天津大学

2022秋季编译原理大作业第六组开发报告

编译器前端的设计开发

学　　院	智能与计算学部
专　　业	软件工程
年　　级	**2020**级
姓　　名	黄丽丽,丁小芮,王悦君,孙思远
学　　号	**3020244288,3020244201,3020244340,3020244341**

2022 年 **11** 月 **13** 日

第一章　开发目的与任务需求分析

1.1　开发目的

编写一个 C - -编译器前端（包括词法分析器、语法分析器）：

C - -语言是本实验的源语言。是一个 C 语言的子集， C - -语言是单文件的，以 .sy 作为后缀，去除了 C 语言中的 include/define/pointer/struct 等较复杂特性。

1. 使用自动机理论编写词法分析器。

编写 C–语言的词法分析器，理解词法分析器的工作原理，熟练掌握基于自动机理论的词法分析器的工作流程。编写源代码识别输出单词的二元属性，填写符号表。

2. 自上而下或者自下而上的语法分析方法编写语法分析器。

编写 C - -语言的语法分析器，理解自上而下/自下而上的语法分析算法的工作原理；理解词法分析与语法分析之间的关系。语法分析器的输入为 C - -语言源代码，输出为按扫描顺序进行推导或归约的正确/错误的判别结果，以及按照最左推导顺序/规范规约顺序生成语法树所用的产生式序列。

1.2　任务需求分析

1.2.1　词法分析

1. 完成 C - -语言的词法分析器，实现有限自动机确定化，最小化算法。词法分析器的输入为 C–语言源代码，输出识别出单词的二元属性，填写符号表。单词符号的类型包括关键字，标识符，界符，运算符，整数。

标识符（IDN）定义与 C 语言保持相同，为字母、数字和下划线组成的不以数字开头的串

整数（INT）的定义与 C 语言类似，整数由数字串表示

2. 实现语言：C++

3. 操作目的：生成符号表；将源代码转化为单词符号序列。

1.2.2　语法分析

1. 完成 C - -语言的语法分析器，采用自上而下的语法分析方法编写语法分析器。根据给出的C - -文法生成预测规约表，并结合词法分析器输出的Token序列作为语法分析器的两个输入，输出结果为规约序列。

2. 实现语言：C++

3. 操作目的：生成规约序列；将文法与词法分析结果转化为规约序列。

1.2.3 整体流程

开发过程的整体流程图如下：

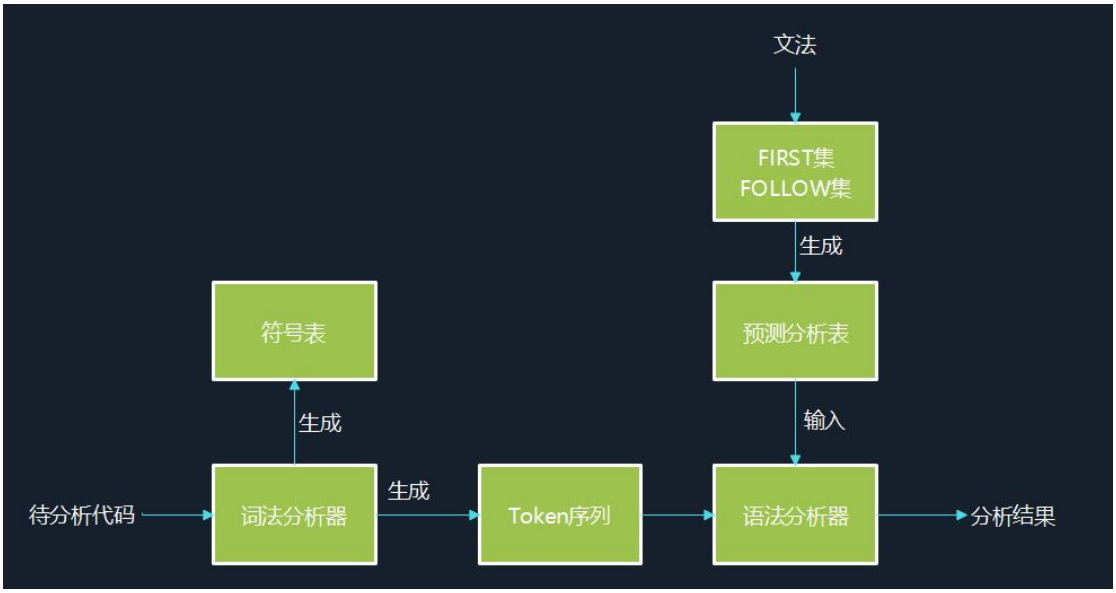

图 1-1　　整体流程图

第二章　　词法分析

2.1　非确定有限自动机构造方式

2.1.1　准备工作

```
typedef struct state{
    int now;
    char input;
    int next;
}state;
```

图 2-1　　状态结构体

如图，我们在代码中构建了一个名为state（状态）的结构体，代表自动机中的状态结点，其中：

now 代表当前的状态;

input 代表输入字符;

next 代表在当前状态与输入字符下，即将去到的下一个状态。

```
vector<char> in_sym = {'S','_','I','0','Y','=','>','<','!','&','|','F'};
map<int,char> nfamap = {{2,'S'},{4,'I'},{6,'Y'},{11,'F'}};
map<int,char> dfamap;
vector<int> NFA_ZT = {2,4,6,11};
```

图 2-2　　辅助存储

in_sym包括了所有我们可能的输入字符，将输入进行了某些处理以简化NFA，具体解释见下下面NFA构造。

nfamap是NFA的终态以及他们的标识符之间的映射，其中四个字母与in_sym中的并不相同，前者为输入，后者的作用是表示这一终态是哪种终态。

下边的dfamap与nfamap起同样作用，是DFA终态与其标识符之间的映射，但因为需要通过程序构造DFA，所以这里并不能赋初值，如何构造将于后边介绍。

这里终态的标识的作用在于，利用自动机识别单词符号时，对最终达到的终态进行判断，处于哪一种终态，其中S代表标识符和关键字终态，I代表整数终态，F代表分隔符终态，Y代表运算符终态，具体如何操作将在后边的单词符号序列输出中进行描述。 NFA_ZT包含了NFA中的所有终态。

2.1.2 具体构造

利用上述的结构体，我们可以构建一个NFA即非确定有限自动机，其中将自动机用state数组储存起来。

```
state NFA[NFA_SIZE]={
    {0,'S',1},{0,'_',1},{1,'S',1},{1,'_',1},{1,'I',1},{1,'0',1},{1,'$',2},
    {0,'0',3},{3,'$',4},{0,'I',5},{5,'0',5},{5,'I',5},{5,'$',4},
    {0,'Y',6},{0,'=',7},{7,'$',6},{7,'=',6},{0,'>',7},{0,'<',7},{0,'!',8},{8,'=',6},{0,'&',9},{9,
    {0,'F',11}
    /*
    第一行为关键字和标识符
    第二行为整型数字
    第三行为运算符
    第四行为界符
    其中2，4，6，11为终态，初态为0
    */
};
```

图 2-3　　NFA存储

第三行后续部分：

```
,{9,'&',6},{0,'|',10},{10,'|',6},
```

图 2-4　　补充

其中对输入进行了一些处理以简化NFA，处理方法如下：

```
char bewhich(char nowchar){
    if((nowchar >= 'A' && nowchar <= 'Z') || (nowchar >= 'a' && nowchar <= 'z')){
        return 'S';
    }else if(nowchar >= '1' && nowchar <= '9'){
        return 'I';
    }else if(nowchar == '0'){
        return '0';
    }else if(nowchar == '+' || nowchar == '-' || nowchar == '*' || nowchar == '/' || nowchar =
        return 'Y';
    }else if(nowchar == '(' || nowchar == ')' || nowchar == '{' || nowchar == '}' || nowchar =
        return 'F';
    }else if(nowchar == '=' || nowchar == '>' || nowchar == '<' || nowchar == '!' || nowchar =
        return nowchar;
    }else{
        return '~';
    }
}
```

