字与运算符过程中使用了字典树算法，避免了字符串匹配的回退，使得关键字匹配的效率大大提高。

```
/* 字典树类 */
class Trie {//方便查找关键字和运算符
private:
    bool isLeaf;//是否是叶子节点
    Trie* next[128];//子节点数组
    LexComponent type;//关键字、运算符类型
public:
    Trie() {//构造函数
        isLeaf = false;
        memset(next, NULL, sizeof(next));
    }

    void insert(const string& word, const LexComponent& t) {//插
```

入函数

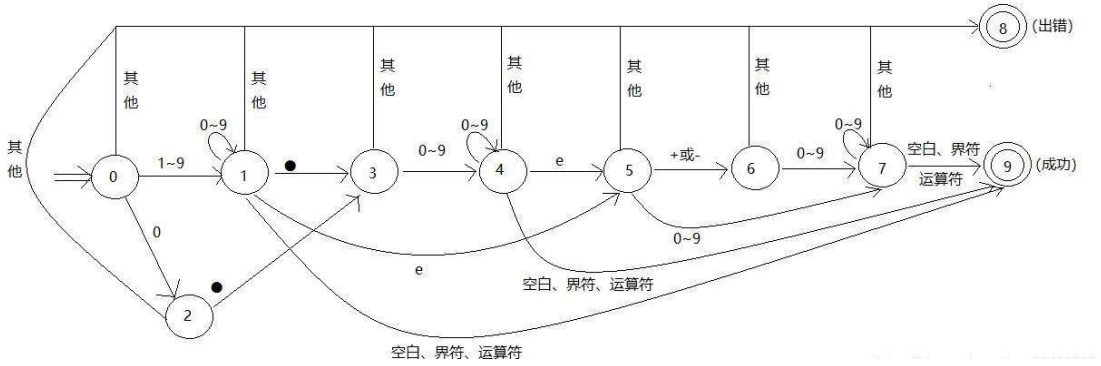
```
            Trie* node = this;
            for (char c : word) {
                if (node->next[c - '\0'] == NULL)
                    node->next[c - '\0'] = new Trie();
                node = node->next[c - '\0'];
            }
            node->isLeaf = true;
            node->type = t;
        }

        vector<_Res> match(const string& word) {//匹配函数
            Trie* node = this,* parent = this;
            int len = word.length();
            vector<_Res> res;//结果容器
            for (int start = 0, i = 0; i < len;) {
                node = this;
                for (i = start; i < len;i++) {
                    const char& c = word[i];
                    parent = node;
                    node = node->next[c - '\0'];
                    if (node == NULL) {
                        if (parent->isLeaf) {//防止过早匹配
                            if ((parent->type == LexComponentPlus ||
parent->type == LexComponentMinus)
                                && (i >= 2 && (word[i - 2] == 'e' ||
word[i - 2] == 'E'))) {//防止匹配到浮点数中的-、+
                                start = start + 1;
                                break;
                            }
                            res.emplace_back(_Res{ parent->type, start,
i});
                            start = i;
```

```
                    }
                    else
                        start = start + 1;
                    break;
                }
            }
        }
        return res;
    }
};
```

浮点数的识别

额外实现了浮点数的识别，可以识别 1.2 普通形式和 1e-3 科学计数形式的浮点数。DFA 转移图如下。



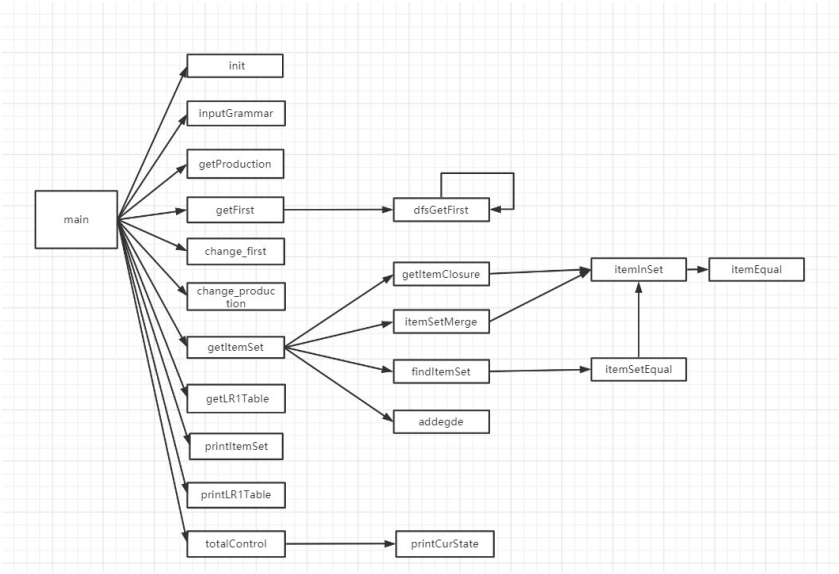

```
/* 无符号浮点数判断过程 */
bool ParseFloating(string& str)
{
    char state = '0';//初始状态
    int i = 0, n = str.length();
    char ch = str[i];
    while (state != '8' && state != '9' && i < n) {
        ch = str[i++];
        switch (state) {
        case '0':
            if ((ch >= '1') && (ch <= '9'))
```

```
                state = '1';//1-9，转状态 1
            else if (ch == '0')
                state = '2';//0，转状态 2
            else
                state = '8';//其他，转状态 8
            break;
        case '1':
            if ((ch >= '0') && (ch <= '9'))
                state = '1';//0-9，转状态 1
            else if (ch == 'e' || ch == 'E')
                state = '5';//e ，转状态 5
            else if (ch == '.')
                state = '3';//. ，转状态 3
            else if (ch == ' ')
                state = '9';//结束，转 9
            else
                state = '8';//其他，转状态 8
            break;
        case '2':
            if (ch == '.')
                state = '3';//. ，转状态 3
            else
                state = '8';//其他，转状态 8
            break;
        case '3':
            if ((ch >= '0') && (ch <= '9'))
                state = '4';//0-9，转状态 4
            else
                state = '8';//其他，转状态 8
            break;
        case '4':
            if ((ch >= '0') && (ch <= '9'))
                state = '4';//0-9，转状态 4
```

```
            else if (ch == 'e' || ch == 'E')
                state = '5';//e，转状态 5
            else if (ch == ' ')
                state = '9';//空白，转 9
            else
                state = '8';//其他，转状态 8
            break;
        case '5':
            if ((ch >= '0') && (ch <= '9'))
                state = '7';//0-9，转状态 7
            else if (ch == '+' || ch == '-')
                state = '6';//+/-，转状态 6
            else
                state = '8';//其他，转状态 8
            break;
        case '6':
            if ((ch >= '0') && (ch <= '9'))
                state = '7';//0-9，转状态 7
            else
                state = '8';//其他，转状态 8
            break;
        case '7':
            if ((ch >= '0') && (ch <= '9'))
                state = '7';//0-9，转状态 7
            else if (ch == ' ')
                state = '9';//空白，转 9
            else
                state = '8';//其他，转状态 8
            break;
        }
    }
    return state == '9';
}
```

