南京理工大学

NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

软件课程设计（II）报告

作　者：叶健博

学　院：计算机科学与工程学院

专　业：计算机科学与技术

班　级：9201062302

学　号：9201080N0139

指导老师：项欣光

2023 年 3 月

一、设计总览

1. 设计要求

（1）创建一个词法分析程序，该程序支持分析常规单词。

要求：必须使用 DFA（确定性有限自动机）或 NFA（不确定性有限自动机）来实现此程序。 程序有两个输入：一个文本文档，包括一组 3º 型文法（正规文法）的产生式；一个源代码文本 文档，包含一组需要识别的字符串（程序代码）。 程序的输出是一个 token（令牌）表，该表由 5 种 token 组成：关键词，标识符，常量，限定符和运算符。词法分析程序可以准确识别:科学计数法形式的常量（如 0.314E+1），复数常量（如 10+12i），可检查整数常量的合法性，标识符的合法性（首字符不能为数字等），尽量符合真实常用高级语言要求的规则。

(2)创建一个使用 LL(1) 方法或 LR(1) 方法的语法分析程序。

要求：程序有两个输入：1）一个是文本文档，其中包含 2º 型文法（上下文无关文法）的产生式集合；2）任务 1 词法分析程序输出的（生成的）token 令牌表。程序的输出包括：YES 或 NO （源代码字符串符合此 2º 型文法，或者源代码字符串不符合此 2º 型文法）；错误提示文件，如果有语法错标示出错行号，并给出大致的出错原因。

2. 项目基本内容

（1）词法分析器

a)输入：一组正规文法、程序源代码

b)输出：Token 表

c)流程：读入正规文法，根据该文法构建 NFA，确定化化该 NFA 得到 DFA，根据 DFA 对输入的程序源代码进行分析，生成 Token 表。

（2）语法分析器

a)输入：一组二型文法、词法分析产生的 Token 表

b)输出：LR(1)分析结果（接收：YES；不接收：NO）

c)流程：读入上下文无关文法，构造 LR(1)项集族，同时生成 LR(1)自动机与 ACTION、GOTO 表，根据 DFA 与 ACTION、GOTO 表对词法分析得到的 Token 表进行语法分析，得到接收结果。