图 2-5　　补充

函数并未截取完全，这里对其意思进行解释。

当输入一个字母时，用S代替这个字母，标识输入一个字母。

当输入一个1-9的数字时，用I代替这个数字，标识输入一个1-9的数字。这里将0单独拿了出去，输入0不返回I而是返回0本身，其原因在于数字不能以0开

头，如果将0包括到I中，自动机将无法区别0与1-9，那么如果出现以0开头的数字也就无法处理，将导致错误。

当输入部分运算符时，返回Y，当输入另一部分运算符时返回他们本身，因为我们需要识别一些符合运算符，而例如加等于等复合运算符没有被包含，还有需要注意的是例如不等于等复合运算符的第一个运算符没有要求识别，故这里需要在终态的设置上进行考虑。

当输入分隔符时，返回F，因为分隔符没有什么特殊要求，所以这里并未做特殊处理。

当输入一些运算符和空格符以及tab符时，返回其本身，前者部分运算符中已经论述原因，后两个为程序书写的基本需要。

当输入其他的字符时，我们一律认为是非法输入，返回一个特殊标记符号，从而能够在词法分析过程中，简单地判断是否有一个非法输入。

2.2 非确定有限自动机的确定化

2.2.1 功能目的

在非确定的有穷自动机种（NFA）中，由于状态的转移需要经过若干步，并且每一步都是不可预测的，对后续输入的字符存在一个试探的过程。而在试探的过程中可能会带来重复的步骤，这无疑是增加了程序的运行时间同时降低了工作效率，将NFA转成DFA进行确定化就是为了增加程序的效率缩短工作时间。

2.2.2 实现逻辑

利用课程教授方法，通过空闭包以及各个输入来对NFA进行确定化操作。基本流程为：

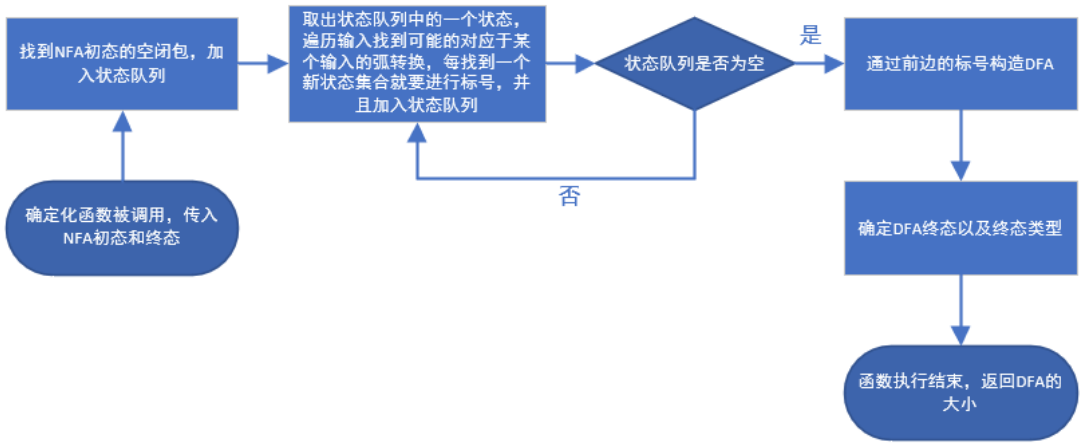

图 2-6　　实现逻辑

2.2.3 辅助函数

```
set<int> closure(set<int> tem){
    set<int> res;
    set<int>::iterator i;
    for(i = tem.begin();i != tem.end();i++){
        res.insert(*i);
        int onestate = *i;
        while(true){
            int j;
            for( j = 0;j<NFA_SIZE;j++){
                if(NFA[j].now==onestate && NFA[j].input=='$'){
                    res.insert(NFA[j].next);
                    onestate=NFA[j].next;
                    break;
                }
            }
            if(j == NFA_SIZE){
                break;
            }
        }
    }
    return res;
}
```

图 2-7　　closure函数

首先是closure函数，该函数的主要功能是基于已有的状态集合，找到该状态集合的空闭包，首先创造一个空的状态集合res，先将传入的状态集合全部放入，然后遍历其中每一个状态，将其能够通过空（程序中用$代替）到达的状态加入res，循环接收后返回res即可。

```
set<int> moveby(set<int> state,char by){
    set<int> res;
    set<int>::iterator i;
    for(i = state.begin();i!=state.end();i++){
        for(int j = 0;j<NFA_SIZE;j++){
            if(NFA[j].now==*i && NFA[j].input==by){
                res.insert(NFA[j].next);
            }
        }
    }
    return closure(res);
}
```

图 2-8　　moveby函数

然后是moveby函数，就像这个函数名字一样，该函数的功能是某一状态集合通过某一输入能够到达的另一个状态集合，套用课上的话来说，该函数所求就是状态集合state的by弧转换，首先，还是创建一个空的状态集合res，先将state中各个状态通过输入字符by到达的全部下一状态加入res中，随后对res求

空闭包进行返回即可。现在，关于确定化最重要的两个所需功能已经实现，接下来我们看确定化的具体操作。

```
int NFA2DFA(int start,vector<int> end)
```

图 2-9　　函数声明

函数需要两个输入，start代表NFA的初始状态，end代表NFA的终态集合，返回值为int，即DFA的大小。所需要用到的变量声明。

```
queue<set<int>> myque;
myque.push(closure({start}));
set<int> nowstate,nextstate;
set<set<int>> allstates;
map<set<int>,int> state_map;
map<pair<set<int>, char>, set<int>> use;
int num = 0;
```

图 2-10　　变量声明

```
while(!myque.empty()){
    nowstate = myque.front();
    myque.pop();
    if(nowstate.size()==0){
        continue;
    }
    if(allstates.count(nowstate)==0){
        allstates.insert(nowstate);
        vector<char>::iterator i;
        for(i = in_sym.begin();i != in_sym.end();i++){
            nextstate = moveby(nowstate,*i);
            if(nextstate.size()!=0){
                use[{nowstate,*i}]=nextstate;
                myque.push(nextstate);
            }
        }
        state_map[nowstate] = num++;
    }
}
```

图 2-11　　循环体

首先，创建一个myque以供我们进行循环，每找到一个状态集合便将其放入myque中，在循环中不断取出去找新的状态集合，直到myque为空为止，初始我们将NFA初始状态的空闭包放入，nowstate后边用来代表当前取出的状态集合，nextstate代表通过no wstate和某一输入我们找到的下一个状态集合，

allstates中存储了我们遍历过的每一个状态集合，state_map则是这些状态集合与它们新的编号之间的映射，use则是存储了两个集合和一个字符，第一个集合为某一状态集合，字符为输入字符，第二个集合为第一个状态集合通过该输入字符到达的另一个状态集合，这里边提到的状态集合即是 DFA的某一状态，num用来统计DFA的大小。

随后进入循环，每此循环取出状态集合后，首先判断一下其是否为空集合，空集合跳过即可，随后判断allstates是否已经包含了这个状态集合，未包括则需要进行添加操作，随后对我们可能输入的字符进行循环，寻找当前状态集合与每一个字符进行moveby的结果（moveby前边我们已经进行介绍，其功能为实现一个弧转换），如果成功找到了一个大小不为0的状态集合，在use中对当前状态、输入字符、下一状态进行标记，将新找到的状态集合放入myque中以便后边进行遍历，然后对新找到的状态集合进行编号以便后边生成DFA。

接下里我们遍历use:

```
map<pair<set<int>, char>, set<int>>::iterator it;
for(it=use.begin();it!=use.end();it++){
    if(state_map[it->second]!=0){
        DFA.push_back({state_map[it->first.first],it->first.second,state_map[it->second]});
    }
}
```

图 2-12　　use遍历

利用state_map将标记过的状态集合用新状态的标号放到DFA中，同时放入的还有输入字符和下一状态，这样我们就会迭代构造一个DFA。

但存在一个问题，DFA中的终态如何寻找呢？

```
for(set<int> onestate : allstates){
    char whi = 'n';
    for(int one : onestate){
        for(int another : NFA_ZT){
            if(one == another){
                whi = nfamap[one];
                break;
            }
        }
        if(whi!='n'){
            break;
        }
    }
    dfamap[state_map[onestate]] = whi;
}

return num;
```

图 2-13　　allstates遍历

遍历allstates中的每一个状态集合，如果这个状态集合包含NFA的某一个终态，我们便将其认作终态，同时将其进行标号，利用n标识不是终态的状态集合，最后返回DFA大小计数器，NFA的确定化到此也就结束了。

