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

班　　级　9191062301

学生姓名　李 骜

学　　号　919106840127

指导教师　项欣光

南京理工大学计算机科学与工程学院制

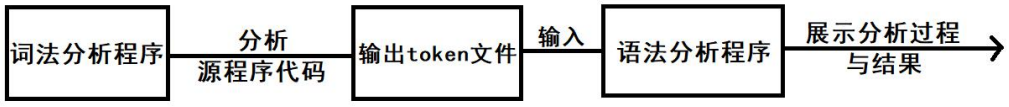
目录

1、项目介绍

1.1 项目框架

词法分析与语法分析有 2 种接口方式。词法也是语法的一部分，词法描述完全可以归并到语法描述中去，在考核要求中也只规定了两个任务的输入、输出与细节要求，并没有强制性地将词法分析器与语法分析器分离。

然而考虑到将词法分析工作分离可以使得整个编译程序的结构更加简洁、清晰和条理化，并改进编译程序的效率、增强编译程序的可移植性，本项目最终选择了将二者分离实现的接口方式。



在项目文件夹中可以把项目分成两部分：

词法分析：test.txt，grammar_3.txt，Lexical_Analyzer.cpp，token.txt

语法分析：token.txt，grammar_2.txt，Syntactic_Analyzer.cpp

其中 test.txt 是用于测试的代码文件，grammar_3.txt 是用于词法分析的 3 型文法，Lexical_analyzer.cpp 是词法分析程序,token.txt 是词法分析的输出；grammar_2.txt 是用于语法分析的 2 型文法，Syntactic_Analyzer.cpp 是语法分析程序，该程序没有输出文件，分析

过程与结果将直接显示在屏幕上。

1.2 项目内容

根据清华大学出版社的《编译原理》（第 3 版）的定义，编译程序完成从源程序到目标程序的翻译工作，是一个复杂的整体的过程。从概念上来讲，一个编译程序的整个工作过程是划分成阶段进行的，每个阶段将源程序的一种表示形式转换成另一种表达形式，各个阶段进行的操作在逻辑上是紧密连接在一起的。一种典型的划分方法将编译过程划分成词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成 6 个阶段。根据考核要求，本项目实现了词法分析与语法分析两个过程。

词法分析是编译过程的第一个阶段。这个阶段的任务是从左到右一个字符一个字符地读入源程序，对构成源程序的字符流进行扫描和分解，从而识别出一个个单词（一些场合下也称为单词符号或符号）。

语法分析是编译过程的第二个阶段。语法分析的任务是在词法分析的基础上将单词序列分解成各类语法短语，语法分析所依据的是语言的语法规则，即描述程序结构的规则，通过语法分析确定整个输入串是否构成一个语法上正确的程序。

要求：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、词法分析程序

2.1 词法分析步骤

词法分析的主要任务是从左至右逐个字符地对源程序进行扫描，产生一个个单词序列，用于语法分析。

主要步骤如下：首先，词法分析从 grammar_3.txt 文件中读取输入的 3 型文法，记录其中的表达式，随后根据读取的表达式构建 NFA，并将 NFA 确定化，转换为 DFA。最后读入 test.txt 文件中的源程序，用 DFA 对其进行分析，并将分析结果以 token 表的形式表示出来，并储存在 token.txt 文件中以便语法分析器使用。

主要流程图如下：

2.2 输入的三型文法

根据考核要求，生成的 token 共有 5 种类型：关键字，标识符，常量，限定符与运算符。想要程序能正确识别以上类型，需要构造正确的三型文法。

值得注意的是，为了在程序中识别方便，本项目把文法中所有非终结符用单个大写字母表示，所有终结符用单个非大写字母字符表示，每个字符具体的含义会在下面叙述，且也会在文法文档中以“$”开头的形式注释出来。

2.2.1 关键字与标识符

可以看出关键字的组成逻辑与标识符是一样的，都是字母或下划线开头的字母、数字与下划线的组合，且关键字的数量有限，因此我选择直接在文法中把二者视为一体。

A 代表标识符/关键字，也就是字母、下划线开头的字母、数字与下划线的组合

l 代表除了 e 与 i 之外的小写字母，之所以把 e 和 i 摘出去是因为科学计数法与复数中要用到

由于标识符与关键字不能由数字开头，所以使用了一个 B——字母、数字、下划线的组合

A->_|l|e|i|eB|iB|_B|lB

B->_|l|e|i|z|o|_B|lB|eB|iB|zB|oB

考虑到关键字数量有限，本项目直接使用一个 string 类型的数组来表示其内容。

```
string
keyword[MAXN]={"main","cin","cout","au
to","break","case","char","const","con
tinue","default","do","double","else",
"enum","extern","float","for","goto","
if","int","long","register","return","
short","signed","sizeof","static","str
uct","switch","typedef","unsigned","un
ion","void","volatile","while"};
```

值得注意的是该数组中其实并不全是 C 语言关键字，也包含了一部分预编译内容，如 main，它们并不是关键字，但考虑到它们在程序中的作用，在本项目中暂且视作关键字来使用。

2.2.2 限定符

由于限定符本来数量就不多且使用较为简单，本项目的限定符用一条语句直接写出。

C 代表限定符

C->,|;|(|)|[|]|{|}

2.2.3 运算符

运算符与限定符类似，大部分都可以直接写出，而双目运算符要在一部分单目运算符的后面加一个等号。

D 代表运算符，由于要表示双目运算符所以加了一个 E，代表双目运算符后半部分的等号

D->+|-|*|/|%|&|^|=|>|<|>E|<E|=E|+E|-E|*E|/E

E->=

2.2.4 常量

相比于以上三种类型，常量要复杂一些，需要看情况将其分为整数、普通小数、科学计数法与复数。

F 代表整型，其中 z 代表 1-9 的数字，o 代表 0

F->z|zG|o

G->z|o|zG|oG

H 代表科学计数法以及普通十进制小数的开头，由于以上二者必须以数字开头，使用 I 来代表第一个数字后边的部分

H->zI|oI

I->zI|oI|.J|eK

J 代表小数点后的部分，由于后面可能还要接科学计数法所以加一个 N

J->z|zN|oN

K 代表 e 后的第一个符号，可以是正负号或者整数，L 代表 e 后边正负号后的整数

N->z|o|zN|oN|eK

K->+L|-L|z|zM

M 代表 L 第一个字符后的部分

L->z|zM

M->z|o|zM|oM

O 代表复数，P 代表复数前半部分与后半部分连接用的正负号

O->zP|oP|zO|oO

Q 代表复数后边的数字，并在数字结尾加 i

P->+Q|-Q

Q->zQ|oQ|i

这里要注意，复数其实还可以继续细化，不过其细化方式与科学计数法类似，因此这里不再赘述。

2.2.5 开始符号

在确定好了以上种类的 3 型文法表示之后，还需要一个开始符号 S 指向这些类型，因此加入以下表达式：

S->A|C|D|F|H|O

2.3 读取 3 型文法

词法分析程序的首要工作就是读取 3 型文法，通过读取到的每一行的表达式来生成 NFA，由于本项目在构造文法的时候就考虑到了读取问题，所以本项目的 3 型文法每一个表达式都单独占一行，且其中的每个终结符或非终结符都只占 1 个 char 类型，除此之外还保证每个表达式的格式都是 S->AB（例），即表达式左右两侧是通过->连接的，因此在识别的时候只需把用 getline()按行读取，将读取的结果存在 string nows 中，nows[0]就是表达式左部，nows[3]到 nows[nows.length()-1]就是表达式右部。

