语法分析器

一、实验内容

自实现语法分析器：

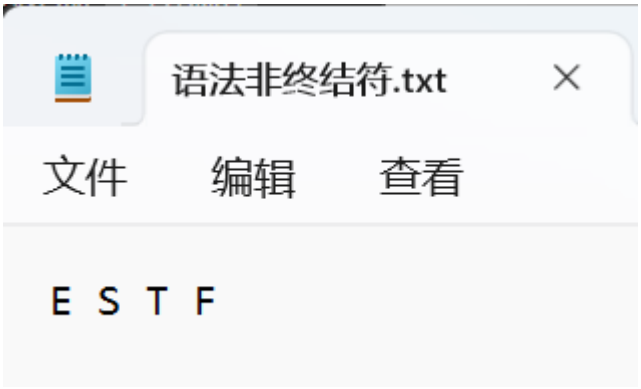
1. 程序输入：终结符集合，非终结符集合，文法，待分析的语句
2. 语法分析算法：LR(1)
3. 程序实现：
   - **根据输入的终结符集合、非终结符集合和文法构造LR(1)的GOTO图**
   - **根据LR(1)GOTO图自动化的构造LR(1)语法分析表**
   - **根据语法分析表分析输入的语句，得出语法分析动作表**
4. 输出
   - LR(1)文法GOTO图
   - LR(1)语法分析表
   - 语法分析动作表
   - 错误报错