2.3 确定有限自动机构造方式

```
vector<state> DFA;
```

图 2-14 存储方式

利用vector容器进行储存，用到了state结构体，同时便于后边我们的添加与删除等操作。

在确定化后我们进行输出，得到如图所示结果：

```
0 ! 6
0 & 7
0 0 3
0 < 5
0 = 5
0 > 5
0 F 9
0 I 2
0 S 1
0 Y 4
0 _ 1
0 | 8
1 0 1
1 I 1
1 S 1
1 _ 1
2 0 2
2 I 2
5 = 4
6 = 4
7 & 4
8 | 4
```

图 2-15 确定化结果

其中每一行都代表了一种状态迁移方式，左边为当前状态，中间为输入，后边为下一状态，代表了自动机中的边，共22行。

2.4 确定有限自动机的最小化

在介绍NFA确定化的操作前，先来介绍需要用到的几个函数。

2.4.1 功能目的

将DFA 中将相同的状态合并，简化自动机。

2.4.2 代码逻辑流程图

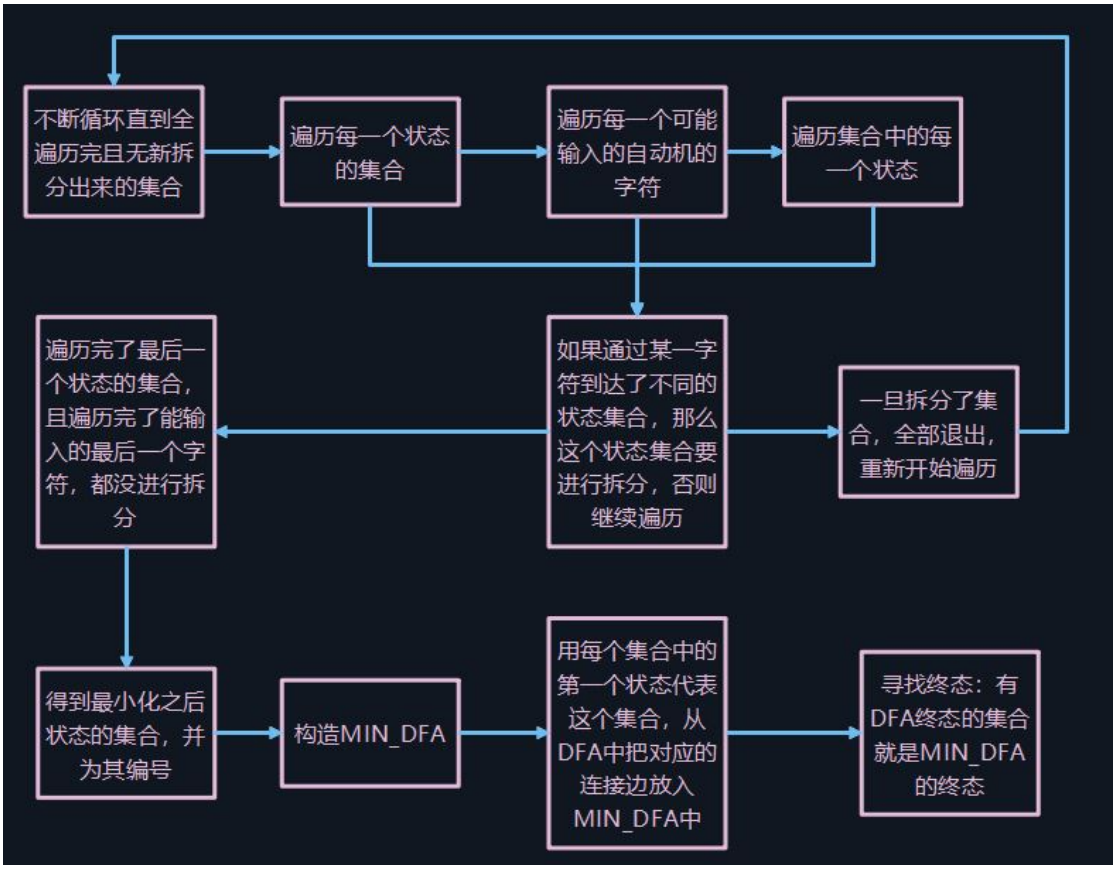

图 2-16　　自动机最小化流程图

2.4.3 程序伪代码

下面是程序伪代码，主要体现功能逻辑实现以及执行步骤。

```
void min_DFA(int num){
    用来保存状态集合的集合con;
    把所有的状态分为两类：终态和非终态
    while(allflag == 0 || allflag ==1){//都代表还没结束
        对每一个状态的集合进行集合编号，
        for(遍历con中的每一个状态的集合){
            设置是否遍历完全部可能输入的字符的变量charflag = 0;

            for(遍历每一个可能输入自动机的字符){
                map<pair<char,int>,set<int>> tempmap;//新分出的集合，谁经过了字符到达了哪个集合(的编号)。进行存储
                for(遍历状态集合中的每一个状态){
                    找到当前遍历状态a经过当前遍历字符所到的下一个状态b（如果无法到达默认到达状态-1）
                    找到状态b所属的集合编号。
                    去存储的tempmap中找，是否已经有存储经过这个字符到达状态b的状态的集合。
                    if(有)：把这个状态b的编号加入这个集合，重新放回到tempmap
                    else ：  新创建集合，把对应信息放入tempmap
                }

                进行判断：
                如果遍历到能输入的最后一个字符（这里是'F'），说明这个集合不能再拆了,结束对字符的遍历，设置charflag =
                if(tempmap.size()==1){//没拆分，不需要采取动作
                    continue;
                }
                else{
                    在con中除去当前遍历的集合，把tempmap中存储的所有集合都加到con中
                    allflag = 1;
                    重新对con进行集合编号
                    break;//只要拆分了集合我们就大退，重新从头遍历，避免遍历少了/
                }

            }

            进行判断：
            if(allflag == 1){//没遍历完进行了拆分，那么我们大退
                break;
            }
            if(已经遍历到最后一个集合，而且已经遍历完最后一个字符，而且没拆分allflag ==0){
                那么我们遍历结束！成功！
                allflag = 2;
                break;//可以退出全部循环了！遍历完成！
            }

        }
        if(allflag ==2) break;
    }

        //到此为止我们遍历完了所有集合也拆完了集合。接下来我们要判断每个多元素集合中没有输入的终态是否是一样的终态！
        for(遍历con中的每一个状态的集合){
            if(是多元素集合){
                如果里面的终态不是一致的终态，那么也进行拆分。
                //注意这里所说的不一致的终态是指，不是通过同一方式到达的终态，必须进行拆分
            }
        }

        //接下来构造MIN_DFA
        for(遍历con中的每一个状态的集合){
            为con编号；
            找每个集合中的第一个元素代表整个集合
            把DFA中的对应关系加入MIN_DFA中，其中状态要变成其所属集合的编号
        }

        //最后是终态的寻找
        将dfamap中存储的终态存储到min_dfamap中即可


}
```

图 2-17　　最小化伪代码

2.4.4 其他功能函数的编写

最小化功能代码中调用的一些函数做如下说明：

（1）

```
int DFA_find_next_state(int now_num,char c,int DFA_SIZE){
```

图 2-18 调用函数1

在DFA中给定现态和输入字符找次态的函数。

（2）

```
int findcollectnum(set<set<int>> con,int first_to_state,map<set<int>,int> bianhaomap)
```

