

# 同济大学计算机系

## 词法和语法分析工具设计与实现 实验报告



院 系 电子与信息工程学院

专 业 计算机科学与技术

组 员 1 1953461 高志成

组 员 2 1953492 李强

授课老师 卫志华

# 目录

一、实验内容及需求分析.....	4
1.1. 实验内容.....	4
1.2. 程序功能.....	4
1.3. 输入信息.....	4
1.4. 输出信息.....	5
1.5. 测试数据.....	5
二、项目总体设计.....	5
2.1. 项目总体设计说明.....	5
2.2. 数据结构定义.....	6
2.2.1. 文法非终结符和终结符的定义.....	6
2.2.2. 词法分析器类型定义.....	6
2.2.3. 语法分析器类型定义.....	7
2.3. 主程序流程图.....	9
三、词法分析器设计方法.....	9
3.1. 词法分析器的任务.....	9
3.2. 词法分析方法.....	10
3.3. 词法分析器的输出.....	12
四、语法分析器设计方法.....	12
4.1. LR(1)分析方法介绍.....	12
4.2. LR(1)分析表构造方法.....	13
4.3. 关键函数的实现方法.....	13
4.3.1. 求 FIRST 集函数.....	13
4.3.2. 求项目集规范族函数.....	15
4.3.3. 生成识别活前缀的 DFA 函数.....	17
4.3.4. 语法分析函数.....	20
五、图形界面设计.....	23
5.1. 数据结构.....	23
5.2. 绘制 DFA 和语法树.....	23
六、调试分析与结果展示.....	25
6.1. 调试-源程序 1.....	26

6.2. 调试-源程序 2 .....	28
6.3. 结果展示: .....	29
七、程序设计过程中的新思考.....	34
7.1. 非终结符和终结符的表示方法.....	34
7.2. 词法单元的属性值存储方式.....	34
7.3. LR(1)项目都存储方式.....	34
7.4. epsilon 产生式的特殊处理 .....	34
7.5. 语法树的存储和建立过程.....	35
八、总结与收获.....	35
8.1. 项目总结.....	35
8.2. 小组分工与贡献.....	36
8.3. 项目收获.....	36
九、参考文献.....	37

# 一、实验内容及需求分析

## 1.1. 实验内容

本实验要求根据 LR(1)分析方法,编写一个类 C 语言的 LR(1)语法分析程序,完成对包含过程调用的类 C 示例程序的词法和语法分析。

## 1.2. 程序功能

本程序使用 LR(1)分析方法编写,支持输入自定义文法,可以自动生成语法分析程序。

向本程序输入已知文法的产生式以及源程序,本程序负责进行词法分析和语法分析,并输出语法分析结果,包括:

1. 是否通过语法分析
2. 若通过语法分析,则还要输出源程序的规约过程和语法树
3. 若无法通过语法分析,输出报错信息以及出错位置


## 1.3. 输入信息

本程序的输入分为两部分,分别是类 C 语言文法产生式和源程序。该两部分均以文件形式输入。

类 C 语言文法产生式的输入格式示例如下。基本形式为

产生式左部 -> 产生式右部 1 右部 2 ...

需要注意的是,->符号的两边、不同的右部符号之间都需要以空格隔开,便于程序一次读入一个完整的文法符号。

 grammar.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
program -> dec_list
dec_list -> dec
dec_list -> dec dec_list
dec -> int id dec_type
dec -> void id func_dec
dec_type -> var_dec
dec_type -> func_dec
var_dec -> ;
```

本程序对源程序的格式无要求,并且支持各种类型的浮点数。

是否支持过程调用取决于输入的文法产生式。示例输入支持过程调用。

## 1.4. 输出信息

详见 3.3 和 6 中词法分析器的输出和语法分析器的输出部分。

## 1.5. 测试数据

在 test 目录下给出了 grammar.txt 和 test\_program.c，作为测试文件。程序输出以图形化界面进行展示。

具体的测试细节参考调试分析与结果展示部分。

# 二、项目总体设计

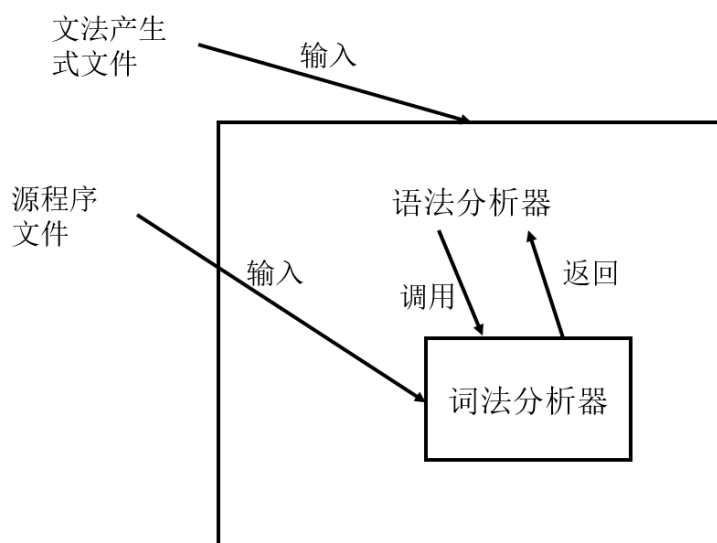
## 2.1. 项目总体设计说明

整个项目采用 LR(1)分析方法进行设计,分为词法分析和语法分析两大部分。

在词法分析部分,程序读入源程序文件,并按照词法规则对源程序进行整理,使得语法分析器可以更方便地处理输入串。

语法分析部分,程序首先读入文法产生式,由产生式构造出对应的 ACTION 表和 GOTO 表;然后每当语法分析器需要读输入串时,就去调用词法分析器相应的接口,获得一个输入符号,并根据 LR(1)分析表进行相应移进、归约、接受、报错动作,直到源程序分析完毕或者出现错误为止。

词法分析器是语法分析器的子模块,每当语法分析器需要下一个符号时,便去词法分析器处取得。输入文件、词法分析器、语法分析器的关系如下:



## 2.2. 数据结构定义

### 2.2.1. 文法非终结符和终结符的定义

为节省内存、提高效率起见，程序中文法符号使用枚举类型进行定义。Tag 枚举类存储着所有的终结符和非终结符，并且提供 isVT 和 isVN 函数来区分终结符和非终结符。

由于枚举类型只能预先写在程序之中，故文法符号不支持自动输入。实际使用时可以根据文法需要对 Tag 类型进行简单的修改。

定义如下，由于篇幅限制仅展示一小部分：

```
enum class Tag
{
    //终结符
    epsilon = 0, //空
    the_end,     //#, 表示终止
    id,          //标识符
    num,         //数字
    //符号 symbol
    sb_add,      //+
    sb_sub,      //-
    sb_time,     //*
    .....
}
```

### 2.2.2. 词法分析器类型定义

词法分析器类为 Lexer，对外提供三个方法。

openFile 方法用于指定待分析的源程序文件。

getNextLexcial 方法是词法分析器的核心。调用该方法后，可以返回下一个词法单元。词法单元由 Token 结构体表示，记录着该词法单元对应的 Tag 类型及其属性值。

scanFile 方法用于一次扫描完整个源程序，并且将词法单元统一输出到一个文件中。由于本程序中词法分析作为语法分析的子模块出现，故不使用该方法。

词法分析器类型定义如下：

```
class Lexer
{
private:
    char peek;
    int line;      //当前行
    int col;       //当前列
```

```

    ifstream file_in;

    bool getNextChar(char&, const bool = true);

public:
    Lexer();
    ~Lexer();

    //打开输入文件
    bool openFile(const char*);

    //获取下一个词法单元
    State getNextLexical(Token& next_token);
    //扫描整个文件并输出词法分析结果
    State scanFile(const char*);
};

```

### 2.2.3. 语法分析器类型定义

语法分析器类为 LR1\_Parser。语法分析模块为整个编译器前端的核心模块，故这部分的数据结构较多，内部结构较复杂。

语法分析主要涉及到的两个数据结构为产生式和 LR1 项目。在程序中，Grammar 结构为文法产生式，其中包含一个左部符号和一个右部符号数组。GrammarProject 结构为项目，包含一根指向文法产生式的指针（避免重复存储）、一个 point 变量指向项目中的 • 的位置，一个 follows 集合表示该项目的后续输入符号。其结构定义如下：