二、程序输入输出展示

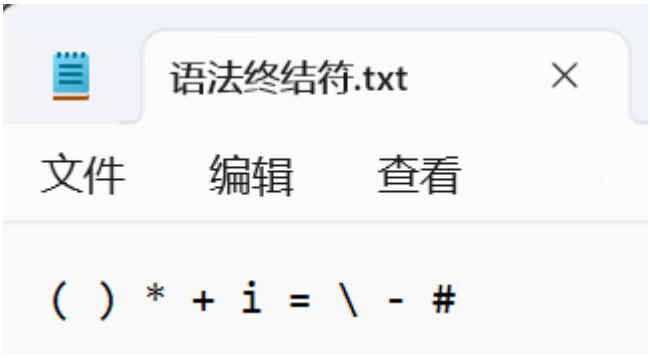
2.1 输入展示

以下输入均以.txt文件的格式读入程序

1. 语法非终结符



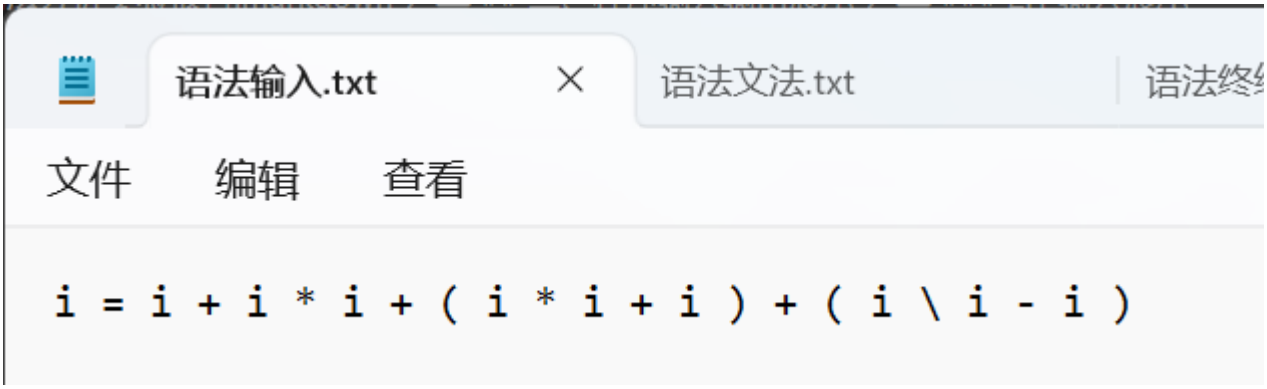
2. 语法终结符



3. 文法

> 包括对加减乘除、等号、括号、变量的分析

4. 语句输入



2.2 输出展示

GOTO图：以下展示每个状态，状态之间的转换关系和读入字符后续说明

F:\编译原理\labs\Syntactic-Analyzer\x64\Debug\Syntactic-Analyzer.exe

```
0
E->·S,  #
S->·i=S,  # + -
S->·S+T,  # + -
S->·S-T,  # + -
S->·T,  # + -
T->·T*F,  # + - * \
T->·T\F,  # + - * \
T->·F,  # + - * \
F->·(S),  # + - * \
F->·i,  # + - * \
------------------------

1
E->S·,  #
S->S·+T,  # + -
S->S·-T,  # + -
------------------------

2
S->i·=S,  # + -
F->i·,  # + - * \
------------------------

3
S->T·,  # + -
T->T·*F,  # + - * \
T->T·\F,  # + - * \
------------------------

4
T->F·,  # + - * \
------------------------

5
F->(·S),  # + - * \
S->·i=S,  ) + -
S->·S+T,  ) + -
S->·S-T,  ) + -
S->·T,  ) + -
T->·T*F,  ) + - * \
T->·T\F,  ) + - * \
T->·F,  ) + - * \
F->·(S),  ) + - * \
F->·i,  ) + - * \
```

```
6
S->S+·T,  # + -
T->·T*F,  # + - * \
T->·T\F,  # + - * \
T->·F,  # + - * \
F->·(S),  # + - * \
F->·i,  # + - * \
------------------------

7
S->S-·T,  # + -
T->·T*F,  # + - * \
T->·T\F,  # + - * \
T->·F,  # + - * \
F->·(S),  # + - * \
F->·i,  # + - * \
------------------------

8
S->i=·S,  # + -
S->·i=S,  # + -
S->·S+T,  # + -
S->·S-T,  # + -
S->·T,  # + -
T->·T*F,  # + - * \
T->·T\F,  # + - * \
T->·F,  # + - * \
F->·(S),  # + - * \
F->·i,  # + - * \
------------------------

9
T->T*·F,  # + - * \
F->·(S),  # + - * \
F->·i,  # + - * \
------------------------

10
T->T\·F,  # + - * \
F->·(S),  # + - * \
F->·i,  # + - * \
------------------------
```

```
--------------------------------------
11
F->(S •),  # + - * \
S->S • +T,  ) + -
S->S • -T,  ) + -
--------------------------------------
12
S->i • =S,  ) + -
F->i •,  ) + - * \
--------------------------------------
13
S->T •,  ) + -
T->T • *F,  ) + - * \
T->T • \F,  ) + - * \
--------------------------------------
14
T->F •,  ) + - * \
--------------------------------------
15
F->( • S),  ) + - * \
S-> • i=S,  ) + -
S-> • S+T,  ) + -
S-> • S-T,  ) + -
S-> • T,  ) + -
T-> • T*F,  ) + - * \
T-> • T\F,  ) + - * \
T-> • F,  ) + - * \
F-> • (S),  ) + - * \
F-> • i,  ) + - * \
--------------------------------------
16
S->S+T •,  # + -
T->T • *F,  # + - * \
T->T • \F,  # + - * \
--------------------------------------
17
F->i •,  # + - * \
--------------------------------------
```

```
--------------------------------------
18
S->S-T •,  # + -
T->T • *F,  # + - * \
T->T • \F,  # + - * \
--------------------------------------
19
S->i=S •,  # + -
S->S • +T,  # + -
S->S • -T,  # + -
--------------------------------------
20
T->T*F •,  # + - * \
--------------------------------------
21
T->T\F •,  # + - * \
--------------------------------------
22
F->(S) •,  # + - * \
--------------------------------------
23
S->S+ • T,  ) + -
T-> • T*F,  ) + - * \
T-> • T\F,  ) + - * \
T-> • F,  ) + - * \
F-> • (S),  ) + - * \
F-> • i,  ) + - * \
--------------------------------------
24
S->S- • T,  ) + -
T-> • T*F,  ) + - * \
T-> • T\F,  ) + - * \
T-> • F,  ) + - * \
F-> • (S),  ) + - * \
F-> • i,  ) + - * \
--------------------------------------
```

```
------------------------------
25
S->i=•S,  ) + -
S->•i=S,  ) + -
S->•S+T,  ) + -
S->•S-T,  ) + -
S->•T,  ) + -
T->•T*F,  ) + - * \
T->•T\F,  ) + - * \
T->•F,  ) + - * \
F->•(S),  ) + - * \
F->•i,  ) + - * \
------------------------
26
T->T*•F,  ) + - * \
F->•(S),  ) + - * \
F->•i,  ) + - * \
------------------------
27
T->T\•F,  ) + - * \
F->•(S),  ) + - * \
F->•i,  ) + - * \
------------------------
28
F->(S•),  ) + - * \
S->S•+T,  ) + -
S->S•-T,  ) + -
------------------------
29
S->S+T•,  ) + -
T->T•*F,  ) + - * \
T->T•\F,  ) + - * \
------------------------
30
F->i•,  ) + - * \
------------------------
```

```
31
S->S-T•,  ) + -
T->T•*F,  ) + - * \
T->T•\F,  ) + - * \
------------------------
32
S->i=S•,  ) + -
S->S•+T,  ) + -
S->S•-T,  ) + -
------------------------
33
T->T*F•,  ) + - * \
------------------------
34
T->T\F•,  ) + - * \
------------------------
35
F->(S)•,  ) + - * \
```

