代码说明文档

对整个开发流程有了具体思路之后，发现本次的**软件课程设计有很鲜明的特征：无论是词法还是文法分析，无论对状态集还是项目集，大大小小的数据结构增删查改都涉及到表项重复的问题**（而针对庞大的词法分析产生式数量，DFA_move 线性矩阵的表项已经上万了），而对表项的查找通过遍历或者多维数组下标转化的形式还是存在一定问题。的确可以使用大数组用空间换取时间，但是在**各种数据结构大小未知**的情况下不断试错，找到合适的数组大小浪费时间不说，查找起来**遍历的时间复杂度 O(n)**也是很慢的。

为了**兼顾快速查询，并减少是否重复的判断函数操作**，编程过程中的**数据结构大量使用了 STL 模板类中的 set 和 map（O(ln N)）、unordered_map（O（1））通过比较函数定义和运算符重载，使得严格弱序成立，自定义类型的自动去重排序为简明的数据结构和代码开发逻辑带来了极大遍历**。

这一数据结构的使用为编程简易性带来的影响是巨大的：

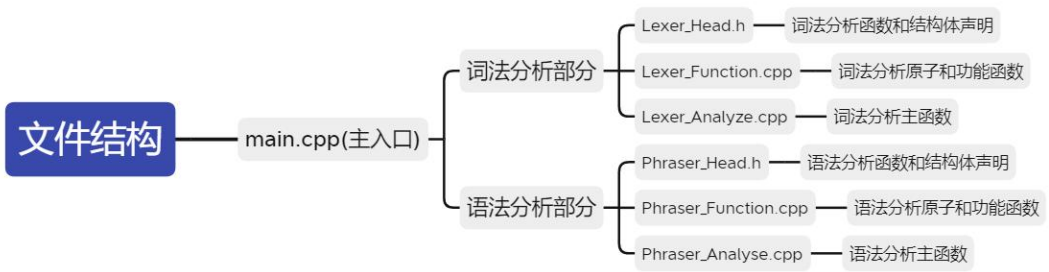
遍历方便、查询方便、插入方便、按序输出方便

不过要时刻注意.find()返回是否是尾部迭代器地址的问题，处理空类的问题层出不穷，还需要对各种运算符重载，但毕竟鱼与熊掌不可兼得，对我而言 STL 模板类的大量应用为我的开发和数据结构设计带来了帮助。

整个程序一共有 2k 余行，所有的函数和数据结构都进行了非常详尽的备注，因此在代码说明文档中只对关键结构和重要函数进行说明。

一、文件结构

为了文件结构的清晰，将头文件和源程序、函数部分分离，使用标准的项目开发结构，将数据结构和函数声明全部放在头文件中，任务 1 和 2 的全局变量使用头文件配合 extern 进行引用，方便数据共享，而原子和功能函数在_Functinon 命名的源文件中定义，全部封装到 Lexer_Analyze 和 Phraser_Analyse 程序中。Main 函数只需要使用词法和语法分析的两个函数就可以实现全部功能。



