

南京理工大学计算机科学与工程学院

软件课程设计(II)报告

姓名:	李兴昊
班级:	9191062301
学号:	919106840131
指导老师:	项欣光

目录

1. 引言.....	3
1.1 选题背景.....	3
2. 词法分析器.....	3
2.0 项目流程.....	3
2.1 设计 3 型文法（正规文法）.....	4
2.2 读取 3 型文法构造出 nfa.....	8
2.3 用子集法将 NFA->DFA.....	9
2.4 Scan()源程序.....	12
2.5 输出 token 组.....	15
3. 语法分析器.....	16
3.0 项目流程.....	16
3.1 设计 2 型文法（上下文无关文法）.....	17
3.2 处理输入的 token 串.....	20
3.3 计算 FIRST 集合.....	21
3.4 求 Clousure 闭包.....	22
3.5 画出 LR(1)的 ACTION GOTO 表.....	28
3.6 用 ACTION GOTO 表对 token 语法分析.....	29
4. 心得体会.....	31

1.引言

1.1 选题背景

本学期要做编译原理的软件课程设计(II)，恰好上学期学过有关编译原理的课程，并取得了一个不错的分数，故我对这次的词法分析器、语法分析器有点兴趣，考虑到 C++ 的 STL 已经为我们封装好了 set map 等高效存储的数据结构，且我们大学三年对 C++ 也更加熟悉一点，所以我想要用自己所学的 C++ 有关的知识写一个简易的词法、语法分析器。

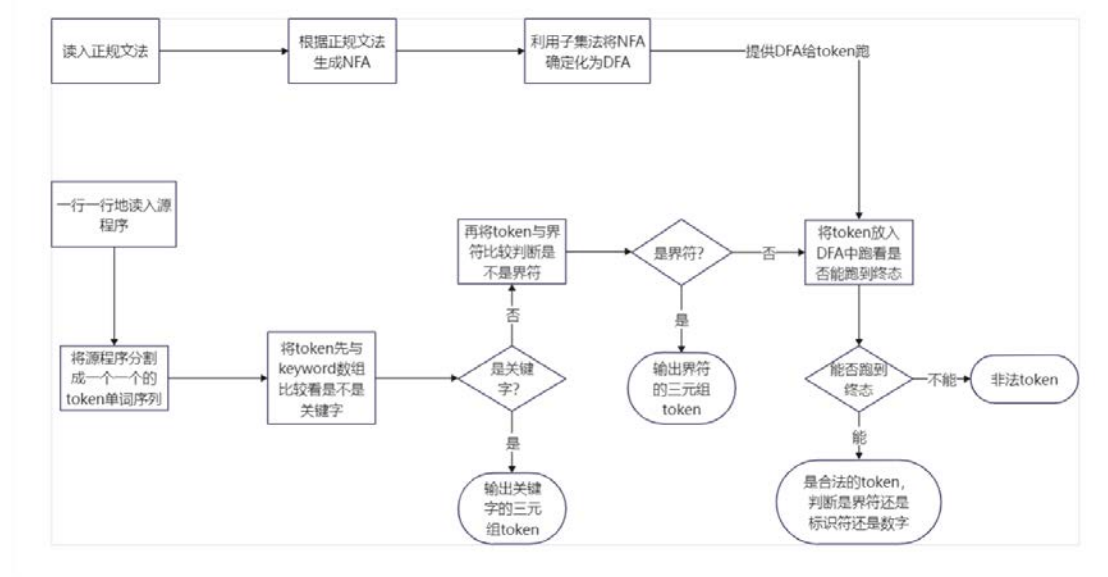
本项目实现了编译过程中的词法分析和语法分析。

词法分析：从左到右一个字符一个字符地读入源程序，对构成源程序的 token 进行扫描和分解，从而识别出一个个单词。

语法分析：在词法分析的基础上，将三元组 token 分解成各类语法短语，如“赋值语句”、“条件语句”、“循环语句”、“函数体”等，并判断其是否合乎二型文法。

2. 词法分析器

2.0 项目流程



2.1 设计 3 型文法（正规文法）

三型文法部分我是自己设计的递推式，并参考了清华大学编译原理第三版教材的第三章内容，设计了词法分析的三型文法。

井号部分是文法的注释，3 型文法如下：

#grammar3_Lxh 全部采用右线性的三型文法

#####

#Tips:

#用 while 循环读取，每行正规文法被一次 getline 读取到 content 中

#content[0]='#'时表示这一行是注释，continue

#content[0]为文法左边

#content[1]为-，和 content[2]为>，合起来是->用在规则（产生式）中，称作“定义为”

#content[3]为文法右边第一个终结符，content[4]表示文法右边的第二位

#若文法右只有一个非终结符，则 content[4]为 space 空格

#若文法右为一个终结符加非终结符，则 content[4]为非终结符

#####

#L 表示界符 limiter

L->,

L->;

L->[

L->]

L->(

L->)

L->{

L->}

#O 表示运算符 operator，引入 E 为双目运算符

O->+

O->-

O->*

O->/

O->%

O->^

O->&

O->=

O->>

O-><

O->+E

O->-E

O->*E

O->/E

O->=E

O->>E

O-><E

E->=

#I 表示标识符，引入 J 因为标识符不能以数字开头,引入 a 表示 26 个字母，引入 d 表示 0-9

#####

#后续 debug 时发现文法的 bug，故将文法中所有的 d 换成 0 | c，文法的意思不改变
注释由于是最开始写的，故不做更改

#####

I->_

I->a

I->e

I->i

I->_J

I->aJ

I->eJ

I->iJ

J->_

J->0

J->c

J->a

J->e

J->i

J->_J

J->0J

J->cJ

J->aJ

J->eJ

J->iJ

#A 表示整形数，引入 B 由于整形数不能以 0 开头，故引入 c 表示 1-9，引入 d 表示 0-9

A->c

A->0

A->cB

B->0

B->c

B->0B

B->cB

#S 表示科学计数法例如 3.21e+10，引入 T 因为前面部分不能以 0 开头，且可以是小数

#注意，不能直接 S->c，T->d 这样直接到终态就退出 FA 了

#引入 H 表示后面部分不能是小数得是一个十进制数(不能以 0 为开头)，引入 C 表示小数部分不能以 0 为开头

#引入 G 表示科学计数法 e 后面的符号，也可以直接没有符号（默认是+）

#引入 D 表示 e 后面的系数得是整数，引入 F 表示 e 后面的整数不能以 0 开头

S->0T

S->cT

T->0T

T->cT

$T \rightarrow .H$

$T \rightarrow eG$

$H \rightarrow c$

$H \rightarrow 0$

$H \rightarrow cC$

$C \rightarrow 0$

$C \rightarrow c$

$C \rightarrow 0C$

$C \rightarrow cC$

$C \rightarrow eG$

$G \rightarrow +D$

$G \rightarrow -D$

$G \rightarrow c$

$G \rightarrow cF$

$D \rightarrow c$

$D \rightarrow cF$

$F \rightarrow 0$

$F \rightarrow c$

$F \rightarrow 0F$

$F \rightarrow cF$

#引入 M 表示复数例如 $3+4i$, 引入 N 表示复数的符号

#引入 Q 表示复数后面的部分

#同理注意, 不能直接 $M \rightarrow c$, $N \rightarrow d$ 这样直接到终态就退出 FA 了

#具体复数构造非 0 开头的数字步骤与科学计数法一致, 故不花时间构造

#只弄一个大概的识别标准 $a+bi$ (a, b 都非 0, 且不能以 0 为开头), 后面可以对这一部

分文法加以完善

$M \rightarrow 0M$

$M \rightarrow cM$

$M \rightarrow 0N$

$M \rightarrow cN$

$N \rightarrow +Q$

$N \rightarrow -Q$

$Q \rightarrow 0Q$

$Q \rightarrow cQ$

$Q \rightarrow i$

#endread

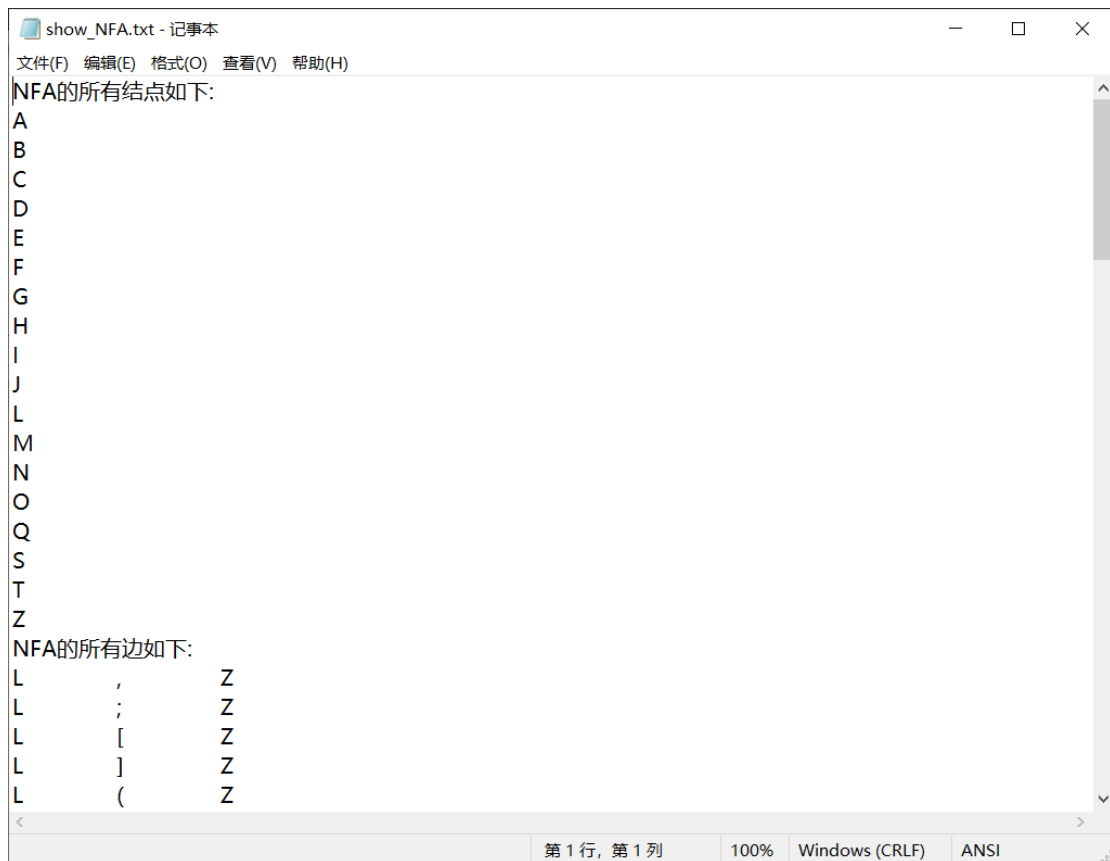
源程序一行一行读入，遇到#则会跳过该行，根据正规文法生成项目的 NFA。可以看到正规文法里已经注意到数字不能以 0 开头，标识符不能以数字开头，且能识别二元运算符、科学计数法与虚数的情况。