LR(1)语法分析表

LR(1)分析表:

	(	)	*	+	i	=	\	-	#	E	S	T	F
0	S5				S2						1	3	4
1				S6				S7	acc				
2			r9	r9		S8	r9	r9	r9				
3			S9	r4			S10	r4	r4				
4			r7	r7			r7	r7	r7				
5	S15				S12						11	13	14
6	S5				S17							16	4
7	S5				S17							18	4
8	S5				S2						19	3	4
9	S5				S17								20
10	S5				S17								21
11		S22		S23				S24					
12		r9	r9	r9		S25	r9	r9					
13		r4	S26	r4			S27	r4					
14		r7	r7	r7			r7	r7					
15	S15				S12						28	13	14
16			S9	r2			S10	r2	r2				
17			r9	r9			r9	r9	r9				
18			S9	r3			S10	r3	r3				
19				S6				S7	acc				
20			r5	r5			r5	r5	r5				
21			r6	r6			r6	r6	r6				
22			r8	r8			r8	r8	r8				
23	S15				S30							29	14
24	S15				S30							31	14
25	S15				S12						32	13	14
26	S15				S30								33
27	S15				S30								34
28		S35		S23				S24					
29		r2	S26	r2			S27	r2					
30		r9	r9	r9			r9	r9					
31		r3	S26	r3			S27	r3					
32				S23				S24	acc				
33		r5	r5	r5			r5	r5					
34		r6	r6	r6			r6	r6					
35		r8	r8	r8			r8	r8					

动作分析表:

分析过程为:

步骤	状态栈	符号栈	输入符	动作
1	0	#	i=i+i*i+(i*i+i)+(i\i-i)#	S2
2	0 2	#i	=i+i*i+(i*i+i)+(i\i-i)#	S8
3	0 2 8	#i=	i+i*i+(i*i+i)+(i\i-i)#	S2
4	0 2 8 2	#i=i	+i*i+(i*i+i)+(i\i-i)#	r9
5	0 2 8 4	#i=F	+i*i+(i*i+i)+(i\i-i)#	r7
6	0 2 8 3	#i=T	+i*i+(i*i+i)+(i\i-i)#	r4
7	0 2 8 19	#i=S	+i*i+(i*i+i)+(i\i-i)#	S6
8	0 2 8 19 6	#i=S+	i*i+(i*i+i)+(i\i-i)#	S17
9	0 2 8 19 6 17	#i=S+i	*i+(i*i+i)+(i\i-i)#	r9
10	0 2 8 19 6 4	#i=S+F	*i+(i*i+i)+(i\i-i)#	r7
11	0 2 8 19 6 16	#i=S+T	*i+(i*i+i)+(i\i-i)#	S9
12	0 2 8 19 6 16 9	#i=S+T*	i+(i*i+i)+(i\i-i)#	S17
13	0 2 8 19 6 16 9 17	#i=S+T*i	+(i*i+i)+(i\i-i)#	r9
14	0 2 8 19 6 16 9 20	#i=S+T*F	+(i*i+i)+(i\i-i)#	r5
15	0 2 8 19 6 16	#i=S+T	+(i*i+i)+(i\i-i)#	r2
16	0 2 8 19	#i=S	+(i*i+i)+(i\i-i)#	S6
17	0 2 8 19 6	#i=S+	(i*i+i)+(i\i-i)#	S5
18	0 2 8 19 6 5	#i=S+(	i*i+i)+(i\i-i)#	S12