二、数据结构、主要函数说明

<1>数据结构说明

词法分析器：

使用各种 set 对状态集，NFA 状态转移矩阵，DFA 状态转移矩阵等一系列多个重要数据结构进行设计并进行 typedef，**重载 operator（）使其自动去重**。

```
//      结构体及类型定义
typedef struct NFASET{...}NSET;           // NFA 状态集 || 可用于组成nfamove矩阵的item(if_finalstate无效)
using turnto = std::pair<char,char>;
namespace std {...}


typedef pair<pair<char,char>,NSET> nfa_move_item;   //    NFA_movetable 中的单元    根据pair<状态,VT> 状态(VT)->NSET(状态集)
typedef pair<pair<int,char>,int> dfa_move_item;
struct cmp{...};                          // 结构体比较函数—用于set去重

typedef set<pair<int,bool>,cmp> temp_for_print;          // 用于有序输出
typedef set<nfa_move_item ,cmp> NFA_movetable;
typedef set<dfa_move_item ,cmp> DFA_movetable;
typedef set<pair<int,set<char>> ,cmp> NEW_of_OLDstate;     //  纪录新状态的组成部分(旧状态)  first 不能取char（不能保证唯一性）
typedef unordered_map<turnto,set<char>> SEARCH_NFA;      // 将move_nfa_matrix_final 转化成可以用下标访问的结构

int VT_NUM=0;  //84  终结符个数不变
int NFA_VN_NUM=0;  //33
int DFA_VN_NUM=0;
char start_state;
int DFA_start_state=0;
int pu=0;             // 动态纪录epsilon闭包和a弧转化 次数
int error_number=0;
char final_bef='S',final_new;
set<char> VT;  //NFA终结符
set<char> VN;  //NFA非终结符
set<int> DFA_VN; // DFA非终结符  无效设计，set自排序可能不同状态开头代表元素也相同
NEW_of_OLDstate O_IN_NEW;    // 真正的DFA非终结符结构设计
set<char> nfa_final_state;
map<int,pair<pair<int,string>,char>> for_task2; // 根据字符号查询行号\位置; 字符index，<行号，字符>原本类型
unordered_map<char,set<char>> VN_CLOSURE_SET;   // 终结符的闭包
unordered_map<int ,set<char>> DFA_FINAL_STATE_FOR_PRINT;  // DFA的终态map 数字号-原组成状态 由dfa_final_state 和 O_IN_NEW生成
unordered_map<int ,set<char>> DFA_NOTFINAL_STATE_FOR_PRINT;  // 同上
map<set<char>,int> dfa_state_search;   // 辅助提供通过nfa状态集合寻找dfa状态的可能 和O_IN_NEW 互反
unordered_map<char,set<char>> useable_vt_for_NFA_VN;     // 处理收集nfa中单个vn可用的vt
unordered_map<int,bool> dfa_state_ifcircleandepsilon;   // 当前状态是否已经经过了i弧转换和词法语法分析
unordered_map<int,bool> dfa_final_state;     // DFA的状态是否为终结态 （！！！！！！！！还是人为规定转换终态只要有$就算）  在O_IN_NEW中
int dfa_vn_counter=0;          // 用于dfa状态计数插入 输出记得char int转换
int for_task2_num=1;
int line_index_for2=0;
NFA_movetable move_matrix_NFA;          // for auto 只能遍历而不能插入
NFA_movetable move_matrix_NFA_final;       // 真正的集合形式
SEARCH_NFA  move_matrix_NFA_MAP;        // 从final转化而来
DFA_movetable move_matrix_DFA;
map<pair<int ,char>,int > move_matrix_DFA_FOR_SEARCH;  // 由move_matrix_DFA演化而来，方便进行dfa寻路，词法分析
```

语法分析器：

设计思路同词法分析器，增加了更多的自定义类内操作，进一步提升代码复用性。

其中遇到了 set 中放入自定义结构类型无法成功去重的情况（不相同的元素结构体无法插入），经过学习发现必须重定义函数使得满足**严格弱序**（相同元素 a 和 b 分别作为左值和右值进行比较，再调换顺序进行比较，如果两者返回都为空才符合严格弱序）

```
//      结构体及类型定义
typedef struct right_source{...}rs;                // 用于储存产生式右部

typedef struct project_set_item {...}project_set_ITEM;    //项目集中的单条产生式（包含各种接收和tail信息） 可以直接==判等
typedef struct action_item{...}ACTION_item;  // 用于ACTION表的构建second
struct compare{...};


typedef set<rs,compare> RIGHT_SET;  // 存放一个vn对应所有的右部
typedef set<project_set_ITEM,compare> PROJECT_SET; // 项目集  项目默认按照输入时的顺序进行排序  使用if_equal判等
```

```
set<char> VT_2;
set<char> VN_2;   // 不包括@
set<char> VN_FOR_TITLE; // GOTO 表头
vector<char> chars_for_analyse; // 用于存储要移进—规约的字符串
unordered_map<char,bool> VN_ifcanbe_epsilon;    // 各个vn是否可以推出空  GOAL 1
map<int,pair<char,string>>  PRODUCTION_FOR_ACTION;    // 用于action表进行规约时根据行号查找规约到哪个vn
map<char,RIGHT_SET> PRODUCTION_SOURCE;      // init将正规式存入的形式
map<char,set<char>> VN_FIRST;   // VN的first集
map<char,bool> VN_getfirst_ifdone;  // 各个是否已经完成求取first集的操作
map<char,set<int>> line_search_for_closure;  // 统计同一vn为左部的产生式行号
set<pair<pair<int,char>,int>,compare> MOVE_OF_NEW_STATE;  // 纪录新状态的转化matrix
set<pair<pair<int,char>,int>,compare> GOTO_TABLE;  // GOTO表
set<pair<pair<int,char>,ACTION_item>,compare> ACTION_TABLE; // ACTION表
map<pair<int ,char>,int > goto_for_search;          // 用于打印和查询扫描
map<pair<int ,char>,ACTION_item > action_for_search;
unordered_map<int,PROJECT_SET> NEW_STATE;  // 用数字做index的新状态
int STATE_NUM=0;   // 状态从0开始计数
extern map<int,pair<pair<int,string>,char>> for_task2; // 根据字符号查询行号\位置; 字符index，<行号，字符>

ofstream output_info_2;           //  输出文法分析的具体信息
ofstream output_2;                //  输出yes|no 并给出错误行号和提示信息
```