3. 开发环境

（1） 程序设计语言：C++

（2） 操作系统：Microsoft Windows 11

（3） 编译工具：MinGW64 9.0 CMake 3.23.2GDB12.1

（4） IDE：JetBrain CLion 2022

4. 项目结构

（1） 文件夹：Compiler 项目文件夹

（2） 源代码文件

a) FA.cpp / FA.h 自动机类，包含词法分析相关代码

b) LR.cpp / LR.h 包含 LR(1)语法分析相关代码

c) main.cpp 程序主函数，用于调试代码以及展示结果

（3） .txt 文件

a) GRAMMAR.txt: 词法分析需要的正规文法（三型文法）

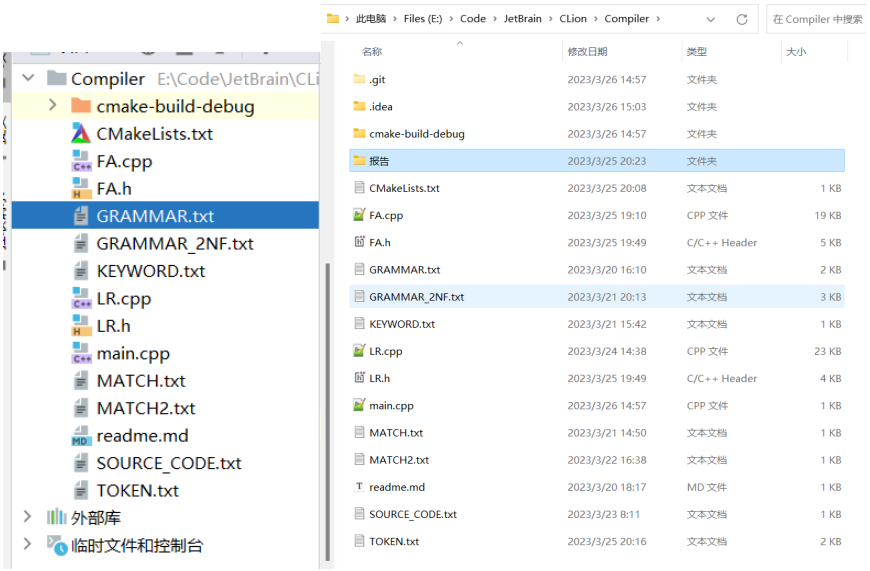
b) SOURCE_CODE.txt: 词法分析中需要的源程序

c) KEYWORD.txt: 关键字列表，用于区分标识符与关键字

d) TOKEN.txt: Token 表，词法分析程序处理后得到的 Token 表保存在这里，同时作为语法分析程序的输入

e) GRAMMAR_2NF.txt: 语法分析程序需要的上下文无关文法（二型文法）

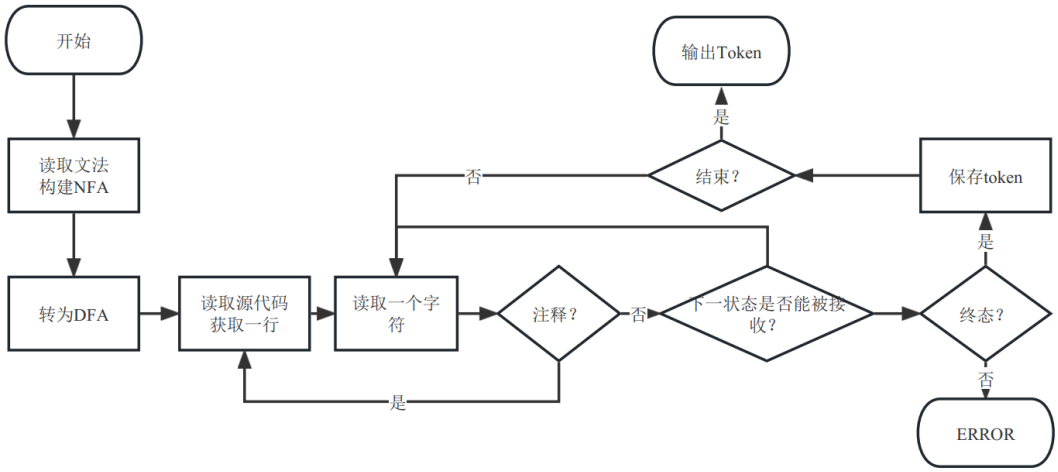
f) MATCH.txt/MATCH2.txt: 语法分析程序中字符与字符串（特定类型）的对应，方便语法分析的处理



二、各模块详细设计

1. 词法分析器（主体为 FA 类）

原理：基于类似 C/C++的语法，添加复数类型，先根据语法的输入生成 NFA 再确定化为 DFA，存储到对应的数据结构中，逐行读取输入的源代码，每行逐个字符分析，若遇到空格则判断是否可推出终态；若不是空格，则根据 DFA 中存储的映射判断是继续读入字符还是推出终态。每推出一次终态就将当前的分析结果存储到 token 列表中，并记录行号和判断类型。分析完毕输出结果。



（1）识别类型：

（2） 正规文法

类型	具体说明
关键字	"if"、"then"、"else"、"bool"、"char"、"int"、"double"、"const"、"while"、"do"、"begin"、"end"
标识符	可以由数字、字母、下划线组成。
常量	可以识别整数、小数、科学计数法、复数
运算符	单目："="、"+"、"-"、">"、"<"、"*"、"/"、"%" 双目："==" ">=" "<="
界符	";"、"("、")"、"{"、"}"、","

```
// 开始 -> 标识符（关键字）|常量|分隔符|运算符
Start -> Identifier | Constant | Separator | Operator
//标识符（关键字）：字母或者下划线开头
Identifier -> aI | bI | cI | dI | eI | fI | gI | hI | iI | jI | kI |
lI | mI | nI | oI | pI | qI | rI | sI | tI | uI | vI | wI | xI | yI |
zI | _I
I -> $ | aI | bI | cI | dI | eI | fI | gI | hI | iI | jI | kI | lI |
mI | nI | oI | pI | qI | rI | sI | tI | uI | vI | wI | xI | yI | zI |
_I | 0I | 1I | 2I | 3I | 4I | 5I | 6I | 7I | 8I | 9I
//常量：整数、小数、虚数、科学计数法
Constant -> A | B | C | D
//整数
A -> +A0 | -A0 | A0 | 0
A0 -> 1A1 | 2A1 | 3A1 | 4A1 | 5A1 | 6A1 | 7A1 | 8A1 | 9A1
A1 -> $ | 0A1 | 1A1 | 2A1 | 3A1 | 4A1 | 5A1 | 6A1 | 7A1 | 8A1 | 9A1
//小数
B -> 0B3 | 1B1 | 2B1 | 3B1 | 4B1 | 5B1 |6B1 | 7B1 | 8B1 | 9B1
B1 -> .B2 | 1B1 | 2B1 | 3B1 | 4B1 | 5B1 |6B1 | 7B1 | 8B1 | 9B1 | 0B1
```

```
B2 -> 0B2 | 1B2 | 2B2 | 3B2 | 4B2 | 5B2 |6B2 | 7B2 | 8B2 | 9B2 | $
B3 -> .B2
//虚数
C->1C1|2C1|3C1|4C1|5C1|6C1|7C1|8C1|9C1
C1->0C1|1C1|2C1|3C1|4C1|5C1|6C1|7C1|8C1|9C1|iC2|C2
C2->$|+C
//科学计数法
D -> +D0 | -D0 | D0
D0 -> 1D1|2D1|3D1|4D1|5D1|6D1|7D1|8D1|9D1
D1 -> .D2 | D3
D2 -> 0D4 | 1D4 | 2D4 | 3D4 |4D4 | 5D4 | 6D4 | 7D4 | 8D4 | 9D4
D3 -> eD5
D4 -> eD5 | 0D4 | 1D4 | 2D4 | 3D4 |4D4 | 5D4 | 6D4 | 7D4 | 8D4 | 9D4
D5 -> +D6 | -D6 | D6
D6 -> 0D7 | 1D7 |2D7 | 3D7 | 4D7 | 5D7 |6D7 | 7D7 | 8D7 | 9D7
D7 -> $ | 0D7 | 1D7 |2D7 | 3D7 | 4D7 | 5D7 |6D7 | 7D7 | 8D7 | 9D7
//分隔符
Separator -> , | ; | ( | ) |{ | } | [ | ] | \t | \n | \0 | #
//运算符
Operator -> +O | -O | *O | /O |%O | =O | <O | >O
O -> $ | =
```