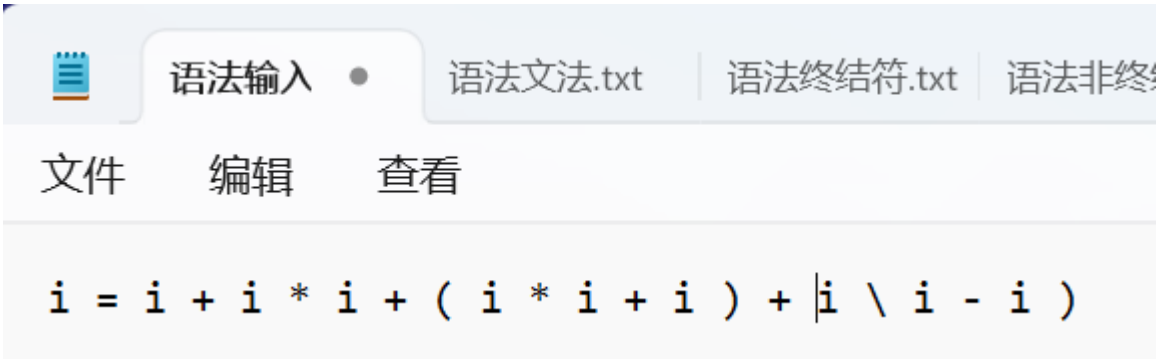
```
19          0 2 8 19 6 5 12          #i=S+(i          *i+i)+(i\i-i)#     r9
20          0 2 8 19 6 5 14          #i=S+(F          *i+i)+(i\i-i)#     r7
21          0 2 8 19 6 5 13          #i=S+(T          *i+i)+(i\i-i)#     S26
22       0 2 8 19 6 5 13 26         #i=S+(T*           i+i)+(i\i-i)#     S30
23    0 2 8 19 6 5 13 26 30        #i=S+(T*i            +i)+(i\i-i)#     r9
24    0 2 8 19 6 5 13 26 33        #i=S+(T*F            +i)+(i\i-i)#     r5
25          0 2 8 19 6 5 13          #i=S+(T            +i)+(i\i-i)#     r4
26          0 2 8 19 6 5 11          #i=S+(S            +i)+(i\i-i)#     S23
27       0 2 8 19 6 5 11 23         #i=S+(S+             i)+(i\i-i)#     S30
28    0 2 8 19 6 5 11 23 30        #i=S+(S+i             )+(i\i-i)#     r9
29    0 2 8 19 6 5 11 23 14        #i=S+(S+F             )+(i\i-i)#     r7
30    0 2 8 19 6 5 11 23 29        #i=S+(S+T             )+(i\i-i)#     r2
31          0 2 8 19 6 5 11          #i=S+(S             )+(i\i-i)#     S22
32       0 2 8 19 6 5 11 22         #i=S+(S)              +(i\i-i)#     r8
33             0 2 8 19 6 4            #i=S+F            +(i\i-i)#     r7
34           0 2 8 19 6 16             #i=S+T            +(i\i-i)#     r2
35                0 2 8 19                #i=S           +(i\i-i)#     S6
36             0 2 8 19 6               #i=S+            (i\i-i)#     S5
37           0 2 8 19 6 5               #i=S+(            i\i-i)#     S12
38          0 2 8 19 6 5 12           #i=S+(i              \i-i)#     r9
39          0 2 8 19 6 5 14           #i=S+(F              \i-i)#     r7
40          0 2 8 19 6 5 13           #i=S+(T              \i-i)#     S27
41       0 2 8 19 6 5 13 27          #i=S+(T\               i-i)#     S30
```

```
42    0 2 8 19 6 5 13 27 30        #i=S+(T\i                -i)#     r9
43    0 2 8 19 6 5 13 27 34        #i=S+(T\F                -i)#     r6
44          0 2 8 19 6 5 13          #i=S+(T                -i)#     r4
45          0 2 8 19 6 5 11          #i=S+(S                -i)#     S24
46       0 2 8 19 6 5 11 24         #i=S+(S-                 i)#     S30
47    0 2 8 19 6 5 11 24 30        #i=S+(S-i                 )#     r9
48    0 2 8 19 6 5 11 24 14        #i=S+(S-F                 )#     r7
49    0 2 8 19 6 5 11 24 31        #i=S+(S-T                 )#     r3
50          0 2 8 19 6 5 11          #i=S+(S                 )#     S22
51       0 2 8 19 6 5 11 22         #i=S+(S)                  #     r8
52             0 2 8 19 6 4            #i=S+F                 #     r7
53           0 2 8 19 6 16             #i=S+T                 #     r2
54                0 2 8 19                #i=S                #     acc
语法识别成功
```

