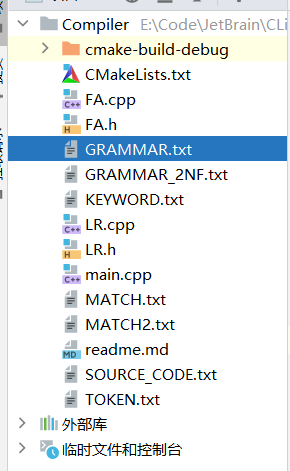
代码说明

一、文件结构

1. 文件夹：Compiler项目文件夹
2. 源代码文件
3. FA.cpp / FA.h 自动机类，包含词法分析相关代码
4. LR.cpp / LR.h 包含LR(1)语法分析相关代码
5. main.cpp 程序主函数，用于调试代码以及展示结果
6. .txt文件
7. GRAMMAR.txt: 词法分析需要的正规文法（三型文法）
8. SOURCE\_CODE.txt: 词法分析中需要的源程序
9. KEYWORD.txt: 关键字列表，用于区分标识符与关键字
10. 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
11. GRAMMAR\_2NF.txt: 语法分析程序需要的上下文无关文法（二型文法）
12. 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、宏定义

// 词法分析需要的正规文法（三型文法）

#define **GRAMMAR\_PATH** "GRAMMAR.txt"

// 词法分析中需要的源程序代码  
#define **SOURCE\_PATH** "SOURCE\_CODE.txt"

// 关键字列表，用于区分标识符与关键字  
#define **KEYWORD\_PATH** "KEYWORD.txt"

// Token表，词法分析程序处理后得到的Token表保存于此，同时作为语法分析程序的输入  
#define **TOKEN\_PATH** "TOKEN.txt"

// 语法分析程序需要的上下文无关文法（二型文法）

#define **GRAMMAR\_2NF\_PATH** "GRAMMAR\_2NF.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  
#define **MATCH\_PATH** "MATCH.txt"  
#define **MATCH2\_PATH** "MATCH2.txt"

三、基本数据结构

* 词法分析  
  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(Node:当前状态 map：不同输入对应的状态转移)* 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 printNFA();  
   *//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 getDFA(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);  
};

* 语法分析  
  *//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);  
};

四、函数说明

FA、LR类中的函数已经在数据结构部分出现过，此处不再说明

*//去除字符串前后的空格*string trim(const string& str);  
  
*//检查节点集合中是否已指定字符串开头的Node（状态）*bool nodeStartsWith(const set<Node>& nodes, const string& prefix);  
  
*//检查两个set是否相等*bool setsAreEqual(const set<Node>& s1, set<Node> s2);  
  
*//将sourceSet中的所有元素加入到destSet中*template <typename T>  
void addAllElements(std::set<T>& destSet, const std::set<T>& sourceSet);  
  
*//检查字符串是否以指定子串开头*bool startsWith(const std::string& prefix, const std::string& str);  
  
*//检查状态集中是否有以指定字符串开头并且以指定字符串结尾的状态*bool hasNode(set<Node>& nodes, const string& string1, const string& string2);  
  
*//检查状态集中是否有以指定名称结尾的状态*bool isNodeNameEndsWith(const Node& node, const string& string2);  
  
*//从文件中读取关键字文件*set<string> readWordsFromFile(const string& path);  
  
*//打印并输出Token表*void printTokens(vector<Token> tokens);  
  
*//语法分析*int LexicalAnalyze(const FA& dfa,const string& path);  
  
*//单行的语法分析*vector<Token> LAbyLine(const FA& dfa, const string& line, int n);

*//计算符号s的FIRST集合*set<char> calc\_first(char s, map<char, set<char>>& first\_set, set<char>& visited, vector<Production>& prods);  
  
*//计算输入串s的FIRST集合*set<char> calc\_first\_s(string& s, map<char, set<char>>& first\_set, vector<Production>& prods);

//将sourceSet中的内容加入到destSet中

void addAllElement(set<char> &destSet, const set<char> &sourceSet)

//将栈倒置

template<class T>  
void reverseStack(stack<T> &s)

main函数：

主要功能：对输入的源代码进行词法分析，使用词法分析得到的Token表进行语法分析，得到语法分析结果。

int main() {  
 *//根据文法获取NFA* FA fa;  
 fa.GrammarToNFA(**GRAMMAR\_PATH**);  
 *//控制台输出该NFA* cout<<"NFA:"<<endl;  
 fa.printNFA();  
 FA dfa;  
 dfa.TransToDFA(fa);  
 *//将该NFA转为DFA,然后最小化,输出最小化的DFA* cout<<"DFA:"<<endl;  
 dfa.printDFA();  
 *//进行词法分析* int i = LexicalAnalyze(dfa,**SOURCE\_PATH**);  
 if (!i){  
 cout<<"程序错误，现已终止！"<<endl;

cout << endl << "按Enter键退出...";  
getchar();

return -1;

}

LR lr;

*//语法分析*

*//读取语法* lr.readGrammar(**GRAMMAR\_2NF\_PATH**);

*//输出产生式*  
 lr.printProduction();

*//构建LR(1)项集族*  
 lr.construct\_LR1\_itemSets();

*//输出LR(1)项集族*  
 lr.printItemSet();

*//对指定path的token进行语法分析*  
 lr.parse(**TOKEN\_PATH**);

cout << endl << "按Enter键退出...";  
getchar();  
return 0;

}