数据结构：

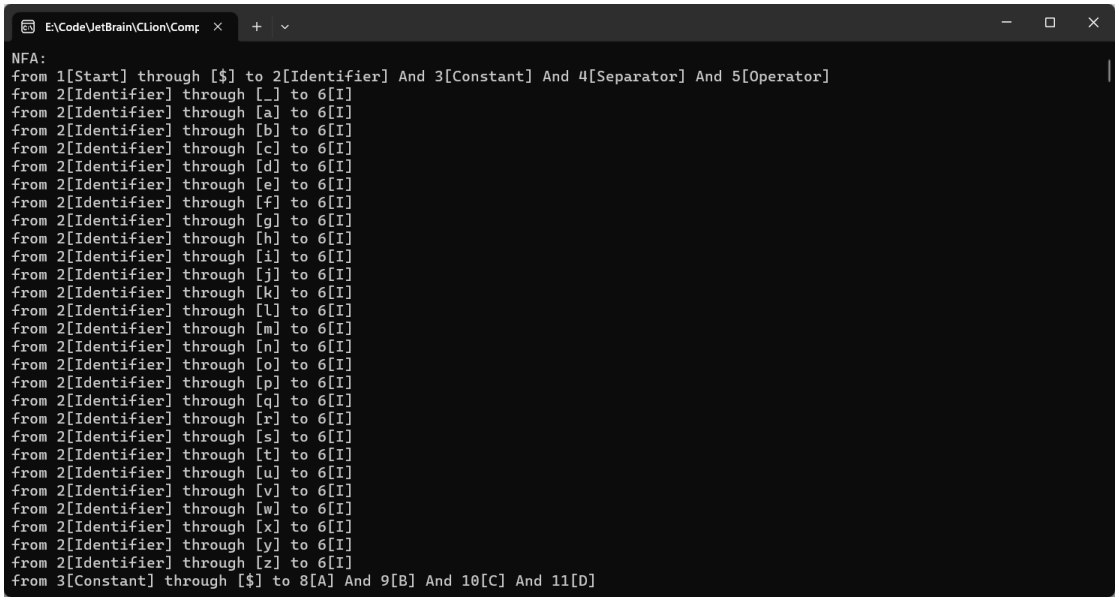
```
struct Node{      //节点（状态）
    int id;
    string name;
    bool operator < (const Node &o) const;
    bool operator == (const Node &o) const;
};

enum TokenType {    // Token 类型枚举
    KEYWORD,       //关键词，比如if、for、while、int
    IDENTIFIER,   //标识符，一般的单词，字母或下划线开头
    CONSTANT,   //常量，数字，比如29+52i（虚数）、1.14（小数）、514（整数）
    DELIMITER,   //界符，比如{}、（）、""、''、[]
    OPERATOR,    //运算符，比如"+"、"-"、"*"、"/"、"+="
    ERROR          //单独的错误信息标识
};

// Token 结构体
struct Token {
    TokenType type; // Token 类型
    string value;    // Token 值
```

```
    int line;           // 所在行数
};

class FA{                                                   //自动机 M = (K, ∑, f, S, Z)
private:
    Node startState;                                        //初态 S
    set<Node> endState;                                     //终态集 Z
    set<Node> States;                                       //状态集 K
    set<char> charSet;                                      //字母表
    map<Node,map<char,set<Node>>> transNFA; //NFA 状态转移 fN
    map<Node,map<char,Node>> transDFA;          //DFA 状态转移 fD
    int count = 0;                                          //节点计数
    map<string,set<Node>> stateCorr;            //状态对应关系
public:
    //获取非终结符集合
    const set<char> &getCharSet() const;
    //获取 DFA 状态转移
    const map<Node, map<char, Node>> &getTransDfa() const;
    //分析输入的语法，将其转换为 NFA
    void GrammarToNFA(const string& path);
    //输出 NFA 的状态转移关系
    void printEdge();
    //输出 DFA 的状态转移关系
    void printDFA();
    //输出字母表中所有的字母
    void printCharSet();
    //分析处理一行数据
    void deal(const string& l, const string& r);
   //将一个状态加入初态集（同时加入状态集），如果存在，返回该节点
   Node insertIntoStartState(const string& name);
   //将一个状态加入终态集
   Node insertIntoEndState(const string& name);
   //将一个状态加入状态集
   Node insertIntoState(const string& name);
   //求输入节点的ε-闭包
   set<Node> closure(const Node& node);
   //NFA 转 DFA
   void TransToDFA(FA nfa);
   //NFA 转 DFA 的处理
   void deal2(FA nfa,const Node& start, const set<Node>& n);
   //获取该自动机的初态
   const Node &getStartState() const;
   //获取该自动机的终态集
   const set<Node> &getEndState() const;
```