2.2 读取 3 型文法构造出 nfa

这一步非常简单，由于我写的都是右线性文法，按行读入文法，分为两种情况

1. 形如 $A \rightarrow b$ 的文法，推导符右侧只有一个终结符，只需要在 NFA 中加入一条由 A 指向终态结点 Z 的边，边上的权值 val 为 b 即可
2. 形如 $A \rightarrow bB$ 的文法，推导符右侧是一个终结符和一个非终结符，只需要在 NFA 中加一条由 A 指向结点 B 的边，边上的权值 val 为 b 即可

循环遍历，直到读到#endread 代表文法已经全部存入 NFA。实现过程如截图（已保存到 show_NFA.txt 文件）：



2.3 用子集法将 NFA->DFA

将 NFA 转成 DFA 我用的是教材上的子集法，先将 NFA 的所有初态手动插入 T_0 。这里的初态包括正规文法中的，指向界符 L、运算符 O、标识符 I、数字 A、科学计数法 S、虚数 M 的这几个非终结符。然后要先对 T_0 做一次 $\epsilon - \text{closure}(T_0)$ 状态闭包，得到 T_1 作为 DFA 的第一个子集，然后进一步求弧转闭包的所有的不重复的子集。

这里要用到两个集合的运算法则：

- (1) 状态集合 I 的 ϵ -闭包，表示为 $\epsilon - \text{closure}(I)$ ，定义一个状态集合，是状态集 I 中任何状态经过任意条 ϵ 弧而能到达的状态的集合。
- (2) 状态集合 I 的 a 弧转换，表示为 $\text{move}(I, a)$ ，定义为状态集合 J，其中 J 是所有那些可从 I 中的某一状态经过 a 弧能到达的状态全体。

具体的算法实现过程我参考了清华大学编译原理教材的第 51 页求集合的 $\epsilon - \text{closure}$ 伪代码，具体实现流程如下：

- (1) 开始，另 $\epsilon - \text{closure}(K_0)$ 为 C 中的唯一成员，并且它是未被标记的。
- (2) While (C 中存在尚未被标记的子集) do {

```

    标记 T;
    for 每个输入字母 a do{
         $U := \varepsilon - \text{closure}(\text{Move}(T, a))$  ;
        If U 不在 C 中 then
            将 U 作为未被标记的子集加在 C 中
        }
    }

```

从求 $\varepsilon - \text{closure}$ 的闭包算法中我们不难看出，STL 封装好的函数 set 集合是非常适合作为求 DFA 子集这一数据结构的，set 的特点是：set 集合是一个排序好的且不会重复的集合。我也是用 set 来求的 DFA 的所有子集，set 函数需要包含头文件#include<set>，set.insert();可以对 set 集合进行插入操作，我们只需要再每次对一个文法中存在的终结符进行弧转闭包操作后，可以直接对集合进行遍历操作（因为 set 集合是排序好的），然后判断是否是重复的集合，再决定是否插入。利用直接封装好的函数，省去了很多行无效代码。下面展示的是我的这部分 NFA_To_DFA 的核心代码，可以看到，很短的几行就将 DFA 的所有闭包都求出来。因为 move 和 $\varepsilon - \text{closure}$ 每次求子集都要使用，故我提前对这两个返回值为 set<char>的函数进行了封装。这两个函数的原理在上面已经讲过，对照伪码用堆栈可以很快地解决，故此处不再赘述。

因为要展示实现过程，我将过程保存到了 show_DFA.txt，里面包括 DFA 里面的每个子集里所有结点、DFA 的所有边、DFA 的终态结点（包含 Z 的子集序号），截图如下所示。

```

//集合移入函数，用于dfa求弧转闭包的时候状态移入集合的操作，对照书上的算法原理实现
set<char> move(set<char> I, char a, NFA_node N) {
    set<char> temp;
    for (auto it = I.begin(); it != I.end(); it++)
        for (int i = 0; i < N.edge_num; i++) {
            if (N.edge[i].start == *it && N.edge[i].val == a)
                temp.insert(N.edge[i].end);
        }
    return temp;
}

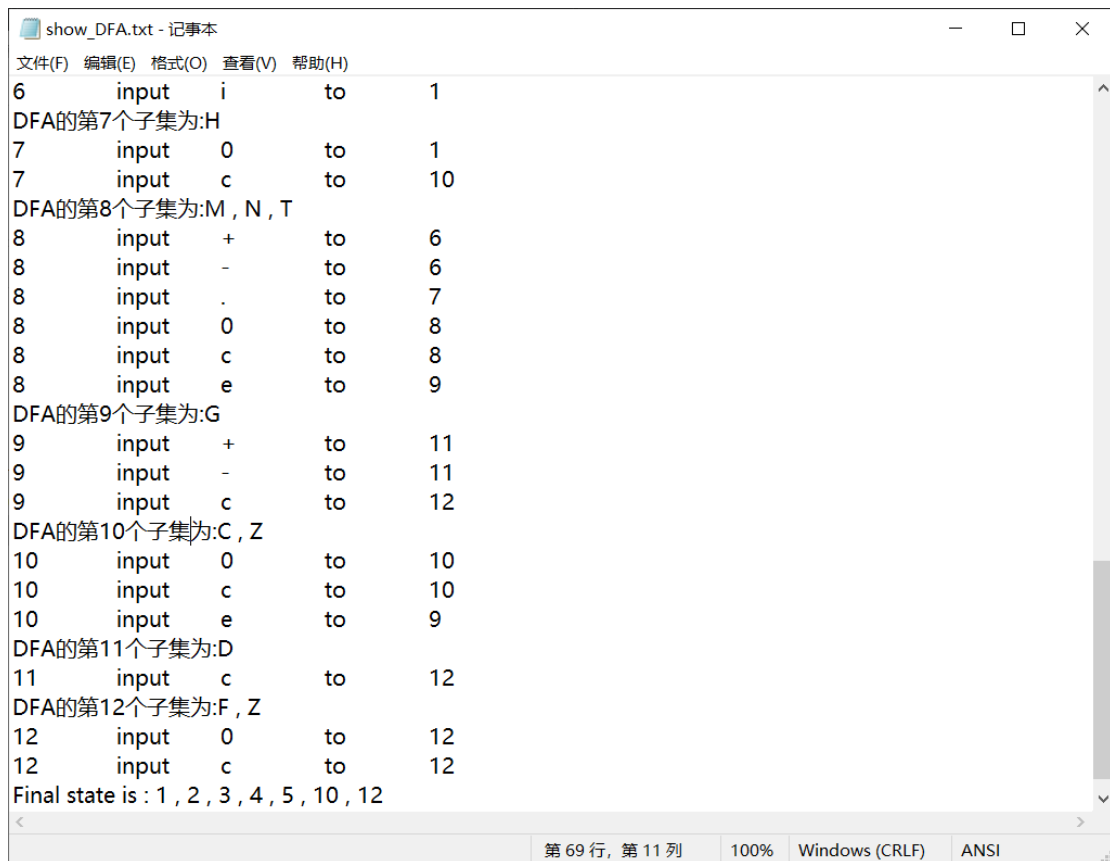
//求Epsilon状态闭包的函数，对照书上算法原理实现
set<char> e_closure(set<char> I, NFA_node N) {
    set<char> temp = I;
    stack<char> st;
    for (auto it = temp.begin(); it != temp.end(); it++)
    {
        st.push(*it);
    }
    char t;
    while (!st.empty()) {
        t = st.top();
        st.pop();
        for (int i = 0; i < N.edge_num; i++) {
            if (N.edge[i].start == t && N.edge[i].val == '@') {
                temp.insert(N.edge[i].end);
                st.push(N.edge[i].end);
            }
        }
    }
    return temp;
}

```

```

for (auto it = weight.begin(); it != weight.end(); it++) { //weight.size()为nfa总的val值数量
    int j = 0;
    if (*it != '@') {
        //非空的弧转
        set<char> temp = e_closure(move(C[i], *it, nfa), nfa); //N为正规文法条数，G为Edge的start val end
        if (!temp.empty()) { //C为DFA每一个集合里结点
            bool inC = false;
            int k = 0;
            while (!C[k].empty() && k < MAX_NODES) {
                //判断是否是重复子集
                if (temp == C[k]) {
                    inC = true;
                    break;
                }
                k++;
            }
            if (!inC) {
                //不是重复子集，把弧转新出现的子集插入C
                k = 0;
                while (!C[k].empty() && k < MAX_NODES)
                {
                    k++;
                }
                C[k] = temp;
            }
            dfa.node.insert(i); //插入初态
            dfa.node.insert(k); //插入终态结点
            dfa.edge[dfa.edge_num].start = i;
            dfa.edge[dfa.edge_num].val = *it;
            dfa.edge[dfa.edge_num].end = k;
            dfa.edge_num++;
            out << i << " input " << *it << " to " << k << endl;
            //debug << i << " " << k << " " << *it << endl; //https://csacademy.com/app/graph_editor/的输出格
        }
        j++;
    }
    i++;
}