每读取到一行 nows，就可以依托该字符串生成对应 NFA 了。

对应部分代码如下：

```
string nows;  //当前字符串
while(getline(myfile,nows)){
    if(nows[0]=='$') continue;  //美元符代表注释，跳过
    create_NFA(nows);
}
```

2.4 NFA 生成

生成 NFA 的第一步是构造 NFA 的数据结构，为了清晰地表示 NFA 各结点的内容与关系，本项目将所有 NFA 放置在一个大的结构体 nfa 中，在 nfa 中包含一个 char 类型的数组 K 用来代表 NFA 中的各个结点（也就是表达式中的各个非终结符），一个 int 类型变量 nk 代表结点的个数，一个 int 类型变量 nf 代表 NFA 中的边的个数，对于边的表示，本项目创建了一个结构体 F，其中含有三个 char 类型变量，分别代表每条边的箭头左侧结点，箭头右侧结点，与边上的终结符，因此本项目又在 nfa 中放置了一个 F 类型的数组，代表所有边。数据结构定义如下：

```
struct F  //从K到I的映像，如A-(a)>B
{
    char zuo; //式子左边，即A
    char zho; //式子中间，即a
    char you; //式子右边，即B
};
struct NFA  //NFA数据结构
{
    char K[MAXN]={0}; //结点集合
    int nk; //结点个数
    F f[MAXN]={0};  //K到I的映射，即NFA图上各结点之间的“边”
    int nf; //“边”的个数
} nfa;
```

在创建了 nfa 数据结构之后，首先需要对其初始化，具体操作为将终态结点 Z 放入 K 中，并将 nk 置为 1（因为此时只有 Z 一个结点），

将 nf 置为 0（0 条边）。

随后对于读到的每个表达式，通过 create_NFA()函数来创建 NFA。

创建 NFA 的基本原则是对于 A->b 类型的表达式，创建一条从 A 到 Z 的边，边上的终结符是 b，对于 A->aB 类型的表达式，创建一条从 A 到 B 的边，边上的终结符是 a，对于 A->B 类型的表达式，创建一条从 A 到 B 的边，边上的终结符是@，也就是空（无条件转移）。

需要注意由于每一行的表达式中都可能含有多个|，在遇到| 的时候应将表达式拆解开，比如说 A->b|aB，这种情况下就需要将 A->b 与 A->aB 分开处理，也就是说一个表达式创建两条不同的边，当遇到更多的|的时候处理方法也类似。

```
if(nows.length()==1){ //只有一个字符，要看是大写还是小写
    if(nows[0]>='A'&&nows[0]<='Z'){  //假如是大写字母，即无条件转移结点
        nfa.f[nfa.nf].zuo=Left; //边的左边是Left，即箭头左侧的结点
        nfa.f[nfa.nf].zho='@';  //边的中间是'@'，即无条件转移
        nfa.f[nfa.nf].you=nows[0];  //边的右边是nows[0],由于nows长度为1，因此是0
        nfa.nf++; //边数加一
    }
    else{ //不是大写字母，说明是小写字母或其它字符，直接连到结束结点
        nfa.f[nfa.nf].zuo=Left; //边的左边是Left，即箭头左侧的结点
        nfa.f[nfa.nf].zho=nows[0];  //边的中间是nows[0]，由于nows长度为1，因此是0
        nfa.f[nfa.nf].you=nfa.K[0]; //K[0]是Z即结束结点
        nfa.nf++; //边数加一
    }
}
```

```
else{ //否则长度为2，第一个为终结符，第二个为终结符
    nfa.f[nfa.nf].zuo=Left; //边的左边是Left，即箭头左侧的结点
    nfa.f[nfa.nf].zho=nows[0];  //边的中间是nows[0]即字符串中的终结符
    nfa.f[nfa.nf].you=nows[1];  //边的右边是nows[1]即字符串中的非终结符
    nfa.nf++; //边数加一
}
tk++; //跳过'|'，直接到下一个字符
```

需要注意的是，本项目取表达式的方式是将左右部分别取得，再组合处理的方式，其中右部的取得方法是遇到|则停止，因此当整个串都处理完之后仍有一个右部未处理，需要将其与左部连接再处理一

次。

每调用一次函数，会将 3 型文法的一行生成 NFA，因此实际调用的时候应搭配循环语句多次使用。

2.4 DFA 生成

本项目生成 DFA 的方法是子集法，构造逻辑参考了清华大学出版社《编译原理》（第 3 版）的方法。

由于 DFA 的每个结点代表了一个结点集合，内含多个非终结符，因此 DFA 的数据结构与 NFA 类似，只不过将 K 从 char 变成了 char 数组，而它们的表示方式也从 NFA 的直接输出内容变成了输出数组编号。数据结构定义如下：

```
struct FF{    //DFA的转换函数的数据结构
    char zu[MAXN]={0};    //以下三个分别是式子的左、中、右,左、右是NFA结点的集合，中是终结符
    char zh;
    char yo[MAXN]={0};
};
struct K{
    char K[MAXN]={0};
};
struct DFA{ //DFA数据结构
    K Kk[MAXN]={0}; //结点集合的集合
    int nk=0;    //结点集合数量
    FF f[MAXN]={0}; //转换函数的集合，即“边”
    int nf=0; //“边”的个数
} dfa;
```

构造完数据结构后，为了使用子集法构造 DFA，需要先定义闭包运算 closure()与 a 弧转换运算 move(I,a)，闭包的实现方法是状态集 I 中任何状态 S 经过任意条@（也就是空）能到达的状态的集合，move 的实现方法是状态集 I 中某一状态能通过一条 a 弧到达的状态的全体集合。部分实现代码如下：

```
for(int i=0;Sl[i];i++){ //遍历Sl中每个元素，把能通过'@'转到的非终结符放入Sl中
  for(int j=0;j<nfa.nf;j++){  //遍历每一条边
    if((nfa.f[j].zuo==Sl[i])&&(nfa.f[j].zho=='@')){ //假如有一条边起点为Sl[i]，且边上是'@'
      char temr=nfa.f[j].you; //取边的右边结点
      bool pan=false;
      for(int k=0;Sl[k];k++){ //判断该结点是否已经在Sl中
        if(Sl[k]==temr){  //假如结点已经在Sl中，pan取true，否则默认是false
          pan=true;
          break;
        }
        else continue;
      }
      if(pan) continue; //假如已在Sl中，判断下一条边
      Sl[l]=temr; //否则将temr即右边的结点放到Sl中
      l++;
    }
    else continue;  //假如该边不满足条件则跳过,看下一条边
  }
}
```

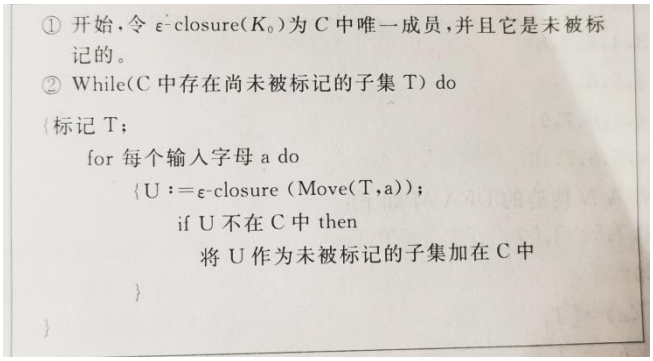
以上为 closure。

```
void MoveS(char *A,char a,char *B){ //move(A,a)，为了防止撞关键字大写了一下，加了个S,B储存结果
  int nb=0; //记录B中元素个数
  for(int i=0;A[i];i++){  //遍历A中的每个结点
    for(int j=0;j<nfa.nf;j++){  //对每个结点，遍历nfa中每条边
      if((nfa.f[j].zuo==A[i])&&(nfa.f[j].zho==a)){  //假如边左侧是A[i]且终结符为a
        B[nb]=nfa.f[j].you; //把边右侧结点存入B中
        nb++; //nb加一
      }
      else continue;
    }
  }
  SortS(B); //把需要存的全存完之后把B排列一遍
}
```