语法分析

函数调用图



重点函数

语法分析总程序

语法分析的主要步骤有：首先规范输入的文法，即合并左侧相同的产生式，然后计算得到所有非终结符的 first 集合，接着根据产生式和非终结符的 first 集得到项目集族，然后根据项目集族得到 ACTION 表和 GOTO 表，最后根据这两个表通过总控程序分析句子。

```
/* 语法分析总程序 */
void Analyze(){
    //初始化
    init();
    //输入终结符、非终结符、产生式
    inputGrammar("./input.txt", getNum, getString, production);
    //合并左侧相同的产生式
    getProduction(production, getNum, getProduce);
    //得到 first 集
```

```
        getFirst(production, getNum, getProduce, first);
        //非终结符去重
        change_first(first);
        //特殊处理空产生式
        change_production(production);
        //得到项目集族
        getItemSet(getNum, getProduce, production, first);
        //得到 LR1 分析表（含 ACTION 表和 GOTO 表）
        if (!getLR1Table(production)) {
            cout << "此文法在生成分析表时候有多重入口，非 LR(1)文法！" <<
endl;
            return 0;
        }
        //...输入句子
        word.emplace_back(INT_MAX);
        //利用总控程序分析句子
        if (!totalControl(getString, getNum, production))//总控程序无
法识别此句子
            cout << "error!" << endl;
    }
```

得到 FIRST 集

GetFirst 算法原理：

使用如下规则，直至每一个非终结符的 FIRST 集合不在增大为止：

1）若 X 属于 VT，则 FIRST（X）= {X}

2）若 X 属于 VN，且有产生式 X -> a…，则把 a 加入 FIRST；若 S -> ε也是产生式，则把 ε也加入到 FIRST（X）中

3）若 X -> Y……是一个产生式且 Y 属于 VN，则把 FIRST（Y）中所有的非空元素都加入到 FIRST（X）中，若 X -> Y1Y2…Yk 是一个产生式，Y1,Y2,…Yi-1 都是非终结符，而且对于任何满足 1 <= j <= i - 1，first（Yj）都含有ε，则把 FIRST（Yi）中所有的非ε元素都加入到 FIRST（X）中，特别的，若所