```

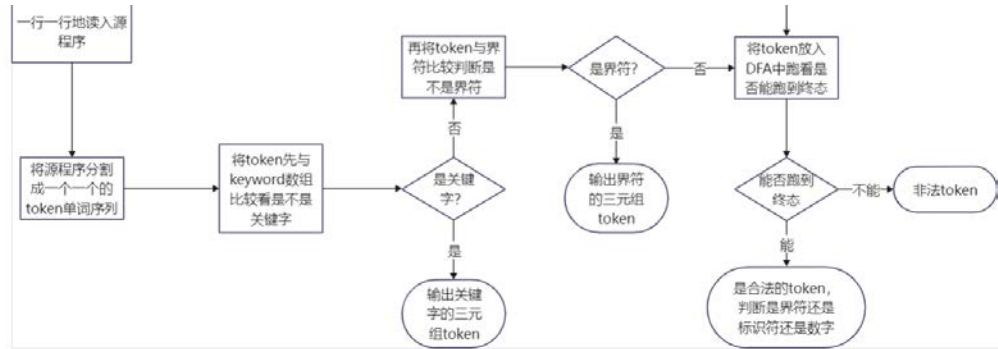


```
show_DFA.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
6      input    i      to      1
DFA的第7个子集为:H
7      input    0      to      1
7      input    c      to      10
DFA的第8个子集为:M, N, T
8      input    +      to      6
8      input    -      to      6
8      input    .      to      7
8      input    0      to      8
8      input    c      to      8
8      input    e      to      9
DFA的第9个子集为:G
9      input    +      to      11
9      input    -      to      11
9      input    c      to      12
DFA的第10个子集为:C, Z
10     input    0      to      10
10     input    c      to      10
10     input    e      to      9
DFA的第11个子集为:D
11     input    c      to      12
DFA的第12个子集为:F, Z
12     input    0      to      12
12     input    c      to      12
Final state is : 1, 2, 3, 4, 5, 10, 12
第 69 行, 第 11 列 100% Windows (CRLF) ANSI
```

2.4 Scan()源程序

这一步主要是将源程序分割好，然后调用一个功能函数 `bool token_can_run_final(string str)`;他的效果是判断一个 `str` 是否能在 DFA 跑到终态，如果能跑到终态，那说明这是一个合法的 token，`return true`;否则就说明这个 token 是个非法 token，`return false`;具体的实现就是从 `str` 的第一个字符串开始取，依次找边，没找到可以走的结点，或者最后没有走到终态结点，就会 `return false`，记得每次递归结束前（`return` 前），要先将 `final_state` 置 0，否则这个函数就只能判断一次是否是合法的终态，第一次 debug 时发现同样的产生式第二次输入居然会 `return false` 时发的问题。

然后就是对分割好，存储在 `vector<string> token` 里的 token 进行挨个处理的过程，处理流程图在最开始完整程序的流程已经展示过，这里附上一张部分截图，省去读者来回翻阅的时间。



主要实现函数如下，这一部分就是一行一行处理，由于全是调用的之前的函数，故实现起来并不复杂，细心 debug 几次没遇到什么大的难题：

```

//将扫描进来的token放进dfa跑，看能不能到终态，能跑到终态的才能说明其是合法的token，否则就是错误的token
bool token_can_run_final(string str) {
    if (str.size() == 0) {
        for (int i = 0; i < dfa_final_state.size(); i++) {
            //若能在dfa跑到终态
            if (now_dfa_state == dfa_final_state[i])
            {
                //每次递归结束要记得把now_dfa_state = 0，回到初态，否则只能判断一次了
                now_dfa_state = 0;
                return true;
            }
        }
    }

    //取token的第一个字符，并删去第一个字符后，用递归法判断能否跑到终态
    char ch = str[0];
    string str_deletefirst;
    for (int i = 1; i < str.size(); i++) str_deletefirst += str[i];
    for (int i = 0; i < dfa.edge_num; i++) {
        if (dfa.edge[i].start == now_dfa_state && dfa.edge[i].val == ch) {
            now_dfa_state = dfa.edge[i].end;
            return token_can_run_final(str_deletefirst);
        }
    }

    //每次递归结束要记得把now_dfa_state = 0，回到初态，否则只能判断一次了
    now_dfa_state = 0;
    return false;
}

```

```

for (int temp1 = 0; temp1 < token.size(); temp1++) {
    //token[temp1]表示每一行的第(temp1-1)个待识别的token
    //求出keyword二维数组总共有多少行
    int all = sizeof(keyword) / sizeof(char);
    int column = sizeof(keyword[0]) / sizeof(keyword[0][0]);
    int row = all / column;
    //一个bool型变量, 假如一个token 已经被判断过就要continue判断下一个token
    bool is_judged = false;
    //...
    for (int temp2 = 0; temp2 < row; temp2++) {
        if (token[temp1] == keyword[temp2]) {
            is_judged = true;
            cout << line_number << ", " << token[temp1] << ", " << "keyword" << endl;

            output << line_number << ", " << token[temp1] << ", " << "keyword" << endl;
        }

        if (is_judged) continue;
        //...
        if (token[temp1] == "+" || token[temp1] == "-" || token[temp1] == "[" || token[temp1] == "]"
            || token[temp1] == "(" || token[temp1] == ")" || token[temp1] == "{" || token[temp1] == "}"
            || token[temp1] == "<=" || token[temp1] == ">=" || token[temp1] == "<" || token[temp1] == ">") {
            is_judged = true;
            cout << line_number << ", " << token[temp1] << ", " << "limiter" << endl;

            output << line_number << ", " << token[temp1] << ", " << "limiter" << endl;
        }

        if (is_judged) continue;
        //3、再判断操作符operator
        //先取operator的第一位, 若是运算符, 则放进dfa看看能不能遇到终态, 若能则说明是合法的运算符
        //因为有无运算符, 所以不能只靠读取一个字符识别
        if (token[temp1][0] == '+' || token[temp1][0] == '-' || token[temp1][0] == '*'
            || token[temp1][0] == '/' || token[temp1][0] == '%' || token[temp1][0] == '^'
            || token[temp1][0] == 'a' || token[temp1][0] == 'w' || token[temp1][0] == 't'
            || token[temp1][0] == 'u' || token[temp1][0] == 'v' || token[temp1][0] == 'c') {
            is_judged = true;
            if (token_can_run_final(token[temp1])) {
                cout << line_number << ", " << token[temp1] << ", " << "operator" << endl;

                output << line_number << ", " << token[temp1] << ", " << "operator" << endl;
            }
            else {
                cout << line_number << ", " << token[temp1] << ", " << "wrong operator" << endl;

                output << line_number << ", " << token[temp1] << ", " << "wrong operator" << endl;
            }
        }

        if (is_judged) continue;
        //4、再判断标识符identifier
        string str_token = token[temp1];
        for (int temp2 = 0; temp2 < token[temp1].size(); temp2++) {
            if ((token[temp1][temp2] >= 'a' && token[temp1][temp2] <= 'z' && token[temp1][temp2] != '1'
                || token[temp1][temp2] >= 'A' && token[temp1][temp2] <= 'Z' && token[temp1][temp2] != 'i'
                || token[temp1][temp2] >= '0' && token[temp1][temp2] <= '9' && token[temp1][temp2] >= '1') && token[temp1][temp2] != 'c') {
                //以字母或者下划线开头的, 一定是标识符, 判断合法性即可
                if (token[temp1][0] == '_' || token[temp1][0] == 'a') {
                    if (token_can_run_final(token[temp1])) {
                        cout << line_number << ", " << str_token << ", " << "identifier" << endl;

                        output << line_number << ", " << str_token << ", " << "identifier" << endl;
                    }
                    else {
                        cout << line_number << ", " << str_token << ", " << "wrong identifier" << endl;

                        output << line_number << ", " << str_token << ", " << "wrong identifier" << endl;
                    }
                }

                //...
                bool is_wrong_identifier = false;
                if (token[temp1][0] == 'c' || token[temp1][0] == '0') {
                    for (int l = 0; l < token[temp1].size(); l++) {
                        if (token[temp1][l] == 'a') {
                            is_wrong_identifier = true;
                            break;
                        }
                    }

                    if (is_wrong_identifier) {
                        //如果是错误的标识符
                        cout << line_number << ", " << str_token << ", " << "wrong identifier" << endl;

                        output << line_number << ", " << str_token << ", " << "wrong identifier" << endl;
                        continue;
                    }
                    else {
                        //不是错误的标识符, 则是常量, 用dfa判断常量合法性
                        if (token_can_run_final(token[temp1])) {
                            cout << line_number << ", " << str_token << ", " << "number" << endl;

                            output << line_number << ", " << str_token << ", " << "number" << endl;
                        }
                        else {
                            cout << line_number << ", " << str_token << ", " << "wrong number" << endl;

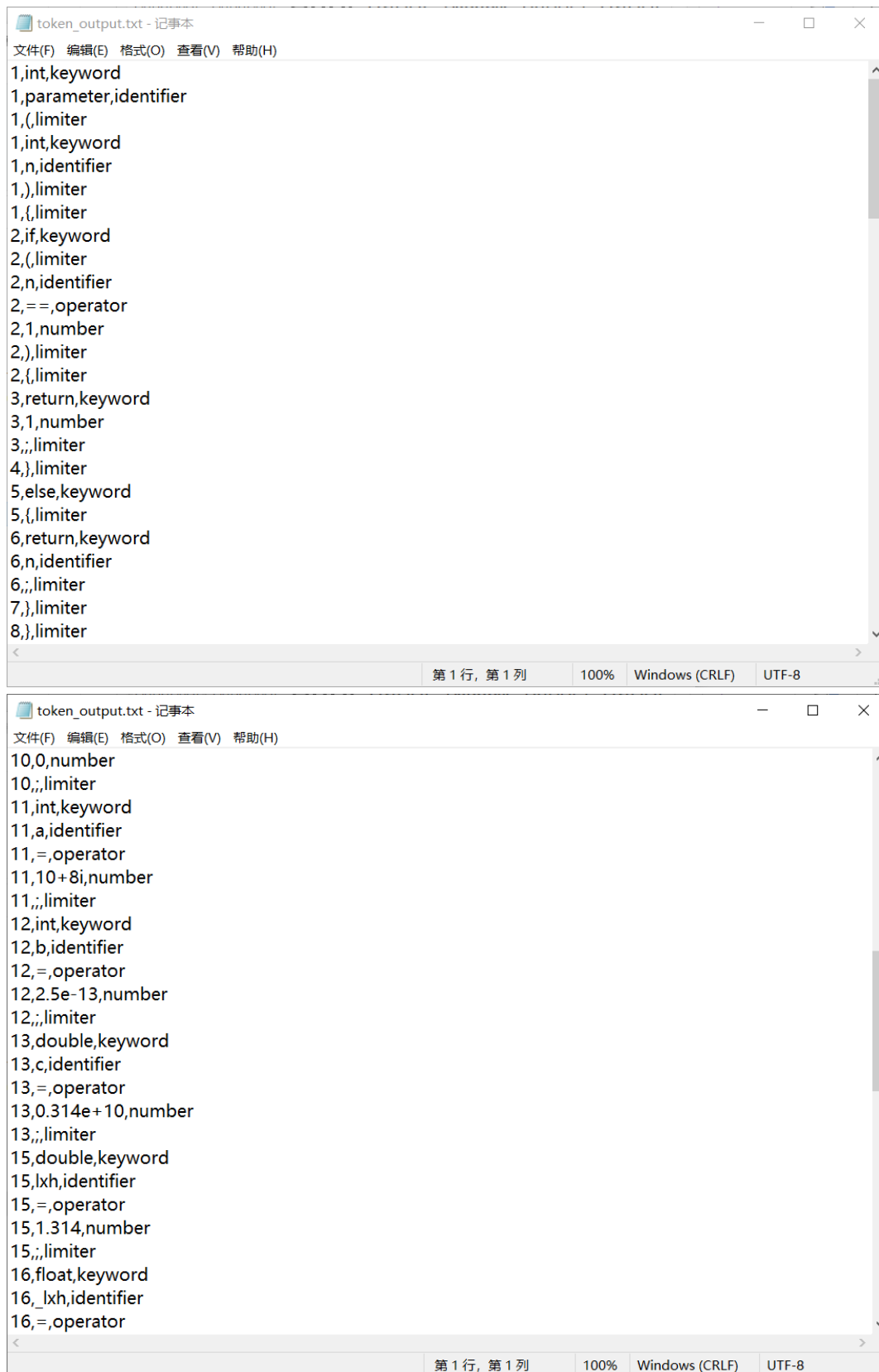
                            output << line_number << ", " << str_token << ", " << "wrong number" << endl;
                        }
                    }
                }
            }
        }

        line_number++;
    }
}