以上为 move。

生成 DFA 的步骤如下：

（1）DFAM 的状态集 S 由 K 的一些子集组成（构造 K 的子集的算法如下图），用[S1,S2,……,Sj]表示 S 的元素，其中 S1……Sj 是 K 的状态。并且约定，这些 S 按照某种规律排列。



① 开始，令 ε-closure(K_0)为 C 中唯一成员，并且它是未被标记的。

② While(C 中存在尚未被标记的子集 T) do

{标记 T；

for 每个输入字母 a do

{U：=ε-closure (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

}

}

（2）DFAM 和 NFAN 的输入字母表是相同的。

（3）转换函数 D 这样定义：

D([S1,……Sj],a)=[R1……Ri]

其中 closure(move([S1,……,Sj],a))=[R1,……,Ri]

（4）S0=closure(K0)是 DFAM 的开始状态

（5）St={[Sj,Sk,……,Se]，其中[Sj,……,Se]属于 S 且{Sj……Se}与 Kt 的交集不为空}

部分实现代码如下：

```
void create_DFA(){  //根据已生成的NFA生成DFA，使用子集法
  char Zero[MAXN]={'S'};  //代表头结点
  closure(dfa.Kk[0].K,Zero); //求头节点的闭包，成为第一个子集
  dfa.nk=1; //刚开始只有一个dfa结点，就是dfa.K[0]
  char Ter[MAXN]={0}; //用来存放所有终结符
  getTerminator(Ter); //取得所有终结符
  for(int i=0;i<dfa.nk;i++){  //用一个整型变量i来代替“标记”，i走到哪个位置说明哪个子集被标记处理了
    for(int j=0;Ter[j];j++){  //每个终结符处理一遍
      char TemS[MAXN]={0};  //工具子集
      MoveS(dfa.Kk[i].K,Ter[j],TemS);  //求move(dfa.K[i],Ter[j])
      char PointS[MAXN]={0};  //目标子集
      closure(PointS,TemS); //求-closure(move(dfa.K[i],Ter[j]))，储存在PointS中
      if(!(PointS[0])) continue; //假如PointS第一个字符为空，直接看下一个终结符
      bool pan=false;
      for(int t=0;t<dfa.nk;t++){  //遍历一遍dfa.K，看PointS是否已经在dfa.K中
        if(equalS(dfa.Kk[t].K,PointS)){  //假如存在于PointS相同的，pan为true，推出遍历
          pan=true;
          break;
        }
        else continue;  //否则继续遍历
      }
      if(pan){ //假如pan==true，说明PointS已在K中，只需加一条边即可
        if(Ter[j]=='@') continue; //假如边上是'@'，则不存
        CopyS(dfa.f[dfa.nf].zu,dfa.Kk[i].K); //f[dfa.nf]的左侧就是dfa.K[i]代表的子集
        dfa.f[dfa.nf].zh=Ter[j];  //f[dfa.nf]的中间就是Ter[j]代表的终结符
        CopyS(dfa.f[dfa.nf].yo,PointS); //f[dfa.nf]的右侧就是PointS代表的子集
        dfa.nf++; //dfa边数加一
      }
      else{ //否则说明得到的子集不在K中，需要把结点放进去，并加入一条边
        if(Ter[j]=='@') continue; //假如边上是'@'，则不存
        CopyS(dfa.f[dfa.nf].zu,dfa.Kk[i].K); //f[dfa.nf]的左侧就是dfa.K[i]代表的子集
        dfa.f[dfa.nf].zh=Ter[j];  //f[dfa.nf]的中间就是Ter[j]代表的终结符
        CopyS(dfa.f[dfa.nf].yo,PointS); //f[dfa.nf]的右侧就是PointS代表的子集
        dfa.nf++; //dfa边数加一
        CopyS(dfa.Kk[dfa.nk].K,PointS);  //加入完边之后把PointS结点放入K中
        dfa.nk++; //结点（子集）数加一
      }
    }
  }
}
```

2.5 对提取好的单词进行识别判断

为了进行识别本项目创建了一个 distinguish_Word(int node,string word)函数，用于判断输入 word，从 DFA 的 node 号结点出发，能否一路识别到终结点。其中 word 代表的是已经提取好的一段单词，比如 string 或 i 或 100。

具体识别方法是从 node 号结点开始，将单词的第一个字符输入进去，得到对应得下一个结点 n，然后删除单词的首字符，假如此时单词长度为 1，判断 n 是不是终结结点，否则把剩下的单词和下一个结点 n 继续调用本函数，直到识别成功或失败。

部分实现代码如下：

```
if(word.length()==0)  return false; //假如word是空串，直接return false
else if(word.length()==1){ //假如word长度为1
  for(int i=0;i<dfa.nf;i++){  //遍历dfa的每一条边，找出其中从node号结点起，边上是word[0]的边
    if(equalS(dfa.Kk[node].K,dfa.f[i].zu)&&dfa.f[i].zh==word[0]){ //找到边
      if(is_EndNode(dfa.f[i].yo)) return true;  //假如该边右边的结点是结束节点，返回true
      else continue;  //否则继续考虑其它边
    }
  } //for循环结束还没有return说明没有找到符合条件的边
  return false; //找不到符合条件的边，return false
}
else{ //word长度大于1
```

```
char noww=word[0];  //当前要判断的字符
string Nword="";
Nword=del_FirstS(word); //取Nword为word删去第一个字符
for(int i=0;i<dfa.nf;i++){  //递归调用每一条从node号结点出发，边上是noww的边
  if(equalS(dfa.Kk[node].K,dfa.f[i].zu)&&dfa.f[i].zh==noww){  //找到边
    int nex=0;  //代表该边右边结点号
    for(int j=0;j<dfa.nk;j++){
      if(equalS(dfa.Kk[j].K,dfa.f[i].yo)){  //找到子集内容与当前边右侧相同的结点
        nex=j;  //得到结点号
        break;
      }
      else continue;
    }
    if(distinguish_Word(nex,Nword)) return true;  //假如下一结点，Nword字符串识别成功
    else continue;  //否则看下一条边
```

2.6 对一行连续的单词进行识别判断

对一行单词，设置三个 int 类型变量 head，now，pre，用 head

到 now 位置对应的字符串代表当前录入的单词，用 head 到 pre 代表上一个能通过 DFA 的字符串,每遍历一次判断 head 到 now 能否被 DFA 识别，假如能则让 pre 等于 now，now 自增，直到 head 到 now 不能被 DFA 识别位置。此时识别 head 到 pre，然后存储识别结果，之后给 head 和 pre 重新赋值，让它们等于 now，随后开始新一轮的循环，直到一行字符串结束。