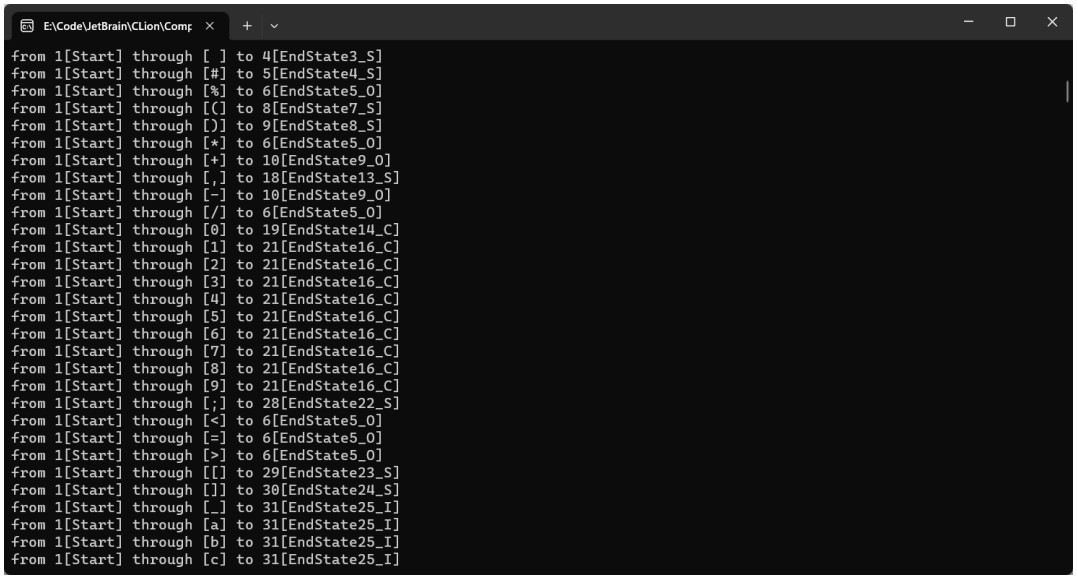
```
    //输入字符，求在传入的NFA 经过该字符到达的下一状态的集合，将集合返回
    set<Node> move(char input,const set<Node>& node,FA nfa);
};
```