2.3 报错提示

当语法分析出错时，会有相应的报错

1. 输入句子如下时：



报错：

```
37          0 2 8 19 6 17          #i=S+i                \i-i)#    r9
38           0 2 8 19 6 4          #i=S+F               \i-i)#    r7
39          0 2 8 19 6 16          #i=S+T               \i-i)#    S10
40       0 2 8 19 6 16 10         #i=S+T\                i-i)#    S17
41    0 2 8 19 6 16 10 17        #i=S+T\i                 -i)#    r9
42    0 2 8 19 6 16 10 21        #i=S+T\F                 -i)#    r6
43          0 2 8 19 6 16          #i=S+T                 -i)#    r2
44             0 2 8 19             #i=S                 -i)#    S7
45           0 2 8 19 7             #i=S-                  i)#    S17
46          0 2 8 19 7 17           #i=S-i                  )#error，意外的右括号
请按任意键继续. . .
```

2. 输入句子如下时：

报错：

```
分析过程为:

步骤                状态栈          符号栈                  输入符     动作

  1                   0               #    i=+i*i+(i*i+i)+(i\i-i)#    S2
  2                 0 2              #i     =+i*i+(i*i+i)+(i\i-i)#    S8
  3               0 2 8             #i=      +i*i+(i*i+i)+(i\i-i)#error，意外的+
请按任意键继续. . .
```

三、重要的数据结构和变量

文法结构体

```
struct wf          //定义文法
{                                  //e.g. 文法(0)E->S -- left = E || right = S || num = 0 || leng
        string left;  //文法左部
        string right; //文法右部
        int num;      //文法序号
        int length;   //归约长度
};
```

GOTO图中一条推导式的结构体

```
struct pro              //一条文法式子+圆点后会跟的终结符
{                                           //e.g. I0包括E->·S,# || num = 0 || position = 0 || sea
        int num;            //第几条文法
        int position;       //圆点位置
        vector<char>  search;  //原点后会跟的终结符
};
```

GOTO图中一个状态的结构体

```
struct itemset      //pro的集合，一个状态（里面有很多个pro，GOTO图中框起来的东西）
{
        string name;    //状态名称
        vector<pro> t;  //定义状态中的项目t
};
```

语法分析表

语法分析表结构 `map<string,string> m`

- m的key是初始状态数+读入字符，value是对应的操作。
- 比如（'0S','1'）表示S_0状态读入字符S后跳转到S_1状态，（'A+','r9'）表示状态S_{17}读入字符+对应的动作是r9