此方法对于大部分情况适用，但是当遇到科学计数法与复数的时候可能会出现问题，因此需要在判断的时候将这两种情况单独摘出，实现可能略微麻烦。但是这种办法克服了必须用空格或换行符分割的限制，遇到并未标准化的代码如 for(int i;;)也能识别，而不需要改成 for (int i ; ;)，实现之后使用的鲁棒性较好。

函数名称为 distinguish_Sentence()，部分实现代码如下：

```
int head=0,now=0,pre=0; //三个int变量起到指针作用，head到pre代表前一个能识别的字符串，head到now代表
while(now<sentence.length()){ //遍历整个输入的句子
  if(create_SonS(sentence,head,now)==" "){  //假如是空的，说明head与now在同一位置且当前位置为空的，
    now++;
    head++;
    pre++;
    continue;
  }
  while(distinguish_Word(0,create_SonS(sentence,head,now))){  //只要head到now还能识别
    pre=now;  //让pre等于now
    now++;  //now指向下一个字符
    if(sentence[now]=='.') now++; //以下四种情况用于识别小数、科学计数法与复数
    if((sentence[now]=='+'||sentence[now]=='-')&&(sentence[now+1]=='i'||sentence[now+1]=='z'))
    if(sentence[now]=='z'&&sentence[now+1]=='i') now++;
    if((sentence[now]=='e')&&(sentence[now+1]=='z')) now++;
  } //while结束说明head到now已无法匹配
  if((sentence[head]=='z'||sentence[head]=='o')&&(sentence[now]=='l'||sentence[now]=='_')){ //
    char saveChar=SentencE[head];
    head++;
```

2.7 对代码文件进行识别判断，并生成 token 文件

由于本项目前面的功能全部封装良好，因此读取并判断整个代码文件的步骤较为简便，只需要按行读取代码文件，然后每一行依次调

用 distinguish_Sentence()即可。为了将分析结果输出为文件的形式，需要创建一个 string 类型的 vector，每分析完一行代码就将分析结果放入 vector 中，全部分析完后使用 ofstream 来保存结果。

部分实现代码如下：

```
ofstream fout;  //用于向token文件输出
fout.open("d://vscode//vsCode//Curriculum_design//token.txt");  //打开文件
int tem=0;
for(int i=0;i<vec.size();i++){
    fout<<vec[i]; //为了防止内容丢失，先全部存在vec中，一起输出
    tem++;
    if(tem!=3) fout<<"        ";
    if(tem==3) fout<<"\n";  //每三个内容就换一次行
    if(tem==3) tem=0;
}
fout.close(); //关闭以保存文件
```

这里需要注意的是在输出结束之后不要忘了将 ofstream 关闭，否则结果无法保存。

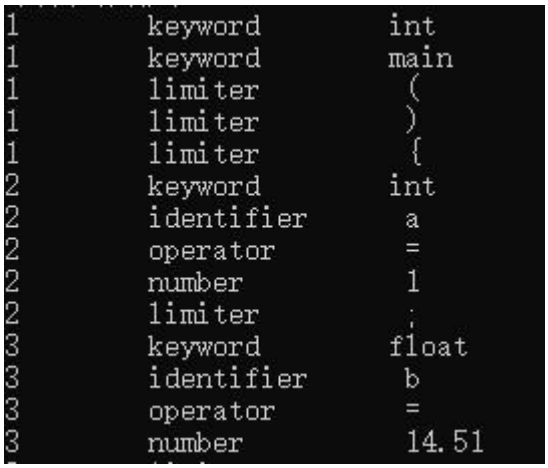
在尝试使用 vector 暂存分析结果、最后一起输出 txt 文件之前本项目曾尝试在分析每一句的时候就将结果保存进 txt 文件，后来发现这种方法由于需要不停打开和关闭 ofstream，最后只能保存最后一行 token，因此经过考虑本项目最终使用了 vector。

2.8 运行实例和截图

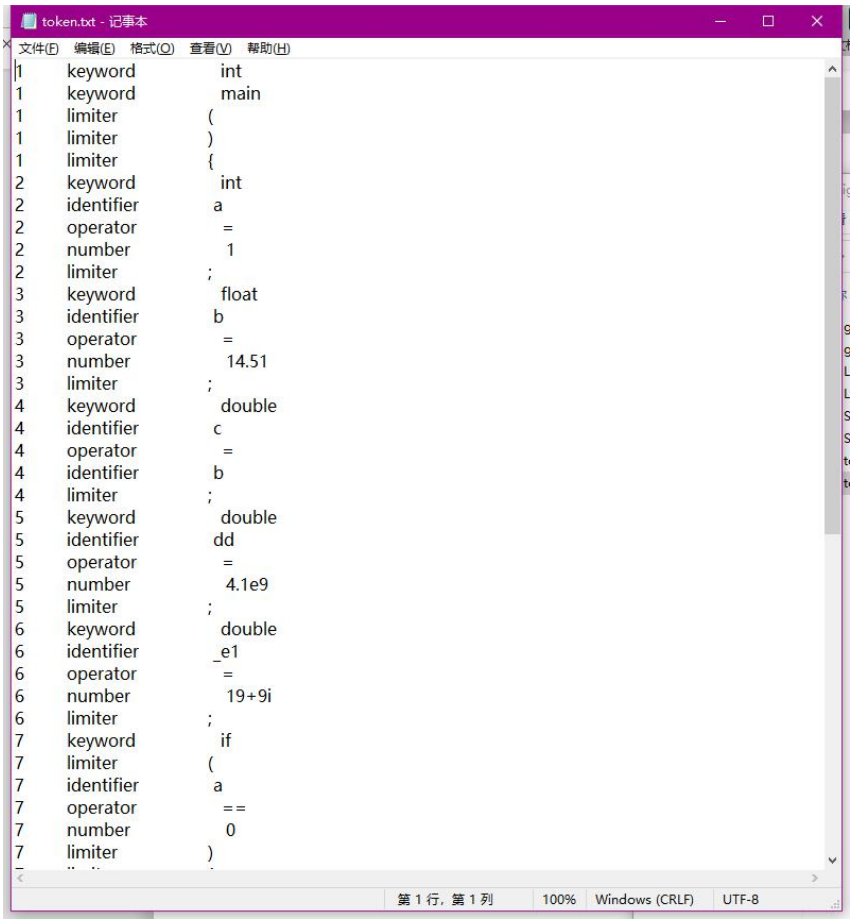
用于测试的源代码截图如下：

```
int main(){
  int a=1;
  float b=14.51;
  double c=b;
  double dd=4.1e9;
  double _e1=19+9i;
  if(a==0){
    a=1;
  }
  while(a>0){
    a=114;
    break;
  }
  return 0;
}
```

部分分析结果如下：

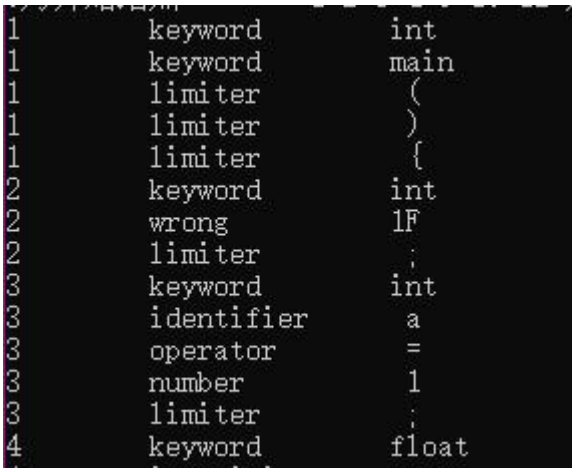


token 文件截图如下：



可以看到在 token 表中正常完成了考核要求，科学计数法与复数都可以识别。

接下来验证查错功能，在源程序第二行加一个错误的标识符。



可以看到该错误标识符被以“wrong”的形式表示了出来。

3、语法分析

3.1 语法分析概述与步骤

语法分析是编译程序的核心功能之一。语法分析的作用是识别由词法分析给出的单词符号串是否是给定文法的正确句子（程序）。语法分析常用的方法可分为自顶向下分析和自底向上分析两大类。而这两大类又都各自分为几种实现方式。