运行相关信息：

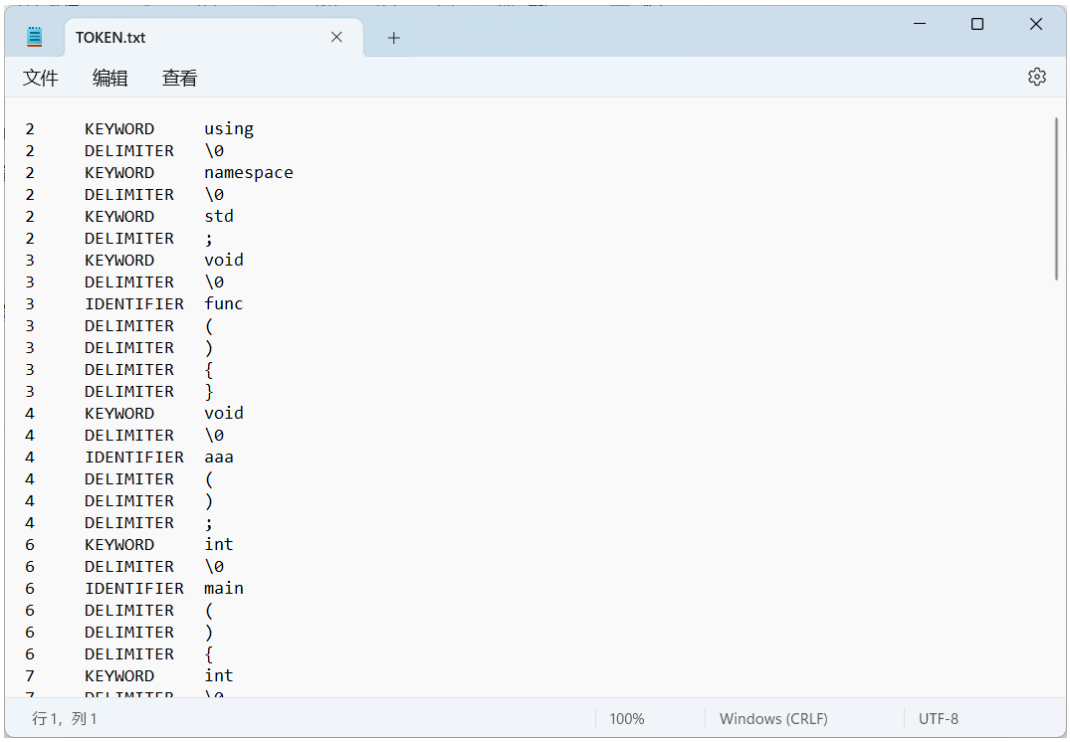
输出通过语法构建的 NFA（部分）：



输出 NFA 确定化后得到的 DFA（部分）：

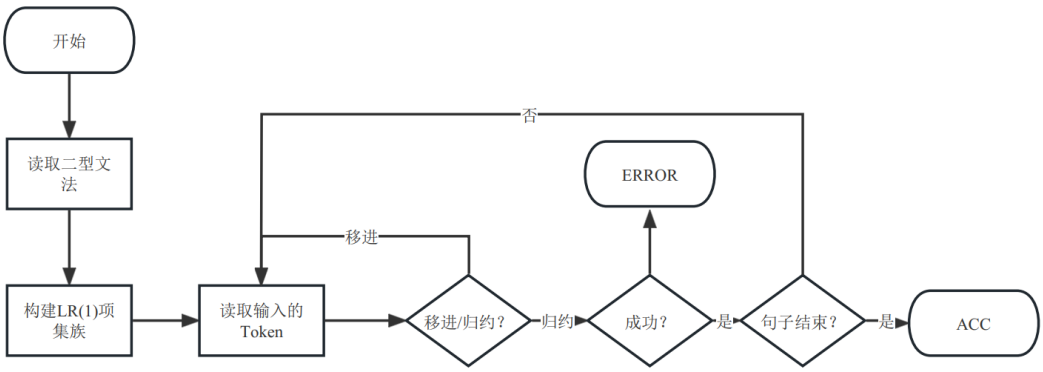


得到的 Token 表(部分)：



2. 语法分析器

原理：读取上下文无关文法（文件中给出），保存产生式，增广产生式，计算 LR(1)项集族，通过项集族构造 LR(1)自动机，根据自动机得到 ACTION 表及 GOTO 表，根据得到的 LR(1)分析表对输入的 Token 表进行语法分析，得到分析结果。