有 FIRST（Yj）对于 j = 1，2，…，k 都含有 ε，则将 ε也加入到 FIRST(X)中。

```
    /* 求项目的 FIRST 集子函数 */
    void dfsGetFirst(const vector<vector<int>>& production,
map<int, int>& getNum, string getProduce[MAX_N],
        string first[MAX_N], int nv, int nump, bool getfirst[MAX_N])
//当前的符号，和对应产生式编号
    {
        int temp = getNum[production[nump][1]];  //产生式推出来的首符
        getfirst[nump] = true;                    //标记
        if (temp <= numvt)                       //是终结符，直接加入返回
            first[nv] += char('\0' + temp);
        else {                                   //是非终结符
            for (int j = 1; j < production[nump].size(); j++) {
                temp = getNum[production[nump][j]];
                //所有 temp 可以推出来的符号对应的产生式
                for (int i = 0; i < getProduce[temp].size(); i++) {
                    //左递归的产生式不用 不影响求 fisrt 集
                    if (production[nump][0] == production[nump][1])
                        continue;
                    dfsGetFirst(production, getNum, getProduce, first,
temp,
                        getProduce[temp][i] - '\0', getfirst);
                }
                int pos = first[temp].find('\0' + numvt);
                if (pos == -1) {//没有空字符
                    first[nv] += first[temp];
                    break;
                }
                first[nv] += first[temp].substr(0, pos);
                first[nv] += first[temp].substr(pos + 1);
                if (j == production[nump].size() - 1)
                    first[nv] += '\0' + numvt;//加上空字符
```

```
            }
        }
    }
    /* 求项目的 FIRST 集 */
    void getFirst(const vector<vector<int>>& production, map<int,
int>& getNum, string getProduce[MAX_N], string first[MAX_N])
    {
        bool getfirst[MAX_N];
        memset(getfirst, 0, sizeof(getfirst));//初始化
        //终结符 first 集合是它自己。 first 集合通过终结符的编号来存储
        for (int i = 1; i <= numvt; i++)
            first[i] = char('\0' + i);
        for (int i = 0; i < production.size(); i++) {
            //左递归的产生式不用 不影响求 fisrt 集
            if (production[i][0] == production[i][1])
                continue;
            //已经生成
            if (getfirst[i])
                continue;
            int temp = getNum[production[i][0]];
            dfsGetFirst(production, getNum, getProduce, first, temp,
i, getfirst);
        }
    }
```

构造项目集闭包

假定 I 是一个项目集，它的闭包 CLOSURE（I）可按如下方式构造：

1） I 的任何项目都属于 CLOSURE（I）

2） 若项目[A -> $\alpha.B\beta,a$] 属于 CLOSURE（I），B -> γ是一个产生式，那么，对于 FIRST（βa）中的每个终结符 b，如果[B -> $.\gamma,b$]原来不在 CLOSURE（I）中，则把它加进去；

3） 重复执行步骤 2）直至 CLOSURE（I）不再增大为止。

在实际代码编写中，我们主要采用广度优先搜索算法求解项目的闭包，在

确定项目的展望字符时，需要用到之前计算的 FIRST 集。

```
    /* 利用 BFS 算法求解项目集闭包 */
    vector<Item> getItemClosure(Item t, map<int, int>& getNum,
string getProduce[MAX_N], vector<vector<int>>& production, string
first[MAX_N])
    {
        vector<Item> temp;//项目集
        temp.push_back(t);//项目集初始项目
        queue<Item> q;
        q.push(t);
        while (!q.empty()) {
            Item cur = q.front();
            q.pop();
            //归约项舍去
            if (cur.ppos == production[cur.nump].size())
                continue;
            //tt 是'.'之后的符号
            int tt = getNum[production[cur.nump][cur.ppos]];
            //若是终结符，不必寻找
            if (tt <= numvt)
                continue;
            //若是非终结符，对应产生式的编号
            for (int i = 0; i < getProduce[tt].size(); i++) {
                Item c;
                c.ppos = 1;
                c.nump = getProduce[tt][i] - '\0';
                //这种是[S->a.B, c]的情况
                if (production[cur.nump].size() - cur.ppos == 1)
                    c.forward += cur.forward;
                //这种是[S->a.BC...d, c]的情况
                else {
                    //以下是求解 FIRST（C...d）的过程
                    for (int j = 1; j < production[cur.nump].size();
```

```
j++)
                    //...
            }
            //项目去重
            if (!itemInSet(c, temp)) {
                q.push(c);
                temp.push_back(c);
            }
        }
    }
    return temp;
}
```

构造项目集族

构造有效的 LR（1）项目集族的办法本质上和构造 LR（0）项目集规范族的办法是一样的。都需要两个函数 CLOSURE 和 GO，上一小节已经构造出 CLOSURE 函数，因此还需要构造 GO 函数，定义如下：

令 I 是一个项目集，X 是一个文法符号，函数 GO（I，X）定义为 GO（I，X）= CLOSURE（J），其中 J = {任何形如[A -> $\alpha B.\beta,a$]的项目 | [A -> $\alpha.B\beta,a$] ∈ I}。

关于文法 G 的 LR（1）项目集族的构造算法是：

BEGIN：
　　$C := \{Closure([S \to \bullet S, \#])\}$;
　　REPEAT：
　　　　FOR C中的每一个项目集I和G的每个符号X
　　　　　　IF $Go(I,X)$非空且不属于C，*THEN*
　　　　　　把 $Go(I,X)$加入C中
　　UNTIL C不再增大
END