这其中对各种原始数据结构的处理，包括 NFA->DFA 和项目集规范族的转化过程中对**新状态建立的处理统一使用 set**，避免重复插入的未知错误，而**打印输出和高效查询时使用 unordered_map**(时间复杂度来到了 O（1），和数组随机读取比几乎没有差别，也可以抵消 set 的遍历时间差)

<2>主要函数说明

词法分析器：

1. void initialization ();

对各种常量数据类型进行初始化，并用可读可写的方式打开词法分析要输入输出的文件流

```
void initialization (){
    VT_NUM=0;
    NFA_VN_NUM=0;
    DFA_VN_NUM=0;
    dfa_vn_counter=0;
    VT.clear();
    VN.clear();
    start_state=' ';
    nfa_final_state.clear();
    dfa_final_state.clear();
    VN_CLOSURE_SET.clear();
    DFA_NOTFINAL_STATE_FOR_PRINT.clear();
    DFA_FINAL_STATE_FOR_PRINT.clear();
    dfa_state_search.clear();
    move_matrix_NFA.clear();
    move_matrix_NFA_final.clear();
    move_matrix_NFA_MAP.clear();
    move_matrix_DFA.clear();

    source_code= fopen( _Filename: "../1_in_code.txt", _Mode: "r+");  // 打开要分析的源代码文件，用于scanner函数进行词法分析
    grammar_1=fopen( _Filename: "../1_in_grammar.txt", _Mode: "r+");              // 打开文法流 LexicalAnalysis_lhk
//    grammar_1=fopen("../1_test.txt","r+");
    output_string.open( s: "../2_in_string.txt");
    output.open( s: "../2_in_token.txt");               // 输出token表
    output_info.open( s: "../info_about_task1.txt");    // 输出文法转化和token识别相关详细信息
}
```

2. void Grammar_to_NFA ();

3 型文法 -> NFA（状态转移矩阵生成）读入文法，先读入行号，再将产生式的左部和右部分别处理，如果右部出空则对左部分做特殊处理（说明可以出空，是终态）

```
void Grammar_to_NFA (){
    int n;
    char line[100];             //   使用fgets必须先分配内存，不可空指针
//    line.resize(7);  // 分配内存后才能用fscanf 尽量不用resize(变capacity)
    char state_left,state_right;
    char vt;
    fseek(grammar_1, _Offset: 0,SEEK_SET);
    fgets(line, _MaxCount: 5,grammar_1);
    string numofline(line);
    n= stoi(numofline);
    output_info<<"                                  词法分析文法 相关信息统计(已排序处理)  "<<endl;
    output_info<<"正规式条数 : "<<n<<endl;
    _for(i, a: 0,n){                            // less ABKY IY
//        fscanf(grammar_1,"%s",&line[0]);
        fgets(line, _MaxCount: 100,grammar_1);
        state_left=line[0];
        vt=line[3];
        if(i==0){               // 初态设置
            start_state=state_left;
        }
        if(!if_inVN(state_left, VN: VN))  // 对左部进行添加
            VN.insert(state_left);
        if(!if_inVT(vt, VT: VT))     // 对右部非终结符进行添加
            VT.insert(vt);
        if(((int)line[4]>=(int)'A'&&(int)line[4]<=(int)'Z')||((int)line[4]>=(int)'a'&&(int)line[4]<=(int)'f')){
            state_right=line[4];
        }
```