```

2.5 输出 token 组

与此同时，输出到 token_output.txt 文件，作为语法分析器的输入，截图如下：



```
token_output.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
1,int,keyword
1,parameter,identifier
1,(,limiter
1,int,keyword
1,n,identifier
1,),limiter
1,{,limiter
2,if,keyword
2,(,limiter
2,n,identifier
2,=,operator
2,1,number
2,),limiter
2,{,limiter
3,return,keyword
3,1,number
3,,,limiter
4,),limiter
5,else,keyword
5,{,limiter
6,return,keyword
6,n,identifier
6,,,limiter
7,),limiter
8,),limiter
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8

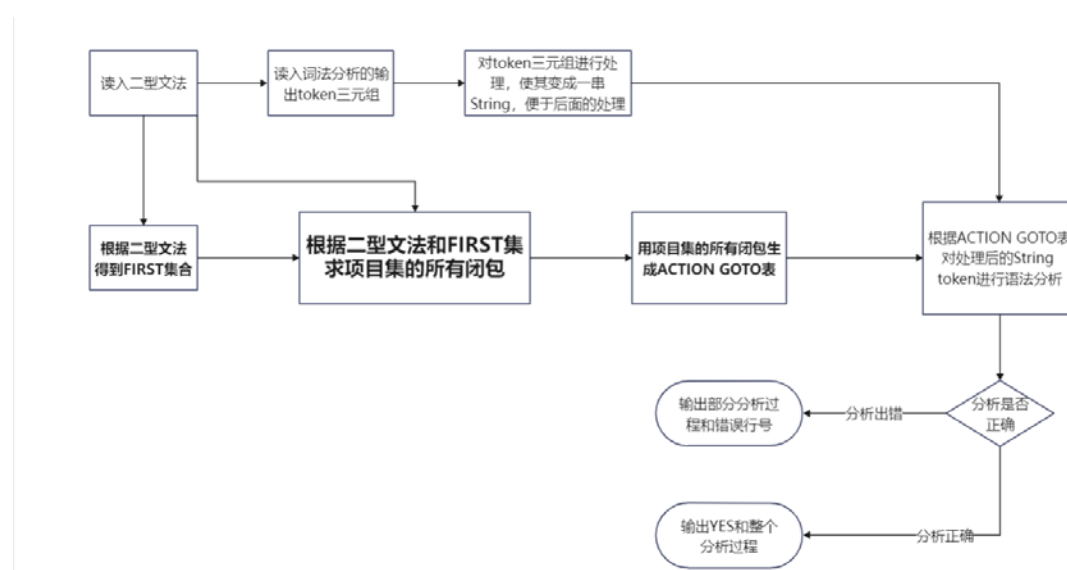
token_output.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
10,0,number
10,,,limiter
11,int,keyword
11,a,identifier
11,=,operator
11,10+8i,number
11,,,limiter
12,int,keyword
12,b,identifier
12,=,operator
12,2.5e-13,number
12,,,limiter
13,double,keyword
13,c,identifier
13,=,operator
13,0.314e+10,number
13,,,limiter
15,double,keyword
15,lxh,identifier
15,=,operator
15,1.314,number
15,,,limiter
16,float,keyword
16,_lxh,identifier
16,=,operator
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8
```

至此，词法分析部分基本完成，基本实现了老师的要求，能够识别科学计数法、虚数，遇到 0 开头的数字也会报错，遇到非法的标识符也会报错，由于语法分析要求一个正确的 token 结果，所以最后都改成了正确的标识符，但是不难测试，定义类似 1a 的标识符还有 02 的数字这种，是会在三元组中报错的。测试结果如下，为了语法分析的正确性，最终的 source_program.txt 中我没有输入错误的词法。

```
选择 Microsoft Visual Studio 调试控制台
22,;,limiter
23,;,limiter
25,while,keyword
25,(,limiter
25,lxh,identifier
25,>=,operator
25,_lxh,identifier
25,),limiter
25,{,limiter
26,lxh,identifier
26,-=,operator
26,c,identifier
26,;,limiter
27,break,keyword
27,;,limiter
28,;,limiter
29,int,keyword
29,[a,wrong identifier
29,;,limiter
30,[a,wrong identifier
30,=,operator
30,02,wrong number
30,;,limiter
31,return,keyword
31,0,number
31,;,limiter
32,;,limiter
C:\Users\Xiaohao Li\source\repos\软件课程设计(II)\Analyzer\x64\Debug\Lexical_Analyzer.exe (进程 6088) 已退出，代码为 0。
按任意键关闭此窗口。 . . .
```

3. 语法分析器

3.0 项目流程



其中最复杂的部分就是求 FIRST 集合和求项目集的闭包，这两部分的函数花了我很长

时间，基本语法分析的一大半时间都还在实现这两个函数与 debug 这两个函数的缺陷上。

下面我来具体讲讲各个部分的工作量与需要注意的地方。

3.1 设计 2 型文法（上下文无关文法）

#grammar2_Lxh

#S 是整个源代码 source_program 的开始，A 是一个个函数（因为源代码是由一个个函数定义的）

$S \rightarrow A$

#B 表示每个函数体的定义,A 能推向空（@表示空）是因为源程序可以由 0 个及以上的函数体组成

$A \rightarrow BA$

$A \rightarrow @$

#B 表示每个函数体的定义，每个函数体的定义由：函数返回值类型 函数名(参数){函数声明} 这几个部分组成

#函数返回类型 type 为 t，函数名为一个合法的标识符，函数名因为是一个 identifier 标识符，故用终结符 i 表示

$B \rightarrow ti(D)\{E\}$

#在 input_token 读入语法分析时，type 分为 vcifd 分别对应 void char int float double，读到他们转为 t 即可

#(D)为函数的参数部分，可以由 0 个及以上的：类型 标识符组成

$D \rightarrow ti$

$D \rightarrow ti,D$

$D \rightarrow @$

#{E}为函数的主体声明部分，也就是主函数体，可以为空

#E 还可以由一些返回语句，声明语句，赋值语句，判断语句，循环语句，条件分支转移语句组成，这些语句用 F 推出

$E \rightarrow FE$

$E \rightarrow @$

#G 为返回语句，r 表示 return，n 表示 number，i 表示标识符

#return 也可以返回运算式子, 比如说 `return 2+a*(b+c)/3` (也就是 K, 下面有定义 K 表示一串运算式子)

`F->G`

`G->rn;`

`G->ri;`

`#G->rK;`

#H 为声明语句, I 为赋值语句, 由一个: `type identifier;` 或者 `type identifier o= value;` 组成其中 o 为计算运算符 `+ - * / & ^ % |`

#t 为 type, J 表示 value, 可以直接是一个 number (常量包括科学计数法和虚数)