图 2-19 调用函数2

给定状态集合的集合，当前状态，和存储集合对应编号的map 寻找状态所在集合的编号。

（3）

```
map<set<int>,int> bianhao(set<set<int>> con){//对集合进行编号
```

图 2-20 调用函数3

给定集合，对集合进行编号的函数。

2.4.5 测试样例的编写

在debug时编写了一个测试样例，采用的是课程作业2中的第四题：

题目如下：

4．将下面的 DFA 最小化：M=({S0,S1,S2,S3,S4}, {a,b},f, S0 , {S4})　　5'

	a	b
S0	S1	S2
S1	S1	S3
S2	S1	S2
S3	S1	S4
S4	S1	S2

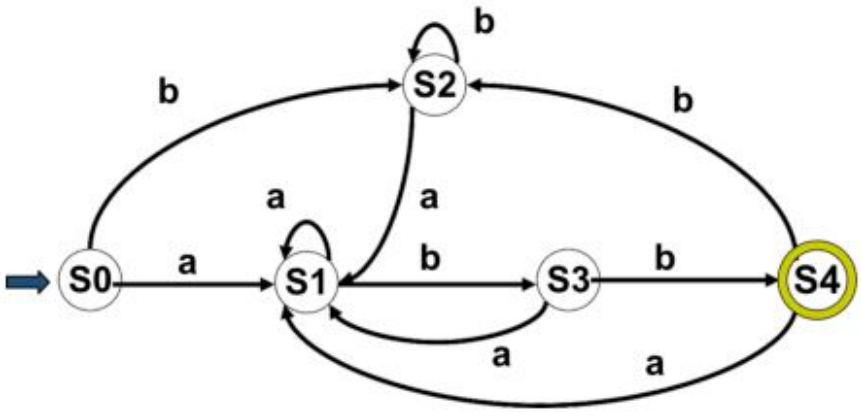

图 2-21　　测试样例使用的题目

其测试结果输出如下：（前四行为拆分出来的状态的集合，最后一行为最小化后的有限机的状态）

图 2-22　　测试样例的输出结果

输出结果正确，初步证明了所设计编写的最小化算法的正确性。只需要再细化细节（如：不同终态的判断等）即可。

2.4.6　测代码测试结果

以下的是词法分析中最小化后的自动机输出的结果（按照现态，输入符号，次态的方式存储）：

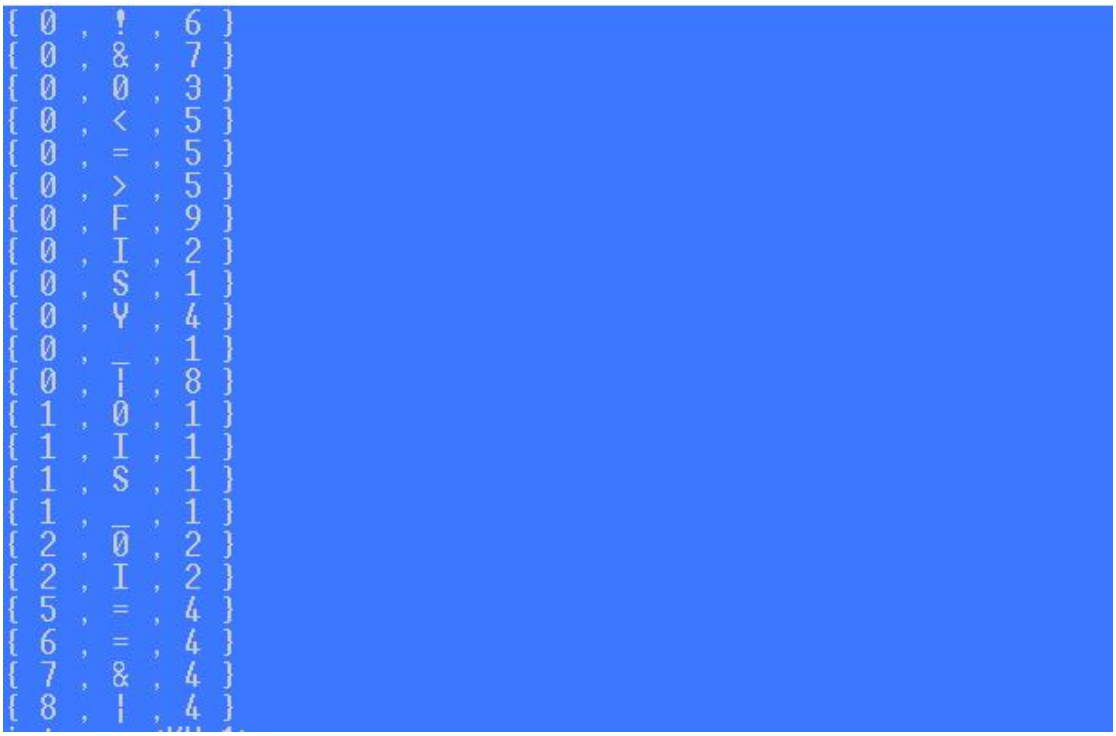

图 2-23　　代码测试结果

2.4.7 不足与改进

在完成代码编写之后的检查过程中发现可以改进的地方：如果状态的集合只有一个元素，那可以直接跳过，判断下一个集合，因为无论如何这个集合都不能再拆分了。这是本功能代码还可以进行改进的地方。

2.5 单词符号序列的生成

实现逻辑：

利用课程教授方法，通过确定化、最小化之后的DFA进行单词符号的识别，通过判断每个token到达的终态，判断其类型，进行相应的输出即可。基本流程为：

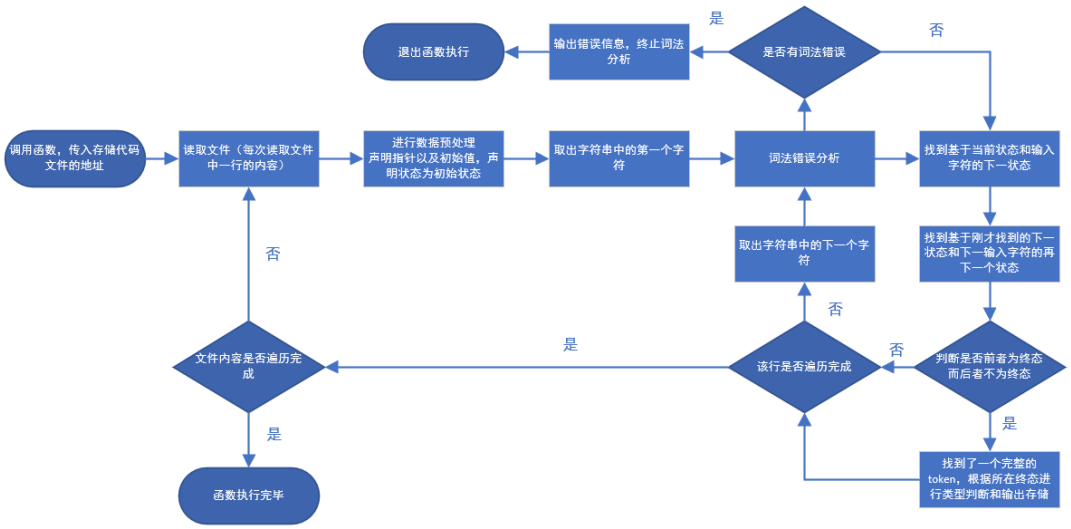

图 2-24　　实现逻辑

这一部分是词法分析最重要的部分，进行单词符号序列的生成与存储。

```
vector<string> Lexical_analysis(string address)
```

图 2-25　　函数声明

首先，函数传入了一个参数，该参数为需要进行词法分析的程序代码所在文件的地址，以方便在函数中进行内容读取。

通过ifstream、while以及string，将文件中的代码内容一行一行的读出以进行分析。

随后我们进行了数据预处理过程，和一些变量的声明。

```
ifstream file;
file.open(address, ios_base::in);
string use;
while(getline(file,use)){
```

图 2-26　　文件读入

```
use = Data_Preprocessing(use);
if(use==""){
    continue;
}
int front = checkNotEmpty(use);
int end = front;
int state = 0;
int errorflag = 0;
```

图 2-27　　准备工作

```
string Data_Preprocessing(string use){
    string res = use;
    int k;
    for(k = 0;k<res.length();k++){
        if(res[k]=='/' && res[k+1]=='/'){
            break;
        }
    }
    res = res.substr(0,k);
    return res;
}
```

图 2-28　　数据预处理

对于数据预处理，主要的任务就是将注释删掉，因为注释是以//开头，这会被识别为除号，所以我认为先删去好过后判断，这里我们找到了连着两个/的位置，利用substr函数取子串即可，如果是单纯的注释行，改行将为空，后去之后经过判断进行了continue操作，如果为代码加注释，那么注释部分将被删掉，返回代码部分。

然后，解释一下后边声明的四个变量，front和end为我们用来取单个单词的指针，front为这个单词在字符串中的开始位置，end为结束位置，通过这两个指针的不停移动，我们可以比较容易地将单词一个个识别出来，利用了checkNotEmpty函数，如下：

```
int checkNotEmpty(string use){
    int res = 0;
    while(use[res]==' ' || use[res]=='\t'){
        res++;
    }
    return res;
}
```