  > （这里A表示17，我是使用字符对应ASCII相对于'0'的偏移量来表示数字的，'A'-'0'=17，这样可以保证我每个状态的序号都是一位字符，便于后续操作）

四、算法思想

（一）求First集的函数实现

- 函数输入：文法，终结符集，被求First集的字符串X
- 算法思想：
  - 如果X的第一个字符为终结符，则FIRST(X)={X[0]}。
  - 如果X的第一个字符是非终结符，且X形如a...,则FIRST(X)={a}。
  - 如果X的第一个字符是非终结符，且X形如ABCdEF...(A、B、C均属于非终结符且包含空串，d为终结符)需要把FIRST(A)、FIRST(B)、FIRST(C)、FIRST(d)加入到FIRST(X)中。
  - 如果X经过一步或多步推导出空字符，将空串加入FIRST(X)。

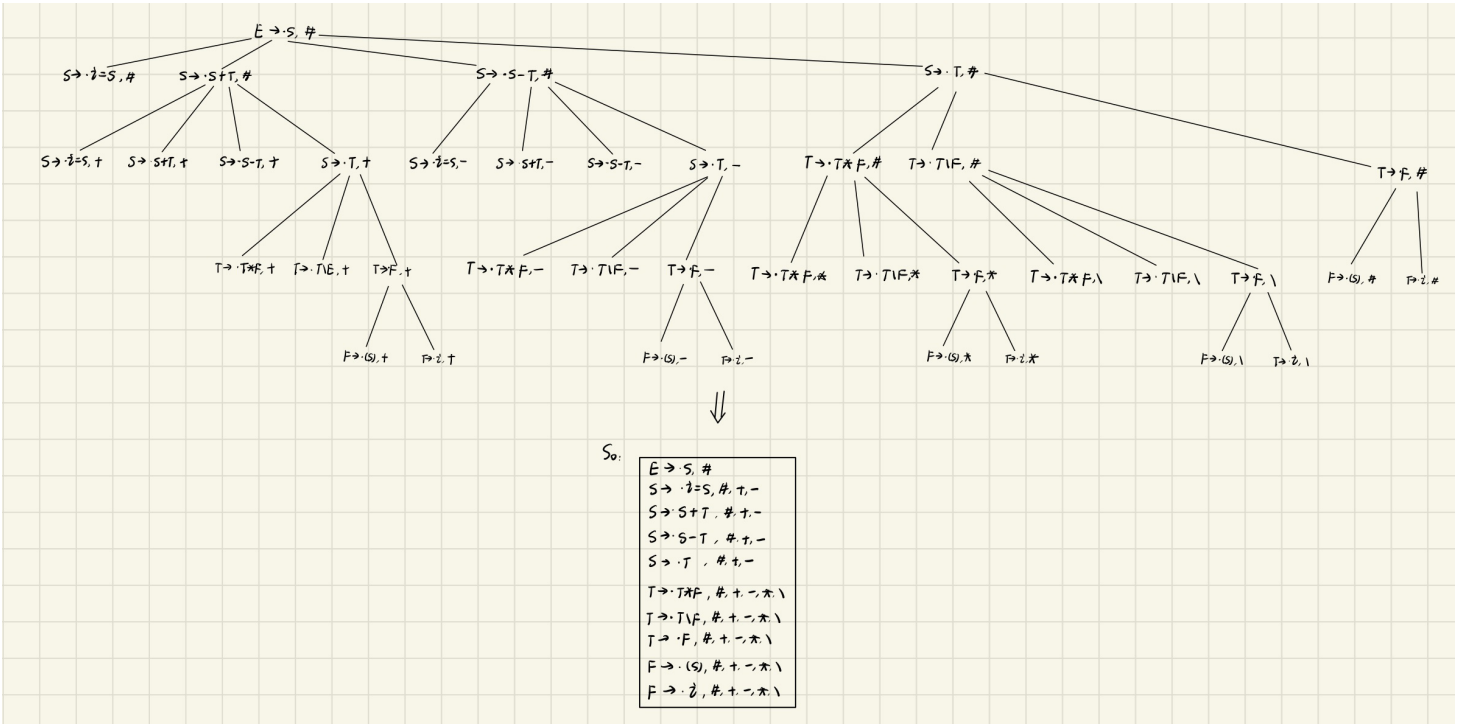
> 源码可见代码
> 中 `vector<char> getFirst(vector<wf> phaser, string s, vector<string> Vt)//得到First集` 函数