#K 表示是 number, 标识符和运算符的结合, 是运算语句, 可以有括号运算

#L 表示判断语句 (没有; 因为判断语句是在 ifwhile 等语句的括号中写的), 其中 p 为 `< > == != <= >=` 表示判断的运算符

`F->H`

`H->ti;`

`H->ti=J;`

`#H->ti=K;`

`F->I`

`I->i=J;`

`I->io=J;`

`#I->i=K;`

`#I->io=K;`

`J->i`

`J->n`

`#K->JoK`

`#K->J`

`#K->(K)`

`L->JpJ`

#M 为循环语句, f 为 for, N 为 for 括号里的内容, E 为函数体的声明, 上面定义过了

#for 循环体的三个变量其实都可以为空, 多 7 种排列组合, 不再加以赘述

#H 为声明语句, I 为赋值语句, L 为判断语句, 这两个是上面定义过的, 注意 H 和 I

后面是有分号的，所以 N 的定义中不需要分号

#O 为不带分号的赋值语句，直接把上面的赋值语句复制过来去掉分号

#w 表示 while，L 为判断语句

F->M

M->f(N){E}

N->HL;O

N->IL;O

O->i=J

O->io=J

#O->i=K

#O->io=K

M->w(L){E}

#P 为分支转移，j 表示 if，k 表示 else

F->P

P->j(L){E}

P->j(L){E}k{E}

#Q,R 为循环语句中的跳转语句 b 表示 break,c 表示 continue

F->Q

Q->b;

F->R

R->c;

#endread

有了词法分析的 3 型文法的经验，语法分析文法写的很快，写的都是一些比较简单的语法，且注释都写在旁边了，应该能很快看懂，我写的二型文法包含：每个函数体的定义 B，函数参数 D，函数的主题部分 E，函数的所有种类语句 F，声明语句 H，赋值语句 I，判断语句 L，循环语句 while for，分支转移语句 break continue，条件语句 if，是一些比较基本且常用的语句，那些比较复杂的比如说包含限定符（const public private 这种）的语句和函数没有实现。

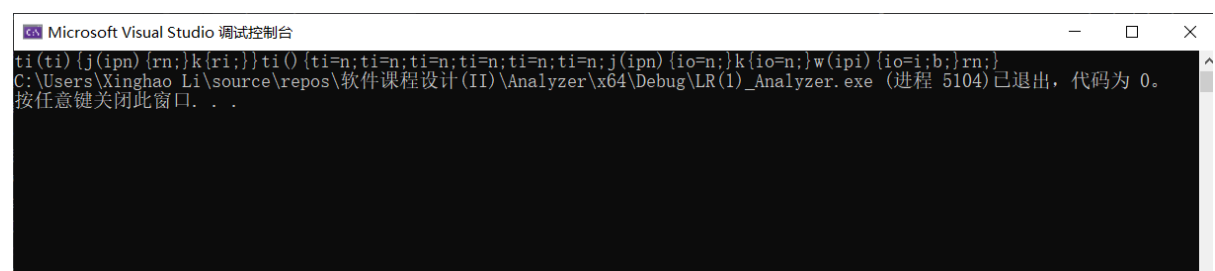
读入文法时需要对文法进行拓广，书上用的 S'->S，由于文法我用的是 vector<vector<char>> G;的二维 char 数组存的，假如用 S'就破坏了文法整体的规整性，所

以我用了 Z 来代替拓广文法的 S'，且由于我在 debug 时经常要用到 test.txt，然后切换回 grammar2.txt 时老是少打一个 2，导致数组越界，所以我加了一行判断 vector 为空直接 exit(0);退出程序以避免越界的 fatal error。具体的函数实现如下：

```
//读入二型文法
void read_grammar2() {
    ifstream read("grammar2.txt");
    string content;
    vector<char> temp; //存放每一行除了推导符以外的字符，并读入文法G
    set<char> t; //用于给map增加结点（非终结符和终结符）
    while (getline(read, content)) {
        if (content[0] == '#') {
            if (content == "#endread") break;
            else continue;
        }
        nonterminal[content[0]] = t;
        //接下来开始读入二型文法
        for (int i = 0; i < content.size(); i++) {
            char ch = content[i]; //ch存放当前的content的第i个字符
            if (ch != '-' && ch != '>') {
                temp.push_back(ch);
            }
            if (i >= 3 && (content[i] < 'A' || content[i] > 'Z')) terminal[content[i]] = t;
        }
        G.push_back(temp);
        temp.clear();
    }
    //debug时输错了ifstream的文件名导致vector out of range的严重错误，故加上exit(0)，当没读到文件时
    if (G.empty()) exit(0);
    //拓广文法, S' -> S, 这里方便表示将S'用Z表示
    temp.push_back('Z');
    nonterminal['Z'] = t;
    temp.push_back(G[0][0]);
    G.insert(G.begin(), temp);
}
```

3.2 处理输入的 token 串

对词法的输出 token 三元组进行处理，将他改成我们将要处理的一串 string，方便后面用项目集进行处理。主要的实现思路就是一堆 if else 没有什么含金量，故在这里不附上源码了，只将结果的终结字符串展示出来，这个串后面要用于语法分析。



3.3 计算 FIRST 集合

这一部分和 Clousure 闭包是整个项目最花时间的部分，这两个功能函数比较复杂，首先是求 FIRST 集合，终结符的 FIRST 集合很简单，就是他的本身，非终结符的 FIRST 集合很复杂。我用了一个 bool 类型的 `ischanged = true;` 作为循环变量，每次循环开始都要将其置为 false，然后定义一个 size 变量，表示当前非终结符的这次循环找新的 FIRST 集合的 FIRST 集大小，每次插入操作后，将插入后的 `FIRST.size()` 与这个 size 变量比较，如果变化了，说明了还能继续找 FIRST 集合，那就将 `ischanged` 置为 true，继续循环找，直到所有非终结符的 FIRST 集合都不改变了，说明已经找到了所有符号的 FIRST 集合，退出循环。

求 FIRST 集要分为一下几种情况，伪码我已经写在代码的注释中，我对其格式进行规整，并将其附在报告中：

1. $A \rightarrow a$ 直接是一个终结符的情况 || $A \rightarrow \epsilon$ 推向空的情况，直接将 `a||@` 加入 `FIRST(A)`

2. $A \rightarrow BC$ 的情况下

2.1 B 不能推出空，将 `FIRST(B)` 加入 `FIRST(A)`

2.2 B 能推出空

2.2.1 将 `{FIRST(B)-{@}}` 加入 `FIRST(A)`

2.2.2 `{FIRST(C)}` 加入 `FIRST(A)`

2.2.2 再分两种情况

(1) 若该产生式只有 $A \rightarrow B$ ，即 C 为空，`{FIRST(C)} = '@'`，将空加入 `FIRST(A)`

(2) $A \rightarrow BC$ 将 `FIRST(C)` 加入 `FIRST(A)`，若 C 也能推空的情况在下次循环中会再一次处理（类似于递归的思想）

这一部分我认为是求 FIRST 集合的核心，很容易考虑不周，我也是 debug 了很久，查阅了教材和各种博客，总结出的如下伪码，一开始心急想要直接编程解决，结果总是漏考虑到一些情况，浪费了很多时间，后期写出伪码后，直接对着伪码实现，反而进度较快，这也给了我启发，写代码还是要先构思再动手，实现起来都是一些重复的步骤，因为 set 集合的 insert 等函数 STL 已经为我们封装好了，这些操作还是很快的。