图 2-29　　checkNotEmpty函数

该函数的作用在于跳过改行一开始的空格符以及tab符，找到该行第一个单词的第一个字符的位置，随后赋值给front作为front和end的初始值。

state代表了当前所处的状态，state随着输入字符而变化，当然也不是每一个字符都会变化，最终单词识别完毕的state将帮助我们判断这个单词属于哪一类型，初始赋值为DFA的初始状态也就是0。

errorflag字面意思就已经很明显了，是用来判断是否发生了错误的标志位，后便会将错误以及错误处理方法进行统一说明。

```
for(int j = 0;j<DFA.size();j++){
    if(DFA[j].now==state && DFA[j].input==bewhich(use[i])){
        state = DFA[j].next;
        end = i;
        break;
    }
}
```

图 2-30　　利用DFA判断状态

对DFA进行循环，找到前一状态和输入等于我们分析过程的当前状态以及输入的那条边，如果存在这么一条边，state将会迁移到下一状态，这里我们在if中进行了break操作，因为我们这只是针对一个输入字符，如果没有这

个break，可以想象当输入&&时，同一个输入字符就会进行两次状态变化，会导致一些意想不到的错误。

```
int nextstate = -1;
for(int j = 0;j<DFA.size();j++){
    if(i+1 == use.length()){
        break;
    }
    if(DFA[j].now==state && DFA[j].input==bewhich(use[i+1])){
        nextstate = DFA[j].next;
    }
}
```

图 2-31　　利用DFA判断下一状态

```
string token;
if(checkZT(state) && !checkZT(nextstate)){
    token = use.substr(front,end-front+1);
    cout<<token<<"\t"<<checkWhich(state,token)<<endl;
    outwrite<<token<<"\t"<<checkWhich(state,token)<<endl;
    state = 0;
    int k = i+1;
    while(use[k]==' ' || use[k]=='\t'){
        k++;
    }
    front = k;
    end = k;
}
```

图 2-32　　判断是否找到完整token

然后我们对当前输入字符的下一个字符进行分析，在当前状态下，输入下一个字符会到达什么状态，这里需要注意的是这个循环必须在前边对state进行改变的循环的后边执行，因为这里面的state必须要用基于当前输入字符已经进行判断或改变的状态，这样我们才能判断输入下一个字符，状态将怎样进行改变，首先将nextstate初始值赋为-1方便后边进行判断，状态的寻找和刚才的过程基本相同，但多了一条判断，如果已经遍历到字符串尾了，直接break就可以了，避免访问超限。

找到了这么两个状态，我们就可以判断当前是否找到一个完整的单词了。

如果state是终态，而nextstate不是终态，那么可以这样认为，该单词已经无法在继续增大下去，也就是说找到了一个完整的单词，已经可以进行输出了，那么我们利用substr取到我们找到的单词，进行输出并写入文件，并且要对指针front和end进行修改，由于每个单词之间可能存在很多空格符以及tab符，所以这里我们利用while循环跳过他们，找到下一个单词的初始位置，赋给front和end，写入文件用到了ofstream已经在前边进行了声明：

```
ofstream outwrite;
outwrite.open("Lexical_Result.txt", ios::out);
```

图 2-33　　写入文件

这里边用到了checkZT和checkWhich函数，如下：

该函数的功能在于判断当前状态是否处于终态，由于我们对终态都进行了标记，所以这里我们利用dfamap即可判断当前状态是否处于终态，还记得前边我们将nextstate初值赋为-1吗，如果是-1传进来证明没找到下一个状态，那么直接返回false即可。

checkWhich函数的作用为根据当前状态以及找到的单词返回一个我们需要的具有固定格式的字符串，如果是终态的标识为I，证明处于数字终态，以此类推，前边已经介绍过I S F Y分别代表什么，需要注意的是指导书中强调关键字是不区分大小写的，所以先创造一个临时的字符串，其内容为token的小写形式，随后利用usemap判断其是关键字还是标识符，再进行相应的操作即可。

```
bool checkZT(int state){
    if(state == -1){
        return false;
    }
    if(dfamap[state]=='I' || dfamap[state]=='S' || dfamap[state]=='F' ||dfamap[state]=='Y'){
        return true;
    }
    return false;
}
```

图 2-34　　checkZT函数

```
string checkWhich(int state,string token){
    if(min_dfamap[state]=='I'){
        return "<INT,"+token+">";
    }else if(min_dfamap[state]=='Y' || min_dfamap[state]=='F'){
        return "<"+usemap[token].first+","+usemap[token].second+">";
    }else{
        string temp;
        for(int i=0;i<token.length();i++){
            if(token[i]>='A' && token[i]<='Z'){
                temp += token[i]+'a'-'A';
            }else{
                temp+=token[i];
            }
        }
        if(usemap[temp].first=="" && usemap[temp].second==""){
            symbol_table.insert(map<string,string>:: value_type(token,"int"));
            return "<IDN,"+token+">";
        }else{
            return "<"+usemap[temp].first+","+usemap[temp].second+">";
        }
    }
}
```

图 2-35　　checkWhich函数

checkWhich函数中用到的usemap是需要用到的格式化信息和token之间的映射关系，内容如下：

```
map<string,pair<string,string>> usemap={
    {"int",{"KW","1"}},{"void",{"KW","2"}},{"return",{"KW","3"}},{"const",{"KW","4"}},
    {"main",{"KW","5"}},{"+",{"OP","6"}},{"-",{"OP","7"}},{"*",{"OP","8"}},
    {"/",{"OP","9"}},{"%",{"OP","10"}},{"=",{"OP","11"}},{">",{"OP","12"}},
    {"<",{"OP","13"}},{"==",{"OP","14"}},{"<=",{"OP","15"}},{">=",{"OP","16"}},
    {"!=",{"OP","17"}},{"&&",{"OP","18"}},{"||",{"OP","19"}},{"(",{"SE","20"}},
    {")",{"SE","21"}},{"{",{"SE","22"}},{"}",{"SE","23"}},{";",{"SE","24"}},
    {",",{"SE","25"}}
};
```

图 2-36　usemap

此外，我们在识别到一个标识符的同时，应将其添加到符号表中，符号表格式无严格要求，经过网上查询资料以及词法分析所作内容，这里利用map¡string,string¿存储符号表,第一个string为我们找到的标识符，第二个string为队标识符的信息描述，因为词法分析能做到的有限，这里只对其类型进行描述，由于关键字中涉及变量声明的只有int和const（这里不再对整形和常整形进行区分），故我们所有变量的类型都可以描述为int。

由于并未将符号表存入文件，可用symbol_table_output对符号表进行输出以直观的看到符号表的内容及作用，函数如下：

```
void symbol_table_output(){
    map<string,string>::iterator it;
    for(it = symbol_table.begin();it != symbol_table.end();it++){
        cout<<"标识符: "<<it->first<<"\t"<<"类型: "<<it->second<<endl;
    }
}
```

图 2-37　符号表输出函数

到此，就完成了单词符号序列的生成、输出及写入文件和符号表的生成存储。

2.6　错误分析

共考虑三种错误：

非法字符输入

标识符以数字开头

数字以0开头

（1）非法字符输入

```
if(use[i]=='&' || use[i]=='|'){
    if(!checkNUM(i,use,use[i])){
        cout<<"输入: "<<use[i]<<" 不在识别范围内！停止词法分析！"<<endl;
        errorflag = 1;
        break;
    }
}
if(use[i]=='!' && use[i+1]!='='){
    cout<<"输入: "<<use[i]<<" 不在识别范围内！停止词法分析！"<<endl;
    errorflag = 1;
    break;
}
if(bewhich(use[i]) == '~'){
    cout<<"输入: "<<use[i]<<" 不在识别范围内！停止词法分析！"<<endl;
    errorflag = 1;
    break;
}
```

图 2-38　　非法字符错误处理

首先输入字符经过处理之后若不在我们的可输入字符集合中那么一定是一个违法字符，输出错误信息停止词法分析即可。

```
bool checkNUM(int pos,string senten,char whi){
    int j = pos;
    int res = 0;
    while(senten[j]==whi){
        j--;
    }
    j++;
    while(senten[j]==whi){
        res++;
        j++;
    }
    if(res % 2 == 0){
        return true;
    }else{
        return false;
    }
}
```