具体实现代码时，我们仍采用 BFS 算法求解项目集族，实际上最后构造的项目集转移图也是一个 DFA，我们采用记录图的方式来记录这个 DFA。

```
    /* 获得项目集族 */
    void getItemSet(map<int, int>& getNum, string
getProduce[MAX_N], vector<vector<int>>& production, string
first[MAX_N])
    {
        vector<Item> temp;
        //初始的项目：S->...,#
        Item t;
        //...
        //初始的项目集
        temp = getItemClosure(t, getNum, getProduce, production,
first);

        //队列初始化
        queue<vector<Item>> q;
        q.push(temp);
        itemSet.push_back(temp);

        while (!q.empty()) {
            //取出队首的项目集
            vector<Item> cur = q.front();
            q.pop();
            //遍历所有符号
            for (int i = 1; i <= num; i++) {
                //空字符无法作为转移终结符
                if (i == numvt)
                    continue;
                vector<Item> temp;
                //遍历该项目集中的所有项目，看看是否可以转移
                for (int j = 0; j < cur.size(); j++) {
                    //是规约项目，无法再读入
                    if (cur[j].ppos == production[cur[j].nump].size())
                        continue;
```

```
                    int tt =
getNum[production[cur[j].nump][cur[j].ppos]];
                    //如果这个符号和最外层循环的转移终结符相同
                    if (tt == i) {
                        Item tempt;
                        tempt.forward = cur[j].forward;
                        tempt.ppos = cur[j].ppos + 1;
                        tempt.nump = cur[j].nump;
                        //合并项目集，添加到共同的新项目集
                        temp = itemSetMerge(temp, getItemClosure(tempt,
getNum,
                                getProduce, production, first));
                    }
                }
                //该符号无法读入
                if (temp.size() == 0)
                    continue;

                //记录当前项目集和新构造项目集的出入关系
                //...
            }
        }
    }
```

获得 LR（1）分析表

从文法的 LR（1）项目集族 C 构造分析表的算法如下：假定 C = {I0，I1，…，In}，令每一个 Ik 的下标 k 为分析表的状态。令那个含有[S' -> .S, #]的 Ik 的 k 作为分析器的初态。动作 ACTION 和状态转换 GOTO 可构造如下：

1） 若项目[A -> $\alpha.a\beta,b$]属于 Ik 且 GOTO（Ik, a）= Ij，a 为终结符，则置 ACTION[k, a]为“把状态 j 和符号 a 移进栈”，简记为“aj”。

2） 若项目[A -> α., a]属于 Ik，则置 ACTION[k, a]为“用产生式 A -> α”规约”，简记为“rj”；其中假定 A -> α为文法 G’的第 j 个产生式。

3） 若项目[S' -> S., #]属于 Ik，则置 ACTION[k, #]为“接受”，简记为“acc”

4） 若 GO（Ik, a）= Ij，则置 GOTO[k, A] = j。

这里我们采用两个全局数组实现 ACTION 表和 GOTO 表的表示，

```
int table[MAX_N][MAX_N];      //表示移进的符号编号、规约的产生式编号
int tb_s_r[MAX_N][MAX_N];     //区分是移进项还是规约项, -1, -2
```

```
    /* 获得 LR1 分析表 table[i][j] = w:状态 i --> j,读入符号 W */
    bool getLR1Table(vector<vector<int>>& production)
    {
        //遍历图
    for (int i = 0; i < itemSet.size(); i++)
    for (int j = head[i]; j != -1; j = edge[j][1]) {
        //多重入口，报错.
        if (table[i][edge[j][2]] != -1)
            return 0;
        //移进的符号
        table[i][edge[j][2]] = edge[j][0];
        //移近项-1
        tb_s_r[i][edge[j][2]] = -1;
    }
    //遍历所有项目
    for (int i = 0; i < itemSet.size(); i++)
    for (int j = 0; j < itemSet[i].size(); j++)
        //归约项
        if (itemSet[i][j].ppos ==
production[itemSet[i][j].nump].size()) {
            for (int k = 0; k < itemSet[i][j].forward.size(); k++) {
                //多重入口，报错.
```

```
                if (table[i][(itemSet[i][j].forward)[k] - '\0'] != -
1)
                    return 0;
                //接受态
                if ((itemSet[i][j].forward)[k] == '\0'
                    && itemSet[i][j].nump == 0)
                    table[i][(itemSet[i][j].forward)[k] - '\0'] = -3;
                //归约态
                else {
                    //规约的产生式编号
            table[i][(itemSet[i][j].forward)[k] - '\0'] =
                    itemSet[i][j].nump;
                    //规约项-2
                 tb_s_r[i][(itemSet[i][j].forward)[k] - '\0'] = -2;
                }
            }
    }
    return true;
    }
```

实现亮点

空产生式的处理

如果输入的产生式里面包含空产生式的话，对于求 FIRST 集和求项目集来说都是一个头疼的问题，我们采用的处理方法是，求 FIRST 集时ε显式表示，即求得的 FIRST 集可能包含ε，而在求项目集族的时候ε隐式表示，即空产生式从一开始就是一个规约项目，应该直接表示成$S \rightarrow .$而不是$S \rightarrow .\varepsilon$，（这一点可以由一道例题来证明，见下图）这样的话项目集族之间的转移符号就不可能是ε，而这恰恰是合理的，因为最后根据算法生成项目之间的转移关系图是一个 DFA，不可能有空转移的存在。最后经过实际测试，这种处理方法也是完全可行的。

11. 设文法G[S]为：S→AS|ε　　A→aA|b
(1) 证明G[S]是LR(1)文法
(2) 构造出它的LR(1)分析表
(3) 给出输入符号串abab#的分析过程