3. **void NFA_to_DFA ();**

词法分析中的核心函数，这其中闭包求取的过程作为重心进行了三次迭代，第一次设置了五层循环对嵌套的结构体进行分析，但最多只能对三层连续的闭包进行分析，逻辑混乱且速度缓慢，对文法设计的包容度不高，程序健壮性低。最终采用了 BFS 和 DFS 混合的思想：先对当前状态集遍历(BFS),找出可以出空的 Vn，再对每个可以出空的 Vn 进行 DFS 深入，利用递归函数解决多层出空的问题，这样无论连续多少层出空我的程序就都可以正确求取闭包（除非太多层导致递归爆栈了）

NFA 确定化，生成新状态转移矩阵，是整个词法分析中最重要的函数，首先对初态求闭包，建立一个栈，将初态压入 stack 中，对栈顶元素求取所有可能的 a 弧转换并取闭包，如果是新状态集则作为 DFA 的新状态存入相应数据结构，并压栈，否则只对 DFA 的 move 矩阵添加表项就推出本轮循环，直到 stack 为空，NFA->DFA 转化结束。

```
void NFA_to_DFA (){
    NSET pre_set;
    pre_set.set_member.insert(start_state);     // 先处理初态
    stack<NSET> s;
    get_closure( & pre_set, & move_matrix_NFA);  // !!!!!!!wait for write  如遇到->@ 后集合中有$
    s.push(pre_set);
//    for(auto i:pre_set.set_member)
//        cout<<i;
//    cout<<endl;
    DFA_start_state=0;   // 0状态是起始状态
    add_NSET_toState( & pre_set, & O_IN_NEW, & dfa_state_search); // 得到旧状态组成的新状态结构体，方便后续操作      (将NSET变成N
    pu=1;  // 当前行第几个
    set<char> temp_allA_inSET;     // 状态集合中所有状态可以进行的a集合(剪枝，大幅优化计算时间，对多状态计算效果显著)
    map<set<char>,int> cut_tree;   //  同一行的如新增状态相同，则不用再二次取闭包并添加状态了，直接退出即可
    set<char> add_state;        // 当前行状态经过i弧转化后新增的状态
    map<char,char> for_quickcompare;
    map<int,bool> have_gothrough_line_states;  // 一个状态是否已经走过完整的一行了
    have_gothrough_line_states.clear();

    int counter=0;

    while(!s.empty()){

        pre_set=s.top();
        s.pop();
        int bef= if_inNewSet( & pre_set, dfa_state_search: dfa_state_search); // 一行的起始状态不变
        if(have_gothrough_line_states.find(bef)!=have_gothrough_line_states.end())
            continue;
        pu=1;
```

4. void scanner ();

扫描代码源文件进行词法分析,本过程必须在 DFA 分析之后操作，通过分析 DFA 的状态转移以及终态的 Vn 组成，剪掉不符合文法规范的意外终态度转化，不断尝试新的输入类型和错误，查看是否成功，同时对代码进行优化，DFA 有穷自动机走到不同状态代表不同的结果和错误类型,生成 token 表放入 2_in_token.txt

```
void scanner (){
    ifstream source( s: "../1_in_code.txt");
    string line;
    vector<string> words;
    map<pair<int ,char>,int> DFA_SEARCH;  // 方便dfa查询
    bool iffirstxiao=true,iffirstdayu=true;
    DFA_SEARCH.clear();
    for(auto ok:move_matrix_DFA)
        DFA_SEARCH.insert({ok.first,ok.second});
    output<<"                          TOKEN TABLE"<<endl;
    output<<"       line        "<<" type        "<<"        word        "<<endl;
//    while(getline(source,line)){
//        for(auto i:line)
//            cout<<i<<' ';
//        cout<<endl;
//    }
    cout<<endl<<"      task 1 is running ... "<<endl<<endl;
    cout<<"源代码分析结果如下所示 :"<<endl;

    while(getline( &: source, &: line)) {
        line_index_for2++;
        words.clear();
        split( line: line, &: words);
//        for(auto i:words){
//            cout<<i<<" ";
//        }
//        cout<<endl;
```