图 2-39　　checkNUM函数

还需要考虑特殊的情况，词法分析器不能接收！、&和|三种符号，但是后面三种是一个合法的输入，故我们需要判断是否输入了单个的&或者|以及是否输入了!，!的判断比较容易，而&和|的判断则比较困难，考虑这么一种情况，输入了8个&，那我们应识别出4个&&（虽然这在语法上是违法的，但是目前认为词法分析只关注是否符合词法规则），那么我们需要判断连续的|—和&的总个数以进行判断是否输入了单个的&或|，判断方法如上图2-39。

每次都找到输入的这个&或|符的连续&或|符号串的第一个符号，然后向后数，如果是偶数证明没有错误，返回true否则返回false。

（2）标识符以数字开头

```
if(dfamap[state]=='I' && bewhich(use[i+1])=='S'){
    cout<<"错误关键字或以数字开头的标识符！停止词法分析！"<<endl;
    errorflag = 1;
    break;
}
```

图 2-40　　标识符以数字开头

如果当前处于数字状态，后边却突然输入了字母，那么证明先输入了数字后输入了字母即以数字开头的标识符，输出错误信息，停止词法分析。

（3）数字以0开头

```
if(state == 3 && (bewhich(use[i+1])=='I' || use[i+1]=='0')){
    cout<<"数字不能以0开头！停止词法分析！"<<endl;
    errorflag = 1;
    break;
}
```

图 2-41　　数字以0开头

通过对最终DFA的分析，我们知道状态3对应输入0的终态，如果在此终态下，后边仍然输入数字，证明程序中输入了一个以0开头的数字，输出错误信息，停止语法分析。

2.7　阶段性测试

2.7.1　确定化测试

初始NFA以及NFA确定化后形成的DFA如下两图所示：

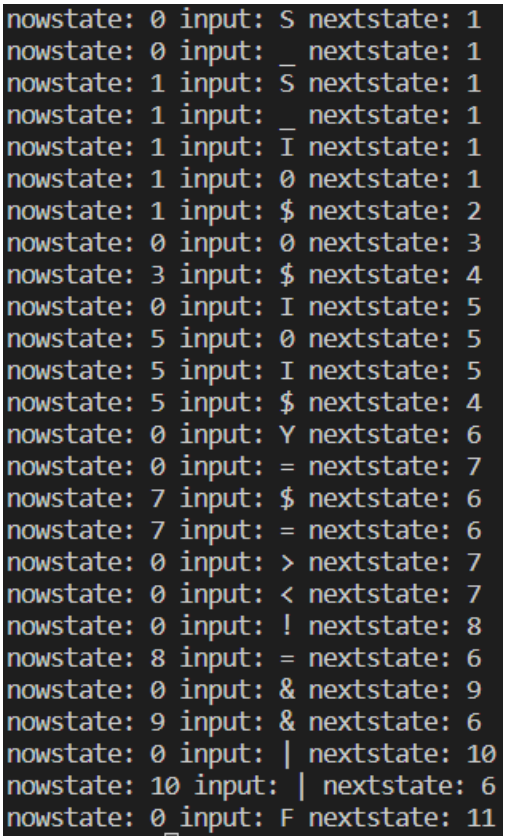

图 2-42　　初始NFA

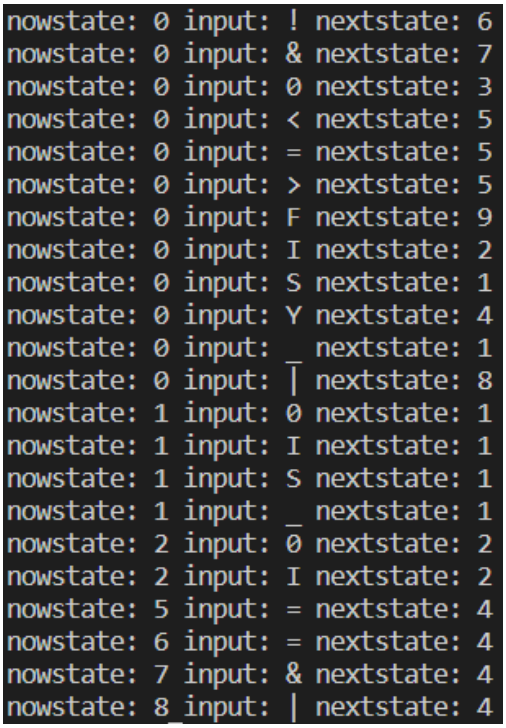

图 2-43　　确定化后得到的DFA

2.7.2 单词符号序列生成与存储测试

采取指导书中的例子：

```
1  int a = 10;
2  int main(){
3      a=10;
4      return 0;
5  }
```

图 2-44　例子

单词符号序列生成结果：

Lexical_Result.txt

```
1   int <KW,1>
2   a   <IDN,a>
3   =   <OP,11>
4   10  <INT,10>
5   ;   <SE,24>
6   int <KW,1>
7   main    <KW,5>
8   (   <SE,20>
9   )   <SE,21>
10  {   <SE,22>
11  a   <IDN,a>
12  =   <OP,11>
13  10  <INT,10>
14  ;   <SE,24>
15  return  <KW,3>
16  0   <INT,0>
17  ;   <SE,24>
18  }   <SE,23>
```

图 2-45　单词符号序列

2.7.3 错误测试

（1）输入非法字符：

代码片段：

```
1  int a = 10;
2  int b = !a;
3  int main(){
4      a=10;
5      return 0;
6  }
```

图 2-46　代码片段

输出结果：

```
int     <KW,1>
a       <IDN,a>
=       <OP,11>
10      <INT,10>
;       <SE,24>
int     <KW,1>
b       <IDN,b>
=       <OP,11>
输入: ! 不在识别范围内! 停止词法分析!
```

图 2-47　输出结果

（2）输入数字开头的标识符：

代码片段：

```
1  int a = 10;
2  int 0b = a;
3  int main(){
4      a=10;
5      return 0;
6  }
```

图 2-48　代码片段

输出结果：

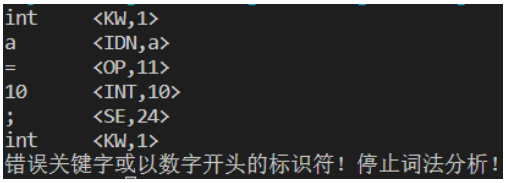

图 2-49　输出结果

（2）输入以0开头的数字：

代码片段：

```
1  int a = 10;
2  int b = 010;
3  int main(){
4      a=10;
5      return 0;
6  }
```

图 2-50　代码片段

输出结果：

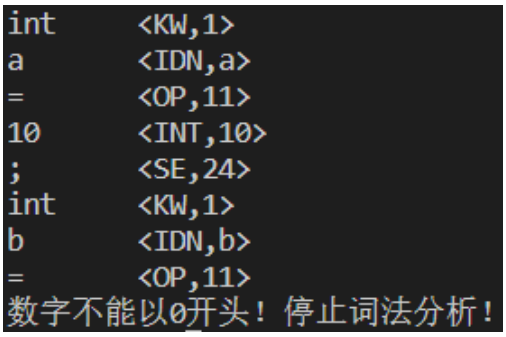

图 2-51　输出结果

2.8　词法分析整体逻辑

首先，提出NFA，在已有NFA的基础上，进行NFA的确定化和DFA的最小化，随后利用生成的最小DFA进行单词符号的识别和单词符号序列的生成与存储

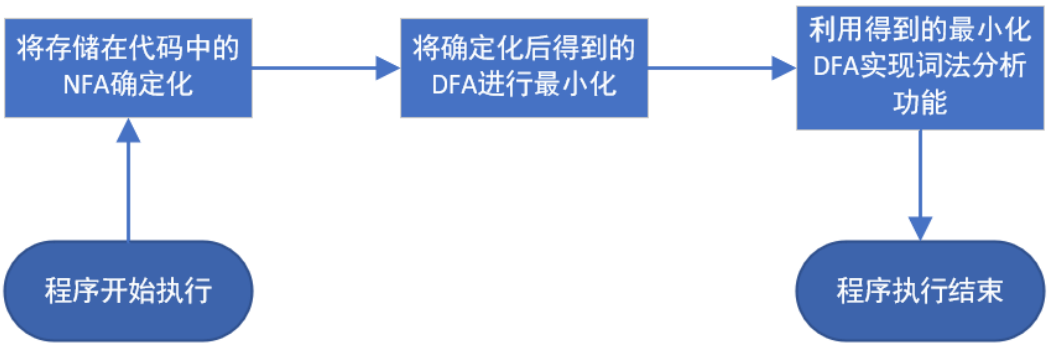
将存储在代码中的NFA确定化
将确定化后得到的DFA进行最小化
利用得到的最小化DFA实现词法分析功能
程序开始执行
程序执行结束