（二）如何获得GOTO图初始状态

算法思想：

1. 从前到后遍历文法中的每一个产生式A
2. 初始圆点都在产生式右部最前面，记圆点后的符号为x
   - 若x为终结符，返回第1步
   - 若x为非终结符A_1，且产生式A中A_1后面的符号的First集为P，跳到第3步
3. 遍历文法，找出产生式左部为A_1的产生式S，S的search集为P
4. 将S和对应的P加入状态，如果有相同的S（圆点位置也相同），合并search集即可，返回第1步

针对这里的文法，分析过程如下：



> 代码实现了上述思想，生成了GOTO图的初始状态

源码如下：

```
for (int i = 0; i < clo.t.size(); i++) //已有核· ,求出closure
{
        if (clo.t[i].position < phaser[clo.t[i].num].right.size()) //圆点位置不在最后
        {
                string first;
                first = phaser[clo.t[i].num].right[clo.t[i].position]; //圆点后面的字符
                if (find(Vn.begin(), Vn.end(), first) != Vn.end()) //如果圆点后面的字符为非终结符
                {
                        string s;
                        vector<char> search;
                        for (int j = clo.t[i].position + 1; j < phaser[clo.t[i].num].\
            right.length(); j++)
                                s += phaser[clo.t[i].num].right[j];
                        for (int j = 0; j < clo.t[i].search.size(); j++)
                        {
                                string t = s;
                                t += clo.t[i].search[j];
                                vector<char> search1 = getFirst(phaser, t, Vt);
                                for (int k = 0; k < search1.size(); k++)
                                {
                                        if (find(search.begin(), search.end(), \
                    search1[k]) == search.end())
                                                search.push_back(search1[k]); \
                         //得到向前搜索符search集合

                                }

                        }
                        temp.search = search;  //得到项目temp的向前搜索符search集合
                        for (int j = 0; j < phaser.size(); j++)
                        {
                                if (phaser[clo.t[i].num].right[clo.t[i].position] == phaser[j]\
                .left[0]) //如果圆点后的字符等于语法左边的字符
                                {
                                        temp.num = phaser[j].num;
                                        temp.position = 0;

                                        if (find(clo.t.begin(), clo.t.end(), temp)\
                    == clo.t.end()) //如果新加的项目是不重复的
                                        {
                                                int flag = 1;
                                                for (int k = 0; k < clo.t.size(); k++)
                                                {
```

```
                                        if (clo.t[k].num == temp.num \
                    &&
                     clo.t[k].position\ == temp.position) //如果是同心集
                                        {
                                                flag = 0;
                                                for (int n = 0; n < temp\
                        .search.size(); n++)
                                                {
                                                        if (find(clo.t[k].\
                            search.begin(), clo.t[k].\
                            search.end(), temp.search[n]) == clo.t[k].\
                            search.end())//如果同心集的search中不含当前temp的字符
                                                {
                                                                clo.t[k].search
                                (temp.search[n]);//就将temp的search加入同心集中
                                                }
                                                }
                                        }
                                }
                                if (flag) //如果新加的项目不重复且不是同心集
                                        clo.t.push_back(temp);  //就将项目加入clo
                                }

                        }
                }

        }
}
C.push_back(clo);//GOTO图初态
clo.t.clear();
```

（三）如何生成完整的GOTO图

得到GOTO图的初始状态S_0之后，需构造出完整的GOTO图，这是一个不断产生新状态的迭代过程

算法思想如下：

```
for 每个状态S_i in 状态集C
{
    遍历S_i每个产生式，得出圆点之后符号集为M
    for 符号m in M
    {
        for 产生式s in S_i
        {
            if s的圆点后的符号=m
            {
                1:将s的圆点向右移一位，作为一个新状态的核，用闭包产生算法得出一个新状态S_new
                2:查看状态集中有没有和这个新状态S_new相同的状态，如果没有，将S_new加入状态集
                3:用map的数据结构记录起始状态、读入字符、对应动作作为语法分析表
            }
        }
    }
}
```

