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\信息安全

组　员 1 2152118　史君宝

组　员 2 2154062　赵书玥

组　员 3 2151638　林天野

授课老师 丁志军

目录

同济大学计算机系 1
词法和语法分析工具设计与实现 实验报告 1
院　　系　　电子与信息工程学院 1
专　　业　　计算机科学与技术\信息安全 1
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.　是否通过语法分析 4
3.　若无法通过语法分析，输出报错信息以及出错位置 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
读入的·文本或文件为ASCII码，首先需要判断是数字、英文字符或其他字符，读入下一个字符并创建容器存储字符串，读取之后的字符内容直到读到文件尾或其他字符，最后对其他字符进行分类讨论处理判断直至所有文本或文件读完，完成词法分析的功能。 7
2.2.3. 语法分析器类型定义 7
2.3. 主程序流程图 10
三、词法分析器设计方法 10
3.1. 词法分析器的任务 10
3.2. 词法分析方法 11
数值：数字（数字）* 11
3.3. 词法分析器的输出 12
四、语法分析器设计方法 13
4.1. LR(1)分析方法介绍 13
4.2. LR(1)分析表构造方法 14
2.　由产生式计算各非终结符的First 集，为后续做准备 14
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 14
1.　I 的任何项目都包含在CLOSURE(I)中 14
3.　重复执行该步骤，直到CLOSURE(I)不再增加 14
4.3. 关键函数的实现方法 14
4.3.1. 求FIRST集函数 14
4.3.2. 求项目集规范族函数 17
4.3.3. 生成识别活前缀的DFA函数 19
4.3.4. 语法分析函数 23
五、图形界面设计 27
5.1. 数据结构 27
5.2. 绘制DFA和语法树 28
while (!Q.empty()) 29
六、调试分析与结果展示 30
6.1. 调试-源程序1 31
DFA：(**部分** 31
6.2. 调试-源程序2 34
6.3.　结果展示： 38
七、总结与收获 41
7.1. 项目总结 41

7.2. 项目收获 42

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，完成对不含过程调用的类 C 示例程序的词法和语法分析，并输出分析结果。

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。

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 若干 test.txt 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

词法分析器可以实现扫描高级语言编写的源程序，将源程序中由特定的“单词符号”组成的字符串进行分解，使得语法分析部分可以规避源程序不同格式造成的影响，使其对源程序进行快速的分析。

在词法分析部分，程序读入源程序文件，按照词法规则对源程序进行整理，使其成为语法分析其可以更方便处理的串。

2.2. 数据结构定义

2.2.1. 文法非终结符和终结符的定义

在程序中对文法符号使用枚举类型进行定义，可以节省内存，提高效率。在unit_type枚举类中存储所需要的终结符和非终结符，并对二者进行区分。

对语法单元的定义如下，以下仅展示部分：

```
enum class Unit_type
{
    //key_word 关键词

    //定义类型关键词
    key_int,            //int类型
    key_double,         //double类型
    key_char,           //char类型
    key_void,           //void类型

    //定义语句关键词
    key_if,             //if关键词
    key_else,           //else关键词
    key_for,            //for关键词
    key_while,          //while关键词
    key_include,        //include关键词
    key_main,           //main关键词
    key_return,         //return关键词
    key_cin,            //cin关键词
    key_cout,           //cout关键词


    //Terminal 终结符

    //终结符类型
    variable,           //变量
    number,             //数字，整数或小数
    the_end,            //#，终止符
    epsilon,            //空符号


    //符号终结符
    symbol_add,         //符号‘+’
    symbol_sub,         //符号‘-’
    symbol_mul,         //符号‘*’
    symbol_div,         //符号‘/’
…
};
```

2.2.2. 词法分析器类型定义

词法分析器类为Lexer，使用analyse_file打开或关闭待分析的源程序文件。getNextchar是词法分析器的核心，调用后可以返回下一个词法单元。其中cur_char是当前读到的字符，next_char是预先看到字符，函数作用是将next_char赋给cur_char，然后next_char变为新看到的字符，实现前瞻，便于对具有相同前缀的词法单元的识别和判断。

在读入时要注意对于非法字符的判断，新字符不是非法的时候才可以读入。即使读到了文件尾新字符是非法的，仍旧要进行错误报告。

词法分析器的类型定义如下：

```
class Lexer_program
{
private:

    //用两个标志变量，来标志读入字符的正确性
    bool cur_flag;
    bool next_flag;

    //记录读到的单词
    char curchar;
    char nextchar;

    ifstream file_in;


public:

    //记录当前读到的位置
    int line;
    int col;


    Lexer_program();
    ~Lexer_program();

    void Lexer_error();

    bool analyse_file(string filename);

    void clear_word();


    bool getNextchar();
    word_unit getNext_word_unit();

};
```

读入的·文本或文件为ASCII码，首先需要判断是数字、英文字符或其他字符，读入下一个字符并创建容器存储字符串，读取之后的字符内容直到读到文件尾或其他字符，最后对其他字符进行分类讨论处理判断直至所有文本或文件读完，完成词法分析的功能。

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。

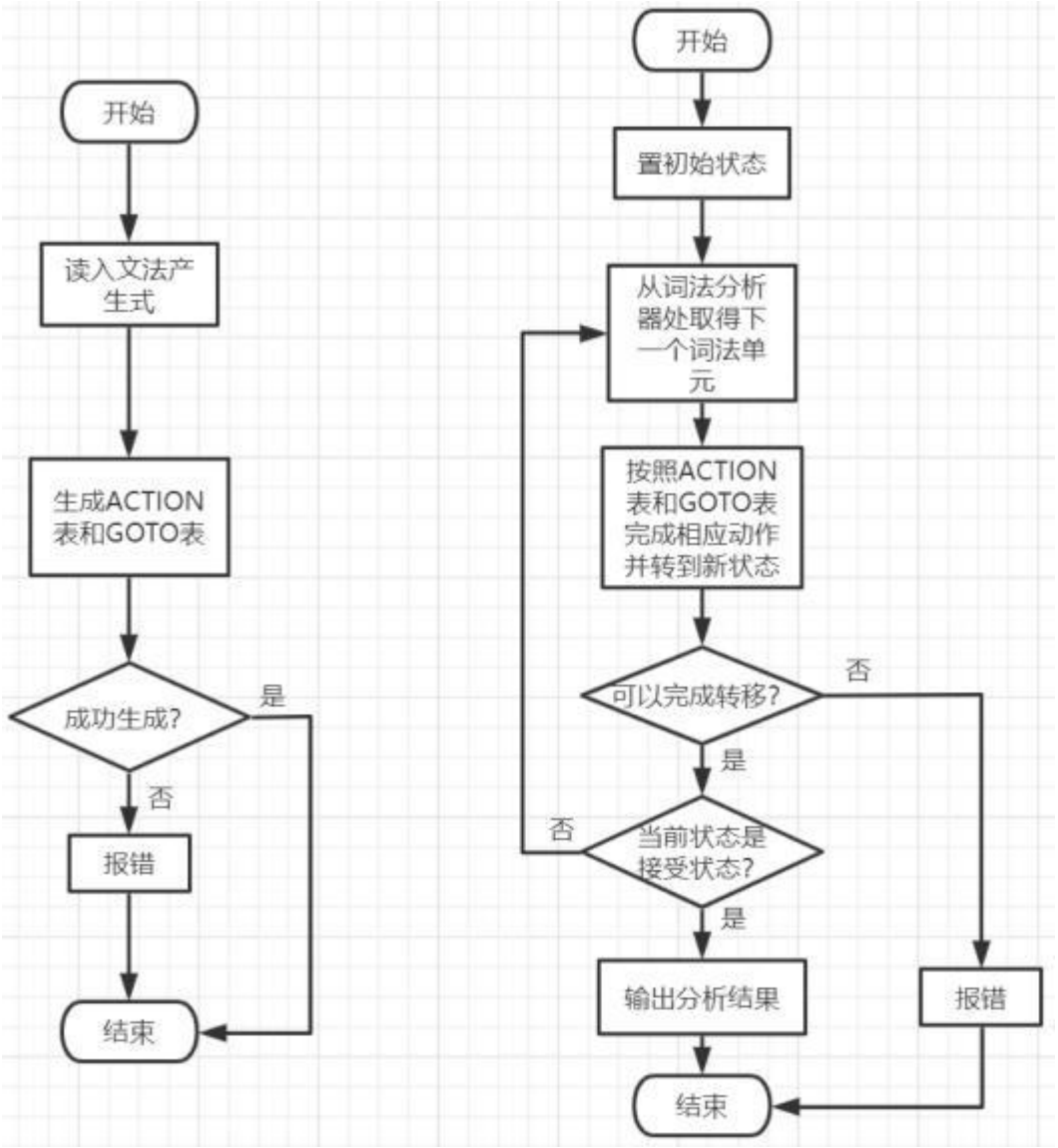
语法分析器类型定义如下：

```
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。

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

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，

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。

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。

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：

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

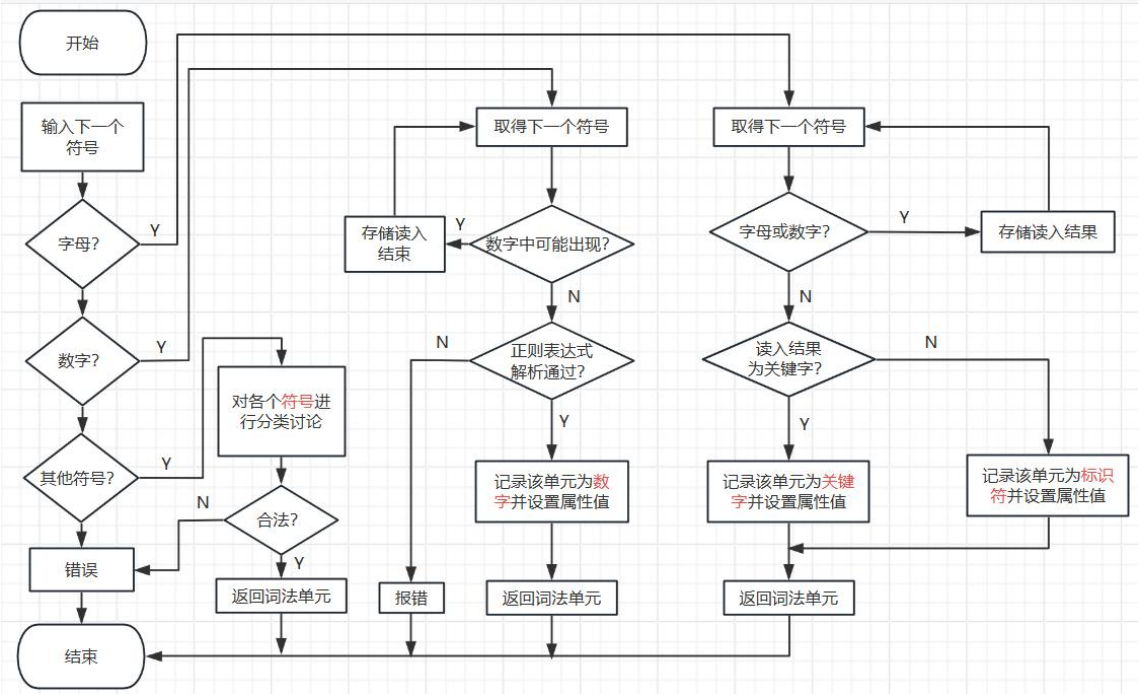
结束符：#

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。

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。

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，向前多读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， 故这种多读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 。

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' -> 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' ->·S，# 的 CLOSURE 集合，作为初始项目集 I0
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->α·Bβ,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->·γ,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，直至某轮遍历后左右数量相等。

代码：