本项目选择使用 LR（1）分析法，这种方法比起自顶向下的 LL（k）分析方法和自底向上的优先分析方法对文法的限制要少得多，也就是说对于大多数用无二义性上下文无关文法描述的语言都可以用相应的 LR 分析器进行识别，与此同时这种方法还有分析速度快，能准确、即时地指出出错位置的特点。它的主要缺点是对于一个实用语言文法的分析器的构造工作量相当大，且 k 越大构造越复杂、实现越困难，因此本次项目仅构造了 LR（1）分析器。

LR（1）分析器由三个部分构成：总控程序、分析表或分析函数、分析栈。本次项目语法分析的步骤是首先读入并将文法进行拓展，也

就是在原本的开始符号 S 前加一个 S’，由于本语法分析器使用的文法中的每个非终结符都只占一个 char，所以用 T 代替 S’；随后正常应该将表达式中由|分隔开的部分拓展为新的表达式，比如 S->a|b 应化为 S->a 和 S->b，但是为了节省工作量在构造文法的时候这步就已经完成了；下一步应该计算每个非终结符的 First 集，在实际使用中发现可能出现需要计算多个连起来的符号的 First 集的情况，因此又增添了计算字符串的 First 的函数；再之后应该根据已取得的 First 集与读入的表达式构造每个闭包的项目集及 GO 函数；随后计算每个状态的 ACTION 表和 GOTO 表；最后根据上一步创建的两表进行语法分析。

流程图如下：

3.2 输入的二型文法

为了验证在词法分析器中写入的源代码，本项目构建了如下二型文法，$符后的内容代表注释。该文法支持对于多个并列的函数的识别，包含主函数，函数中支持返回语句，for 循环语句，while 循环语句，break 语句，continue 语句，if 语句，赋值语句，声明语句以及赋值和声明结合起来的语句，支持 C++常见的语句逻辑。该文法存放在 grammar_2.txt 文件中。

$S 代表开始，A 代表程序，B 代表函数，@代表空

S->A

A->BA

A->@

$t 代表数据类型，具体包括 int double float string long char short void

$i 代表标识符 identifier

$C 代表参数，D 代表完整的声明,E 代表声明的一部分，D 接在 E 后面

B->ti(C){D}

C->ti

C->ti,C

C->@

D->ED

D->@

$F 代表返回声明，G 代表 for 循环的声明，H 代表 while 循环声明，I 代表 break 声明，J 代表 continue 声明

$K 代表 if 声明，L 代表赋值声明，Q 代表声明语句

E->F

E->G

E->H

E->I

E->J

E->K

E->L

E->Q

Q->ti;

$R 代表赋值，也就是等号=或 operator=，P 代表值的表示，o 代表计算运算符

$取 tiRP 的时候说明将声明与赋值结合了起来

L->ti=N;

L->iRN;

R->=

R->o=

$P->(P)

$N 代表项目

$P->NoP

$P->N

$r 代表 return

$n 代表数字 number，包括科学计数法与复数

F->rN;

$F->rP;

N->n

N->i

$M 代表 for 循环的条件，0 代表具体的条件

$f 代表 for

G->f(M){D}

$Y 代表没有;的赋值语句

M->LO;Y

M->QO;Y

Y->iRN

$p 代表有判断作用的运算符，比如<,<=,==,>,>=,!=

O->NpN

$w 代表 while

H->w(O){D}

$b 代表 break

I->b;

$c 代表 continue

J->c;

$z 代表 if

K->z(0){D}

3.3 读取文本文件中的文法与 token 表

首先要把存放在 grammar_2 中的二型文法与存放在 token 文件中的输入串取出来。使用 ifstream 打开二者，二型文法直接存放在一个 string 类型的 vector 中，随用随取即可。这里要注意别忘了把 T->S 放进去，直接在读取的时候完成拓广。

```
string nows="T->S";  //当前字符串，初始化为T->S，其中T为引入新的开始符号，S为原本开始符号
gra.insert(gra.begin(),nows);
while(getline(myfile,nows)){  //按行读取
  if(nows[0]=='$') continue;  //美元符代表注释，跳过
  gra.insert(gra.begin()+gra.size(),nows);  //把当前的字符串放入gra中
}
```

而输入串的读取则要麻烦一些，由于 token 表传进来的是各种类别，因此需要把每个不同的类型以一个固定的长度为 1 的 string（其实 char 更好，一开始写数据结构的时候没有想到）存放起来。

```
  if(Type=="identifier") tok.insert(tok.begin()+tok.size(),"i");  //标识符为i
  if(Type=="number") tok.insert(tok.begin()+tok.size(),"n");  //数字为n，包括科学计数法等
  if(Type=="keyword"){  //关键字要分成多种来处理
    if(Value=="main") tok.insert(tok.begin()+tok.size(),"i"); //由于main函数的特殊性，识别为i
    else if(Value=="return") tok.insert(tok.begin()+tok.size(),"r");  //return为r
    else if(Value=="for") tok.insert(tok.begin()+tok.size(),"f"); //for为f
    else if(Value=="while") tok.insert(tok.begin()+tok.size(),"w"); //while为w
    else if(Value=="break") tok.insert(tok.begin()+tok.size(),"b"); //break为b
    else if(Value=="continue") tok.insert(tok.begin()+tok.size(),"c");  //continue为c
    else if(Value=="if") tok.insert(tok.begin()+tok.size(),"z");  //if为z
    else if(Value=="int"||Value=="double"||Value=="float"||Value=="string"||Value=="long"||Valu
    else tok.insert(tok.begin()+tok.size(),"other_keyword");  //其它关键字，分类原理与上边一致，本
  }
  if(Type=="operator"){ //运算符也要分类
    if(Value=="=") tok.insert(tok.begin()+tok.size(),"=");  //=单独拿出来
    else if(Value=="<="||Value=="=="||Value==">="||Value==">"||Value=="<"||Value=="!=") tok.ins
    else tok.insert(tok.begin()+tok.size(),"o");  //其余的直接视作p
  }
  if(Type=="limiter") tok.insert(tok.begin()+tok.size(),Value); //界符直接试作自己
```

基本遵循以上原则，标识符统一视作 i，常量统一视作 n，限定符由于本来长度就是 1 且几乎全都有独特的用处，全部视作自己。

关键字要分情况，return 视作 r，for 视作 f，while 视作 w，break

视作 b，continue 视作 c，if 视作 z，int、double 等代表数据类型的的统一视作 t（即 type），其它的关键字在本次项目的语法分析过程中并不会用到，防止程序出错视作“other_keyword”。

运算符要分为两类，等号=要单独拿出来，>=、<=等返回 true 或 false 的运算符统一视作 p，其余运算符视作 o。

3.4 求 First 集

在本项目中并不为每个非终结符的 First 单独创造一个数据结构，而是将求 First 集的功能封装在一个函数中，随用随取。使用这种处理的主要原因是在后续计算中不只需要使用非终结符的 First 集，也需要求一些非终结符与终结符组合的字符串的 First 集，这种情况下提前封装好求 First 集的函数不仅节省代码量，也更加灵活。