语法分析器：

1. void vn_epsilon_process ();

根据 task2 文法处理 Vn 判断是否可以推出 epsilon

先遍历一边产生式，找到绝对能出空（存在右部是空的产生式）和绝对不能出空（所有相关产生式右部的第一个字符都是 Vt）的 Vn

再对未确定的 Vn 进行扫描判断，只对右部是 Vn 开头的产生式进行处理，内部终止条件是全部可空（可出空）或遇到 Vt 终止。不断重复上述循环过程直至没有未确定的 Vn。

```
void vn_epsilon_process (){       // 要求vn必须是大写英文字母
    set<char> VN_can_epsilon,VN_not_epsilon,VN_unknow; // 先求出定能|不能求出闭包的状态;
    bool all_have_vt=true;
    bool if_added_to_must=false;       // 是否已加入前两个绝对集合
    int counter=0;
    for(auto i:PRODUCTION_SOURCE){
        counter=0;
        if_added_to_must= false;
        for(auto j:i.second){
            if(j.if_a_single_epsilon){   // if have ->@
                if_added_to_must=true;
                VN_can_epsilon.insert(i.first);
                goto here;
            }
            for(auto k:j.right_inorder){         // 如果一个vn的所有产生式的右部的都含有vn,必不能产生空
                if(j.right_for_compare[0]<'A'||j.right_for_compare[0]>'Z'){
                    counter++;
                    break;
                }     // 如果右部中存在终结符
            }
        }
        if(counter==i.second.size()){
            VN_not_epsilon.insert(i.first);
            if_added_to_must=true;
```

2. void vn_get_FIRST_SET ();

求取 Vn 的 first 集，同上面判断空的函数思想基本一致，如果一个 Vn 右部全部是终结符开头的产生式则 First 集已确定，再根据已经确定 First 集的 Vn，结合是否出空（决定是否顺位判断下一 Vn 以及 First 集是否包含空）对未确定的 Vn 进行 First 集求取。

```
void vn_get_FIRST_SET (){
    map<char,set<char>> first_temp;
    int counter=0;      // 统计 对每一个vn 的所有产生式右部 满足第一个字符为vt|@的个数
    for(auto i:VN_2){      // 对每个vn
        counter=0;
        for(auto j:PRODUCTION_SOURCE.find(i)->second){
            if(j.if_a_single_epsilon){   // 如果有->@
                if(first_temp.find(i)!=first_temp.end()){
                    first_temp.find(i)->second.insert( x: '@');
                }
                else{
                    set<char> tt;
                    tt.insert( x: '@');
                    first_temp.insert( x: {i,tt});
                }
                counter++;
                continue;
            }
            if(j.right_for_compare[0]<'A'||j.right_for_compare[0]>'Z'){ // 如果右部第一个为vt
                if(first_temp.find(i)!=first_temp.end()){
                    first_temp.find(i)->second.insert( x: j.right_for_compare[0]);
                }
                else{
```