```
void Parser_program::Init_First_list()
{
    //我们要求first集
    //对于终结符，它们的first集为自身
    for(int i = 0; i < int(Unit_type::board); ++i)
    {
        first_list[Unit_type(i)] = {Unit_type(i)};
    }

    //之后是非终结符的求法
    vector<int> right_isVN;    //右边是非终结符

    //遍历整个语法产生式集合，将上面分类
    for(int i = 0; i < int(Grammar_list.size()); ++i)
    {
        //首先考虑右边是否是的
        if(isVT(Grammar_list[i].right[0]))

first_list[Grammar_list[i].left].insert(Grammar_list[i].right[0]);
        else
            right_isVN.push_back(i);
    }



    //一次按顺序更新可能不正确，所以需要多次扫描
    while(1)
    {
        //定义first和last用于最后出去
        int first = 0;
        int last = 0;

        //扫描前先确定生成式左侧非终结符的first中元素数量
        for(int i = 0; i < int(right_isVN.size()); ++i)
        {
            int j = right_isVN[i];
            first += first_list[Grammar_list[j].left].size();
        }
```

```
        for(int i = 0; i < int(right_isVN.size()); ++i)
        {
            //右边的首字母是终结符，要进行考虑
            //如果右边首字符无epsilon，则将其first集加入左边的
            int j = right_isVN[i]; //帮助储存

            //对产生式右侧的每个元素都判断
            for (const auto& atom : Grammar_list[j].right)
            {
                //如果是非终结符
                if(isVN(atom))
                {
                    //如果其中包含epsilon，则将其中的加入后看下一个
                    if(first_list[atom].count(Unit_type::epsilon))
                    {
                        for (const auto& atom_atom : first_list[atom])
                        {

first_list[Grammar_list[j].left].insert(atom_atom);
                        }
                        continue;
                    }
                    //如果其中不包含epsilon，则将其中的加入退出
                    else
                    {
                        for (const auto& atom_atom : first_list[atom])
                        {

first_list[Grammar_list[j].left].insert(atom_atom);
                        }
                        break;
                    }
                }
                //如果不是终结符就将其加入后退出
                else
                {
                    first_list[Grammar_list[j].left].insert(atom);
                    break;
                }

            }
        }

        //扫描后再确定生成式左侧非终结符的first中元素数量
        for(int i = 0; i < int(right_isVN.size()); ++i)
        {
            int j = right_isVN[i];
```

```
            last += first_list[Grammar_list[j].left].size();
        }

        //如果二者一样，说明不再更新了，可以退出扫描
        if(first == last)
            break;
    }

    return;
}
```

4.3.2. 求项目集规范族函数

算法：

Step1: 对于给的的项目集，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→a·Bβ属于 CLOSURE(I)，那么，对任何关于 B 的产生式 B→Y，项目 B→·Y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<LR1_process> Parser_program::getClosure
(const set<LR1_process>& project_process)
{
    set<LR1_process> Closure(project_process);          //project_set自身的所有项目都在闭包中
    set<LR1_process> help_project(project_process); //辅助集合
    set<LR1_process> new_project;

    bool flag;
    while (true) {
        flag = false;

        for (const auto& atom : help_project) { //扫描上一次产生的所有项目
            if (Grammar_list[atom.Grammar_num].right.size() >
atom.point_addr &&

isVN(Grammar_list[atom.Grammar_num].right[atom.point_addr]))
            {
                //A->α.Bβ型
                Unit_type VN =
Grammar_list[atom.Grammar_num].right[atom.point_addr];

                //求出first(βa)
                set<Unit_type> first_BA;
                if (atom.point_addr + 1 <
Grammar_list[atom.Grammar_num].right.size())
                {
                    first_BA =
first_list[Grammar_list[atom.Grammar_num].right[atom.point_addr + 1]];
                    auto p = first_BA.find(Unit_type::epsilon);
```

```
                    if (p != first_BA.cend()) {
                        //如果含有epsilon,则删除epsilon并把原项目的follows
加入
                        first_BA.erase(p);
                        for (const auto& foll : atom.follow)
                            first_BA.insert(foll);
                    }
                }
                else {
                    for (const auto& foll : atom.follow)
                        first_BA.insert(foll);
                }

                for (int k = 0; k < Grammar_list.size(); k++) {
                    //扫描所有B->γ型的产生式
                    if (Grammar_list[k].left == VN) {

                        //若CLOSURE中不存在{B->γ,firstba},则加入
                        bool have = false;
                        for (auto it = Closure.begin(); it !=
Closure.end(); ++it) {
                            if (it->Grammar_num == k && it->point_addr
== 0) {
                                //项目在集合
                                have = true;
                                if (it->follow != first_BA) {
                                    //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 ngp = *it;
                                    for (Unit_type atom_first_BA :
first_BA)
                                        ngp.follow.insert(atom_first_BA);
                                    Closure.erase(it);
                                    Closure.insert(ngp);
                                    new_project.insert(ngp);
                                }
                                break;
                            }
                        }
                        if (!have) {
                            //否则插入新项目
                            flag = true;
                            Closure.insert({ k,0,first_BA });
                            new_project.insert({ k,0,first_BA });
                        }
                    }
                }
            }
        }
        if (!flag)    //不再增加,则返回
            break;
        help_project = new_project;    //对新添加项目进行下一轮扫描
        new_project.clear();
    }
```

```
    return Closure;
}
```