使用的上下文无关文法：

//<程序> -> 《头文件》《程序》|《using》《程序》|《变量常量声明》《程序》|《函数》《程序》|《类/结构体》《程序》|《程序 1》

//头文件、类与结构体这里不实现

S -> US | IS | FS | P

//<程序 1> -> 《头文件》| 《using》|《变量常量声明》|《函数》|《类/结构体》

P -> U | I | F | C | Y

//<程序 2> -> 《变量常量声明》|《函数》

C -> I | F

//《头文件》-> #（include）《头文件名》

```
//H -> #aN
//《头文件名》 -> "《名》" | <《名》>
//N -> "j" | <j>
//《using》 -> （using）(namespace) 《命名空间》;| （using）《标识符》=《类型》;
U -> uls; | ut=g;
//《命名空间名》 如 std，用 s 表示
//字母不够，不给字母了
//标识符，用 t 表示
//B -> t
//《类型》 -> （int|bool|complex|char|double|float|void）
//T -> g
//int i = 0;
//const int i;
//《变量常量声明》->(const)《变量常量类型说明 1》|《变量常量类型说明 1》
I -> cJ | J
//《变量常量类型说明 1》->《类型》《标识符》;|《类型》《标识符》=《标识符/常量》;|《类型》《标识符》=《运算结果》;
J -> gt; | gt=t; | gt=d; | gt=O;| t=d;
//《运算结果》->《运算参数》【+-*/】《运算参数》
O -> RQR
//运算符
Q -> m | p
//运算参数
R -> d | t
//《运算 2》->《identifer》【+= -= *= /=】《Identifier/常量》
V -> tWR;
//W -> += | -= | *= | /=
W -> y
//《函数》->《返回值》《函数名（identifier）》(《类型声明》){《程序》}????????
F -> gt()Y | gt();
//for(;;){}
//《for》->for(《语句》;《语句》;《语句》)《单条语句》|{《多条语句》}
K -> n(G;G;G)Y | n(G;G;G)I | n(G;G;G);
//《while》->while(《单条语句》){《多条语句》}
L -> w(G)Y | w(G)I | w(G);
//《do while》->do{<多条语句>}while(《单条语句》);
D -> fYw(G); | fIw(G);
//《空语句》-> ;
//《单语句不带；》
G -> gt=d | gt=t | tWR | trR
//《if》->if(<语句>){}else if{}else{}
M -> i(G)YeMeY|i(G)YeM|i(G)Y|i(G)IeMeI|i(G)IeM|i(G)I | i(G)YeY |
```

```
i(G)IeI | i(G);
//《跳转语句》-> break;|continue;|return|return <identifier/Constant>
X -> o; | q; | qR;
//《代码块》-> {《语句》}
Y -> {A} | {}
//《多语句》 -> 《for》 | 《while》 | 《do while》 | 《》
//注意：for、while 等应属单语句，但是此处作为多语句来处理。
A -> KA | LA | DA | XA | YA |IA | CA | MA | VA | T
T -> K | L | D | X | Y | I | C | M | V
```

数据结构：*//产生式*

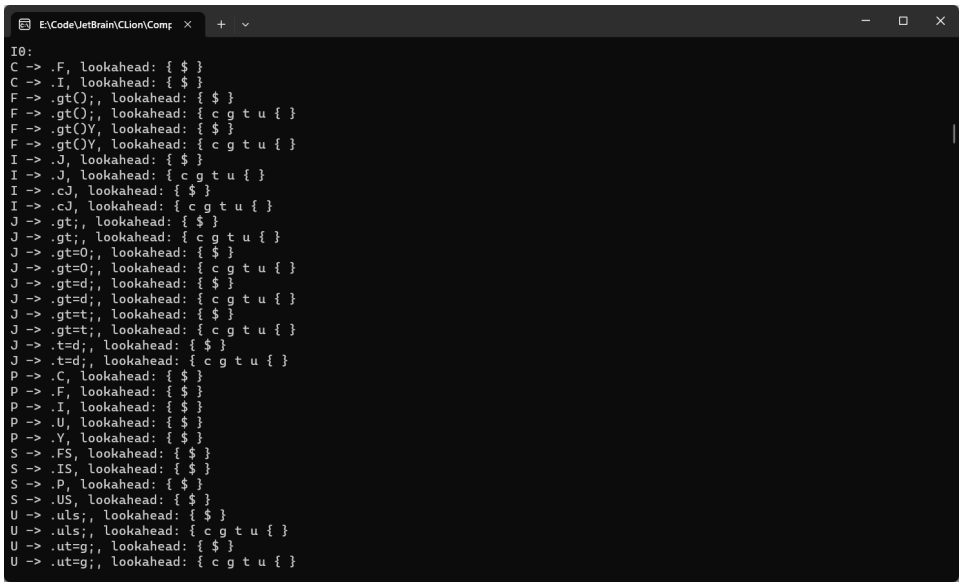
```
struct Production{
    char l;                 //产生式左部
    string r;               //产生式右部
    bool operator==(const Production& o) const;
    bool operator<(const Production& o) const;
};

//项
struct Item{
    Production rule;        //产生式
    int dot;                //产生式中点的位置
    set<char> lookahead;    //展望符
    bool operator==(const Item& o) const;
    bool operator<(const Item& o) const;
};

//项集
struct ItemSet{
    string name;
    set<Item> items;
    bool operator==(const ItemSet& o) const;
    bool operator<(const ItemSet& o) const;
};

/* 语法分析 步骤
 * 读取上下文无关文法（文件给出），保存产生式，增广产生式，计算LR(1)项集族，
 * 通过项集族构造LR(1)自动机，根据自动机得到ACTION 表及GOTO 表，
 * 根据得到的LR(1)分析表对输入的Token 表进行语法分析，得到分析结果*/

class LR {                  //LR 类
private:
```