求 First 集的过程基本遵循以下原则：

终结符的 First 集合就是它自己。

对于非终结符，连续使用以下规则，直到 First 集不再增大为止：

假如有类似于 A->a……的表达式，就把 a 假如 A 的 First 集，假如 a 等于空，把空（也就是@）放入 A 的 First 集中。

假如有类似于 A->B，就把 B 的 First 集加入 A 的 First 集中，包括空。

假如有类似于 A->Ba，就把 B 的 First 集中非空的元素加入 A 的 First 集中，随后把 a 也加入 A 的 First 集中。

假如有类似于 A->BC，就把 B 的 First 集中非空的元素加入 A 的 First 集中，随后把 First（C）也加入 A 的 First 集中。

遵循以上原则，本项目构造的求单个字符的 First 集的函数部分代码如下：

```
string t="";
if(X<'A'||X>'Z'){ //假如终结符开头，即a……，那么First集就是它自己，即a
  t+=X;
  return t;
}
```

```
for(int i=0;i<gra.size();i++){  //遍历每一个表达式
  if(gra[i][0]==X){ //只有X->…… 才会考虑
    if(gra[i][3]==X) continue;  //假如X->X……之类的则跳过到下一表达式
    if(gra[i][3]>='A'&&gra[i][3]<='Z'){ //X->Y……，后跟非终结符，由于1与2是箭头所以判断3
      string tem=getFirst(gra[i][3]); //求First[Y]
      for(int i=0;i<tem.length();i++){
        if(tem[i]!='@') t+=tem[i]; //去掉Y中的空
      }
      bool pan=false; //判断First[Y]中是否有空
      for(int i=0;i<tem.length();i++){
        if(tem[i]=='@'){  //找到空，pan为true
          pan=true;
          break;
        }
        else continue;
      }
```

```
if(pan){  //只有当Y的First集中有空的时候才考虑以下规则
  if(gra[i].length()==4){ //若X->Y，则将 ε 加入First(X)中
    t+='@';
    continue;
  }
  else if(gra[i][4]>='A'&&gra[i][4]<='Z'){  //若X->YZ…，则将First(Z)加入First(X)中
    string temp="";
    temp=getFirst(gra[i][4]);
    for(int k=0;k<temp.length();k++) t+=temp[k];
  }
  else{ //若X->Ya…，则将 a 加入First(X)中
    t+=gra[i][4];
  }
}
else continue;  //否则看下一个表达式
```

值得注意的是在运行完以上函数后一定要把求得的 First 集排序并去除多余元素，具体实现位于工具函数之中，实现较为简单，代码中有详细注释，不再赘述。

除此之外还应编写求字符串的 First 集的函数，实现逻辑和求 A->BC 类型的时候差别不大，不再赘述，具体实现如下：

```
if(s[0]>='A'&&s[0]<='Z'){ //字符串首个字符是非终结符
    string tem1=getFirst(s[0]);  //s[0]的first集
    string tem2=getSFirst(rearS(s)); //s去头的first集
    bool pan=false; //判断s[0]的first集中有没有空
    for(int i=0;i<tem1.length();i++){
        if(tem1[i]!='@') t+=tem1[i];
        else pan=true;
    }
    if(pan){  //假如pan为true，说明s[0]的first集中有@
        for(int i=0;i<tem2.length();i++) t+=tem2[i];  //把去头的s的first集放进去
    }
}
else{ //否则说明字符串首个字符是终结符
    t+=s[0];  //直接返回该终结符
}
```

3.5 计算每个闭包的项目集以及 GO 函数

在这一步首先要生成闭包 IO 的第一个项目 T->·S，#，在实现的时候为了方便本项目将·用`代替，将,用$代替。

随后根据已知项目，计算 getClosure（I）的项目。I 的任何项目都属于 getClosure(I)。若项目 A->α·BC,a 属于 getClosure(I)，且 B->D 是一个产生式，那么对于 First(Ca)中的每个终结符 b，如果 B->·D，b 原来不在 getClosure(I)中，则把它加进去。一直循环以上步骤，直到 getClosure(I)再也不增加为止。

以上为构造原则，过于抽象复杂，在实际实现过程中，本项目更多参考了清华大学出版社《编译原理》第 3 版 147 页的例题中 LR（1）项目集的构造过程，相比于逻辑关系，在实现过程中依靠对于转换图去理解的成分更多。

也因此在本项目中构造每个项目集和把每个项目集用边连成项目集族是分成不同函数进行的。在生成项目集时基本上根据以上逻辑。该函数名为 getClosure，部分实现代码如下：

```
for(int j=0;j<gra.size();j++){  //遍历二型文法中的每一个表达式，看有没有箭头左侧是nc的
  if(gra[j][0]!=nc) continue; //假如gra[j]的第一个字符不等于nc则跳过，看下一条表达式
  else{ //否则有箭头左侧是nc的，需要加入项目集
    string temt="";
    for(int k=0;k<gra[j].length();k++){
      if(k==3) temt+='`'; //在箭头后边加入一个点
      temt+=gra[j][k];
    }
    temt+='$';  //末尾加上逗号
    for(int k=0;k<newr.length();k++) temt+=newr[k]; //把newr加入末尾
    ii.insert(ii.begin()+ii.size(),temt); //把新的表达式放入项目集
  }
```

另一个函数名为 putI，作用判断上一个函数生成的项目集是否已经在项目集族 I 中，若不在则放入，否则不放。在放完项目集后该函数还将生成与该函数相关的边，并按照这些边生成相关的项目集，也像刚才一样判断是否已在 I 中，若不在则放入。部分实现代码如下：

```
int ni=has_existedi(nowi);  //nowi是否已在I中，不在为-1，在为对应序号
if(ni==-1){ //nowi假如并不在I中
  ni=I.size();  //把ni从-1改成对应序号
  I.insert(I.begin()+I.size(),nowi);  //把nowi对应的项目集放进去
}
```

```
int niii=has_existedi(iii); //niii储存iii在I中位置，不在I中则为-1
if(niii==-1){ //假如iii不在I中
  niii=I.size();  //令niii为当前I的项目个数
  I.insert(I.begin()+I.size(),iii); //随后把iii放进去
}
if(!has_existed_edge(ni,nodec[i],niii)){  //假如现在边集合中没有这条
  bian[ne].zuo=ni;  //增加一条边，左边为ni
  bian[ne].zhong=nodec[i];  //中间为当前字符
  bian[ne].yo=niii; //右边为niii
  ne++; //别忘了把边数加一
}
```

3.6 计算 ACTION 表与 GOTO 表

为便于管理，本项目计算的 ACTION 表与 GOTO 表同样不单独设置数据结构，也使用由函数来求取的方式。函数名为 AG，两个输入分别为 int z 与 char c，z 代表状态号，c 代表终结符或非终结符，c 是终结符则求 ACTION 表，否则求 GOTO 表。