一个文法不是SLR(1)时，不能证明它是LR(1)的

解：将文法改写为拓广文法：

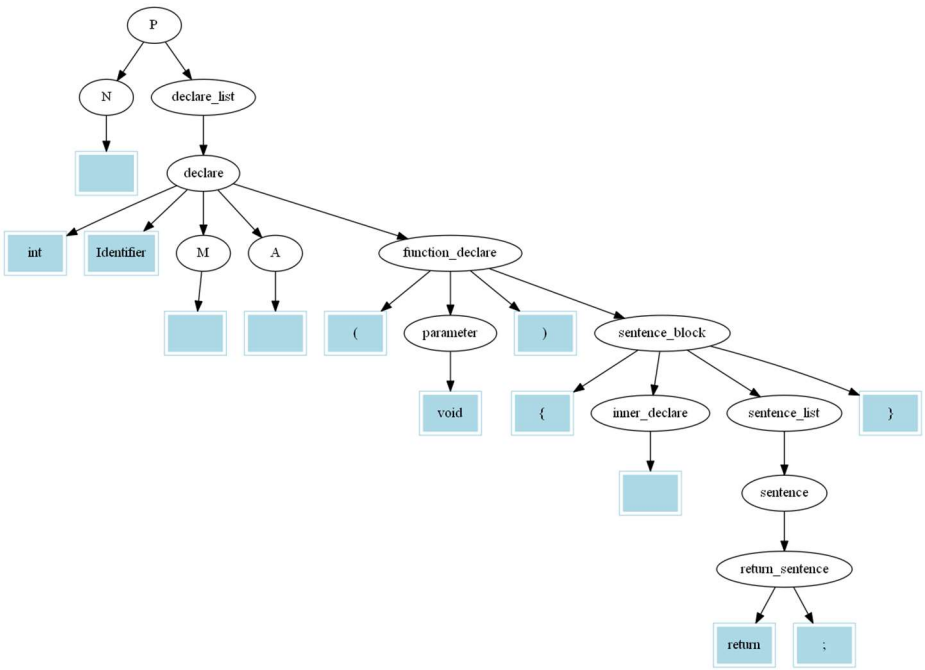
(0) S'→S　(1) S→AS　(2) S→ε　(3) A→aA　(4) A→b

构造其LR(1)项目集规范族：

状态	核集合	项目集（核集合 + 闭包增加项目）
I0	S′ →•S,#	S→•S，# S→•AS，# S→•，# A→•aA，a/b/# A→•b，a/b/#
I1	S→S•,#	S→S•,#

语法树的绘制

语法树绘制的难点在于语法树的构造，因为LR（1）分析是自下而上的分析，而由下而上的构造一棵树是比较困难的，我们的解决办法是在总控程序进行规约的时候，把规约用到的产生式存到一个栈里面，规约成功后再逐个取出，从根节点S开始构造语法树。



因为 LR（1）是最左规约的，对应着最右推导，所以每次只需要取最右边的非终结符节点进行推导展开，将该非终结符节点作为根节点构造子树，如此完成最终语法树的构造。

最终绘制语法树，我们使用 dot 语言和 graphviz2.38.0 软件生成了比较美观的语法树图片，如下图所示。

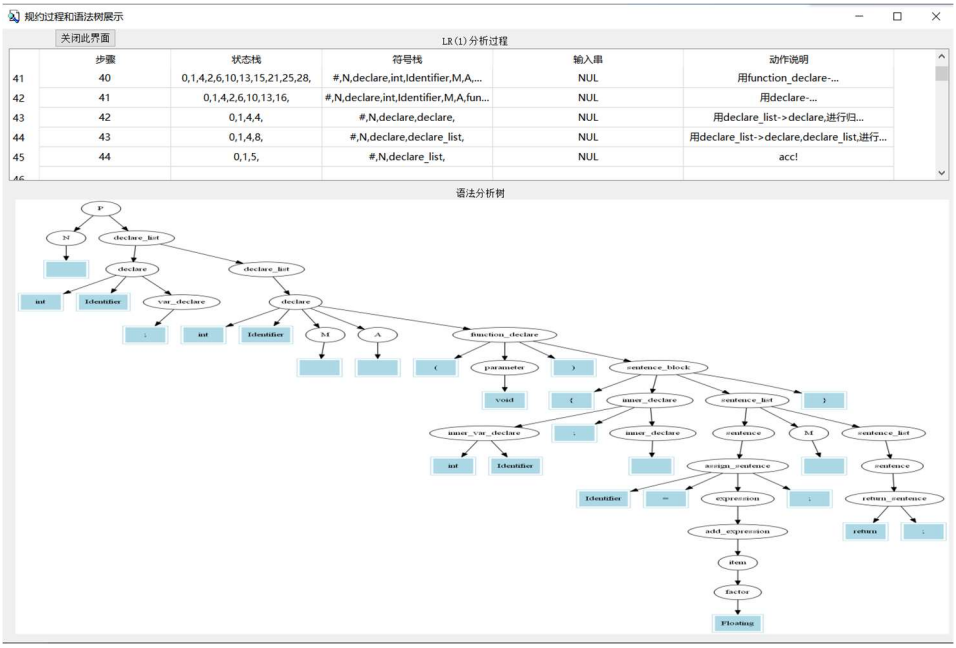
调试分析

测试数据及测试结果

不含空产生式，不含过程调用，但是含有浮点数的测试

```
int  a;
int  main(void)  {
     int  a;
     a  =  1e-3;
     return;
}
```