```

struct Grammar //文法, left->right
{
    Tag left;
    vector<Tag> right;
};

struct GrammarProject //LR(1)项目
{
    int p_grammar;           //该项目的产生式指针, 存储产生式在vector中对应的下标
    int point;               //点的位置
    /* S->.E point=0 , S->E. point=1 */
    set<Tag> follows;        //项目后面可以跟随的终结符
    /* S->.E, #/a/b follows={# a b} */
    ...                      //其他辅助的运算符重载不再赘述
}

```

语法分析器主要为外界提供三个接口。

init 方法用于输入文法产生式文件，初始化语法分析器的文法、项目集规范族，并初始化 ACTION 表和 GOTO 表。该方法的内部实现主要调用 initFirstList 和 initActionGotoMap 两个私有方法。

parser 方法用于解析源程序。输入源程序文件，parser 对源程序进行 LR1 分析，并建立一棵语法树。若源程序存在语法错误，则 parser 方法的第二个参数返回一个 Token，提供出错信息。

语法分析器类型定义如下：

```
class LR1_Parser
{
private:
    Lexer lexer;                //词法分析器

    vector<Grammar> grammar_list;    //文法集合
    map<Tag, set<Tag>> first_list;    //非终结符first集
    //map<Tag, set<Tag>> follow_list; //非终结符follow集
    vector<set<GrammarProject>> project_set_list;    //项目集
    //map<int, map<Tag, int>> state_trans_map;    //项目之间的转移关系(int存储项目集的下标)
    map<int, map<Tag, Movement>> action_go_map;    //action表和goto表,存储在一起

    PTree pTree;                //语法树

private:
    State openGrammarFile(const char*); //读入文法产生式
    set<GrammarProject> getClosure(const set<GrammarProject>&); //求CLOSURE集
    int findSameProjectSet(const set<GrammarProject>&); //查找相同的CLOSURE集,失败返回-1
    void initFirstList(); //初始化First集
    State initActionGotoMap(); //求识别活前缀的DFA

public:    //记得改为private
    LR1_Parser();
    ~LR1_Parser();

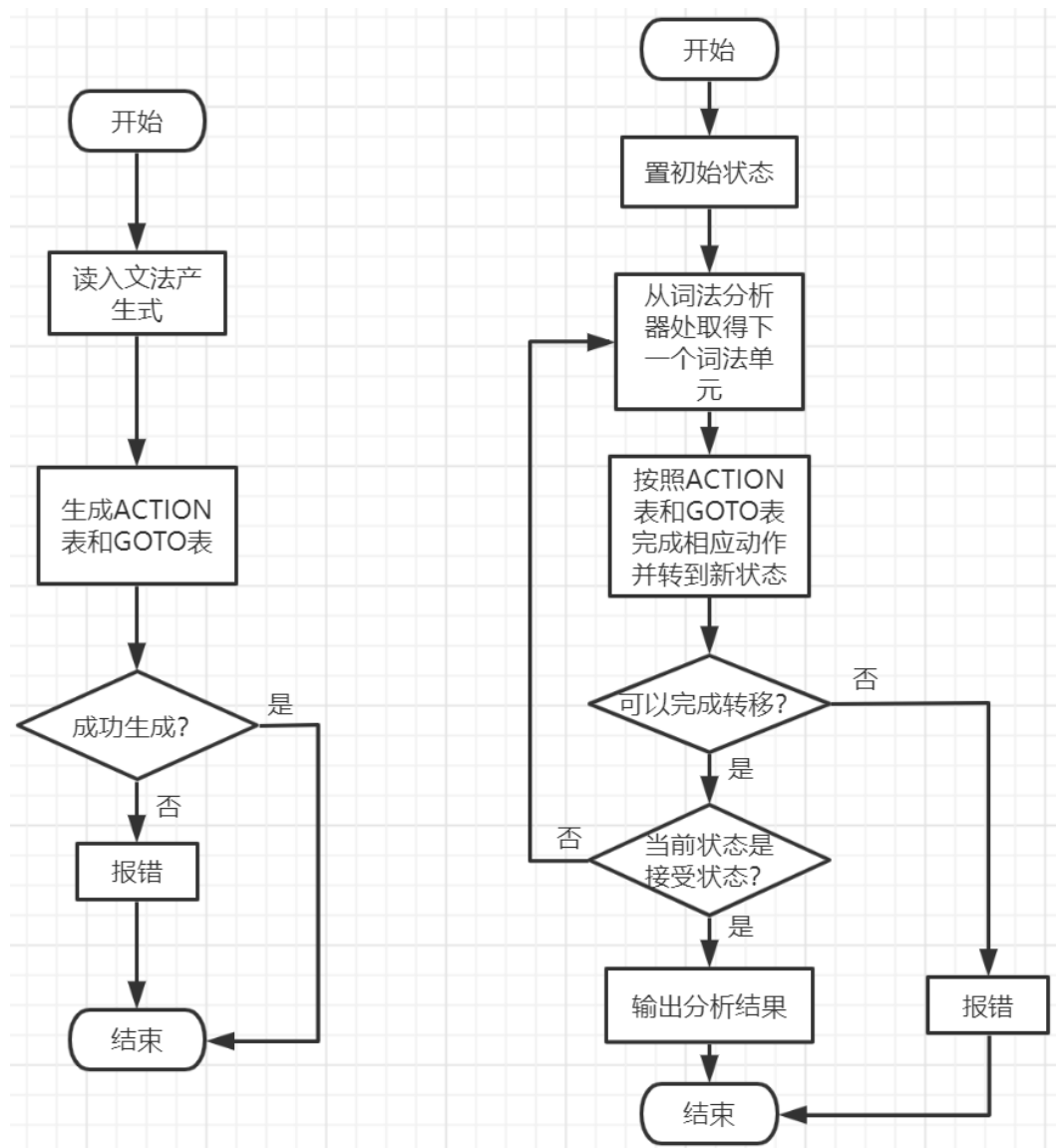
    State init(const char*); //语法分析器初始化
    State parser(const char*, Token&); //语法分析
    void printTree(ostream& out); //打印树
    void printVP_DFA(ostream& out); //打印DFA
};
```



## 2.3. 主程序流程图

主程序流程图如下。左边是 LR(1)语法分析器的自动生成过程，右边是使用该语法分析器进行分析的过程。

流程图给出了整个语法分析器构建和工作的大致过程。其中每部分的实现方法在后面有详细的介绍。



## 三、词法分析器设计方法

### 3.1. 词法分析器的任务

设计词法分析器最关键的一点是要弄清楚词法分析器究竟要做什么。语法分析是编译程序的核心，词法分析器的工作就是对源程序进行一些处理，使得语法

分析部分可以更方便地读懂源程序的每个部分到底包含了什么，而不用花费大量时间来处理源程序千奇百怪的格式。词法分析器的工作，就是围绕“使语法分析器更专注于语法分析本身”这一点来展开的。

譬如，源程序中含有这样一句：