求两表的基本逻辑如下：

若 A->B•aC,b 属于 Ik，且 GO(Ik,a)==Ij，则进行移进操作，在表上的表示方式为 Sj。

若 A->B •,a 属于 Ik，则进行规约操作，在表上的表示方式为 rj，其中 j 为 A->B 对应的表达式号。

若 T->S •,#属于 Ik，则表示识别成功，在表上表达的方式是 acc，位置对应在#所在的那一列。

以上三种情况全在 ACTION 表中。

若 GO(Ik,A)=Ij,其中 A 属于非终结符，则得到 GOTO[k,A]=j，表示转入 j 状态。

其余状态全表示出错，在 ACITON&GOTO 表中本来用 wrong 表示，考虑到排版问题，用三个空格来表示。

实现的部分代码如下：

```
if(c>='A'&&c<='Z'){ //假如c是非终结符，走GOTO
    for(int i=0;i<ne;i++){
        if(bian[i].zuo==z&&bian[i].zhong==c){ //找到对应的边
            int t=bian[i].yo;
            string tt="";
            tt+=to_string(t);
            return tt;
        }
    }
}
```

```
    for(int i=0;i<ne;i++){   //首先判断有没有从Iz出发，中间是c的边
        if(bian[i].zuo==z&&bian[i].zhong==c){ //找到了
            string S="S";
            int Sy=bian[i].yo;
            S+=to_string(Sy);
            return S; //返回Si
        }
        else continue;      //否则继续看下一条边
    } //for循环结束还没return说明不是移进
```

```
if(I[z][i][dian+1]=='$'){ //只有点后面就是逗号的才考虑
  string rS;
  if(I[z][i][dian-1]=='S'){ //假如点前面就是S，返回acc
    rS="acc";
  }
  else{ //否则返回rl
    rS="r";
    string toolS="";
    for(int l=0;l<dian;l++){  //用toolS存放当前表达式点前面的部分
      toolS+=I[z][i][l];
    }
    for(int l=0;l<gra.size();l++){
      if(toolS==gra[l]){
        rS+=to_string(l);
        break;
      }
    }
  }
```

```
string wr="   ";  //代表wrong
return wr;
```

3.7 表示 LR（1）分析表

基本通过 3.6 中叙述的逻辑进行构造，实现较为繁琐，但更多是出于排版与表示问题，重点在 3，6 中的 AG 函数上，不在赘述。

3.8 对输入串进行分析，并输出结果

首先在输入串的末尾加入一个#，随后创建一个状态栈，初始化只有一个元素 0，一个符号栈，初始化只有一个元素#，在实际实现的时候由于需要展示分析过程，栈过于不便，使用 vector 来代替。

根据状态栈栈顶 i 与输入串串头 a，计算 AG（i，a），加入等于 acc 则分析成功；假如等于 Sj，则进行移进操作，j 进入状态栈，并从输入串读取一个字符进入符号栈；假如等于 rj，就用第 j 个产生式 X->Y 来规约，且 GOTO（l，x）入栈，其中 l 为状态栈出栈 Y 的位数次后的栈顶；假如等于三个空格，正常此时应该判错，可是由于本项目使用了@来代替空，所以此时先判断 AG（i，@）是不是空格，是

则判错，否则使用 AG（i，@）的对应结果来规约。

部分实现代码如下：

```
string dang=AG(zhuang,shu); //对应表中的值
if(dang=="acc"){  //若为acc则分析成功
    cout<<"成了，牛逼"<<endl<<"SUCCESS!!!!!";
    return;
}
```

假如报错，此时应该取符号栈顶，先找到该出错符号在输入串中出现的次数 i，然后找出该符号在输入串中出现第 i 次时的位置，再从 token 表中寻找该序号对应的行号，这样就可以找到出错位置并顺势分析出错原因了。

```
if(Aj=="   "){
  cout<<"寄了"<<endl<<"FAILURE";  //假如不是遇到@，分析出错
  string tokenSS="";
  for(int h=0;h<tok.size();h++) tokenSS+=tok[h]; //将tok容器的内容取出来
  int chu=0;  //记录输入串中出错符号出现次数
  for(int h=0;h<tokenSS.length();h++){
    if(Symb[Symb.size()-1]==tokenSS[h]) chu++;
  }
  for(int h=0;h<tokenSS.length();h++){
    if(Symb[Symb.size()-1]==tokenSS[h]){
      if(chu==1){
        cout<<endl<<"出错行号在"<<hangh[h]<<"行，请检查";
        break;
      }
```

其它情况的实现也与以上两种类似。

```
else if(dang[0]=='S'){  //移进
    string Sj="";
    for(int l=1;l<dang.length();l++) Sj+=dang[l]; //取得S后边的部分，用来转成int
    int sj=atoi(Sj.c_str());  //string转int
    Sta.insert(Sta.begin()+Sta.size(),sj); //sj入状态栈
    Symb.insert(Symb.begin()+Symb.size(),shu); //串头入符号栈
    tokenS=rearS(tokenS); //输入串去掉串头
    cout<<"ACTION("<<zhuang<<','<<shu<<")="<<dang<<"，移进操作";
}
```

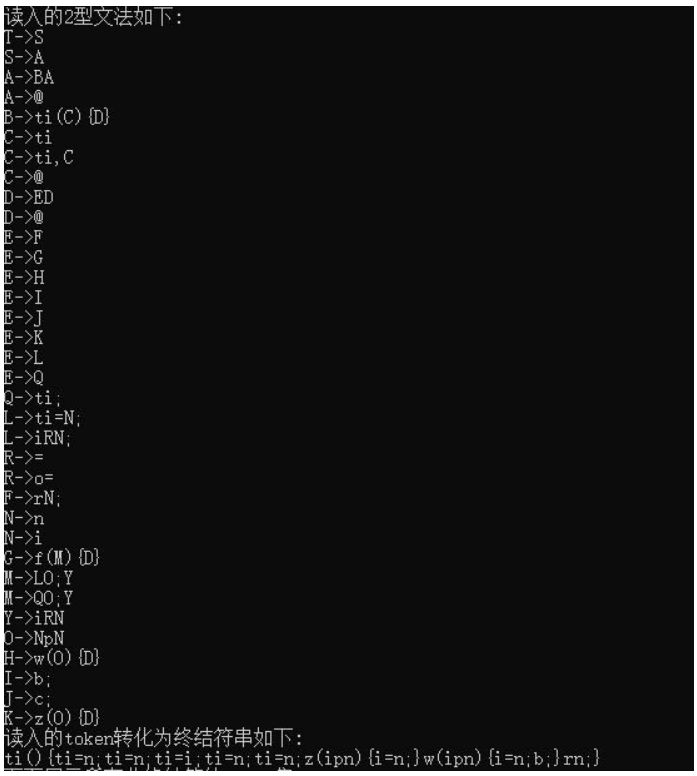
```
else{ //规约,首字符是r
    string rj="";
    for(int l=1;l<dang.length();l++) rj+=dang[l]; //取得r后边的部分，用来转成int
    int nr=atoi(rj.c_str());  //string转int
    char lef=gra[nr][0];  //rj表达式左部
    string rig="";  //rj表达式右部
    for(int l=3;l<gra[nr].length();l++) rig+=gra[nr][l];  //取箭头右边的部分
    int nrig=rig.length();
    for(int l=0;l<nrig;l++) Symb.pop_back();
    Symb.insert(Symb.begin()+Symb.size(),lef); //进行规约
    for(int l=0;l<nrig;l++) Sta.pop_back();  //状态栈出栈
    int kkk=Sta[Sta.size()-1];  //取状态栈栈顶
    string GOTOkX=AG(kkk,lef);  //求GOTO
    int gotokx=atoi(GOTOkX.c_str());  //转int
    Sta.insert(Sta.begin()+Sta.size(),gotokx); //GOTO入栈
    cout<<"GOTO("<<zhuang<<','<<shu<<")="<<dang<<", 规约操作";
}
```