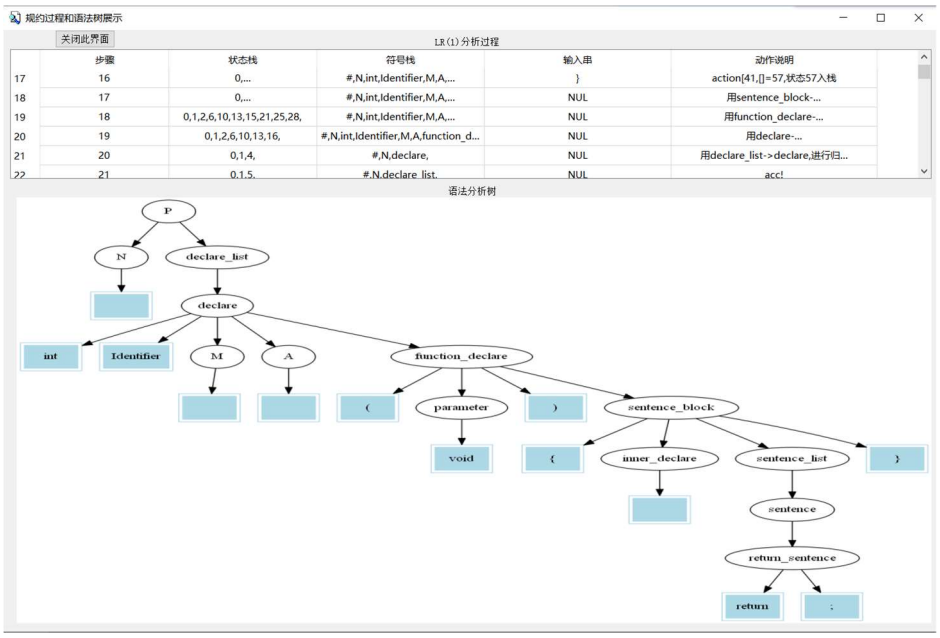
运行结果如下图所示，产生了正确的语法分析树，表格中是整个 LR（1）分析的移进规约过程：



含有空产生式，不含过程调用的测试

```
int  main(void)  {
     return;
}
```

运行结果如下图所示，产生了正确的语法分析树，表格中是整个 LR（1）分析的移进规约过程：



含有过程调用的测试

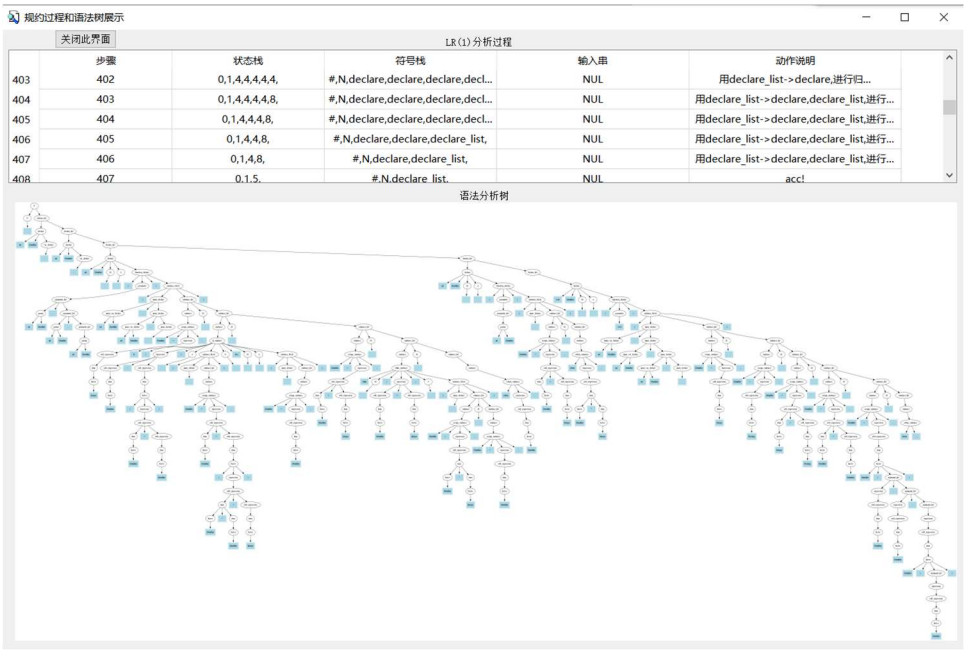
```
int  a;
int  b;
int  program(int  a,  int  b,  int  c){
     int  i;
     int  j;
     i  =  0;
     if  (a  >  (b  +  c)){
               j  =  a  +  (b  *  c  +  1);
     }
     else{
               j  =  a;
     }
     a  =  a  +  15.23;
     while  (i  <=  100){
```

```
            i = j * 2;
            j = i;
        }
        return i;
}

int demo(int a){
        a = a + 2;
        return a * 2;
}

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

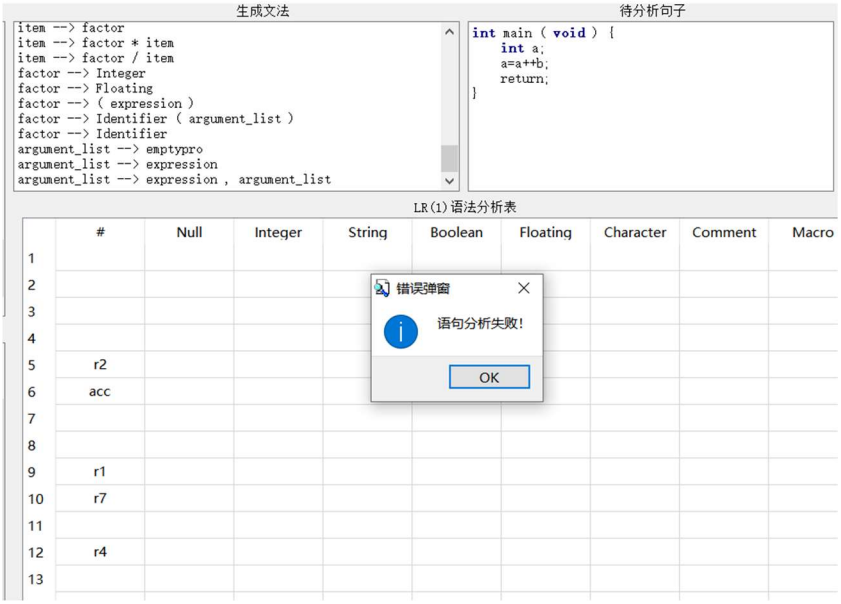
运行结果如下图所示，产生了正确的语法分析树，表格中是整个 LR（1）分析的移进规约过程：



错误代码测试