具体的代码实现如下：

```

//根据二型文法得到各个符号的First集合
void get_First() {
    for (auto &it : terminal) {
        it.second.insert(it.first); //终结符的first是他本身
    }

    bool ischanged = true; //每次insert操作会看该终结符的FIRST.size() 是否改变, 只要改变了, 就要一直循环再找一次
    while (ischanged) {
        ischanged = false;
        for (auto& it : nonterminal) {
            //对于每个非终结符遍历一次所有文法, it.first为当前非终结符, it.second为该终结符的First集合
            for (int i = 0; i < G.size(); i++) {
                //G是一个二维数组, 行表示每行文法, 列表示每行文法的每个字符
                int size = it.second.size();
                if (G[i][0] == it.first) {
                    //遍历到了it迭代器指向的那个nonterminal
                    auto iter = terminal.find(G[i][1]); //iter迭代器去终结符的map中寻找, 找到则返回对应地址, 否则
                    if (iter != terminal.end() || G[i][1] == '@') {
                        //A->a直接是一个终结符的情况 || A->@推向空的情况
                        it.second.insert(G[i][1]);
                        if (it.second.size() > size) ischanged = true;
                    }
                    else {
                        //A->BC的情况下, 把FIRST(B)加入FIRST(A), 若B能推出空, 则把FIRST(C)加入FIRST(A)
                        bool can_to_empty = false;
                        for (auto j = nonterminal[G[i][1]].begin(); j != nonterminal[G[i][1]].end(); j++) {
                            if (*j == '@') can_to_empty = true;
                        }
                        if (!can_to_empty) {
                            //B不能推出空, 将FIRST(B)加入FIRST(A)
                            for (auto j = nonterminal[G[i][1]].begin(); j != nonterminal[G[i][1]].end(); j++) {
                                it.second.insert(*j);
                                if (it.second.size() > size) ischanged = true;
                            }
                        }
                        else {
                            //B能推出空, 将 {FIRST(B)-{@}} 并上 {FIRST(C)} 加入FIRST(A)
                            //1. {FIRST(B)-{@}}
                            for (auto j = nonterminal[G[i][1]].begin(); j != nonterminal[G[i][1]].end(); j++) {
                                if (*j != '@') it.second.insert(*j);
                                if (it.second.size() > size) ischanged = true;
                            }
                            //2. 然后分两种情况
                            //2.1 若该产生式只有A->B, 即C为空, {FIRST(C)}='@', 将空加入A
                            if (G[i].size() == 2) {
                                it.second.insert('@');
                                if (it.second.size() > size) ischanged = true;
                            }
                            //2.2 A->BC将FIRST(C)加入FIRST(A), 若C也能推空的情况在下次循环中会再一次处理 (类似
                            else {
                                for (auto j = nonterminal[G[i][2]].begin(); j != nonterminal[G[i][2]].end(); j++) {
                                    it.second.insert(*j);
                                    if (it.second.size() > size) ischanged = true;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

3.4 求 Closure 闭包

求 Closure 闭包主要分为两个部分

- (1) 先初始化第一个项目集, 根据每个项目集的第一条项目, 构造这个项目集的闭包函数, 算法的原理我用的清华大学第三版编译原理教材的第 145 页伪码:

1.假定 I 是一个项目集, I 的任何项目都属于 **Closure (I)**

2.若有项目 $A \rightarrow \alpha \cdot B\beta, a$ 属于 $Clousure(I)$, $B \rightarrow \gamma$ 是文法中的产生式,

$B \in V^*$, $b \in FIRST(\beta a)$, 则 $B \rightarrow \cdot \gamma$, b 也属于 $Clousure(I)$

3.重复 2 直至 $Clousure(I)$ 不再扩大为止

实现过程中的注意点我也以伪码的形式留在了注释中, 下面是对其总结:

1. 每次循环先找到每条项目 item 的 \cdot 的位置, 因为 \cdot 是中文的圆角字符, 在 ASCII 码里面没有对应的值, 所以无法用 char 表示, 所以我在求闭包时用了 space 空格替代了, 找到了 \cdot 的位置, 开始从 \cdot 的后面一个终结符生成新的产生式。

然后要分成两种情况:

1.1 \cdot 在产生式的最后, 比如说 $S' \rightarrow S \cdot \#$ 那么这条产生式不能推出新的产生式故该项目集的 \cdot 不需要处理, break 处理下一个项目集

1.2 \cdot 在产生式的中间, 比如说 $A \rightarrow \alpha \cdot B\beta, a$, 那么就要到对这条项目集进行处理遍历所有的二型文法 G , 查找 \cdot 后面的非终结符所有的产生式, 并将新的产生式加入项目集

生成新的产生式后不能直接插入到当前的 project 中, 先判断该新的项目是否存在
在当前项目集, 这里又要分情况讨论

1.2.1 若不存在则直接向项目集中插入这条产生式

1.2.2 若存在则记录下编号, 不用插入产生式

2.产生式处理完毕, 就要处理向前搜索符号集了, 这里还是需要分类讨论, 用一个

set<char> temp 暂存计算出的非终结符:

形如产生式: $A \rightarrow \alpha \cdot B\beta, a$ 计算向前搜索符号集, 即计算 $FIRST(\beta a)$, 分4种情况:

2.1 该产生式没有 β , 产生式为 $A \rightarrow \alpha \cdot B, a$, 向前搜索符号集为 $\{a\}$

2.2 β 的第一个字符为终结符 $\{b\}$, 产生式为 $A \rightarrow \alpha \cdot Bb, a$, 那么向前搜索符号集为该终结符 $\{b\}$

2.3 β 的第一个字符为非终结符 C , 产生式为 $A \rightarrow \alpha \cdot BC, a$, 且该非终结符能推出空, 向前搜索符号集为 $FIRST(C) - \{\epsilon\}$ 并上 $\{a\}$, 其中 γ 为去除 β 的第一个符号 C 后的字符串

2.4 β 的第一个字符为非终结符 C , 产生式为 $A \rightarrow \alpha \cdot BC, a$, 且该非终结符不能推出空, 向前搜索符号集为 $FIRST(C)$

3. 现在新项目 closure[i].project[j] 的向前搜索符号集暂存在 set<char> temp2 中对向前搜索符号集的处理分为2种:

1.temp1 原先不在该项目集中, 根据原产生式后面部分的 FIRST 集合, 来确定其向前搜

索符号集

2.temp1原先在项目集中，根据原产式（之前记录下的编号），得到新的向前搜索符号集插入原先产生式的向前搜索符号集合

这一部分真的很复杂，从伪码就可以看得出来，很容易漏考虑到一些情况，也是建议先写伪码，理解原理，防止写源代码时漏考虑一些情况。第一部分的代码实现如下：


```

//求每个项目集第一条产生式扩展出来的闭包项目
for (int j = 0; j < closure[i].project.size(); j++) {
    //遍历已有项目集, j为每个项目集中的第j个项目
    for (int k = 0; k < closure[i].project[j].size(); k++) {
        //遍历该项目集, 找到·的位置, k为每个项目中各个终结符非终结符的编号
        if (closure[i].project[j][k] == '·') {
            //找到了·的位置, 开始从·的后面一个终结符生成新的产生式
            if (k == closure[i].project[j].size() - 1) {
                //·在产生式的最后, 比如说S' -> S·, # 那么这条产生式不能推出新的产生式
                //故该项目集的·不需要处理, break处理下一个项目集
                break;
            }
            for (int l = 0; l < G.size(); l++) {
                //·在产生式的中间的情况, 比如说A -> α·BB, a
                //遍历所有的二型文法G, 查找·后面的非终结符所有的产生式, 并将新的产生式加入项目集
                if (G[l][0] == closure[i].project[j][k + 1]) {
                    vector<char> templ(G[l]);
                    templ.insert(templ.begin() + 1, ' ');
                    //生成新的产生式后不能直接插入到当前的project中
                    //先判断该新的项目是否存在在当前项目集, 若存在则记录下编号, 若不存在则插入
                    int project_is_exist = 0;
                    for (int m = 0; m < closure[i].project.size(); m++) {
                        if (templ == closure[i].project[m]) {
                            project_is_exist = m;
                            break;
                        }
                    }
                    if (project_is_exist == 0) closure[i].project.push_back(templ);
                    //接下来开始处理向前搜索符号集
                    set<char> temp2; //用来保存暂时的向前搜索符号集, 后面用于push_back的
                    /*
                    形如产生式: A -> α·BB, a 计算向前搜索符号集, 即计算FIRST(Ba), 分4种情况:
                    1. 该产生式没有B, 产生式为A -> α·B, a, 向前搜索符号集为{a}
                    2. B的第一个字符为终结符(b), 产生式为A -> α·Bb, a, 那么向前搜索符号集为该终结符{b}
                    3. B的第一个字符为非终结符C, 产生式为A -> α·BC, a, 且该非终结符能推出空, 向前搜索符号集为FIRST(C)
                    4. B的第一个字符为非终结符C, 产生式为A -> α·BC, a, 且该非终结符不能推出空, 向前搜索符号集为FIRST(C)
                    */
                    bool deduce_empty = true; //判断B的第一个字符C能不能推出空
                    int n = 0; //记录去除B的第n个字符
                    while (deduce_empty) {
                        deduce_empty = false;
                        if (k + n + 1 == closure[i].project[j].size() - 1) {
                            //第一种情况B为空
                            for (auto it : closure[i].search_forward[j]) temp2.insert(it);
                        }
                        else if (terminal.find(closure[i].project[j][k + n + 2]) != terminal.end()) {
                            //第二种情况, 首字符为终结符
                            temp2.insert(closure[i].project[j][k + n + 2]);
                        }
                        else {
                            //第三、四种情况B的第一个字符为非终结符
                            //找到C的FIRST集中所有的终结符
                            set<char> temp_nonter(nonterminal.find(closure[i].project[j][k + n + 2]) -> second);
                            for (auto it : temp_nonter) {
                                if (it == 'ε') {
                                    //如果第一个字符能推出空
                                    deduce_empty = true; //再进行一次循环, 找FIRST(γa)并插入
                                    n++;
                                }
                                else {
                                    //如果第一个字符不能推出空
                                    temp2.insert(it);
                                }
                            }
                        }
                    }
                    /*
                    现在新项目closure[i].project[j]的向前搜索符号集暂存在set<char> temp2中
                    对向前搜索符号集的处理分为2种:
                    1. templ原先不在该项目集中, 根据原产生式后面部分的FIRST集合, 来确定其向前搜索符号集
                    2. templ原先在该项目集中, 根据原产生式, 得到新的向前搜索符号集插入原先产生式的向前搜索符号集
                    */
                    if (project_is_exist == 0) {
                        //第一种情况, templ不在项目集中
                        closure[i].search_forward.push_back(temp2);
                    }
                    else {
                        //第二种情况, templ在项目集中
                        for (auto it : temp2) {
                            closure[i].search_forward[project_is_exist].insert(it);
                        }
                    }
                }
            }
        }
    }
}
//已经处理过这个项目集的·了, break处理下一个项目集
break;
}

```

- (2) 判断是否是已经出现过得闭包并，并确定每个项目集间的关系，例如 I_0 输入符号 S 到 I_1 ， I_1 输入符号 a 到 I_4 等等

第二部分的实现注意点如下：

1. 每次循环先找到每条项目 item 的 \cdot 的位置，因为 \cdot 是中文的圆角字符，在 ASCII 码里面没有对应的值，所以无法用 char 表示，所以我在求闭包时用了 space 空格替代了，找到了 \cdot 的位置，

根据 \cdot 的位置，两种情况处理：

1.1 如果 \cdot 在产生式最后，那么这个产生式不会有闭包间的 GOTO 边，break，处理下一个产生式

1.2 否则 \cdot 不在产生式的最后，则将点后移一位 然后计算闭包之间的边，生成新的闭包
这里要定义一个 bool is_new_closure = false; 判断新生成的项目是否是新的闭包（用于判断要不要生成新闭包），遍历所有的闭包集合，将新的 closure 与每个闭包比较，看是否是新的闭包，这里要分成 3 种情况：

1.2.1 跟每个闭包第一条产生式的向前搜索符号集 size 对比，看是否一样，因为如果跟第一条产生式的 project 和 search_forward 都一样，就能知道闭包必然也一样（因为是第一条产生式推出的整个闭包），不一样则说明是新的闭包，break

1.2.2 若一样，则比较向前搜索符号集，因为假设产生式一样，但是向前搜索符号集不一样，在 LR(1) 里面是定义为不需要合并的同心集，也是算新的闭包。不一样则说明是新的闭包，break

1.2.3 若向前搜索符号集也一样，那说明是已有的闭包，求 go 函数（也就是闭包集合 DFA 之间的边，为后面求 ACTION GOTO 表做铺垫）

2.1 如果是新的闭包，直接插入该闭包

注：这种情况容易被忽视，需要单独写一个 if 来判断，if 的条件为是一个新的闭包且 go 函数已经有了值得情况，例如这种情况，第一次第一行 T 的 go 函数会计算出来，第二次要把第二行的产生式加入第一次的 go 函数指向的那个闭包

$$E \rightarrow \cdot T, \quad)/+$$
$$T \rightarrow \cdot T * F, \quad)/+/*$$

2.2 如果不是新的闭包，计算闭包之间的 go 函数，break

第二部分的代码实现如下：