4.3.3. 生成识别活前缀的 DFA 函数

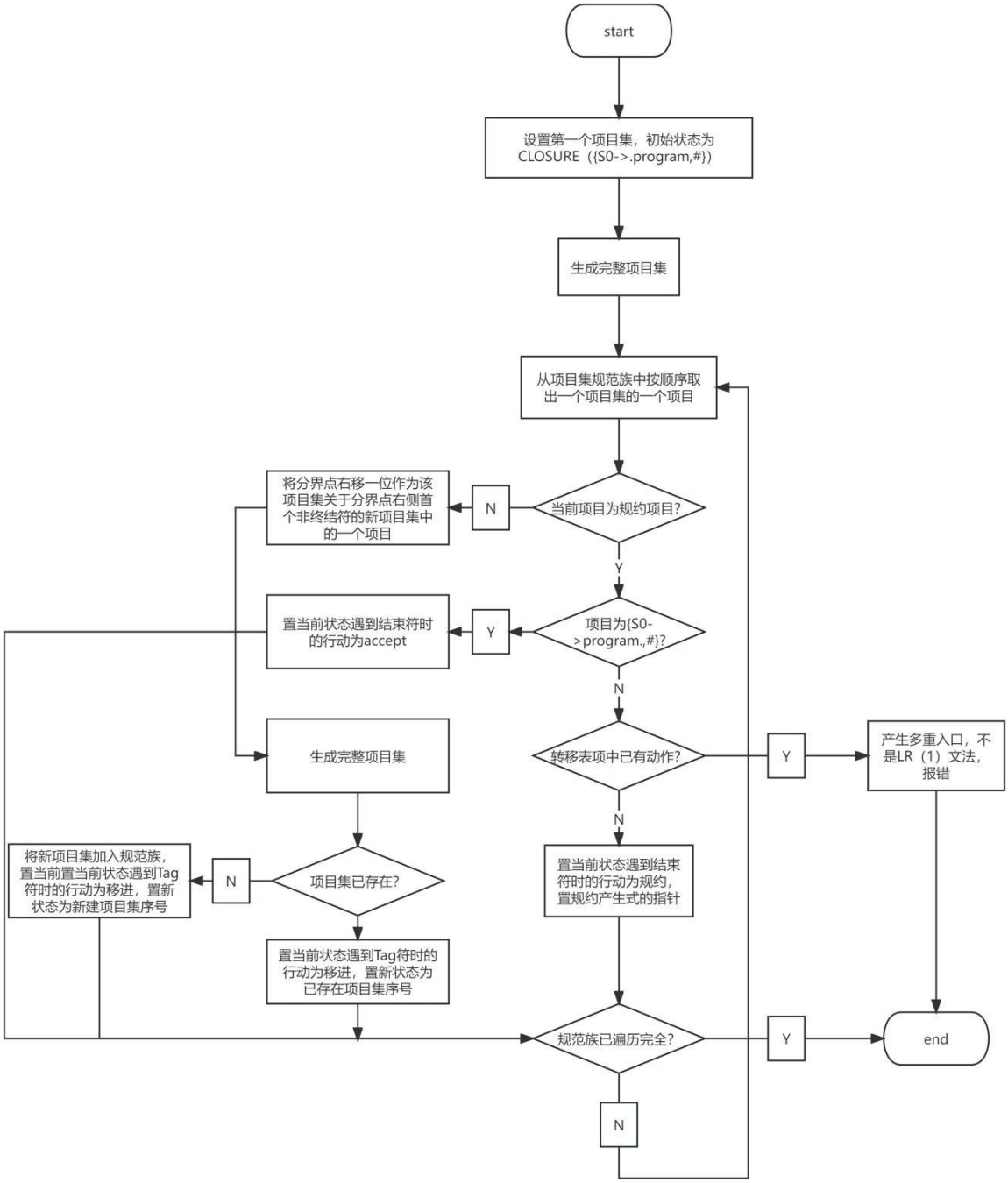
算法：

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＝{任何形如 A→CX·β的项目|A→C ·Xβ属于 I} 。直观上说，若 I 是对某个活前缀 Y有效的项目集，那么， GO(I，X)便是对 YX 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：


start
设置第一个项目集，初始状态为CLOSURE（{S0->.program,#}）
生成完整项目集
从项目集规范族中按顺序取出一个项目集的一个项目
当前项目为规约项目?
N
将分界点右移一位作为该项目集关于分界点右侧首个非终结符的新项目集中的一个项目
Y
项目为{S0->program.,#}?
Y
置当前状态遇到结束符时的行动为accept
N
转移表项中已有动作?
Y
产生多重入口，不是LR（1）文法，报错
N
置当前状态遇到结束符时的行动为规约，置规约产生式的指针
生成完整项目集
项目集已存在?
N
将新项目集加入规范族，置当前置当前状态遇到Tag符时的行动为移进，置新状态为新建项目集序号
置当前状态遇到Tag符时的行动为移进，置新状态为已存在项目集序号
规范族已遍历完全?
Y
end
N

代码：

```
int Parser_program::initAction_Goto()
{
    project_process_list.clear();
    //初始状态为CLOSURE({S0->.program,#})
    set<LR1_process> time_process_set;
    LR1_process time_process;
    time_process.Grammar_num = 0;
    time_process.point_addr = 0;
    time_process.follow = {Unit_type::the_end};
    time_process_set.insert(time_process);
    project_process_list.push_back(getClosure(time_process_set));
    int pro_num = 0;      //新项目集下标
    for(pro_num = 0; pro_num < int(project_process_list.size());
++pro_num) {
        set<LR1_process>& process_pro_num =
project_process_list[pro_num];
        map<Unit_type, set<LR1_process>> new_process_map;        //当前
项目集可以产生的新项目集
        //扫描所有项目
        for (const auto& atom : process_pro_num) {
            if (atom.point_addr <
Grammar_list[atom.Grammar_num].right.size()) {
                //不是归约项目
                LR1_process A_project;
                A_project.Grammar_num = atom.Grammar_num;
                A_project.point_addr = atom.point_addr + 1;
                A_project.follow = atom.follow;

new_process_map[Grammar_list[atom.Grammar_num].right[atom.point_addr]].i
nsert(A_project);
            }
            else {
                //是归约项目
                if (atom.Grammar_num == 0 && atom.point_addr == 1 &&
atom.follow.size() == 1
                        && *atom.follow.cbegin() == Unit_type::the_end)
                {
                    Movement A_movement;
                    A_movement.action = Action::accept;
```

```
                    A_movement.go = atom.Grammar_num;
                    Action_goto[pro_num][Unit_type::the_end] =
A_movement;  //可接受状态
                }
                else {
                    for (const auto& foll : atom.follow) {
                        if (Action_goto[pro_num].count(foll))
                            return 0;  //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
                        {
                            Movement B_movement;
                            B_movement.action = Action::reduction;
                            B_movement.go = atom.Grammar_num;
                            Action_goto[pro_num][foll] = B_movement; //
用该产生式归约
                        }
                    }
                }
            }
        }
        //生成新closure集，填写转移表
        for (const auto& atom : new_process_map) {
            set<LR1_process> new_Closure = getClosure(atom.second);  //
生成新closure集
            int a = findSameProjectSet(new_Closure);                 //
查重
            if (a == -1) {
                project_process_list.push_back(new_Closure);
                Movement C_movement;
                C_movement.action = Action::shift_in;
                C_movement.go = int(project_process_list.size()) - 1;
                Action_goto[pro_num][atom.first] = C_movement;   //移进
            }
            else {
                Movement D_movement;
                D_movement.action = Action::shift_in;
                D_movement.go = a;
                Action_goto[pro_num][atom.first] = D_movement;   //移进
            }
```

```
        }
    }
    return 1;
}
```