3.9 运行实例和截图

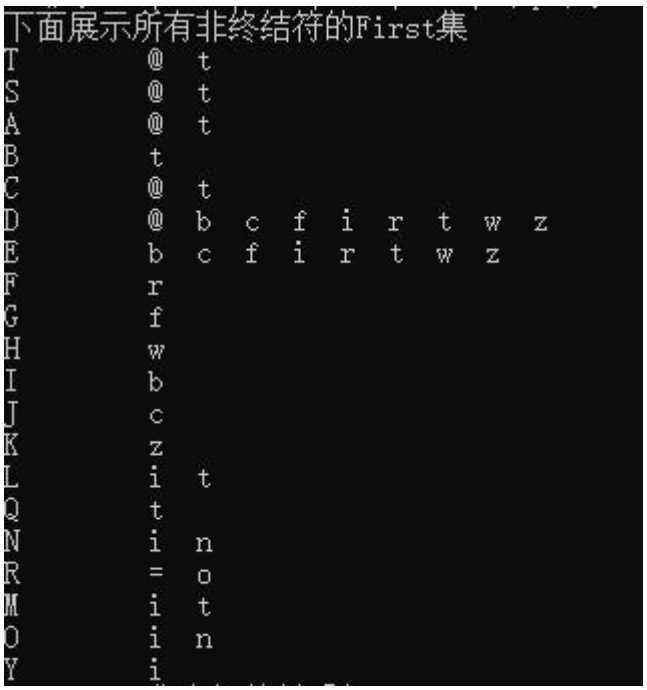
输入的 2 型文法部分截图如下：

读入的 token 表在词法分析过程中已经展示过，不再赘述。

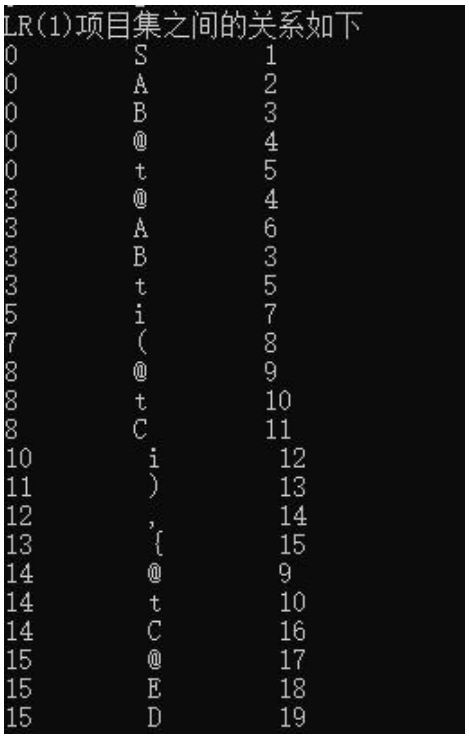
首先展示读入的文法表达式与 Token 表对应的输入串：



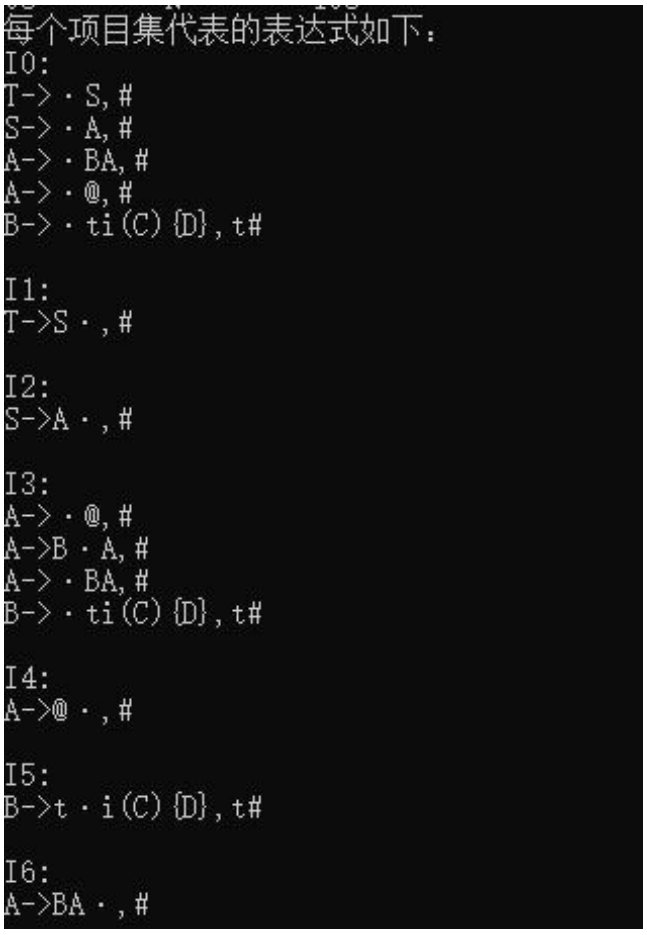
随后展示所有非终结符的 First 集



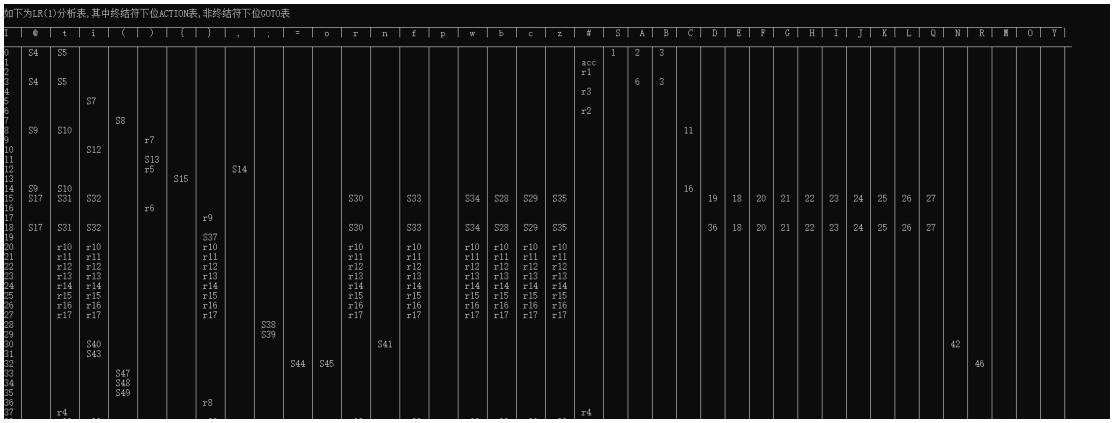
随后展示生成的项目集族种各个项目集之间的关系，由于内容过多，只展示部分。



在下面会介绍每个项目集中的式子。



随后展示 LR（1）分析表，内容过多只截图部分。



随后展示分析过程，内容过多，只展示开头和结尾。

4、心得体会

本次课程设计不仅是对我编程能力的一次检验、对我编译原理知识的一次巩固，更锻炼了我的项目能力，让我过去所学更加通透。在这次开发过程中让我印象尤为深刻的是对于函数封装的使用，本来无比巨大的项目在化为多个小函数之后是如此的条理清晰，这不仅极大地降低了我的开发难度，更减少了我 debug 的时间成本。

总体来说，这次课程设计的开发让我受益良多，本次实现的词法分析器与语法分析器仍有继续改进的空间，等日后我在有空的时候会对本项目继续完善与改进。