3. **void LR_1_CREATE ();**

重要函数，用于求取 LR 项目集合的同时对,新状态转换表进行添加（增广文法 $->S）编程思路和词法分析中 NFA_to_DFA 一致，都是使用 stask 进行项目集求取。先将增广文法进行 CLOSURE 求取，压 stack 后就可以进行重复的循环操作了：弹栈，判断可以进行的转化需要的 Vn||Vt 集合，再进行不同的移进操作，求取 CLOSURE 的同时生成右边式的新 First 集合，更新向前搜索符，如果是新状态则压栈，但对所有的过程，都要插入状态转移的表项，不断重复上述过程直到栈空则过程结束

```
void LR_1_CREATE (){
    // $->S 为第0条 增广文法的加入引发一系列变更
    VT_2.insert( x: '#');  // error reason
    if(VN_ifcanbe_epsilon.find( x: 'S')->second)
        VN_ifcanbe_epsilon.insert( x: { x: '$', y: true});
    else
        VN_ifcanbe_epsilon.insert( x: { x: '$', y: false});
    PRODUCTION_FOR_ACTION.insert( x: { x: 0, y: { x: '$', y: "S"}});
    rs the_begin( right: "$->S");
    RIGHT_SET first_temp;
    first_temp.insert(the_begin);
    PRODUCTION_SOURCE.insert( x: { x: '$',first_temp});
    VN_FIRST.insert( x: { x: '$',VN_FIRST.find( x: 'S')->second});
    // 建立第一个项目集
    PROJECT_SET pre_set;
    pre_set.insert( x: init_project_set_item( line_index: 0));
    get_CLOSURE( &: pre_set);           // 取CLOSURE
    NEW_STATE.insert( x: {STATE_NUM++,pre_set});  // 加入新状态
    stack<PROJECT_SET> s;
    s.push(pre_set);
    set<char> vt_or_vn_forconvert; // 用于GOTO的可能字符
    while(!s.empty()){
        pre_set=s.top();
        s.pop();
        int pre= if_in_new_state( aft_set: pre_set),aft;  // 用于辅助LR（1）状态转化表进行插入
        vt_or_vn_forconvert.clear();
        for(auto i:pre_set){               //  查找所有.的右部终结和非终结符
            for (int j = 0; j < i.right.size(); ++j) {
                if(i.right[j]=='.'){
                    if(j+1<i.right.size()){
```

4. void get_ACTION_and_GOTO ();

根据 ppt 中的规则和含义对新的项目集合转化关系进行统计得到 ACTION 和 GOTO 表，主要是分析状态转移矩阵，如果转化条件是终结符，则更新 GOTO 表（代表状态转移），如果是非终结符号有三种情况（移进，规约和 acc），需要分别进行处理，置状态位方便后面 LR(1)分析过程查表操作和格式化输出。

```
void get_ACTION_and_GOTO (){
    // 先对goto表添加
    for(auto i:MOVE_OF_NEW_STATE){
        if(VN_2.count(i.first.second))   // MOVE_OF_NEW_STATE中character为Vn直接添加GOTO表
            GOTO_TABLE.insert({i.first,i.second});
    }
    // 再对action表添加
    //  S r acc添加
    for(auto i:NEW_STATE){
        for(auto j:i.second){
            if(*j.right.rbegin()=='.'){    // r和acc情况
                if(j.tail=='#'&&j.left_Vn=='$'&&*j.right.begin()=='S'){     // acc情况
                    ACTION_item tempt( movement_first_char: 'a', number: 0);
                    if(!if_is_new_ACTION_item( temp: { x: {i.first, y: '#'},tempt}))
                        ACTION_TABLE.insert( x: { x: {i.first, y: '#'},tempt});
                    continue;
                }
                if(VT_2.count(j.tail)){           // r情况
```

5. void LR_1_ANALYSIS ();

LR(1)分析，建立状态栈和符号栈，根据 ACTION、GOTO 表进行压栈弹栈，查询产生式的 map 进行规约，如果达到 acc 则接受，查找表项失败则规约不成功，失败位置对应的字符需要回溯对应 tocken 表行号和列号返回分析的错误原因。扫描返回结果 YES 或者 NO，如果规约失败，输出错误行号和可能的错误信息

```
void LR_1_ANALYSIS (){
    ifstream input_string( s: "../2_in_string.txt");
    string temp;
    bool if_accept_success=false;   // 是否被成功接受
    getline( &: input_string, &: temp);
    for (int i = 0; i < temp.length(); ++i)     // 将待分析字符串读入
        chars_for_analyse.push_back(temp[i]);
    chars_for_analyse.push_back('#');
    cout<<"移进规约分析过程详见 info_about_task2.txt ( 实时生成 )"<<endl<<endl;
    // step 1 单独处理
    stack<int> state;           // 状态栈
    stack<char> character;       // 符号栈
    vector<int> state_fornow;   // 当前行使用方便查询和输出
    vector<char> character_fornow;
    state.push( x: 0);
    character.push( x: '#');
    int step_num=0;       // 步骤号
    int char_for_access=0;
    output_info_2<<"                              LR(1) 规约分析过程展示"<<endl;
    while(!if_accept_success){         // 注意引入无表项时的break
        output_info_2<<step_num++<<" ";
        if(step_num==2)
            ;
```