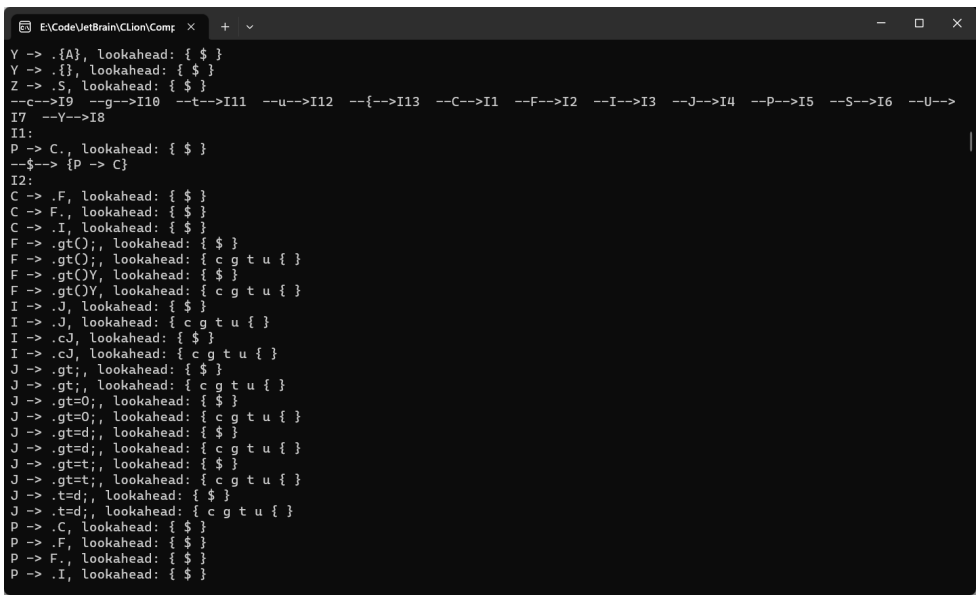
```
    //终结符集合
    set<char> terminals;
    //非终结符集合
    set<char> nonTerminals;
    //产生式集合
    vector<Production> Productions;
    //DFA 对应关系，一个 Node，输入一个 Symbol1，展望符为 Symbol2 到达新状态
    map<Node,map<char,map<char,Node>>> transDFA;
    //ACTION 表，项集（状态）输入一个 Vn（小写字母），到达一个新的 ItemSet，此时需要归约，bool 取 true，移进取 false
    map<ItemSet,map<char,pair<ItemSet,bool>>> Action;
    //GOTO 表，itemSet（状态）输入 Vt，到达新项集，此时为待约项
    map<ItemSet,map<char,ItemSet>> Goto;
    //FIRST 表
    map<char, set<char>> first_set;
    //项集族
    set<ItemSet> is;
    //Token 表
    vector<Token> tokens;
    //token 处理后的移入串
    stack<char> tokenString;
    //符号与字符的对应关系
    map<string,char> dic;
    //记录 token
    map<int,Token> tokenLine;
    //字符与符号的对应关系
    map<char,string> dic2;

public:
    //从 path 中读取上下文无关文法，保存到产生式集合中，并将该集合返回
    vector<Production> readGrammar(const string& path);
    //对文法进行增广（添加一个新的符号，以便起始状态仅在产生式左边出现一次）
    static void augmentGrammar(vector<Production>& grammar);
    //计算 LR(1) 项集族，并生成 LR(1) 自动机
    set<ItemSet> construct_LR1_itemSets();
    //计算项集族闭包
    ItemSet closure(ItemSet &productions);
    //输出项集族以及状态转移关系
    void printItemSet();
    //输出读入的产生式
    void printProduction();
    //语法分析
    //path:Token 文件的路径
    void parse(const string& path);
```

```
    //输出读入的Token，用于验证
    void printToken();
    //读取Token
    string readToken(const string& path);
    //读取MATCH.txt，用于匹配字符串到字符的映射
    void readDic(const string& path1, const string& path2);
};
```