```
int  main(void)  {
     int  a;
     a  =  a++b;
     return;
}
```

运行结果如下图所示，因为第三行的++符号不是产生式能够推出的符号，因此不能正确规约，产生弹窗错误。



时间复杂度分析

词法分析过程

函数	复杂度分析
preAnalyze	O(（测试C程序的长度）*（测试C程序的长度))=O(n^2)
analyzeSpliters	O(（测试C程序的长度）*（测试C程序的长度))=O(n^2)

analyzeKeywordsAndOperators	O(测试 C 程序的长度+3*测试 C 程序的长度*测试 C 程序的长度)=O(n^2)
analyzeWords	O(（测试 C 程序的长度）*（测试 C 程序的长度))=O(n^2)

故词法分析器的时间复杂度为 ：O(n^2)　　（n 为测试 C 程序的长度）

语法分析过程

函数	复杂度分析	
getFirst	O(终结符数量+产生式数量*dfsGetFirst)=O(m^3)	
getItemSet	getItemClosure	O(项目集中产生式数量*单个非终结符的产生式*单个产生式的长度)=O(m^2)
	O(项目集的数量*符号的数量*单个项目集中项目的数量*（getItemClosure）)=O(m^3)	
getLR1Table	O(项目集的数量*单个项目集的边的数量+项目集的数量*单个项目集中项目的数量*单个项目的向前搜索符的数量)=O(m^3)	
totalControl	测试 C 程序的长度*测试 C 程序的长度=O(n^2)	

故语法分析器的时间复杂度为 ：O(n^2) +O(m^3)　（n 为测试 C 程序的长度，m=max(产生式数量，文法符号数量，产生式长度）

存在问题及解决方法

从课本单个字符到整个单词

课本经典的 LR1 文法的终结符与非终结符都是单个字母或数字，而现实中程序所属文法的终结符与非终结符是字符串类型的，所以为了方便存储与使用，我们将终结符与非终结符用数字进行替代，并一一对应。在使用时，字符串与数字可以来回切换。

有关 QT 使用技巧

使用的开发工具需要位数相同，而且 C++标准一致，在 QT 中，如果全局变量在头文件中定义，则需要使用 extern 方法进行定义。

由于小组分工，每个人负责不同的模块，所以模块设计时需要协商好入口参数与返回参数，避免模块间信息传递错误的尴尬。

文法产生式出现空串

如果输入的产生式里面包含空产生式的话，对于求 FIRST 集和求项目集来说都是一个头疼的问题，我们采用的处理方法是，求 FIRST 集时ε显式表示，即求得的 FIRST 集可能包含ε，而在求项目集族的时候ε隐式表示，即空产生式从一开始就是一个规约项目，应该直接表示成$S \to \cdot$而不是$S \to .\varepsilon$，因为最后根据算法生成项目之间的转移关系图是一个 DFA，不可能有空转移的存在。最后经过实际测试，这种处理方法也是完全可行的。

总结与收获

成果总结

本次实验我们以小组合作的形式，完成了词法分析器和语法分析器大作业。在课程的理论学习基础之上，小组成员都积极进行了动手实践，通过编程实现了词法分析并生成单词流、求给定文法中终结符、非终结符、产生式的 First 集、计算 LR(1) 项目簇、构建对应 ACTION 表和 GOTO 表，完成了类 C 语言的词法分析和语法分析过程。在实验进行过程中也遇到了很多困难，例如产生式空串的处理、语法树构建、各个结构体成员的数据类型定义等问题，但都在小组合作下对其进行了解决。

本实验完整的完成了语法分析器，能够正确地根据给定类 C 语言的文法对类 C 语言程序进行词法和语法分析，并能正确输出 LR（1）分析表、LR（1）分析过程和语法树。程序设计与编写结束后撰写了完整的实验报告，对本次实验的过程以及所设计的程序进行了清晰全面的说明。总体来说完成了课程作业

的要求，实验进展顺利，先将重要的成果总结如下：

1.词法分析器及 LR（1）语法分析器源代码一份。

2.基于 QT 的编译原理演示程序界面程序一份。

3.编译原理 LR（1）词法分析报告一份。

实验过程中对课程的认识

通过本次大作业的编程实践，我们将编译原理课堂上的较为抽象的理论知识转化为实际工程上的应用，从而加深了对课程知识的理解，并在实际编程的过程中建立了不同编译原理知识间的联系，将零碎的知识进行了有机整合并进行应用。

通过编程实现类 C 语法的分析过程，我们了解了高级编程语言程序在一台计算机上进行程序内容识别和语言文法正误检查的流程和机制，这能让我们更好的理解程序设计语言深层的设计理念和设计思想，这对于我们日后的计算机工程实践是一份宝贵的经验。

心得收获

继词法分析器设计后，我们又迎来编译器设计过程中重要的语法分析器的设计，相对于之前的词法分析器，语法分析器的工作量明显提高了不少，不仅需要调用先前设计好的词法分析器，还需要为后续调用语义分析模块做考虑。

由于课本内容只介绍了 LR 分析器的概念与 LR(0)分析器中分析表的构造，而我们这次的任务是创建基于 LR(1)分析方法的语法分析器，所以我们一开始在网上查阅了大量的资料，组织多次小组会议，互相讨论，分享自己关于语法分析器设计的想法，在多次讨论后设计了所要用到的数据结构，并构思了整体框架与程序流程。之后，我们为整体程序分成了多个模块，每个人设计其中的部分模块，经过交流后统一了模块之间衔接的接口。

接下来，我们几个人开始了自己负责模块的代码撰写，互相督促模块设计进度，在群里面互相解决写代码时遇到的 bug。经过一周的努力，我们写完各自负责的模块代码并进行了整合，运行之后效果还不错，但我们利用空闲时间又对设计程序进行多次优化，最终我们小组实现了一个相对具有交互功能的

LR(1)语法分析器。

这段时间，我们过得十分充实与紧张，虽然语法分析器的设计工作量大且比较复杂，但我们也拿出自己的努力与时间，获得了一个具有演示功能的语法分析器。在实现语法分析器的过程中，我们深入理解了语法分析器的核心工作原理，更充分的掌握编译过程中语法分析这一重要部分。同时，小组成员之间的合作与交流也使我们懂得在日后的项目设计中分工合作的重要性。

附录