4.3.4. 语法分析函数

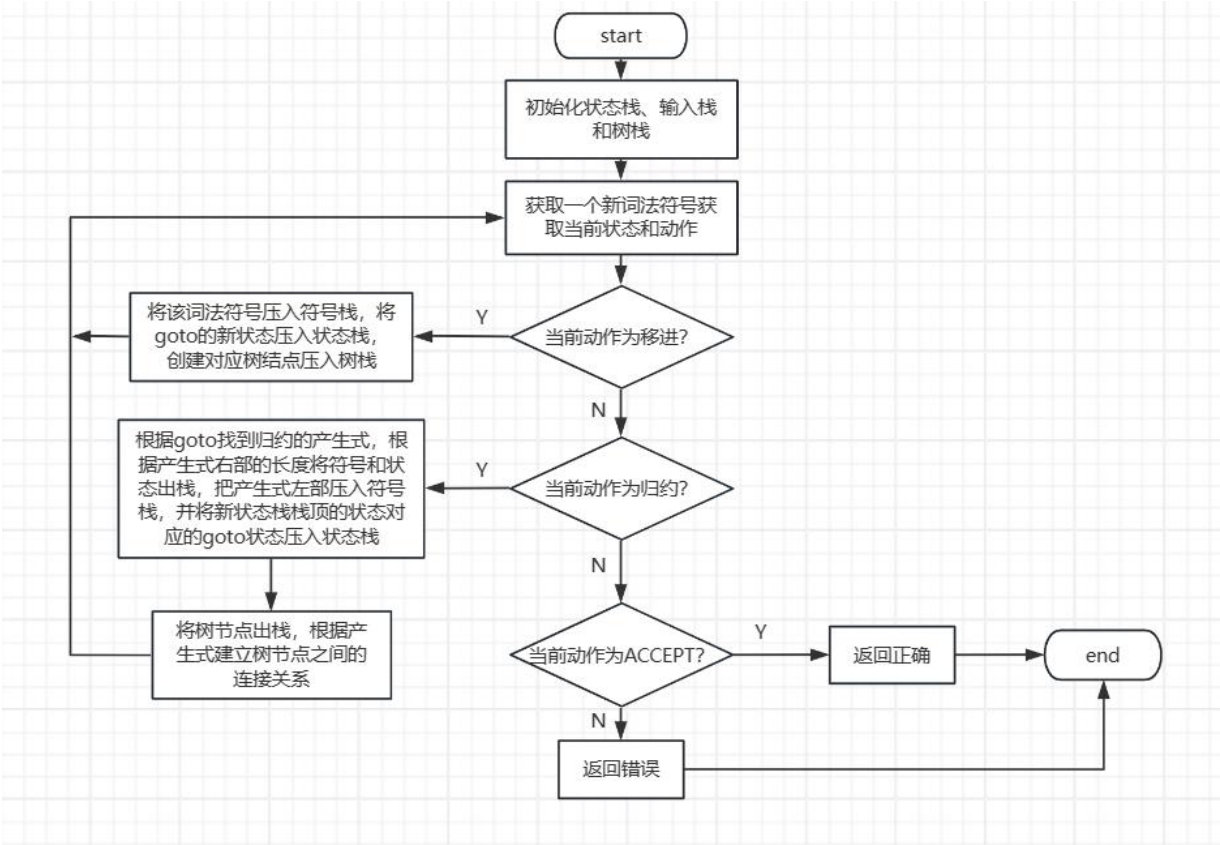
算法：

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：

```
//语法分析函数，对源程序进行分析
int Parser_program::parser(const char* filename, word_unit&
err_word_unit)
{
```

```
    lexer_program.line = 1;
    lexer_program.col = 0;

    if (!this->lexer_program.analyse_file(filename))
        return 0;

    stack<int> StateStack;//状态栈
    stack<Unit_type> InputStack;   //输入栈
    stack<int> NStack;  //树结点栈，存放树节点下标

    StateStack.push(0);        //初始化
    InputStack.push(Unit_type::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
    word_unit now_word_unit;   //当前token

    int now_state;        //当前state
    Movement now_movement;//当前动作

    while (true) {
        //需要新获取一个token
        if (!use_lastToken) {
            err_word_unit = this->lexer_program.getNext_word_unit();

            qDebug()<<QString::fromStdString(err_word_unit.unit_value);

            if (err_word_unit.unit_state == Error)
            {
                qDebug()<<"5";
                return 0;
            }
        }
        now_state = StateStack.top();                          //获取当前状
态

        if (Action_goto.count(now_state) == 0 ||
Action_goto[now_state].count(now_word_unit.unit_type) == 0) {
            //若对应表格项为空,则出错


            err_word_unit = now_word_unit;
            qDebug()<<"6";

            return 0;
        }
        now_movement = Action_goto[now_state][now_word_unit.unit_type];
    //获取当前动作
            //移进


        if (now_movement.action == Action::shift_in) {
            StateStack.push(now_movement.go);
            InputStack.push(now_word_unit.unit_type);

            TNode node_in;//移进的树结点
            node_in.tag = now_word_unit.unit_type;   //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 (now_movement.action == Action::reduction) {

            int len = Grammar_list[now_movement.go].right.size(); //产生
式右部长度

            TNode node_left;                                    //产生式左部
            node_left.tag = Grammar_list[now_movement.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) {
                StateStack.pop();
                InputStack.pop();

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

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

            now_state = StateStack.top();   //更新当前状态
            if (Action_goto.count(now_state) == 0 ||
                Action_goto[now_state].count(node_left.tag) == 0) {
                //若对应表格项为空,则出错



                err_word_unit = now_word_unit;
                return 0;
            }

            now_movement = Action_goto[now_state][node_left.tag]; //更新
当前动作
            //入栈操作
            StateStack.push(now_movement.go);
            InputStack.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 0;     //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

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 Parser_program::Parser_DFA(ostream& out)
{
    out << "digraph{" << endl;
    out << "rankdir=LR;" << endl;
    //声明每一个项目集
    for (int i = 0; i < project_process_list.size(); i++)
    {
        out << "node_" << i << "[label=\"";
        //输出项目集中的每一个项目
        for (const auto& atom : project_process_list[i])
        {
            //输出产生式
            out << GetStringFromUnitType(Terminal_to_Unit_type,
Grammar_list[atom.Grammar_num].left) << "->";
            int p;
            for (p = 0; p < Grammar_list[atom.Grammar_num].right.size();
p++)
            {
                if (atom.point_addr == p)
                    out << ".";
                out << GetStringFromUnitType(Terminal_to_Unit_type,
Grammar_list[atom.Grammar_num].right[p]);
            }
            if (atom.point_addr == p)
                out << ".";
            out << ", ";
            //输出follows
            for (auto it = atom.follow.cbegin(); it !=
atom.follow.cend(); it++)
            {
                if (it != atom.follow.cbegin())
                    out << "/";
                out << GetStringFromUnitType(Terminal_to_Unit_type,*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_process_list.size(); i++)
    {
        for (const auto& tag_mov : Action_goto[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=\"" << GetStringFromUnitType(Terminal_to_Unit_type,
tag_mov.first) << "\"];" << endl;
        }
    }

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

打印语法树：

```
void Parser_program::Parser_PTree(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 < int(pTree.TNode_List.size()); i++)
    {
        out << "node_" << i << "[label=\"" <<
GetStringFromUnitType(Terminal_to_Unit_type, 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