图 2-52　　整体逻辑

第三章　语法分析

3.1　语法分析器在编译其中的定位/功能

语法分析器需要接收词法分析器提供的Token序列，检查Token序列是否能够由源程序语言文法产生，对语法错误信息进行提示，并最终生成语法树。首先对给出的C−−文法进行初步处理，使用|符号分割每一条候选产生式，拟采用自顶向下的LL(1)分析法。首先对给出的C−−文法消除左递归、消除回溯，并对所有候选首符集中存在 ε 的终结符进行判断，确定最终获得的文法满足LL(1)分析条件。

3.2　获得LL(1)文法的算法

(1)消除左递归

若存在非终结符P，$P \overset{+}{\Rightarrow} P\alpha$，则判断文法是存在左递归的。左递归问题会导致分析过程陷入无限循环，设计算法如下以消除左递归。

Algorithm 1: 消除左递归

Input: *Grammar*1.*txt*

Output: *Grammar*1.*txt*

将文法G中的所有非终结符，按出现顺序排列成$P1, P2, \cdots, Pn$

for *i in (1,n)* **do**

  for *j in (1,i-1)* **do**

    convert $P_i \rightarrow P_j\gamma$ to $P_i \rightarrow \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$

    where $P_j \rightarrow \delta_1|\delta_2|\&|\delta_k$ is rule of P_j

  end

end

化简文法，去除由开始符号出发永远无法到达的非终结符的产生规则

(2)消除回溯

若非终结符P的所有候选首符集中存在两两相交的情况，则判断文法是存在回溯的。回溯问题会导致在对该非终结符P匹配输入串时，无法根据其面临的第一个输入符号a准确的指派某个特定候选前去执行任务，降低编译器工作效率。基于消除左递归后的文法，设计算法如下进一步消除回溯。

Algorithm 2: 消除回溯

Input: *Grammar*1.*txt*

Output: *Grammar*1.*txt*

while 非终结符的候选首符集非两两不相交 **do**
  for *i in (1,n)* **do**
    if $P \to \delta\beta_1|\delta\beta_2|\cdots|\delta\beta_n|\gamma_1|\gamma_2|\cdots|\gamma_n, \quad (\gamma notbeginwith\delta)$ **then**
      $P \to \delta P'|\gamma_1|\gamma_2|\cdots|\gamma_m$
    end
  end
end

(3)候选首符集存在空

若非终结符P的候选首符集存在 ε ，当P面临不存在于其候选首符集的输入符号a时，只有其后随符号集中存在a才判断文法是满足LL(1)的。基于消除回溯的文法，设计算法如下判断可否使用LL(1)分析。

Algorithm 3: 对候选首符集中存在 ε 的终结符进行判断

Input: *Grammar*1.*txt*

Output: *Grammar*1.*txt*

for *i in (1,n)* **do**
  if $\varepsilon \in FIRST(P)$ **then**
    if $FIRST(P) \bigcap FOLLOW(P) = \phi$ **then**
      do nothing
    else
      break
    end
  end
end

3.3 获取FIRST集合

基于LL(1)文法G生成FIRST集合，遍历每一条产生式 $X \to a_1 a_2 \cdots a_n$ ，设计算法如下构造非终结符 X 的候选首符集 $FIRST(X)$ 。算法过程中面临着以下几种情况：

① 将终结符 X 本身放入其候选首符集

② 对非终结符 X 推出终结符的情况进行处理

③ 对符号 X 能推出以非终结符开头的产生式进行处理

Algorithm 4: 生成FIRST集合

Input: $Grammar1.txt$

Output: $FirstSet$

```
while FirstSet is updating do
    if X ∈ V_T then
        FIRST(X) add X
    end
    else if X ∈ V_N then
        if have X → a··· (a ∈ V_T) then
            FIRST(X) add a
        end
        if have X → ε then
            FIRST(X) add ε
        end
    end
    if have X → Y··· (Y ∈ V_N) then
        FIRST(X) add FIRST(Y) - ε
    end
    if have X → Y_1 Y_2 ··· Y_k (Y_1, ··· , Y_k ∈ V_N) then
        for j in (1, i − 1) do
            if ALL FIRST(Y_j) have ε then
                FIRST(X) add ε
            end
        end
    end
end
```

3.4 获取FOLLOW集合

- FOLLOW（A）：紧跟在非终结符A后边的终结符α的集合
- 如果A是某个句型的的最右符号，则将结束符 # 添加到 FOLLOW（A）中

Algorithm 5: 生成FOLLOW集合

Input: $Grammar1.txt$

Output: $FollowSet$

put # into FOLLOW(S) where s is strat
while *FollowSet is updating* **do**
    if *have* $A \rightarrow \alpha B \beta A \rightarrow \alpha B \beta$ **then**
        $FOLLOW(B)$ add $FIRST(B)$ - ε
    end
    if *have* $A \rightarrow \alpha B A \rightarrow \alpha B$ **then**
        $FOLLOW(B)$ add ALL $FOLLOW(A)$
    end
    if *have* $A \rightarrow \alpha B \beta A \rightarrow \alpha B \beta$ *and* $\varepsilon \in FIRST(\beta)$ **then**
        $FOLLOW(B)$ add ALL $FOLLOW(A)$
    end
end

3.5 生成预测分析表

基于LL(1)文法G和生成的FIRST集合、FOLLOW集合，以所有非终结符为行、所有终结符为列（含#），遍历每一条产生式 $A \rightarrow \alpha$，设计算法如下构造分析表。以 $M[A, a]$ 表示分析表中各单元格的产生式。算法过程中面临着以下几种情况：

① 对产生式右部首符号的FIRST集中包含非终结符的情况进行处理

② 对产生式右部首符号的FIRST集中包含 ε 的情况进行处理

③ 对所有不合文法的情况进行出错处理

Algorithm 6: 生成预测分析表

Input: $FirstSet, FollowSet, non_terminal, terminal, proc$

Output: $Table$

for $all \quad a \in FIRST(\alpha), \quad (a \in V_T)$ **do**
  $M[A,a] \quad add \quad A \to \alpha$
end
if $\varepsilon \in FIRST(\alpha)$ **then**
  for $\forall b \in FOLLOW(A)$ **do**
    $M[A,b] \quad add \quad A \to \varepsilon$
  end
end
if $M[A,b] = null$ **then**
  $synch$
end

3.6 生成规约序列

预测分析程序按照栈顶符号和当前输入符号进行判断，以有序对（X，a）表示当前面临的栈顶符号为X，输入符号a。对于任意有序对（X，a），程序分为三种情况：

① X=a=#，分析成功并停止分析程序。

② X=a!=#，将符号X弹出栈顶，指向下一个输入符号。

③ X为非终结符，根据预测分析表进行判断

以 $M[A,a]$ 表示预测分析表中各单元格的产生式，串 w 表示输入串，S 表示开始符号，$w\#$ 表示未分析串，设计算法如下以生成规约序列：

Algorithm 7: 生成规约序列

Input: $Table, w$

Output: $sequence$

```
push  #S;
while X! = # do
    X− > top;
    p− > a;
    if X ∈ V_T or X = # then
        if X = a = # then
            accept  w
        end
        else if X = a! = # then
            pop  X,  ip− > next
        end
        else
            Error  Message
        end
    end
    else
        if M[X, a] = X → Y_1 Y_2 ··· Y_k then
            pop  X;
            push  Y_k, ··· , Y_2, Y_1
        end
        else
            Error  Message
        end
    end
end
```

3.7 文件结构组织

基于功能点对任务进行划分，每部分功能的实现在.cpp文件中完成，需要提供给其他文件的参数信息在.h中文件声明。最终对语法分析器中需要完成的任务及其输出、输出参数确定如下：

- 初始C−−文法(Grammar1.txt): 包含所设计编译器应用的所有文法规则，对于每个非终结符，以—符号分割其候选产生式。

- 公用参数(Param.h): 对于数据对象规模的确定，如文法中产生式的最多个数、FIRST及FOLLOW集合最大规模等。
- 文法初始化 (Init.h Init.cpp):