```
abdfjsj23jsd=1.2345e9;
```

如果我们使用词法分析器，可以把这句话解析为类似下面这样的格式：

```
{标识符, abdfjsj23jsd} {赋值运算符, =} {数字, 1.2345e9} {界符, ;}
```

这样一来，当语法分析器向词法分析器请求输入时，词法分析器给出“标识符 赋值运算符 数字 界符”这种处理过后的语句。语法分析器清楚地知道了该语句的每部分是什么，便可以直接在语法层面上对这种形式的语句进行统一处理。

如果没有词法分析器，那么语法分析器必须亲自从源程序中读取输入。上面的输入为例，语法分析器必须花费精力读懂“abdfjsj23jsd”是一个整体并且表示一个标识符、“1.2345e9”是一个整体并且表示一个数字、虽然“=”两边没有空格但它是单独的一个整体，等等。这会使语法分析过程变得非常困难。

同时，词法分析部分使用的是正则文法，相比于语法分析的 LR(1)文法，有更简单的处理方法。因此，若不将词法部分使用正则文法来分析，而是统一使用 LR(1)方法分析，会极大降低编译的效率。

在本程序中，词法分析器的任务是：解析源程序，将源程序分隔为一个个便于语法分析器识别的语法单元。同时，词法分析器略去源程序中的注释部分。

### 3.2. 词法分析方法

词法分析的文法属于正则文法，可以使用 DFA 模型进行分析。本实验中类 C 语言的词法规则如下（增加了各种表示的浮点数的处理，未在规则中写出）：

关键字：int | void | if | else | while | return

标识符：字母（字母|数字）\*（注：不与关键字相同）

数值：数字（数字）\*

赋值号：=

算符：+ | - | \* | / | = | > | >= | < | <= | !=

界符：;

分隔符：,

注释号：/\* \*/ //

左括号：(

右括号：)

左大括号：{

右大括号：}

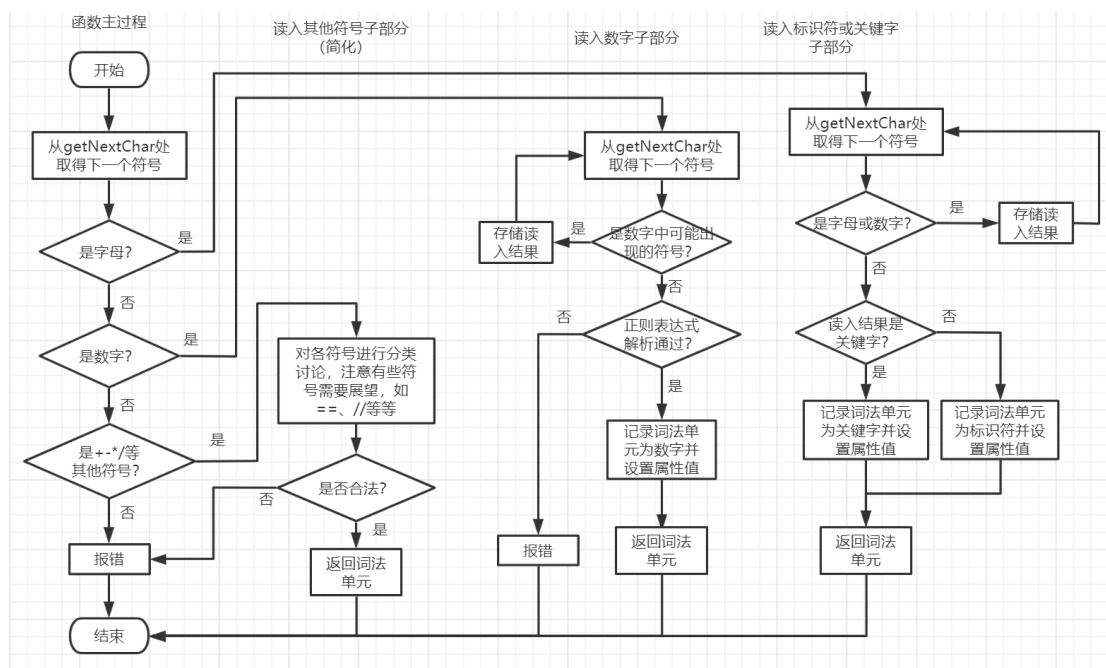
字母: a|...|z|A|...|Z  
数字: 0|1|2|3|4|5|6|7|8|9  
结束符: #

由于本实验中类 C 语言的词法规则较为简单, 故直接使用 switch 语句和 if-else 语句模拟 DFA 分析过程, 部分分析 (如实数) 采用 C++ 的正则表达式进行辅助分析。

Lexer 类中进行词法分析有两个较为关键的方法, 分别是私有方法 getNextChar 和公有 getNextLexical 方法。

词法分析器从源程序读入字符是由 getNextChar 方法完成的。getNextChar 方法是为 getNextLexical 服务的, 它每次按需从源程序中读取单个字符返回给 getNextLexical 方法, 并且存储下一个字符于 peek 变量中, 便于词法分析进行展望。

getNextLexical 是词法分析器的主要方法。该方法返回下一个词法单元。其算法流程如下:



该算法的流程图中较为详细地介绍了读取标识符/关键字和读取数字的方法。由于篇幅问题, 读取其他符号的部分简化处理。看上去算法十分复杂, 实际上并不难, 是一个情况较多的分类讨论问题。

在读入标识符/关键字时, 程序不断从 getNextChar 取得下一个符号。若符号是字母或数字, 则拼接到上次读入结果之后; 否则说明读入完毕。在关键字表中检索该项, 判断读入的字符串是否为关键字, 并赋予该词法单元相应的属性值。

读入数字部分处理的情况非常复杂。程序中支持如 123、123.、

123.4567、.123e9 等等不同格式的数字输入，使用 if-else 结构难以处理。程序中使用了 C++ 正则表达式进行辅助处理，正则表达式定义如下：

$$(\d+|\d+\.\d+|\d+\.\d+\d+)(e\d+)$$

程序逐个读入所有数字中可能出现的字符，包括数字、点、e，最后使用正则表达式进行匹配，若匹配成功则表示是数字，否则进行报错处理。使用正则表达式可以使算法结构更加清晰。

读入符号部分需要做一些特殊说明。某些符号，如=、==、/、/\*\*/，存在相同的前缀。程序中碰到这种情况时，向前展望一位从而判断到底读入了什么符号。特别的，当读入 // 或 /\* 符号时，程序会根据注释规则，去掉源程序中的所有注释。由于源程序中的符号最多只有两位字符，故这种展望一位的方法是很有效的。

### 3.3. 词法分析器的输出

getNextLexical 函数返回一个词法单元 Token，其结构定义如下：

```
struct Token    //词法单元, 由类型和属性值构成
{
    Tag tag;      //词法单元类型
    string value; //属性值, 空值用null表示, 全部用str存储, 需要时使用sstream解析
};
```

例如，对于源程序中的符号 abcd，词法分析器返回 {Tag::id, "abcd"}。

## 四、语法分析器设计方法

### 4.1. LR(1)分析方法介绍

LR(1)分析法是一种自下而上语法分析方法，从输入串开始逐步进行规约，直至得到开始符号。LR(1)分析器利用历史（栈）、现实（当前输入符号）、展望（尚未输入的符号，LR(1)中展望一位），寻找句柄，进行移进-规约操作，完成语法分析。

分析器包括总控程序和分析表两部分。分析器读入一个个词法单元，在总控程序的控制下，参考分析表上记录的动作进行状态转移，最终接受输入串或报错。

在 LR(1)分析法中，总控程序对于所有 LR 分析器都是相同的，参考分析表可以轻松地完成语法分析过程。因此，最关键的是如何由文法产生式构造出分析

表，这也是本程序设计的重难点。

## 4.2. LR(1)分析表构造方法

程序中构造 LR(1)分析表的核心是构造识别活前缀的 DFA。在构造 DFA 的过程中可以一步步构造出分析表的状态，以及状态之间的转移关系。

总的来说，语法分析器的准备工作包括：

1. 读入文法产生式，构造拓广文法  $S' \rightarrow$  起始符号
2. 由产生式计算各非终结符的 First 集，为后续做准备

构造分析表的步骤如下：

1. 求第一个项目  $S' \rightarrow \cdot S$ ，# 的 CLOSURE 集合，作为初始项目集  $I_0$
2. 从初始项目集出发，检索其所有项目。对于所有移进项目，求出其移进后产生的新项目的 CLOSURE 闭包。若该闭包不存在在项目集列表中，则作为新状态加入项目集列表，同时置分析表的相应位置为移进。对于所有归约项目，置分析表的相应位置为归约
3. 以新产生的项目集为当前项目集，重复步骤 2，直到没有新项目集产生

构造过程中，涉及到求项目集  $I$  的 CLOSURE 闭包的问题。构造  $I$  的 CLOSURE 闭包的算法如下：

1.  $I$  的任何项目都包含在 CLOSURE( $I$ )中
2. 若项目  $[A \rightarrow \alpha \cdot B \beta, a]$  属于 CLOSURE( $I$ )， $B \rightarrow \gamma$  是一个产生式，那么对于 First( $\beta a$ ) 中的每个终结符  $b$ ，如果  $[B \rightarrow \cdot \gamma, b]$  不在 CLOSURE( $I$ )中，则把它加入进去
3. 重复执行该步骤，直到 CLOSURE( $I$ )不再增加

## 4.3. 关键函数的实现方法

### 4.3.1. 求 FIRST 集函数

算法：

Step1: 遍历所有产生式，对所有右部首字符为终结符的产生式，将其首终结符放入产生式左部的 FIRST 集中

Step2: 对于每一个右部首字符为非终结符的产生式，对其进行如下操作：

若其首部字符和产生式左部字符不同，则将右部首字符的 FIRST 集去除空值后加入产生式左部的 FIRST 集，否则不进行加入产生式的操作。判断当前非终结符的 FIRST 集是否包含空值，若包含则继续分析产生式右部的下一个字符，以此类推，直到产生式右部所有字符都判断完或者遇到不能推空的非终结符为止。

Step3: 循环执行 Step2，直至某轮遍历后所有 FIRST 集都不再发生变化。

代码:

```
void LR1_Parser::getFirstList()
{
    vector<int> grammar_pointer;    //记录产生式右部第一个符号为非终结符的
    文法
    for (int i = 0; i < grammar_list.size(); i++) {
        if (grammar_list[i].right.size() == 0)
            first_list[grammar_list[i].left].insert(Tag::epsilon);
        else {
            auto first_elem = grammar_list[i].right.front();
            if (isVT(first_elem))
                first_list[grammar_list[i].left].insert(first_elem);
            else
                grammar_pointer.emplace_back(i);
        }
    }

    bool flag;
    while (true) {
        flag = false;
        for (const auto& i : grammar_pointer) {
            bool have_epsilon = false;
            for (const auto& elem_A : grammar_list[i].right) {
                have_epsilon = false;
                if (isVN(elem_A)) {
                    //考虑 A->A..的特殊情况
                    if (grammar_list[i].left == elem_A) {
                        if (first_list[elem_A].count(Tag::epsilon))
                            continue;
                        else
                            break;
                    }
                    //若出现 A->B...,则将 B 的 first 集全部加到 A 中
                    for (const auto& elem_B : first_list[elem_A]) {
                        if (elem_B == Tag::epsilon) {
                            have_epsilon = true;
                            continue;    //epsilon 不加入
                        }
                    }
                }
            }
        }
    }
}
```

```

        }
        int before =
first_list[grammar_list[i].left].size();
        first_list[grammar_list[i].left].insert(elem_B);
        int after =
first_list[grammar_list[i].left].size();
        if (before < after)
            flag = true;
    }
    if (!have_epsilon)
        break; //若不含空,则后续不用继续加入
    }
    else
        break;
}
if (have_epsilon) //如果产生式最后一个符号也含空,则将空加入
First 集
        first_list[grammar_list[i].left].insert(Tag::epsilon);
    }
    if (!flag) //如果 first 集不再增加,则返回
        break;
}
}

```

#### 4.3.2. 求项目集规范族函数

算法:

Step1: 对于给定的项目集, 其所有项目均属于其 CLOSURE(I)

Step2: 遍历当前 CLOSURE(I)中已有的项目, 若  $A \rightarrow \alpha \cdot B \beta$  属于 CLOSURE(I), 那么, 对任何关于 B 的产生式  $B \rightarrow \gamma$ , 项目  $B \rightarrow \cdot \gamma$  也属于 CLOSURE(I)。重复执行以上步骤直至 CLOSURE(I) 不再增大为止。

代码:

```

set<GrammarProject> LR1_Parser::getClosure(const set<GrammarProject>&
project_set)
{
    set<GrammarProject> ret(project_set); //project_set 自身的
所有项目都在闭包中
    set<GrammarProject> old_project(project_set); //辅助集合
    set<GrammarProject> new_project;

    bool flag;
    while (true) {

```

```

flag = false;
for (const auto& i : old_project) { //扫描上一次产生的所有项目
    if (grammar_list[i.p_grammar].right.size() > i.point &&
        isVN(grammar_list[i.p_grammar].right[i.point])) {
        //A-> $\alpha$ .B $\beta$  型
        Tag vn = grammar_list[i.p_grammar].right[i.point];

        //求出 first( $\beta$ a)
        set<Tag> firstba;
        if (i.point + 1 <
            grammar_list[i.p_grammar].right.size()) {
            firstba =
first_list[grammar_list[i.p_grammar].right[i.point + 1]];
            auto p = firstba.find(Tag::epsilon);
            if (p != firstba.cend()) {
                //如果含有 epsilon,则删除 epsilon 并把原项目的
follows 加入

                firstba.erase(p);
                for (const auto& follow : i.follows)
                    firstba.insert(follow);
            }
        }
        else {
            for (const auto& follow : i.follows)
                firstba.insert(follow);
        }

        for (int gp = 0; gp < grammar_list.size(); gp++) {
            //扫描所有 B-> $\gamma$  型的产生式
            if (grammar_list[gp].left == vn) {
                //若 CLOSURE 中不存在{B-> $\gamma$ ,firstba},则加入
                bool have = false;
                for (auto it = ret.begin(); it != ret.end();
                    ++it) {
                    if (it->p_grammar == gp && it->point == 0) {
                        //项目在集合
                        have = true;
                        if (it->follows != firstba) {
                            //若 follows 不完整,则插入新的 follows
                            flag = true;
                            //由于集合元素的值无法修改,故只能覆盖之
                            auto npg = *it;
                            for (Tag firstba_elem : firstba)
                                npg.follows.insert(firstba_elem);
                        }
                    }
                }
            }
        }
    }
}

```



```

        ret.erase(it);
        ret.insert(ngp);
        new_project.insert(ngp);
    }
    break;
}
}
if (!have) {
    //否则插入新项目
    flag = true;
    ret.insert({ gp,0,firstba });
    new_project.insert({ gp,0,firstba });
}
}
}
}
}
if (!flag) //不再增加,则返回
    break;
old_project = new_project; //对新添加项目进行下一轮扫描
new_project.clear();
}

return ret;
}

```

#### 4.3.3. 生成识别活前缀的 DFA 函数

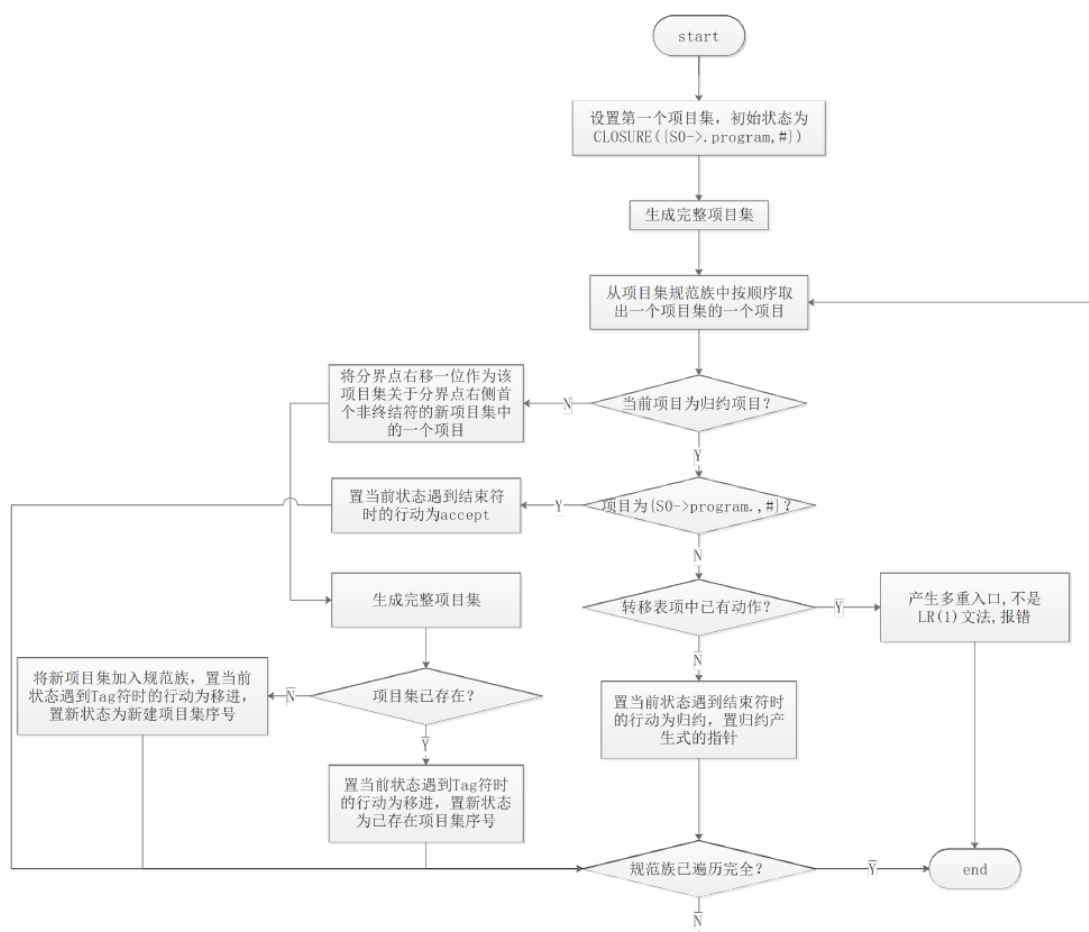
算法:

Step1: 构造 GO 集: GO 是一个状态转换函数。I 是一个项目集, X 是一个文法符号。函数值  $GO(I, X)$  定义为:  $GO(I, X) = CLOSURE(J)$ , 其中  $J = \{ \text{任何形如 } A \rightarrow \alpha X \cdot \beta \text{ 的项目} | A \rightarrow \alpha \cdot X \beta \text{ 属于 } I \}$ 。直观上说, 若 I 是对某个活前缀  $\gamma$  有效的项目集, 那么,  $GO(I, X)$  便是对  $\gamma X$  有效的项目集。

Step2: 对于当前规范族中每个项目集 I 和  $G'$  的每个符号 X, 若  $GO(I, X)$  非空且不属于规范族则把  $GO(I, X)$  放入规范族中, 重复这个过程直至规范族不再增大。

Step3: 用规范族中的项目集和 GO 函数构造 DFA

流程图:



代码:

```

State LR1_Parser::getViablePrefixDFA()
{
    project_set_list.clear();
    //初始状态为 CLOSURE({S'-.>.S,#})
    project_set_list.emplace_back(getClosure({ { 0,0,{Tag::the_end} } }));
};

int new_index = 0;    //新项目集下标
while (new_index < project_set_list.size()) {
    set<GrammarProject>& pset_now = project_set_list[new_index];
    map<Tag, set<GrammarProject>> new_pset_map;    //当前项目集可以产生的新项目集

    //扫描所有项目
    for (const auto& i : pset_now) {
        if (i.point < grammar_list[i.p_grammar].right.size()) {
            //不是归约项目

```

```

        new_pset_map[grammar_list[i.p_grammar].right[i.point]].i
insert({ i.p_grammar,i.point + 1,i.follows });
    }
    else {
        //是归约项目
        if (i.p_grammar == 0 && i.point == 1 && i.follows.size()
== 1 && *i.follows.cbegin() == Tag::the_end)
            action_go_map[new_index][Tag::the_end] =
{ Action::accept,i.p_grammar };    //可接受状态
        else {
            for (const auto& follow : i.follows) {
                if (action_go_map[new_index].count(follow))
                    return State::ERROR;    //如果转移表该项已经有
动作,则产生多重入口,不是 LR(1)文法,报错
            }
            else
                action_go_map[new_index][follow] =
{ Action::reduction,i.p_grammar };    //用该产生式归约
        }
    }
}

//生成新 closure 集, 填写转移表
for (const auto& i : new_pset_map) {
    set<GrammarProject> NS = getClosure(i.second);    //生成新
closure 集
    int it = findSameProjectSet(NS);    //查重
    if (it == -1) {
        project_set_list.emplace_back(NS);
        action_go_map[new_index][i.first] = { Action::shift_in,
int(project_set_list.size()) - 1 }; //移进
    }
    else {
        action_go_map[new_index][i.first] = { Action::shift_in,
it };    //移进
    }
}

++new_index;    //处理下一个项目集的转移关系
}

return State::OK;
}

```

#### 4.3.4. 语法分析函数

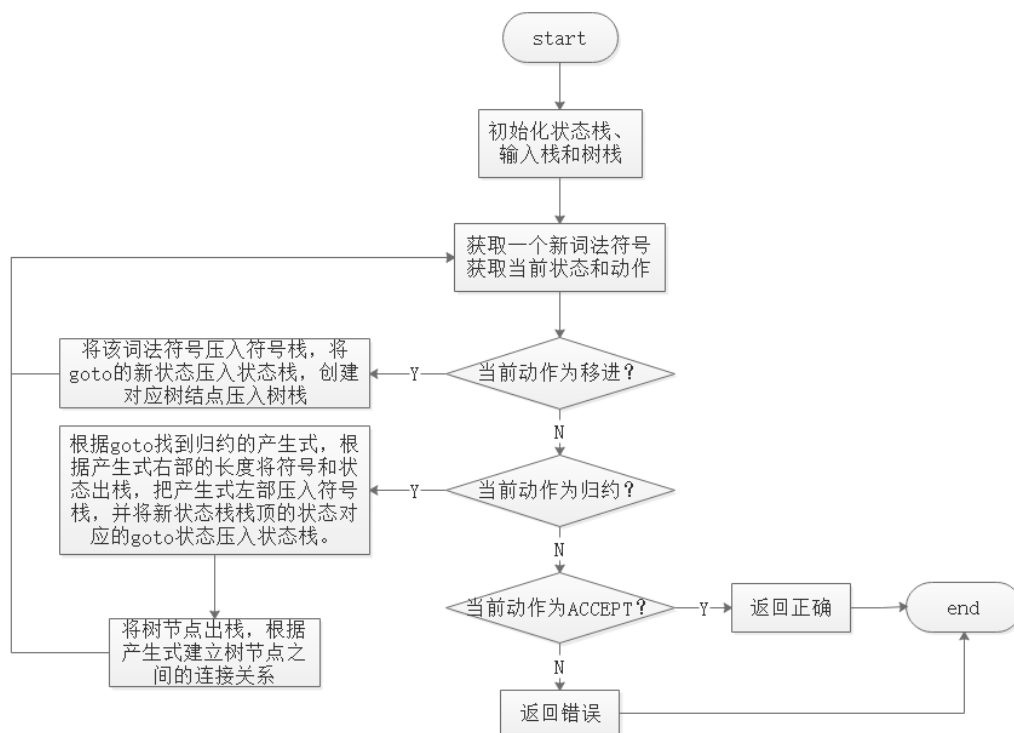
算法:

Step1: 读入源程序, 设置状态栈和符号栈, 初始栈底为#。

Step2: 使用词法分析器分析一个词, 对于该词, 查 action-goto-map 得到与其对应的行动 (action), 若为移进, 则将该词法符号压入符号栈, 并将 goto 的新状态压入状态栈; 若为归约, 则根据 goto 找到归约的产生式, 根据产生式右部的长度将固定数量的符号从符号栈出栈, 同时状态栈也出栈相同数量的状态, 最后把产生式左部压入符号栈, 并将新状态栈栈顶的状态对应的 goto 状态压入状态栈。

Step3: 重复执行 Step2, 直至扫描到文件末尾并达到 accept 状态, 或遇到错误提前退出。

流程图:



代码:

```
State LR1_Parser::parser(const char* src_path)
{
    if (!this->lexer.openFile(src_path))
        return State::ERROR;
```

```

stack<int> SStack; //状态栈
stack<Tag> TStack; //输入栈
stack<int> NStack; //树结点栈, 存放树节点下标

SStack.push(0); //初始化
TStack.push(Tag::the_end); //初始化
//NStack.push(-1); //初始化

bool use_lastToken = false; //判断是否使用上次的 token
Token t_now; //当前 token
int s_now; //当前 state
Movement m_now; //当前动作
while (true) {
    //需要新获取一个 token
    if (!use_lastToken) {
        State ret = this->lexer.getNextLexical(t_now);
        if (ret == State::ERROR)
            return ret;
    }
    s_now = SStack.top(); //获取当前状态
    if (action_go_map.count(s_now) == 0 ||
        action_go_map[s_now].count(t_now.tag) == 0) //若对应表格项为空, 则出错
        return State::ERROR;
    m_now = action_go_map[s_now][t_now.tag]; //获取当前动作
    //移进
    if (m_now.action == Action::shift_in) {
        SStack.push(m_now.go);
        TStack.push(t_now.tag);

        TNode node_in; //移进的树结点
        node_in.tag = t_now.tag; //初始化 tag 值
        node_in.p = pTree.TNode_List.size(); //指定树节点在
TNode_List 中的下标
        pTree.TNode_List.push_back(node_in); //移进树结点
        NStack.push(node_in.p); //将树节点下标移进树栈
        (保证栈内结点和 TNode_List 中的结点一一对应)

        use_lastToken = false;
    } //归约
    else if (m_now.action == Action::reduction) {
        int len = grammar_list[m_now.go].right.size(); //产生式右部
长度

```

```

TNode node_left; //产生式左部
node_left.tag = grammar_list[m_now.go].left; //产生式左部
tag
node_left.p = pTree.TNode_List.size(); //移进树结点

//移出栈
while (len-- > 0) {
    SStack.pop();
    TStack.pop();

    node_left.childds.push_front(NStack.top()); //创建子结点
链表
    NStack.pop();
}

pTree.TNode_List.push_back(node_left); //移进树栈

s_now = SStack.top(); //更新当前状态
if (action_go_map.count(s_now) == 0 ||
    action_go_map[s_now].count(node_left.tag) == 0) //若对应
表格项为空,则出错
    return State::ERROR;

m_now = action_go_map[s_now][node_left.tag]; //更新当前动
作

//入栈操作
SStack.push(m_now.go);
TStack.push(node_left.tag);
NStack.push(node_left.p);

use_lastToken = true;
}
else //接受
{
    pTree.RootNode = pTree.TNode_List.size() - 1; //根结点即为
最后一个移进树结点集的结点
    return State::OK; //accept
}
}
}

```

## 五、图形界面设计

### 5.1. 数据结构

共有两个结构体，分别为树节点和语法树，用于存储语法树。