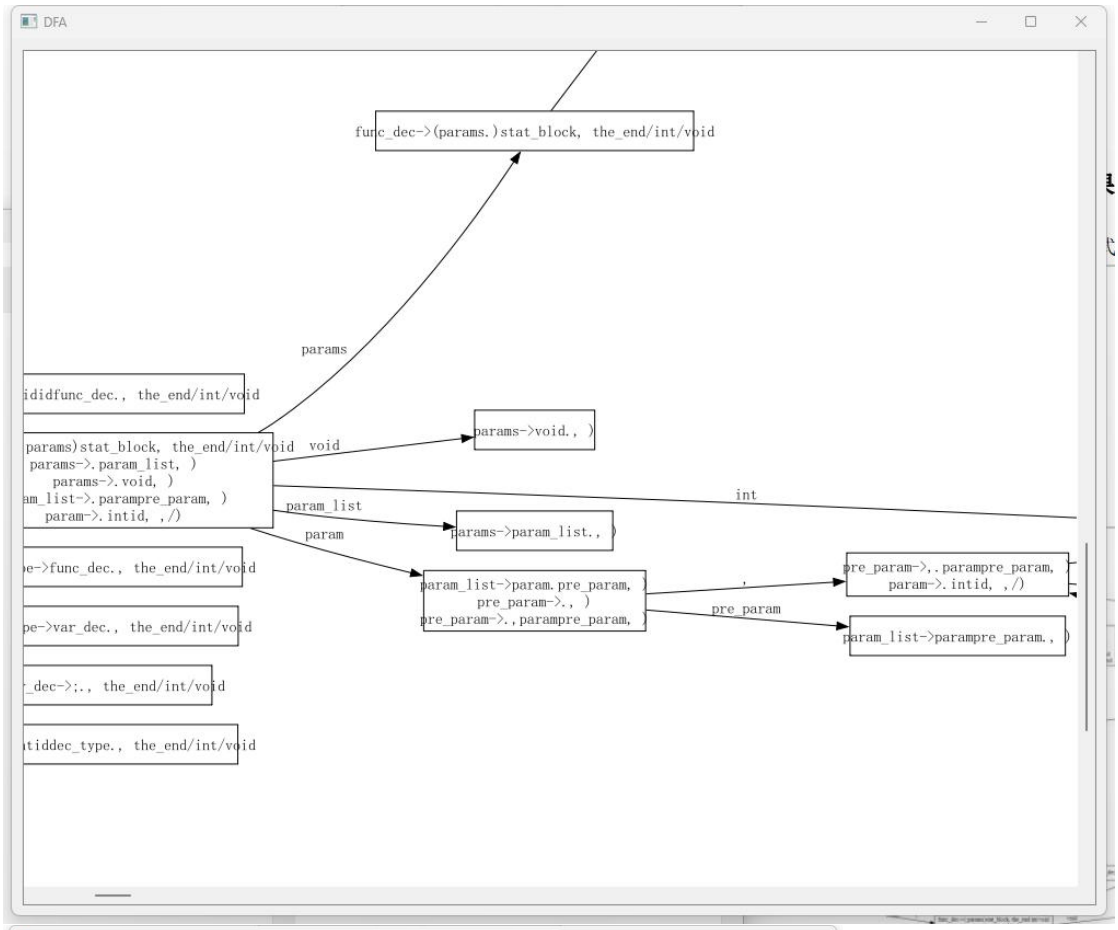
测试使用的文法产生式和程序全部放在 test 目录下。

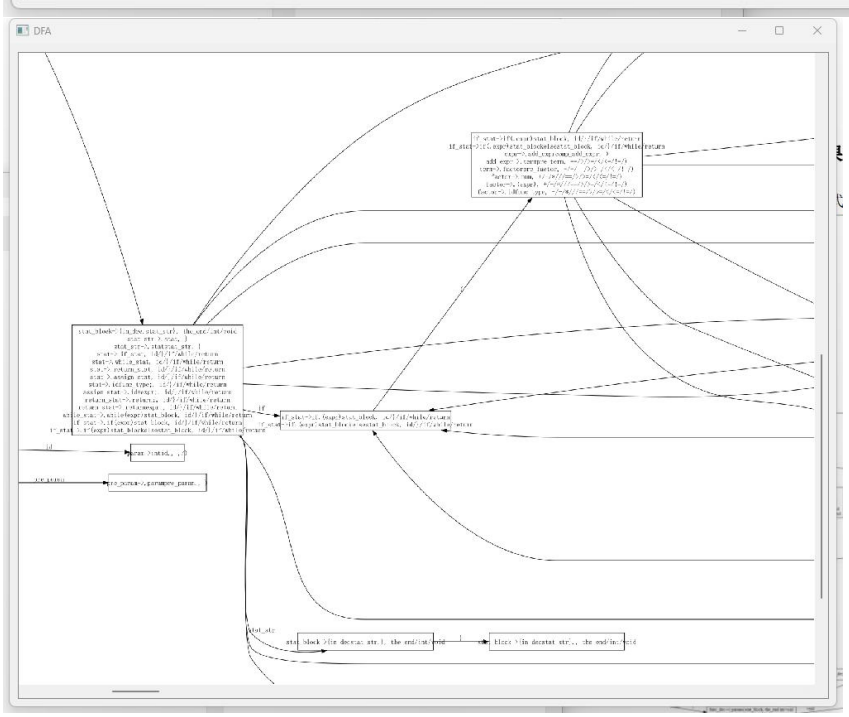
6.1. 调试-源程序 1

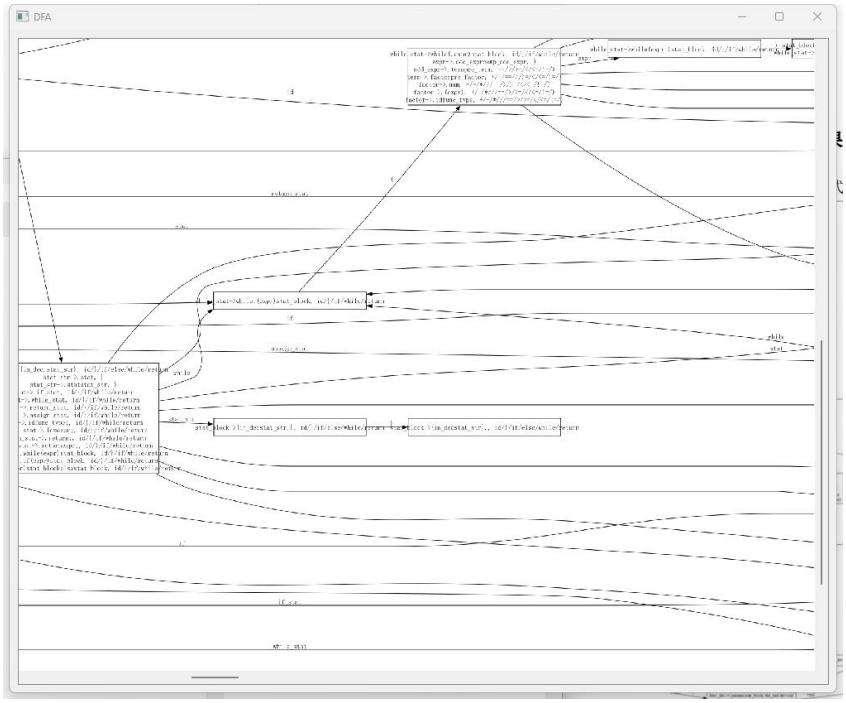
源程序：

```
int main{
int a;
int b;
int c;
c = a + b;
}
```

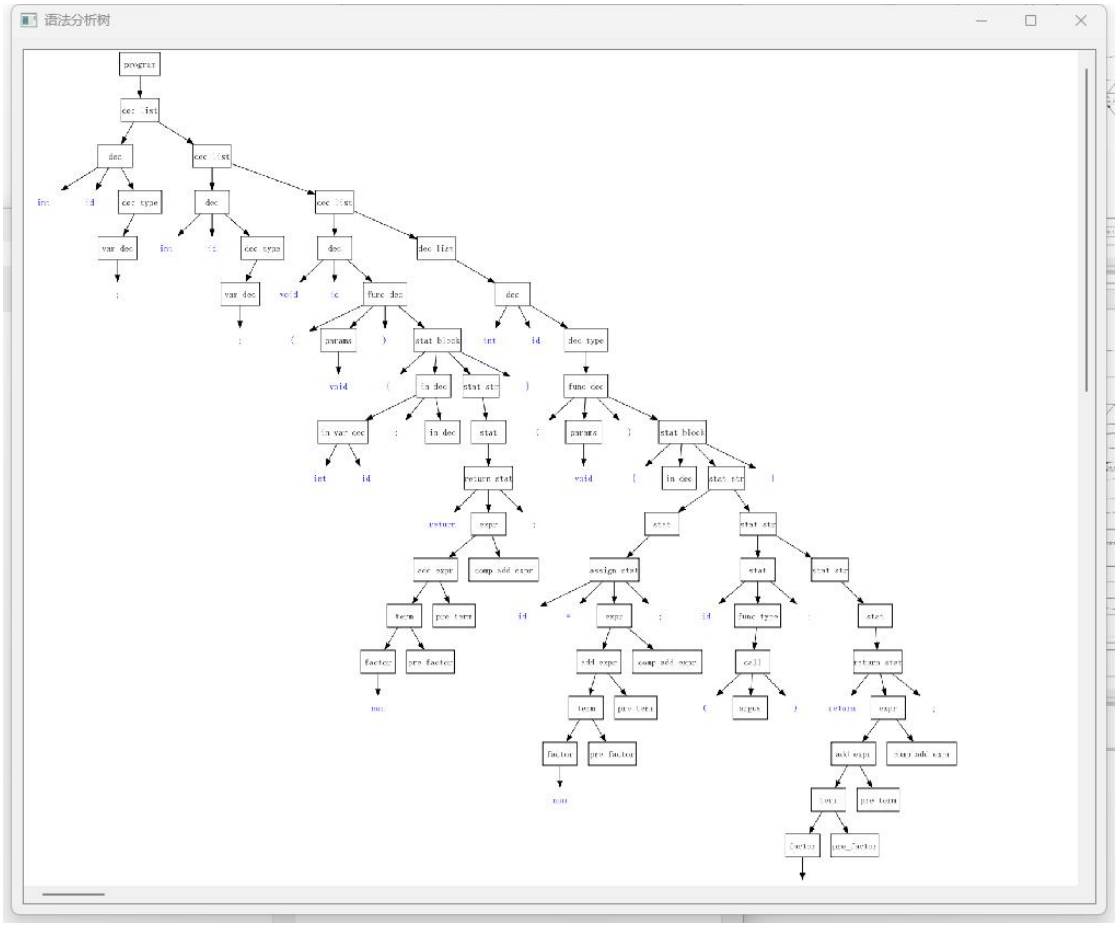
DFA：(部分


DFA
func_dec->(params.)stat_block, the_end/int/void
params
iddfunc_dec., the_end/int/void
params)stat_block, the_end/int/void
params->.param_list,)
params->.void,)
am_list->.parampre_param,)
param->.intid, ,/)
void
params->void.,)
int
param_list
params->param_list.,)
param
e->func_dec., the_end/int/void
pe->var_dec., the_end/int/void
_dec->;., the_end/int/void
tiddec_type., the_end/int/void
param_list->param.pre_param,)
pre_param->.,)
pre_param->.,parampre_param,)
,
pre_param
pre_param->,.parampre_param,
param->.intid, ,/)
param_list->parampre_param.,)