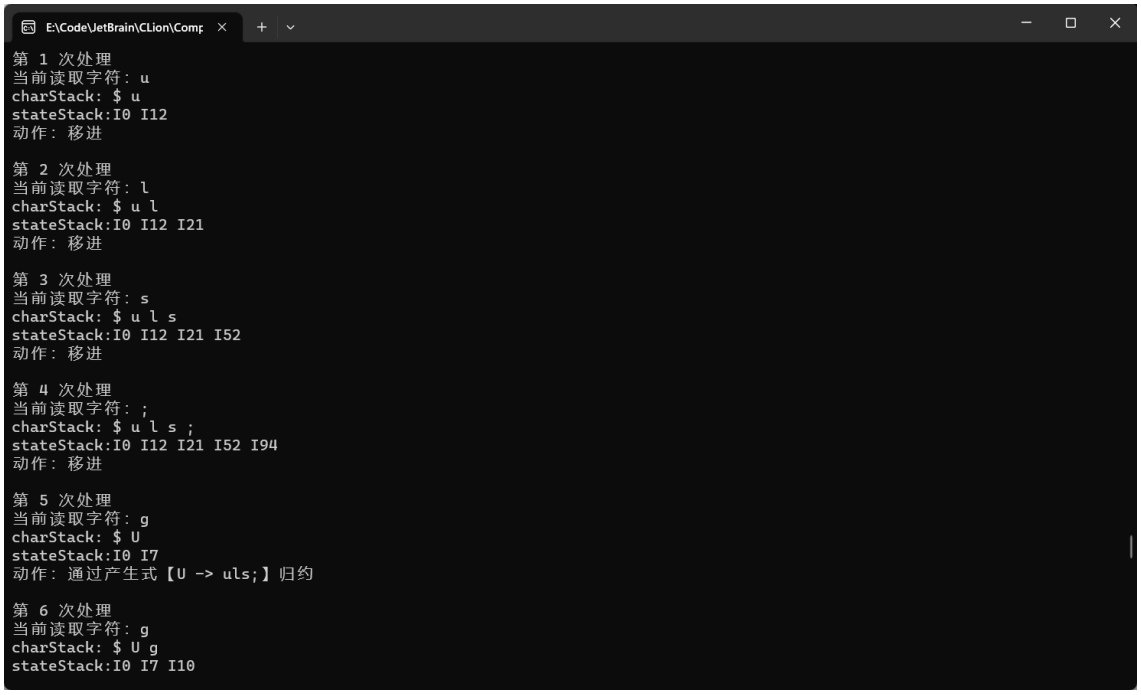
运行结果

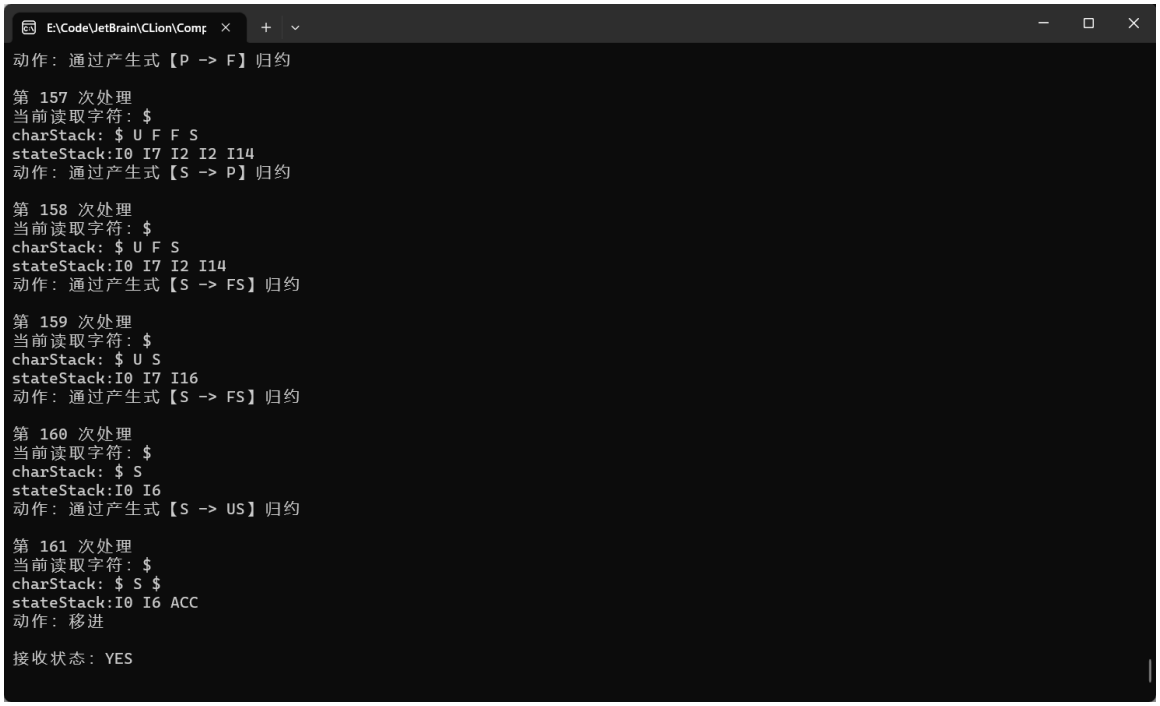
输出通过产生式构建的项集族以及对应的状态转移关系（包含展望符、ACTION 及 GOTO 表）





输出词法分析过程：





可以看到接收状态为 YES，语法分析通过

三、个人体会

经过多日的不断努力，这次课程设计终于完成，通过本次课程设计，我对编译原理相关知识有了进一步的加深，同时，提升了自己的编程能力。通过这次课程设计，我深刻意识到自己仍然有很多不足，知识仍有较多欠缺，还有很多要学。今后我也会认真对待每次项目实践，为今后的学习与工作打下坚实基础。