```
struct TNode    //树结点
{
    Tag tag;    //tag 值
    list<int> childs;    //孩子结点集
};

struct PTree    //语法树
{
    vector<TNode> TNode_List;    //结点集合
    int RootNode = -1;    //根结点指针
};
```

### 5.2. 绘制 DFA 和语法树

为方便观察结果，项目包含了对于生成的项目及规范族和语法树的绘制，使用 dot 语法描述图形信息，并使用 graphviz 工具将生成的 dot 文件转为图片。绘制函数代码如下：

**打印 DFA:**

```
void LR1_Parser::printVP_DFA(ostream& out)
{
    out << "digraph{" << endl;
    out << "rankdir=LR;" << endl;
    //声明每一个项目集
    for (int i = 0; i < project_set_list.size(); i++)
    {
        out << "node_" << i << "[label=\"";
        //输出项目集中的每一个项目
        for (const auto& gp : project_set_list[i])
        {
            //输出产生式
            out << convTag2Str(grammar_list[gp.p_grammar].left) << "->";
            for (int p = 0; p < grammar_list[gp.p_grammar].right.size();
p++)
            {
                if (gp.point == p)
```

```

        out << ".";
        out << convTag2Str(grammar_list[gp.p_grammar].right[p]);
    }
    out << ", ";
    //输出 follows
    for (auto it = gp.follows.cbegin(); it != gp.follows.cend();
it++)
    {
        if (it != gp.follows.cbegin())
            out << "/";
        out << convTag2Str(*it);
    }
    out << "\n";
}
//声明结点属性
out << "\" shape=\"box\"];" << endl;
}

//声明转移关系
for (int i = 0; i < project_set_list.size(); i++)
{
    for (const auto& tag_mov : action_go_map[i])
    {
        //只有移进才会转移
        if (tag_mov.second.action != Action::shift_in)
            continue;
        else
            out << "node_" << i << "->node_" << tag_mov.second.go <<
"[label=\"\" << convTag2Str(tag_mov.first) << "\"];\" << endl;
    }
}

out << "}" << endl;
return;
}

```

打印语法树:

```

void LR1_Parser::printTree(ostream& out)
{
    if (pTree.RootNode == -1)    //没有根节点，树都不存在，没得画咯
        return;
    queue<int> Q;
    out << "digraph parser_tree{" << endl;
    out << "rankdir=TB;" << endl;

```



```

//初始化结点
for (int i = 0; i < pTree.TNode_List.size(); i++)
{
    out << "node_" << i << "[label=\\\" <<
TAG2STR.at(pTree.TNode_List[i].tag) << "\\\" ";
    out << "shape=\\\"";
    if (isVT(pTree.TNode_List[i].tag)) //终结符, 蓝色字体, 无圆框
        out << "none\\\" fontcolor=\\\"blue\\\"";" << endl;
    else //非终结符, 黑色字体, 有圆框
        out << "box\\\" fontcolor=\\\"black\\\"";" << endl;
}
out << endl;

Q.push(pTree.RootNode); //根节点入队列, 即将开始 BFS 输出语法树
while (!Q.empty())
{
    TNode node = pTree.TNode_List[Q.front()]; //取第一个结点, 对其进
行画树
    Q.pop();

    if (node.childs.size() == 0) //若无子结点, 不用画他的子树
        continue;
    //若有子结点, 则画其子树
    for (auto it = node.childs.cbegin(); it != node.childs.cend();
it++) //声明连接关系
    {
        out << "node_" << node.p << "->node_" << *it << ";" << endl;
        Q.push(*it);
    }
}

out << "}" << endl;
return;
}

```

## 六、调试分析与结果展示

测试使用的文法产生式和程序全部放在 test 目录下。

## 6.1. 调试-源程序 1

源程序:

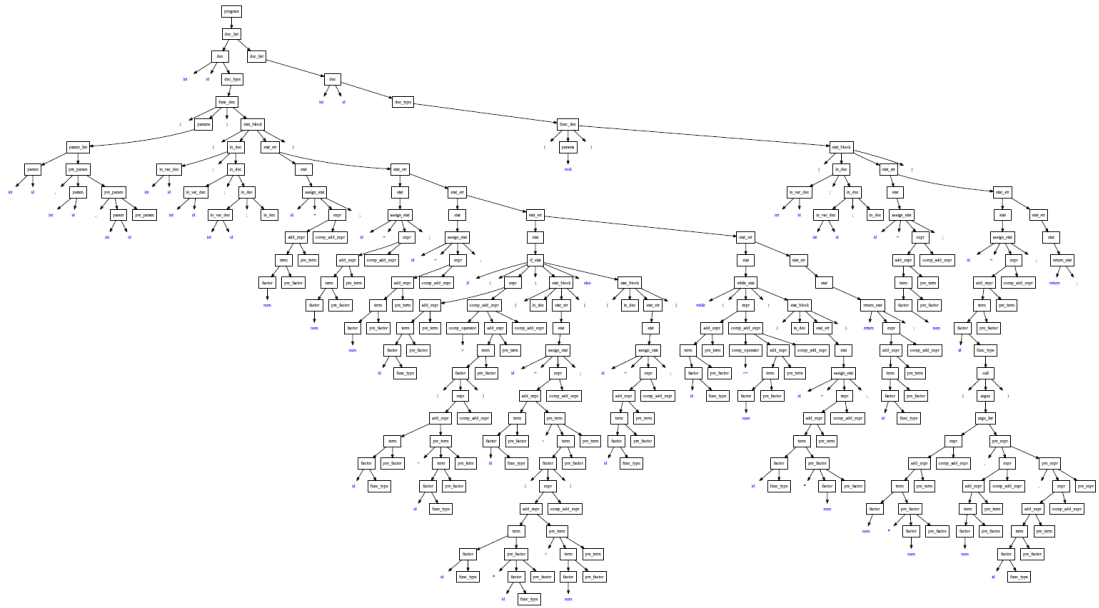
```
int program(int a,int b,int c)
{
    int i238jjsifioer923;
    int i;
    int j;
    i=012.28372;
    j=193e34;
    i238jjsifioer923=1.e6;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;  //乱写滴
    }
    return i;
}