DFA



语法树：



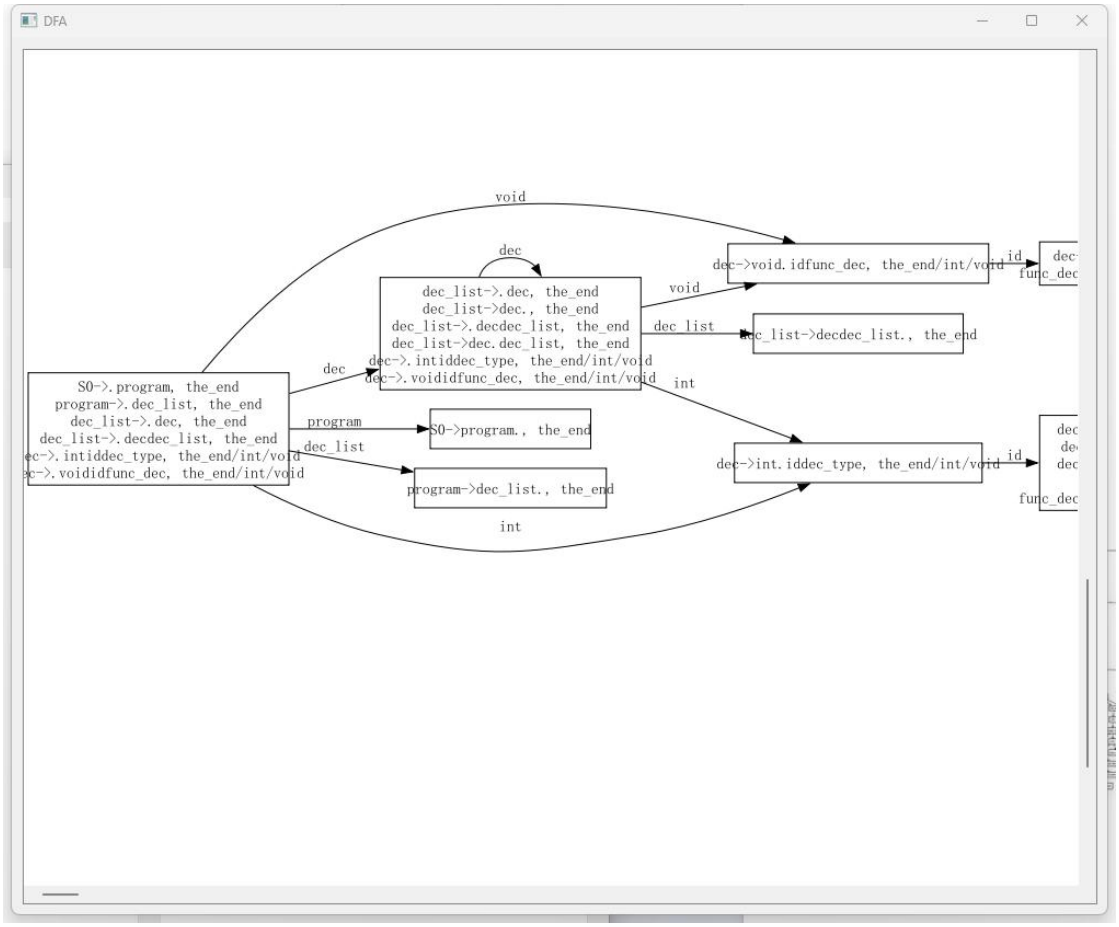
6.2. 调试-源程序 2

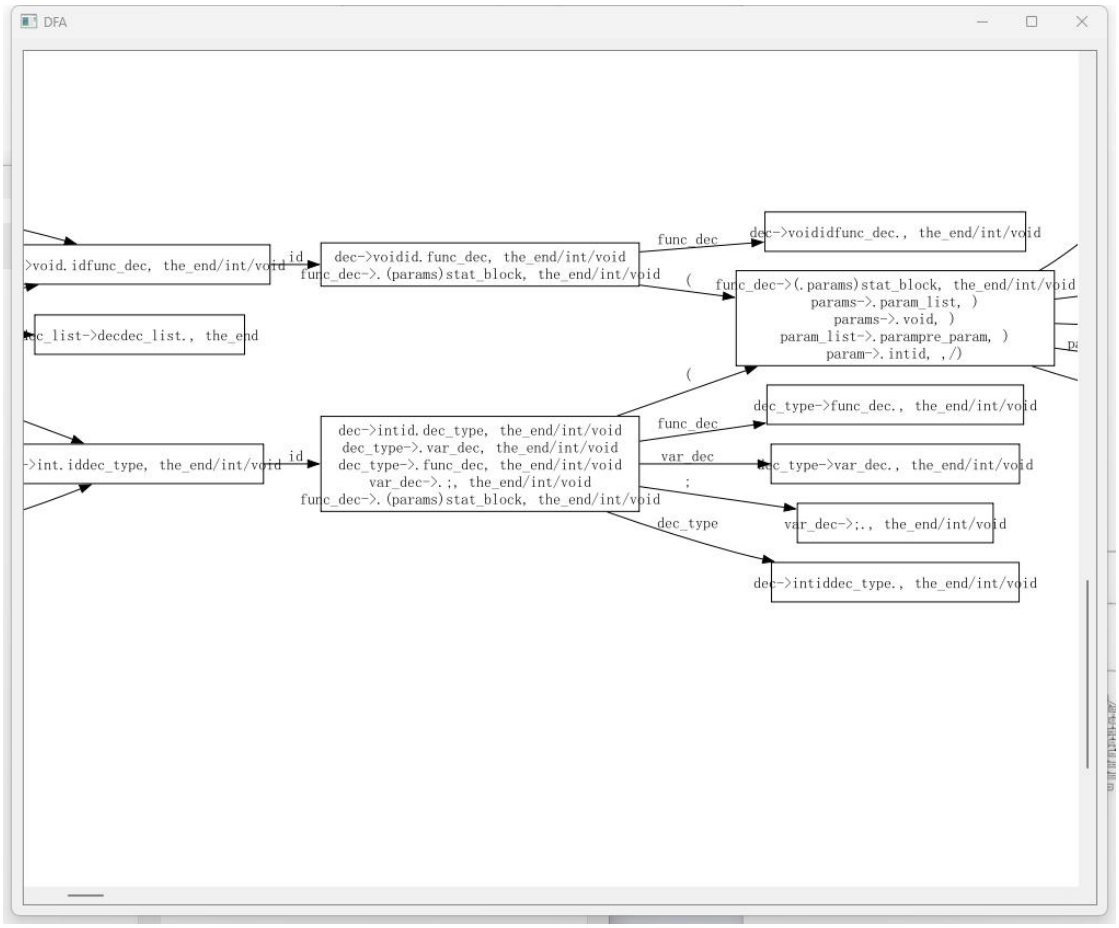
源程序：

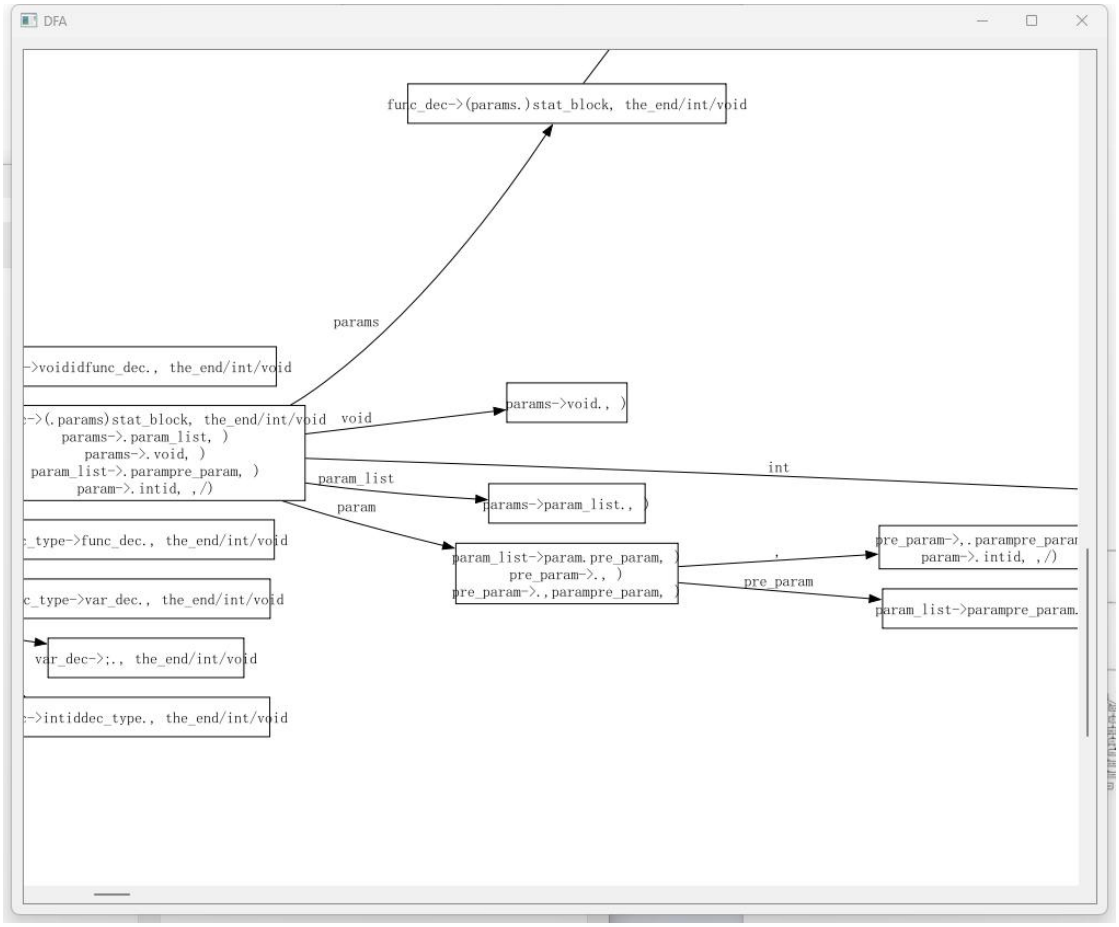
```
int a;
int b;
void demo(void)
{
int a;
return 0;
```

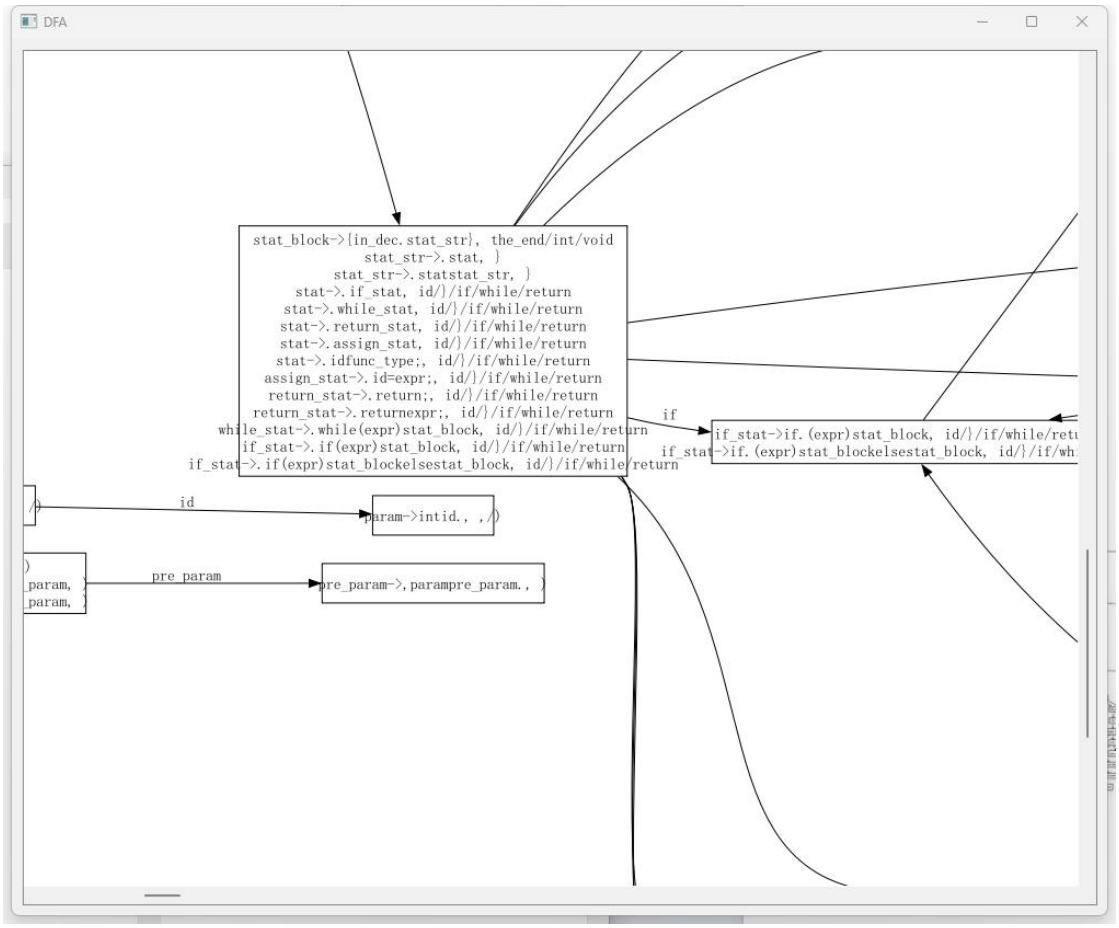
```
}
int main(void)
{
a = 0;
demo();
return 0;
}
int main{
int a;
int b;
int c;
c = a + b;
}
```

DFA(部分)：

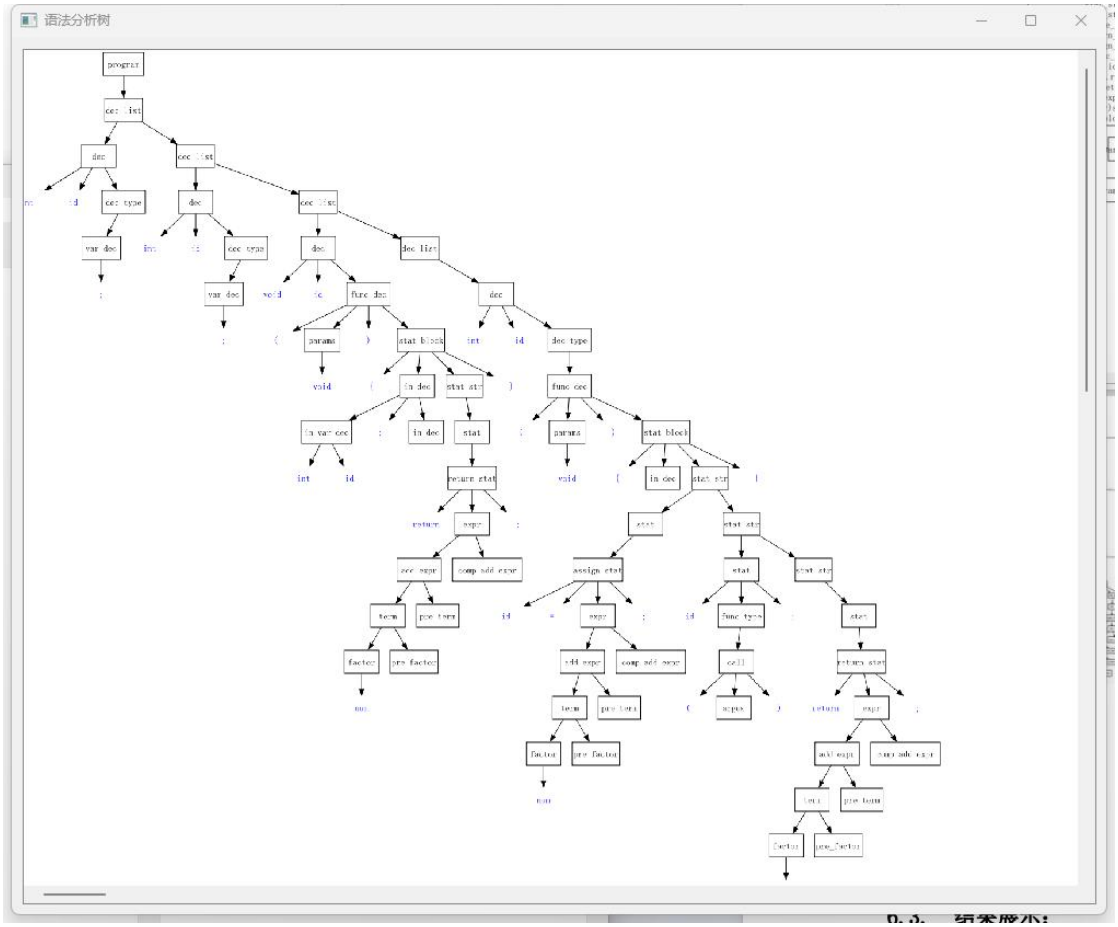

DFA
S0->.program, the_end
program->.dec_list, the_end
dec_list->.dec, the_end
dec_list->.decdec_list, the_end
dec->.intiddec_type, the_end/int/void
dec->.voididfunc_dec, the_end/int/void
void
dec
program
dec_list
int
dec_list->.dec, the_end
dec_list->dec., the_end
dec_list->.decdec_list, the_end
dec_list->dec.dec_list, the_end
dec->.intiddec_type, the_end/int/void
dec->.voididfunc_dec, the_end/int/void
dec->void.idfunc_dec, the_end/int/void
dec_list->decdec_list., the_end
S0->program., the_end
program->dec_list., the_end
dec->int.iddec_type, the_end/int/void
id