```
//判断该闭包是否是已经出现的闭包,并计算闭包之间相互转换的边
for (int j = 0; j < closure[i].project.size(); j++) {
    //遍历本项目集
    for (int k = 0; k < closure[i].project[j].size(); k++) {
        //扫描每个产生式,找到·的位置
        if (closure[i].project[j][k] == '·') {
            //debug时发现project后面,少写了一个[j],导致vector溢出,这么小的错误debug了半天
            //因为现在是对第i个项目集(闭包)的第j个项目(产生式)进行遍历,找到他·的位置,所以显然要用project[j].size()
            if (k == closure[i].project[j].size() - 1) {
                //如果·在产生式最后,那么这个产生式不会有新的闭包, break, 处理下一个产生式
                break;
            }
            //否则·不在产生式的最后,则将点后移一位
            //然后计算闭包之间的边,生成新的闭包
            vector<char> new_closure_project(closure[i].project[j]); //新的产生式(有原产生式推出,故初始化为原产生式closure[i].
            new_closure_project[k] = new_closure_project[k+1]; //new_closure_project[k]存放的是待移入字符
            new_closure_project[k+1] = '·';
            char ch = new_closure_project[k]; //ch存放待移入字符(后面嵌套太多了,用new_closure_project[k]很混乱)
            set<char> new_closure_search_forward(closure[i].search_forward[j]); //新的向前搜索符号集(与原产生式向前搜索符号集一样)
            bool is_new_closure = false; //判断新生成的项目是否是新的闭包(用于判断要不要生成新闭包)
            for (int m = 0; m < closure.size(); m++) {
                //遍历所有的闭包,看看该闭包是否已经出现过(项目集相等且向前搜索符号集也相等)
                for (int n = 0; n < closure[m].project.size(); n++) {
                    //遍历每个闭包的所有项目集
                    is_new_closure = false;
                    if (new_closure_project == closure[m].project[n]) {
                        //这里的下标也是debug了半天,应该是跟第一条产生式的向前搜索符号集size对比,看是否一样,因为如果跟第一条产生式
                        //而不是跟整个项目集搜索符号集的大小比,第一次写的时候search_forward后面少写了个[0],导致debug了半天浪费了
                        if (closure[m].search_forward[0].size() != new_closure_search_forward.size()) {
                            is_new_closure = true;
                            break;
                        }

                        auto it1 = closure[m].search_forward[0].begin();
                        for (auto it2 = new_closure_search_forward.begin(); it2 != it1; it2++) {
                            if (*it2 != *it1) {
                                is_new_closure = true;
                                break;
                            }
                        }
                        /*if (it1 == closure[m].search_forward[0].end()) {
                            is_new_closure = false;
                            break;
                        }*/
                        it1++;
                    }
                    if (is_new_closure == false) {
                        closure[i].go[ch] = m;
                        break;
                    }
                }
                else is_new_closure = true;

                if (is_new_closure == false) break;
            }
            if (is_new_closure == false) break;
        }
    }
}

/*
例如这种情况,第一次第一行T的go函数会计算出来,第二次要把第二行的产生式加入第一次的go函数指向的那个闭包
E->·T, )/+
T->·T*F, )/+/*
*/
if (closure[i].go.count(new_closure_project[k]) != 0 && is_new_closure) {
    closure[closure[i].go[ch]].project.push_back(new_closure_project);
    closure[closure[i].go[ch]].search_forward.push_back(new_closure_search_forward);
    break;
}

//如果是没出现过的新的闭包,插入该闭包
if (is_new_closure == true) {
    Project new_closure;
    new_closure.project.push_back(new_closure_project);
    new_closure.search_forward.push_back(new_closure_search_forward);
    closure.push_back(new_closure);
    //go是一个char到int的映射,表示输入对应的char,该闭包会跳转至哪个闭包,
    //比如A->·BC, a, 那么他的输入符号就是B,也就是对应new_closure_project[k]
    //由于输入的符号是new_closure_project[k],新的闭包是closure.size()-1
    closure[i].go[ch] = closure.size() - 1;
}

}

i++;
```

3.5 画出 LR(1)的 ACTION GOTO 表

这一部分在构造完闭包后倒是不复杂，写好伪码后考虑周全很容易就可以得到。

依旧是找到每个式子的·的位置后（我代码中的space空格）分为四种情况（思想方法见清华大学教材p146）

1. $S' \rightarrow S \cdot, \#$ 属于 Ik ，则ACTION表中[k,#]为acc
2. $A \rightarrow \beta \cdot, a$ 属于 Ik ，则ACTION表中[k,a]为 r_j ，表示用第j条产生式规约
3. $A \rightarrow \alpha \cdot a \beta, b$ 属于 Ik ，且 Ik 移进 a 转移到 Ij ，则ACTION表中[k,a]为 S_j ，表示把移入符号 a 和状态 j 分别移入文法符号栈和状态符号栈
4. $A \rightarrow \alpha \cdot B \beta, a$ 属于 Ik ，且 Ik 移进 B 转移到 Ij ，则GOTO表中[k,B]为 j ，表示置当前文法符号栈顶为 A ，状态栈顶为 j

这个倒是不困难，值得一提的是表是我发现一开始写的二型文法不是 LR(1)文法，产生了填 Table 表时产生了归约归约冲突，我在归约的那个 if 输出了产生冲突的项目集号，并输出所有项目集 debug，确实找到了冲突所在，我将文法中所有运算式子 K 都注释掉后，冲突就解决了，文法又成了 LR(1)文法。

```
bool get_LR1Table() {
    /*
    分为四种情况（思想方法见清华大学教材p146）
    1. S' -> S ·, # 属于 Ik, 则ACTION表中[k, #]为acc
    2. A -> β ·, a 属于 Ik, 则ACTION表中[k, a]为rj, 表示用第j条产生式规约
    3. A -> α · a β, b 属于 Ik, 且 Ik移进a转移到Ij, 则ACTION表中[k, a]为Sj, 表示把移入符号a和状态j分别移入文法符号栈和状态符号栈
    4. A -> α · B β, a 属于 Ik, 且 Ik移进B转移到Ij, 则GOTO表中[k, B]为j, 表示置当前文法符号栈顶为A, 状态栈顶为j
    */
    for (int i = 0; i < closure.size(); i++) {
        for (int j = 0; j < closure[i].project.size(); j++) {
            for (int k = 0; k < closure[i].project[j].size(); k++) {
                if (closure[i].project[j][k] == ' ') {
                    //找到每条文法的·
                    if (k == closure[i].project[j].size() - 1) {
                        //若·在文法的最后,要对式子进行规约
                        //分为1 2两种情况
                        if (closure[i].project[j][0] == 'Z') {
                            //对应最上面注释的第一种情况S' -> S ·, #
                            map<string, char> m;
                            string state = to_string(i);
                            m[state] = '#';
                            if (table.find(m) != table.end() && table[m] != "acc") {
                                cout << "error";
                                return false;
                            }
                            else {
                                table[m] = "acc";
                            }
                        }
                        else {
                            //对应最上面注释的第二种情况A -> β ·, a
                            int G_num = 0;
                            for (int x = 0; x < G.size(); x++) {
                                vector<char> y(closure[i].project[j]);
                                y.pop_back(); //closure[i].project[j]比二型文法多了最后的一个·, 在vector里为space空格, pop出来便可
                                if (y == G[x]) {
                                    G_num = x;
                                    break;
                                }
                            }
                            map<string, char> m;
                            string state = to_string(i);
                            for (auto it : closure[i].search_forward[j]) {
                                m[state] = it;
                                if (table.find(m) != table.end() && table[m] != "r" + to_string(G_num)) {
```

3.6 用 ACTION GOTO 表对 token 语法分析

参考的上学期编译原理 LR(1)分析的作业，因为我在书上没有找到有关分析过程，分为步骤，状态栈，符号栈，ACTION GOTO 操作：

每次分析取状态栈最右边的状态和待输入符号最左边的符号，还是分成 4 种情况：

1.若 ACTION GOTO 表中对应状态对应符号的位置为 ACC，则表示分析成功，return true;

2. 若 ACTION GOTO 表中对应状态对应符号的位置为 Sk，则表示要进行移入操作，在状态栈中加入状态 k，将待输入符号串移入一位符号栈

3. 表里对应位置的字符串第一位为 r，表里的值为 rk，则表示要进行归约操作，操作如下：

3.1 找到对应的文法，文法右边有几个字符，符号栈状态栈就要 pop 几次

3.2 与此同时，符号栈 push_back 该条文法的左边字符，此时的状态栈顶遇到左边的字符，转入 GOTO 表中的那个状态，并将该状态压入栈中

4. 表里那个位置为空，有两种情况：

4.1 去该状态的@那一列，找到 Sk 并将 Sk 的 k 入状态栈，将@入符号栈，continue 进入下一次循环分析

4.2 去该状态的@那一列为空，则表示分析出错

其中 4.1 是整个流程的精髓所在，是对空的产生式的归约处理，一开始我认为只要表的对应位置为空，就直接输出 error 分析出错，但是后来发现空产生式就没法归约了，而我文法中的空产生式都用的@指代的，所以想到了这个 debug 方法。

代码的实现如下：