int main(void)
{
    int fhjhrbghikehsgutrg;
    int k9384;
    fhjhrbghikehsgutrg=303;
    k9384=program(3859*34.23e3,.45868,fhjhrbghikehsgutrg);
    /*
    hello
    */
    return;
}
```

DFA: (部分)



语法树:



## 6.2. 调试-源程序 2

源程序:

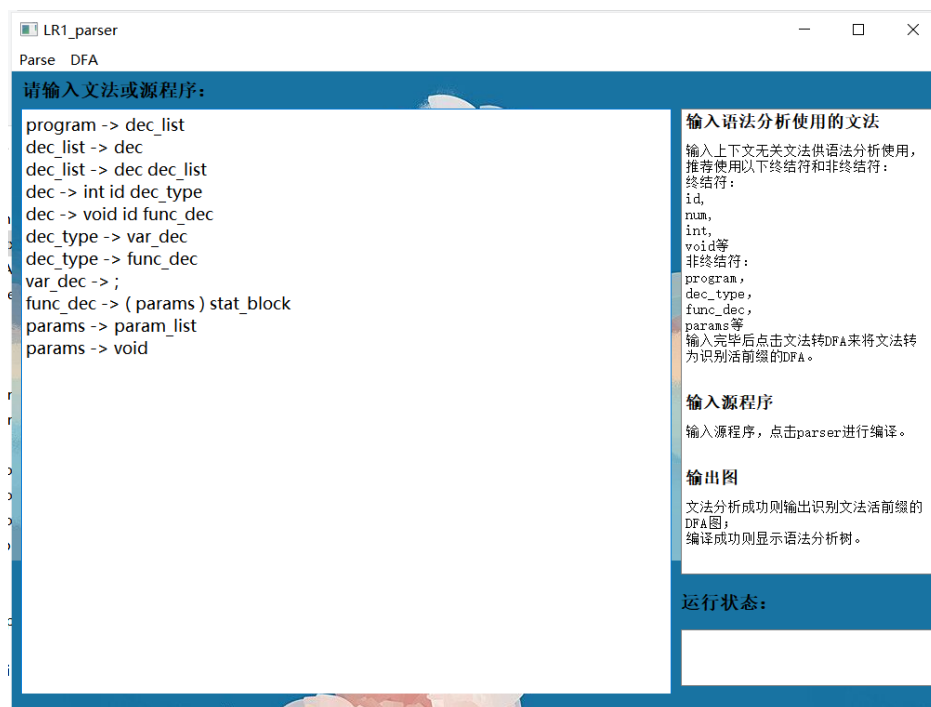
```
int a;
int b;
int program(int a, int b, int c)
{
    int i;
    int j;
    i = 0;
    if (a > (b + c))
    {
        j = a + (b * c + 1);
    }
    else
    {
        j = a;
    }
    while (i <= 100)
    {
        i = j * 2;
    }
    return i;
}
int demo(int a)
{
    a = a + 2;
```



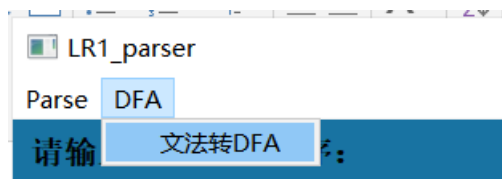


按照右侧的提示信息，在编辑区输入文法或源程序，点击相应的按钮即可进行文法载入和源程序语法分析。右下角有运行状态栏输出运行状态和报错信息。

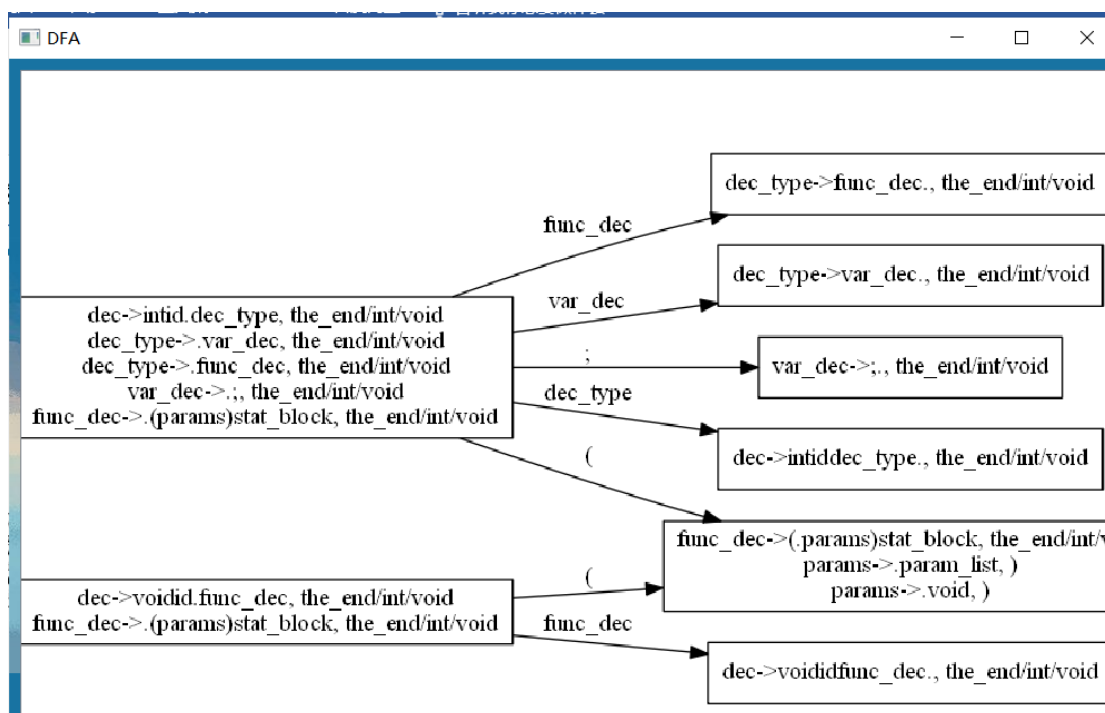
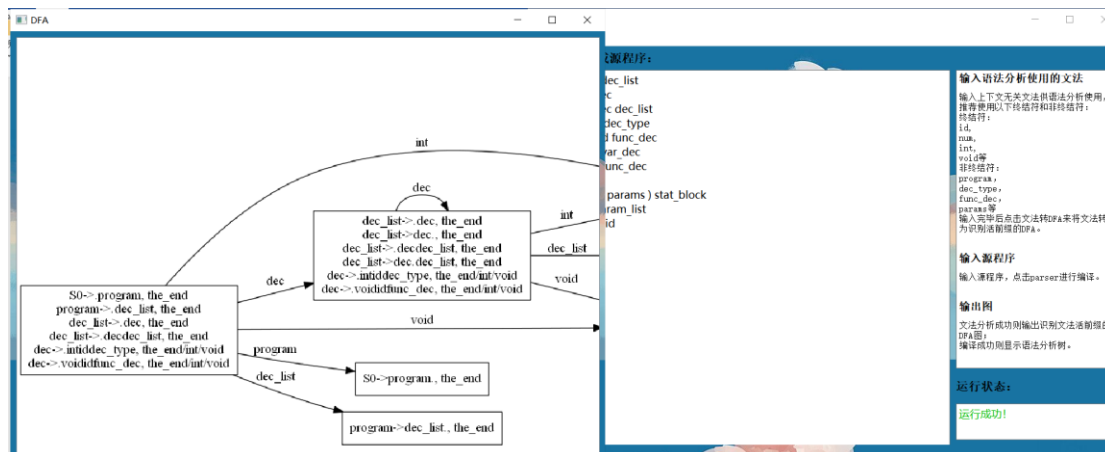
输入正确文法：



点击文法转 DFA 选项：



运行结果:

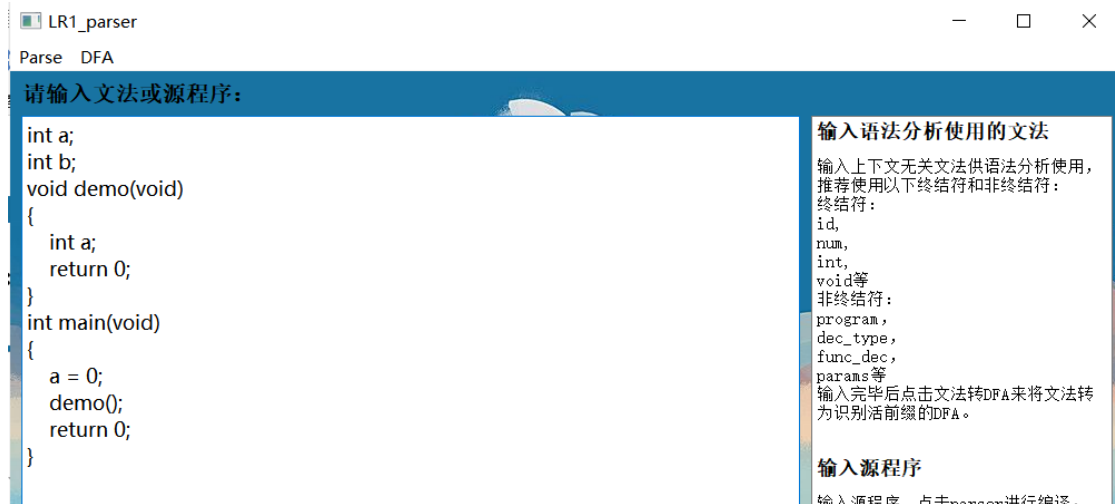


输入错误文法:

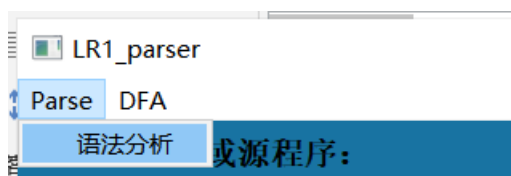
输入错误文法，状态框会显示文法载入错误。



输入正确源程序：

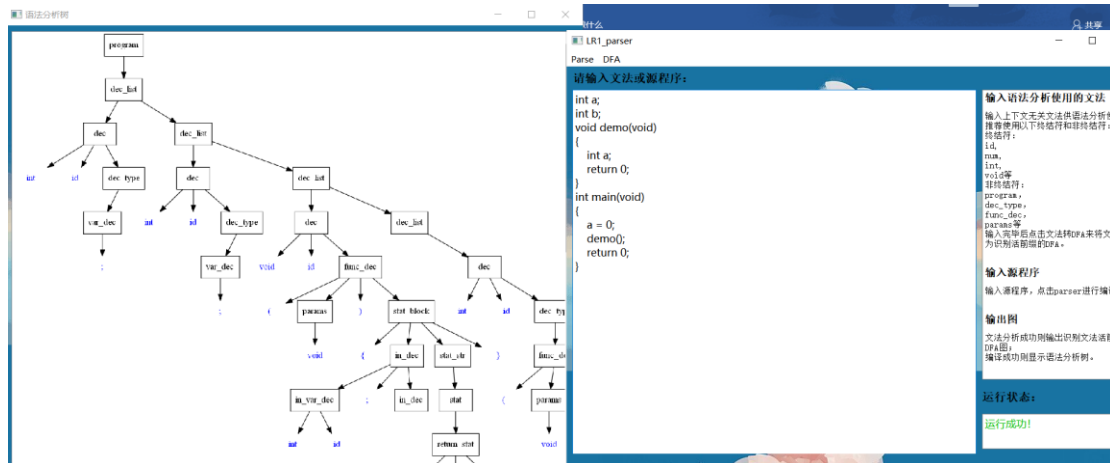


点击 parser 语法分析选项：

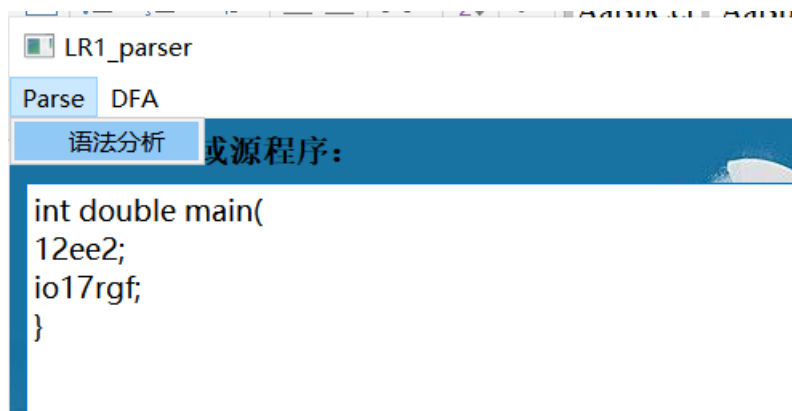


运行结果：





输入错误的源程序:



运行结果:



## 七、程序设计过程中的新思考

### 7.1. 非终结符和终结符的表示方法

程序中，使用 `enum class` 枚举类型来表示非终结符和终结符。这种表示方法可以使得程序在进行词法、语法分析时便于统一，并且在存储文法产生式等数据时便于节省内存。特别的，`enum class` 设置 `vtnboundary` 一项来分隔终结符和非终结符，使得二者的区分变得很简单。

使用 `enum class` 枚举类型也有其弊端。由于 `enum class` 必须在代码中提前给定，故本程序的输入文法的范围实际上仅支持代码中给定的终结符和非终结符。但是因为语言的文法规则一般不会随便改变，并且在代码中进行修改也不困难，故不影响正常使用。

### 7.2. 词法单元的属性值存储方式

一个词法单元的属性值可能有多种类型。如 123 为 `int`，123.45 为 `double`，“hello”为 `string`。如果在结构体中为每种类型都设定一个属性值存储单元，显然很浪费空间。程序中的处理方法是所有值都存储在 `string` 中。在语义分析时，可以按需将该内容转换为相应的类型。

### 7.3. LR(1)项目都存储方式

由于每个 LR(1)项目都依赖于一条文法产生式，而文法产生式的右部为 `vector`，故若为每个项目都存储一次产生式显然很浪费空间，而且没有必要。

程序中，所有文法产生式都存储在语法分析器类中的一个 `vector` 中。每个项目中有一个 `int` 指针，该指针指向项目对应的产生式的下标。这样一来，每条产生式仅存储了一次，减少了信息冗余。

### 7.4. epsilon 产生式的特殊处理

输入文法中有很多空产生式。如果直接引入终结符 `epsilon` 会导致一个问题： $A \rightarrow \epsilon$  这条产生式可能会导出  $A \rightarrow \cdot \epsilon$  和  $A \rightarrow \epsilon \cdot$  这种没有意义的项目，导致归约过程出现问题。