终结符	"Null","Integer","String","Boolean","Floating","Character", "Comment","Macro","WhiteSpace","EndLine","{","}","[","]","(", ")", "+","-","*","/", "%","++","--","==","!=", ">","<",">=", "<=","&&","||","!","&","|", "~","^","<<",">>","=","+=","-=", "*=","/=","%=","<<=",">>=", "&=","|=", "^=",",",";","auto", "break","case","char","const","continue","default","do","if" "double","else","enum","extern","float","for","goto","int", "long","register","return","short","signed","sizeof","Word" "static","struct","switch","typedef","union","unsigned", "void","volatile","while","Identifier","Word"
非终结符	"emptypro","A","P","M","N","add_expression","argument_list", "assign_sentence","declare","declare_list","expression", "factor","function_declare","if_sentence","inner_declare", "inner_var_declare","item","param","parameter","sentence", "parameter_list","return_sentence","sentence_block", "sentence_list","var_declare","while_sentence"
产生式	"P———>N declare_list", "declare_list———>declare,declare_list", "declare_list———>declare", "declare———>'int','Identifier',M,A,function_declare", "declare———>'int','Identifier',var_declare", "declare———>'void','Identifier',M,A,function_declare", "A———>emptypro", "var_declare———>';'", "function_declare———>'(',parameter,')',sentence_block", "parameter———>parameter_list", "parameter———>'void'", "parameter_list———>param", "parameter_list———>param,',',parameter_list",

	"param———>'int','Identifier'", "sentence_block———>'{',inner_declare,sentence_list,'}'", "inner_declare———>emptypro", "inner_declare———>inner_var_declare,';',inner_declare", "inner_var_declare———>'int','Identifier'", "sentence_list———>sentence,M,sentence_list", "sentence_list———>sentence", "sentence———>if_sentence", "sentence———>while_sentence", "sentence———>return_sentence", "sentence———>assign_sentence", "assign_sentence———>'Identifier','=',expression,';'", "return_sentence———>'return',';'", "return_sentence———>'return',expression,';'", "N———>emptypro", "M———>emptypro", "expression———>add_expression", "expression———>add_expression,'>',add_expression", "expression———>add_expression,'>=',add_expression", "expression———>add_expression,'<',add_expression", "expression———>add_expression,'<=',add_expression", "expression———>add_expression,'==',add_expression", "expression———>add_expression,'!=',add_expression", "add_expression———>item", "add_expression———>item,'+',add_expression", "add_expression———>item,'-',add_expression", "item———>factor", "item———>factor,'*',item", "item———>factor,'/',item", "factor———>'Integer'", "factor———>'(',expression,')'", "factor———>'Identifier','(',argument_list,')'", "factor———>'Identifier'", "argument_list———>emptypro", "argument_list———>expression", "argument_list———>expression,',',argument_list", "NULL"