```

//对输入的tokens，根据LR1的ACTION_GOTO表进行分析
bool LR1_Analyze() {
    cout << "步骤\t\t" << "状态栈\t\t\t\t\t" << "符号栈\t\t\t\t\t" << "传入串\t\t\t\t\t" << "ACTION_G0" << endl;
    output << "步骤\t\t" << "状态栈\t\t\t\t\t" << "符号栈\t\t\t\t\t" << "传入串\t\t\t\t\t" << "ACTION_G0" << endl;
    vector<string> status_stack; //状态栈
    status_stack.push_back("0"); //初始化状态栈
    vector<char> symbol_stack; //符号栈
    symbol_stack.push_back("#"); //初始化符号栈
    string str_token = get_token_string(); //传入串
    str_token += "#"; //初始化传入串
    int steps = 1;
    while (true) {
        output << steps << "\t\t";
        cout << steps++ << "\t\t";
        for (int i = 0; i < status_stack.size(); i++) {
            cout << status_stack[i];
            output << status_stack[i];
        }

        cout << "\t\t\t\t\t";
        output << "\t\t\t\t\t";
        for (int i = 0; i < symbol_stack.size(); i++) {
            cout << symbol_stack[i];
            output << symbol_stack[i];
        }

        cout << "\t\t\t\t\t";
        output << "\t\t\t\t\t";
        for (int i = 0; i < str_token.size(); i++) {
            cout << str_token[i];
            output << str_token[i];
        }

        cout << "\t\t\t\t\t";
        output << "\t\t\t\t\t";
        string status_top = status_stack[status_stack.size() - 1]; //状态栈最右边那个状态
        char ch_head = str_token[0]; //传入串最左边那个字符
        map<string, char> m;
        m[status_top] = ch_head; //根据映射，到ACTION_GOTO表里找对应位置的字符串是acc 还是rk 还是Sk
        if (table[m] == "acc") {
            //表里对应位置的字符串为acc，则表示分析成功
            cout << "YES!" << endl;
            output << "YES!" << endl;
            return true;
        }

        else if (table[m][0] == 'S') {
            //表里对应位置的字符串第一位为S，表里的值为Sk，则表示要进行移入操作，在状态栈中加入状态k，将传入串符号串移入一位
            string temp;
            for (int i = 1; i < table[m].size(); i++) temp += table[m][i];
            status_stack.push_back(temp);
            symbol_stack.push_back(str_token[0]);
            str_token = str_token.substr(1);
            cout << table[m] << "：将状态" << atoi(temp.c_str()) << "压入状态栈" << endl;
            output << table[m] << "：将状态" << atoi(temp.c_str()) << "压入状态栈" << endl;
        }

        else if (table[m][0] == 'r') {
            /*表里对应位置的字符串第一位为r，表里的值为rk，则表示要进行归约操作，操作如下：
            1. 找到对应的文法，文法右边有几个字符，符号栈状态栈就要pop几次
            2. 与此同时，符号栈push_back该条文法的左边字符，此时的状态栈遇到左边的字符，转入GOTO表中的那个状态，并将该状态压入*/
            string temp;
            for (int i = 1; i < table[m].size(); i++) temp += table[m][i];
            int pop_frequency = G[atoi(temp.c_str())].size() - 1;
            while (pop_frequency-- > 0) {
                status_stack.pop_back();
                symbol_stack.pop_back();
            }
            symbol_stack.push_back(G[atoi(temp.c_str())][0]);
            char symbol_temp = G[atoi(temp.c_str())][0];
            string status_temp = status_stack[status_stack.size() - 1];
            map<string, char> m_temp;
            m_temp[status_temp] = symbol_temp;
            status_stack.push_back(table[m_temp]);
            cout << table[m] << "：用";
            output << table[m] << "：用";
            for (int i = 0; i < G[atoi(temp.c_str())].size(); i++) {
                cout << G[atoi(temp.c_str())][i];
                output << G[atoi(temp.c_str())][i];
                if (i == 0) {
                    cout << "->";
                    output << "->";
                }
            }

            cout << "归约，并将状态" << table[m_temp] << "入栈" << endl;
            output << "归约，并将状态" << table[m_temp] << "入栈" << endl;
        }

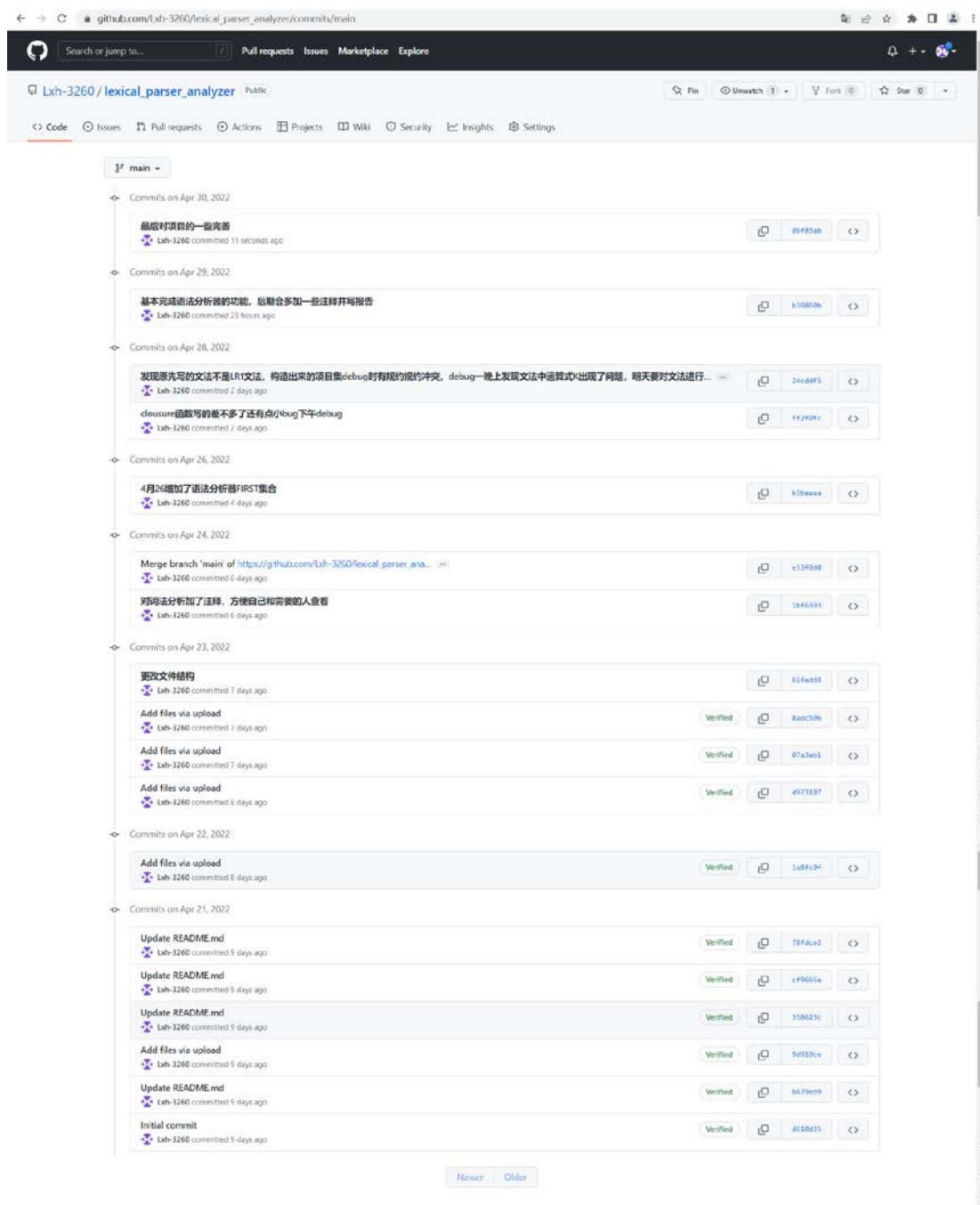
        else {
            /*表里那个位置为空，有两种情况，
            1. 去0那一列找，并将Sk的k入状态栈
            2. 表示分析出错*/
            string temp_status = status_stack[status_stack.size() - 1];
            map<string, char> temp_m;
            temp_m[temp_status] = '0';
            if (table.find(temp_m) != table.end()) {
                //对应上面注释的第一种情况，去0那一列找，并将Sk的k入状态栈
                string temp;
                for (int i = 1; i < table[temp_m].size(); i++) temp += table[temp_m][i];
                status_stack.push_back(temp);
                symbol_stack.push_back('0');
                cout << "要用空符号集归约，本文法的空符号集为0，故在状态栈加入" << temp << "符号栈加入0" << endl;
                output << "要用空符号集归约，本文法的空符号集为0，故在状态栈加入" << temp << "符号栈加入0" << endl;
                continue;
            }

            else {
                //对应上面注释的第二种情况，分析出错
                cout << "NO!" << endl;
                output << "NO!" << endl;
                return false;
            }
        }
    }
}

```


[illegible]

本次课程设计的开发周期从开始复习编译原理的有关知识到整个项目完成大概花了两周时间，我也更加深刻地理解了上学期马勇老师人机交互课上所说的“知识不会消失，只是你的大脑失去了对它的检索能力”，当我再看到书上的例子和之前记得笔记的时候我可以很快地回忆起有关的知识。



仓库的地址为 https://github.com/Lxh-3260/lexical_parser_analyzer，因为我一开始在理解题意方面花了很多时间，我认为假如有人指导我设计正规文法、二型文法，理解设计要求我可能进度又能快一些，省去一些搜集资料的无用功，所以我的代码中有很多注释，报告也写的比较完善。希望后来的人能从我的词法分析器和语法分析器中学到一些东西吧，取其精华去其糟粕，对我的项目进一步优化，也不枉我这两个星期的努力。

最后，祝项老师身体健康，科研顺利。

李兴昊

2022.4.30