程序中的处理方式是，对于  $A \rightarrow \epsilon$  的产生式，不引入 `epsilon` 符号，直接记为  $A \rightarrow \cdot$ 。这条产生式右部长度为 0，仅生成  $A \rightarrow \cdot$  这一条归约项目，故可以

在合适时候进行归约。在求 First 集时，再引入  $\epsilon$  这个终结符。

## 7.5. 语法树的存储和建立过程

由于该语法分析器使用自底向上的 LR(1)分析方法，因此语法分析树的建立过程也是自底向上的。考虑到每次归约需要连接的结点需要由归约产生式以及符号栈和状态栈内的内容来确定，因此设计以下语法树存储和建立方式：设置树结点数组，以下标作为区分各结点的唯一标识；建立树结点栈，与符号栈和状态栈并列，在语法分析的过程中对树栈进行如下操作：

若本次行动为移进，则建立新树节点，其 tag 值为移进的词法符号，并将其压入树栈中（保证树栈与符号栈之间的一一对应关系）；

若本次行动为归约，则建立一个新树结点，其 tag 值为产生式左部词法符号，然后对符号栈进行出栈的同时，对树栈也进行相同的操作，并将出栈的树结点装入产生式左部结点的子结点数组中。最后将新建的结点（对应产生式左部）压入树栈（保证树栈与符号栈之间的一一对应关系）；

若本次行动为接受，则语法树已建立完成，将语法树树结点数组的最后一个元素（即最后一个压入树栈的词法符号，其一定是文法的初始符号）的下标赋给语法树的根结点指针，这样语法树就建成了。

# 八、总结与收获

## 8.1. 项目总结

本次项目中，我们小组实现了基于正则文法的词法分析器和基于 LR(1)分析法的语法分析器，可以用于对类 C 程序进行词法和语法分析，并输出分析结果，绘制源程序对应的语法树。

在词法分析器方面，我们增加了浮点数的识别，可以识别 C 语言允许的所有浮点数表示方式，如 12.34、12.、.34、12.3e9 等。

在语法分析器方面，我们实现了一个类似 YACC 的程序，支持输入文法产生式并自动生成对应的语法分析器。

此外，我们还实现了一个图形化界面，支持图形界面输入源程序和文法产生式，在构建语法分析器完成后输出识别活前缀的 DFA，对源程序进行语法分析后输出语法树的矢量图或报错。

## 8.2. 小组分工与贡献

李强：主要负责设计文法符号相关的数据结构，构建类 C 语言文法产生式，构建词法分析器，设计语法分析器数据结构，完成语法分析器的部分实现，并完成部分实验报告。

高志成：主要负责设计语法分析器和语法树的数据结构，完成语法分析器的大部分实现，完成语法树的构建和输出，设计实现整个图形界面，并完成部分实验报告。

小组贡献：李强 50%，高志成 50%。

## 8.3. 项目收获

在本次项目的完成过程中，我们收获了很多经验。

在课程本身方面，通过这次项目的完成，我们对编译原理这门课程有了更深刻的理解和认识。在初学这门课程时，我们实际上并不清楚词法分析器和语法分析器各自的功能以及联系，对于一些概念如 First 集、Follow 集等等的学习也仅仅停留在学习算法的层面，却不太明白这些概念的引入到底是为了什么。在项目设计过程中，我们充分复习了词法分析器设计、正则文法分析、First 集计算算法、LR(1)分析方法等等知识点，特别的，我们认识到了编译器到底一步步地在做什么，由此将这些知识点全部串接在了一起。

在复杂项目开发方面，我们充分认识到，好的设计可以使得项目的完成事半功倍。实际上本次项目并不是特别复杂，难点便在于数据结构的设计。我们小组在开始写代码之前仔细构想了各模块的数据结构设计，故在动手的时候减少了许多麻烦。

在小组合作方面，我们也获得了一些经验。在项目的开始阶段，做好任务规划和分工是很有必要的。每个人任务明确，可以使冗余工作量达到最小，并使项目并行推进。同时，在小组合作时的一件非常重要的事是让队友知道你做了哪些工作。我们小组仅含两位成员，故时常在一起交流进度和想法。

总而言之，本次项目开发不仅帮助我们复习、强化了编译原理的相关知识，并且锻炼了我们的设计和思考能力，也锻炼了动手编程、小组合作等等综合方面的能力。

## 九、参考文献

- [1]陈火旺,刘春林.程序设计语言编译原理[M].北京:国防工业出版社,2000.1
- [2]Alfred V. Aho ; Monica S. Lam ; Ravi Sethi ; Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools[M].China Machine Press,2011