算法实现了上述思想，最终得出完成的GOTO图，GOTO图截图见 二、程序输入输出展示 部分。

（四）如何用GOTO图构造语法分析表

数据结构

语法分析表结构 `map<string,string> m`

- m的key是初始状态数+读入字符，value是对应的操作。
- 比如（'0S','1'）表示S_0状态读入字符S后跳转到S_1状态，（'A+','r9'）表示状态S_{17}读入字符+对应的动作是r9

  > （这里A表示17，我是使用字符对应ASCII相对于'0'的偏移量来表示数字的，'A'-'0'=17，这样可以保证我每个状态的序号都是一位字符，便于后续操作）

算法思想

在构造GOTO图时，涉及到新状态产生时，将对应的初始状态、读入字符作为key，对应的动作作为value添加到变量m中

- 当读入字符为终结符时，对应的动作即为产生的新状态的序号
- 当读入字符为非终结符时，令新状态为S_new
  - 若产生的新状态的圆点均在产生式末尾，表示需要规约，规约对应的产生式需要为num
    - 当num!=0,对应动作为r_{num}，表示用第num个产生式规约
    - 当num==0，对应动作为acc，表示语句分析成功
  - 否则，表示推导，对应动作为S_{new}，表示跳转到状态S_new

（五）用语法分析表分析语句

算法思想：

1. 已构建的语法分析表结构 `map<string,string> m` ，m的key是初始状态数+读入字符，value是对应的操作。比如（'0S','1'）表示S_0状态读入字符S后跳转到S_1状态，（'A+','r9'）表示状态$S_1$7读入字符+对应的动作是r9
2. 起始状态下，状态栈中放入0，符号栈中放入#
3. 从状态栈中pop出当前状态n，从输入的语句中读入一个字符m，则key为'nm'，用key去语法分析表中找相应的value，即对应动作
   - 如果对应动作是s开头，表示推导，若表示跳转到状态S_i，将m读入符号栈，i读入状态栈
   - 如果对应动作是r开头，表示规约，若对应的产生式序号为num，根据第num号产生式右部的长度，从状态和符号集合中弹出相应数量的元素，并将产生式左部left加入符号栈。现在状态栈顶部为状态n'，n'和left在分析表中对应的数字为num'，将num'加入状态栈
   - 如果对应动作是acc，则分析成功
   - 如果没找到对应动作，报错

伪代码如下：

```
for each character c in input string in:
    key = current state + c
    action = lookup action in parsing table using key

    if action is not found:
        report error and exit

    if action is "acc":
        report successful parsing and exit

    if action starts with "S":
        state = second character of action
        push state onto state stack
        push c onto symbol stack

    else (action is a reduction):
        production = get production rule from action
        pop symbols and states from stacks based on length of production's right-hand side
        push production's left-hand side symbol onto symbol stack
        state = get state from parsing table using current state and production's left-hand side
        push state onto state stack

report parsing failure
```

（六）报错处理

报错时查找当前符号栈的顶层元素，报出相关错误即可。

五、遇到的问题

1. GOTO图中的符号太多，用十进制数字来表示的话，转成字符后位数不定，不利于检索
   答：使用ASCII值，使用字符对应ASCII相对于'0'的偏移量来表示数字，譬如：17用A表示，因为'A'-'0'=17
2. LR(1)分析法实现较为繁琐，生成GOTO图，生成语法分析表的时候细节非常多，需要耐心debug
3. 生成的语法分析表等输出太大，在输出框里用对齐('\t')输出也无法解决字符输出很乱的问题
   答：将输出框的字号调的很小就OK了！不会乱换行
4. 写报告时不知道怎么清晰的用文字来表达自己的逻辑，已经用伪代码等工具尽力表达了。

六、实验心得

通过这次手敲LR(1)文法的实验，感觉对LR(1)文法的步骤已经烂熟于心。LR（1）分析法在编程上面的实现非常繁琐，而且构造了文法之后，我不是手算的语法分析表，而是完成了一个自动化较高的语法分析器：自动构造GOTO图和语法分析表，再利用语法分析表来分析语句。这次实验极大的锻炼了我的编程和debug能力，也加固了编译原理的理论课只是，收获颇丰！