  ① 获取初始C--文法，生成开始符号、产生式个数、产生式、非终结符、终结符等五个参数信息。

  ② 使用3.2中的算法将文法转化为LL(1)文法。
- 获取FIRST集合、 FOLLOW集合(Set.h Set.cpp): 使用3.3及3.4中的算法生成FIRST集合、FOLLOW集合，需要Init.h文件中的参数作为输入。
- 生成预测分析表(Table.h Table.cpp): 使用3.5中的算法生成预测分析表，需要Init.h文件以及Set.h文件中的参数作为输入。
- 进行语法分析(syntax_analysis.h syntax_analysis.cpp): 使用3.6中的算法进行语法分析，需要词法分析器的输出以及Table.h中的参数作为输入。

考虑到工程组织复杂，为简化编译步骤编写Makefile文件建立文件间的依赖关系，可使用make命令进行编译。整体流程及对应文件可由下图简要表示：

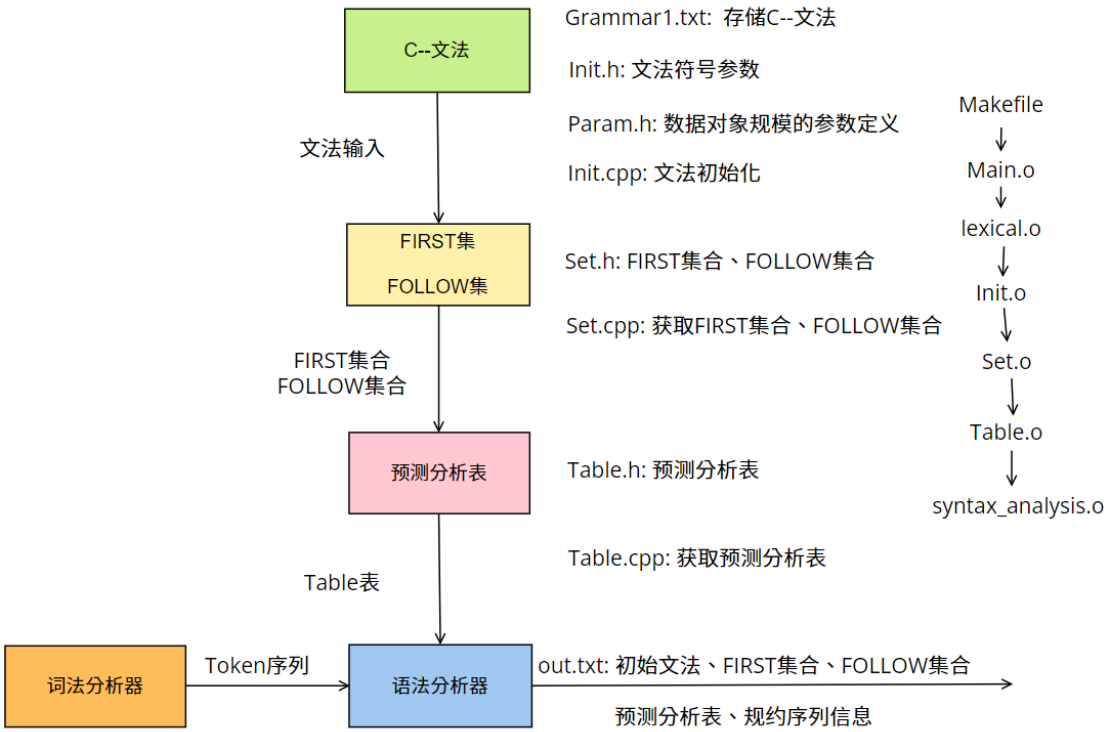

图 3-1 文件依赖关系

3.8 输出格式说明

使用make命令编译后，在终端输入./target.exe > out.txt 命令可将输出结果重定向到指定文件中。对语法分析器中涉及的输出及格式说明如下：

######FirstSet Result######

[终结符1]：[FIRST集合元素1] [空格] ... [FIRST集合元素n]

.....

[终结符n]：[FIRST集合元素1] [空格] ... [FIRST集合元素n]

图 3-2　　FIRST集合输出格式

######FollowSet Result######

[终结符1]：[FOLLOW集合元素1] [空格] ... [FOLLOW集合元素n]

.....

[终结符n]：[FOLLOW集合元素1] [空格] ... [FOLLOW集合元素n]

图 3-3　　FOLLOW集合输出格式

######Table Result######

[空格] [终结符1] [空格] [终结符2] [空格] ..... [终结符n] [空格]

[非终结符1] [产生式1] [空格] [产生式2] [空格] ..... [产生式n] [空格]

.....

[非终结符m] [产生式1] [空格] [产生式2] [空格] ..... [产生式n] [空格]

若规约表中对应位置不含产生式，则使用[空格]代替

图 3-4　　预测分析表输出格式

输出格式：

[序号] [TAB] [栈顶符号]#[面临输入符号] [TAB] [执行动作]

图 3-5　　规约序列输出格式

第四章　实验分工与个人感悟

4.1　实验分工

孙思远：非确定有限自动机设计、非确定有限自动机确定化，单词符号序列的生成、输出与存储、词法错误处理，开发报告、测试报告的编写以及测试样例的编写。

王悦君：确定有限自动机的最小化，符号表的生成、存储与输出，开发报告、测试报告的编写以及测试样例的编写。

丁小芮：文法初始化与FIRST集合、预测分析表的设计和生成，工程结构优化，编写开发文档、测试文档

黄丽丽：FOLLOW集合的生成，规约序列的生成，工程结构组织，编写开发文档、测试文档

4.2　个人感悟

孙思远：在本次实验的过程中，我主要负责非确定有限自动机设计、非确定有限自动机确定化，单词符号序列的生成、输出与存储，我认为最困难的地方在于非确定有限自动机设计，因为总会存在一些状态和输入考虑不周的情况，几经思考和手动推导，我才找到了比较完善的NFA版本随后的确定化和序列的生成、输出与存储紧跟老师上课所讲的内容，在这里我认为比较难的是如何利用自动机取识别一个个token，需要利用循环和状态的转换，总结来说，这次试验让我收获颇丰，理论联系实际，现在的我对课上知识更加掌握并且更加明白为什么我要学习这门编译原理课程，还有，小组共同工作也让我的交流能力、组织能力等得到了提升，我认为对我无论是学习还是生活都有许多益处。

王悦君：在本次实验的过程中，我主要实现了有限自动机的最小化和符号表的编写。编写过程中遇到了一些困难与问题。由于最小化的逻辑较为复杂，因此在实现的时候较为困难，需要考虑清楚每轮遍历的各种情况以及何时跳出循环。此外，在符号表的编写方面由于没有任何的要求与规范，因此刚开始编写的时候也出现了存储错误的情况，经过小组交流得以解决。通过本次实验，我深刻体验到了词法分析的过程，也理解了自动机相关算法的代码实现，获益匪浅。

丁小芮：在本次实验过程中，我主要负责语法分析器中的文法初始化、FIRST集合生成以及预测分析表生成部分。在使用自顶向下的LL(1)分析方法时，需要对原始文法进行判断，在其不满足条件是进行多步操作与转化。在生成FIRST集合与预测分析表时，依据课上学习的算法进行实践。由于算法中

的部分逻辑复杂，在调试过程中遇到很大困难，但也加深了我对语法分析器工作逻辑的理解。由于语法分析器涉及步骤较多，且需要进行合作开发，设计了.h参数文件与.cpp过程文件分离的形式，提供其他成员所需参数及函数，并结合Makefile文件优化工程结构。整体来说，本次实验加深了我对编译器语法分析过程的理解，学习优化工程结构方法的同时提高了工程组织能力，学习运用了自动化编译的方法。

黄丽丽：在本次实验中，通过实现Follow集合的生成，以及预测分析算法，生成规约序列，对于LL(1)文法分析的过程以及算法理解更加深刻熟练。由于采用C++语言编写，起初对于各个变量与函数的调用较为混乱，后来通过对于整个项目文件的组织，提取出全局的变量以及方法，通过编写Makefile来编译项目，解决各个功能之间的依赖问题，对于整个编译器前端流程也更加熟悉。本次实验较好之处在于实现了基本功能并测试，且将小组成员分别实现的模块化功能分离的代码组织成一个项目；不足之处在内部接口没有在实验之前设计和确定，导致变量类型不一致，存在冗余转换过程的问题。