DFA
dec->voidid.func_dec, the_end/int/void
func_dec->.(params)stat_block, the_end/int/void
func_dec
dec->voididfunc_dec., the_end/int/void
func_dec->(.params)stat_block, the_end/int/void
params->.param_list,)
params->.void,)
param_list->.parampre_param,)
param->.intid, ,/)
(
dec->intid.dec_type, the_end/int/void
dec_type->.var_dec, the_end/int/void
dec_type->.func_dec, the_end/int/void
var_dec->.;, the_end/int/void
func_dec->.(params)stat_block, the_end/int/void
dec_type
var_dec
;
dec_type->func_dec., the_end/int/void
dec_type->var_dec., the_end/int/void
var_dec->;., the_end/int/void
dec->intiddec_type., the_end/int/void
id


DFA
func_dec->(params.)stat_block, the_end/int/void
params
->voididfunc_dec., the_end/int/void
->(.params)stat_block, the_end/int/void
params->.param_list,)
params->.void,)
param_list->.parampre_param,)
param->.intid, ,/)
void
params->void.,)
int
param_list
params->param_list.,)
param
_type->func_dec., the_end/int/void
c_type->var_dec., the_end/int/void
var_dec->;., the_end/int/void
->intiddec_type., the_end/int/void
param_list->param.pre_param,)
pre_param->.,)
pre_param->.,parampre_param,)
,
pre_param
pre_param->,.parampre_para
param->.intid, ,/)
param_list->parampre_param.


DFA
stat_block->{in_dec.stat_str}, the_end/int/void
stat_str->.stat, }
stat_str->.statstat_str, }
stat->.if_stat, id/}/if/while/return
stat->.while_stat, id/}/if/while/return
stat->.return_stat, id/}/if/while/return
stat->.assign_stat, id/}/if/while/return
stat->.idfunc_type;, id/}/if/while/return
assign_stat->.id=expr;, id/}/if/while/return
return_stat->.return;, id/}/if/while/return
return_stat->.returnexpr;, id/}/if/while/return
while_stat->.while(expr)stat_block, id/}/if/while/return
if_stat->.if(expr)stat_block, id/}/if/while/return
if_stat->.if(expr)stat_blockelsestat_block, id/}/if/while/return
if
if_stat->if.(expr)stat_block, id/}/if/while/retu
if_stat->if.(expr)stat_blockelsestat_block, id/}/if/wh
id
param->intid., ,/)
pre_param
pre_param->,parampre_param.,)

语法树：

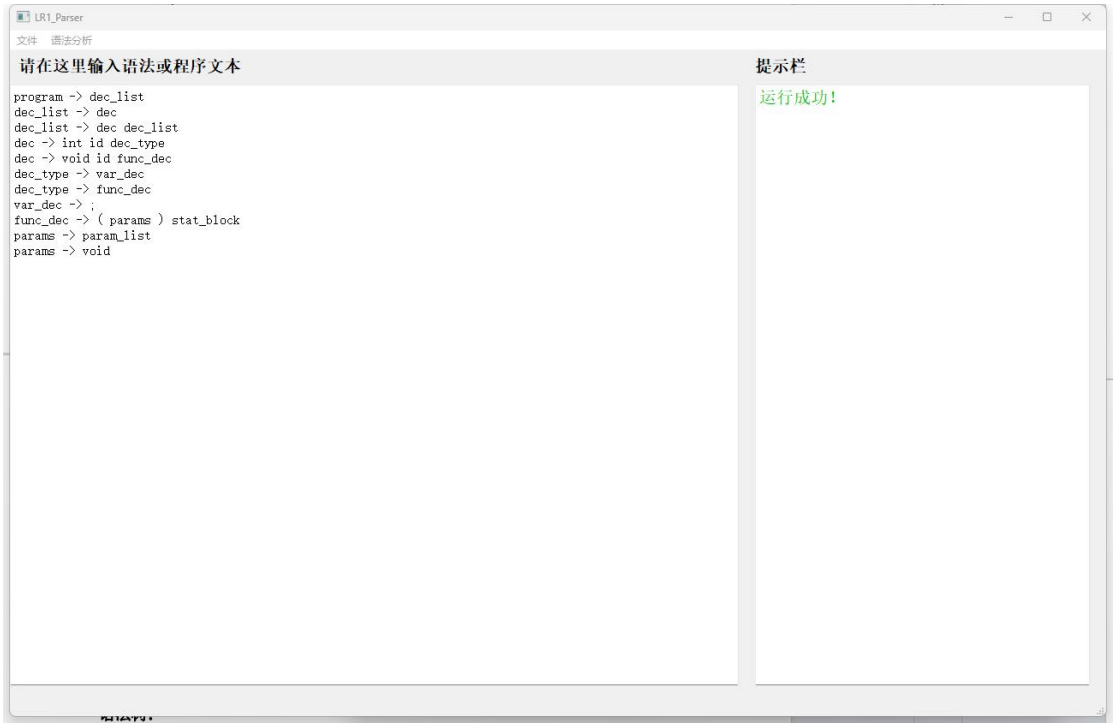


6.3. 结果展示：

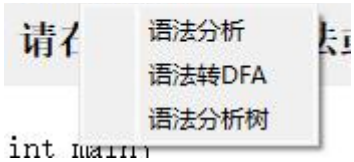
交互界面使用 QT 设计， 具有编辑输入文法、编辑输入源程序、显示提示信息、显示生成图等多种功能。

交互界面：

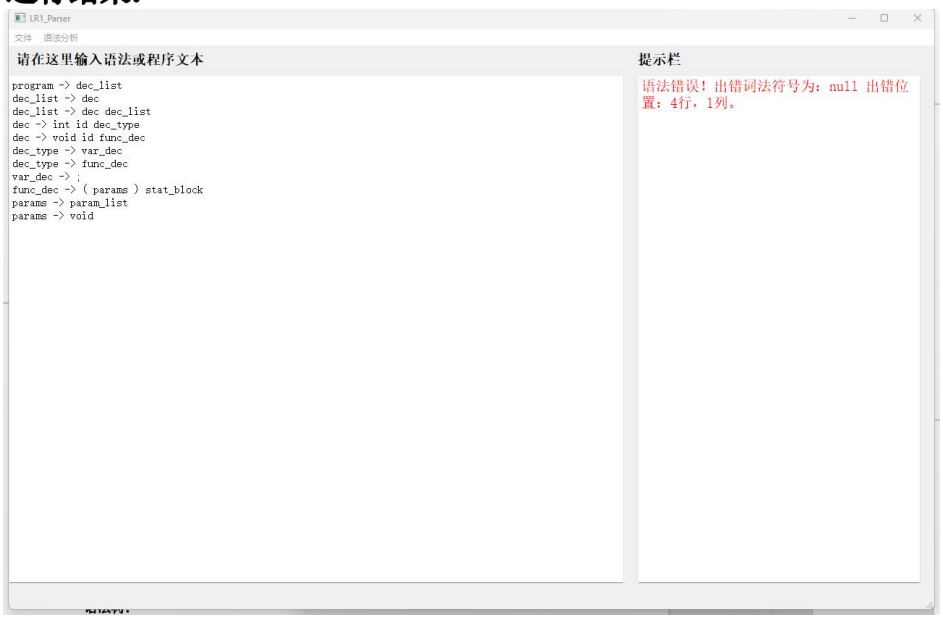
输入错误文法：



点击文法转 DFA 选项：


语法分析
语法转DFA
语法分析树

运行结果：


LR1_Parser
文件 语法分析
请在这里输入语法或程序文本
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 -> ;
func_dec -> (params) stat_block
params -> param_list
params -> void
提示栏
语法错误！出错词法符号为：null 出错位置：4行，1列。

七、总结与收获

7.1. 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。

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，对源程序进行语法分析后输出语法树的矢量图或报错。

7.2. 项目收获

在本次项目的完成过程中，我们收获了很多经验。

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， 但是不知道这些算法在实际的应用中是怎么样的。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。