一、目标

根据 C 语言的词法规则，建立词法分析器展开识别，具体功能包括:

1) 支持标识符、常量（含十进制与十六进制的整数、浮点数）、关键字、分界符、运算符等词法类型;
2) 输出每个词的类型和单词属性;
3) 检查源程序中存在的词法错误，并报告错误所在的位置;
4) 查填符号表，例如，对于变量标识符，需在符号表存入标识符名字、类型等信息。

二、过程或算法

1. 程序实现内容介绍

本程序使用 C++实现了一个简单的 C 语言词法分析器，它可以匹配任何给定标准正则表达式所描述的模式，并且可以获得词素在源程序中的位置，同时支持简单的错误检测，以及建立简易的符号表，此外，它还过滤掉了源程序中的注释和空白。

首先使用调度场算法将标准正则表达式由中缀表达式转化为后缀表达式。然后采用 Thompson 算法根据标准正则表达式的后缀表达式构造 NFA，之后采用子集构造算法根据 NFA 构造 DFA。对于每个标准正则表达式均构造一个 DFA，并按照需要的顺序将它们排序。读入源程序，同时过滤掉注释和空白并保留位置信息。根据需要的优先级顺序使用之前得到的 DFA 依次检测源程序，若成功识别出与某个模式匹配的词素，则输出它的信息，否则报错。

2. 语言说明:

正则文法:

a. 标识符： $V_N = \{Å,\ \mathcal{B}\}, V_T = \{A-Z, a-z, \_0-9\}$, Å 为开始符号，产生式集合为(1) $Å \to A-Z\mathcal{B}|a-z\mathcal{B}|\_\mathcal{B}$ (2) $\mathcal{B} \to A-Z\mathcal{B}|a-z\mathcal{B}|\_\mathcal{B}|0-9\mathcal{B}|\varepsilon$。

注：为简洁方便起见，这里在集合中用 A-Z，a-z,0-9 分别表示列举全部大写字母、小写字母、数字，在产生式中用这些符号分别表示对应字符的并，如 A-Z$\mathcal{B}$实际应为$A\mathcal{B}|B\mathcal{B}|C\mathcal{B}|\ldots|Z\mathcal{B}$。下文中采取同样的记法。

b. 常量(十进制整数): $V_N = \{Å, \mathcal{B}, C\}, V_T = \{+,-,0-9\}$, Å 为开始符号，产生式集合为(1) $Å \to +\mathcal{B}|-\mathcal{B}|\mathcal{B}$ (2) $\mathcal{B} \to 1-9C|0$ (3) $C \to 0-9C|\varepsilon$。

c. 常量(十六进制整数)：$V_N = \{Å, \mathcal{B}, G\}, V_T = \{x, 0-9, A-F, a-f\}$, Å 为开始符号，产生式集合为(1) $Å \to 0x\mathcal{B}$ (2) $\mathcal{B} \to 0-9G|A-FG|a-fG$ (3) $G \to 0-9G|A-FG|a-fG|\varepsilon$。

d. 常量(浮点数)：$V_N = \{Å, \mathcal{B}, C, D, F, G, H\}, V_T = \{+, -, ., e, E, 0-9\}$, Å 为开始符号，产生式集合为(1) $Å \to +\mathcal{B}|-\mathcal{B}|\mathcal{B}$ (2) $\mathcal{B} \to 0-9\mathcal{B}|C$ (3) $C \to .D$ (4) $D \to 0-9D|F$ (5) $F \to EG|eG|\varepsilon$ (6) $G \to +0-9H|-0-9H|0-9H$ (7) $H \to 0-9H|\varepsilon$。

e. 关键字：$V_N = \{Å\}, V_T =$ $\{a, b, c, d, e, f, g, h, i, k, l, m, n, o, r, s, t, u, v, w, x, y, A, B, C, I, \_, S, N, T, G\}$, Å 为开始符号，产生式集合为(1) Å →

auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while|inline|restrict|_Bool|_Complex|_Imaginary|_Alignas|_Alignof|_Atomic|_Static_assert|_Noreturn|_Thread_local|_Generic。

f. 分界符：$V_N = \{Å\}, V_T = \{, ; : \{ \} ' " \#\}$, Å 为开始符号，产生式集合为(1) Å →,|;|:|{|}|'|"|#。

注：这里终结符集合中的字符'{','}','应当不致产生歧义，简洁起见不使用转义字符。

g. 运算符：$V_N = \{Å\}, V_T = \{( ) [ ] + - * / . = \& | ^ \% < > \sim ! ? : \}$, Å 为开始符号，产生式集合为(1) Å → (|)|[|]| + | − | ∗ |/|. | = |&|\||^|%| < | > |~|!|−> | + +| − −| << | >> | <= | >= | == |! = |&&|\|\||?:|+= |−= | ∗= |/= |% = |& = |^ = |\| = | <<= | >>=。

注：这里使用转义字符'\|'表示字符'|'。

正则式：

a. 标识符：
(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|_)_(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|_|0|1|2|3|4|5|6|7|8|9)*

b. 常量(十进制整数)：((\+|-)|\e)_((1|2|3|4|5|6|7|8|9)_(0|1|2|3|4|5|6|7|8|9)*|0)

注：\e 表示空字符ε，这是因为非标准 ASC II 码表中的字符可能无法输入。后文采用同样的记号。

c. 常量(十六进制整数):
0_x_(0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|a|b|c|d|e|f)_(0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|a|b|c|d|e|f)*

d. 常量(浮点数):
((\+|-)|\e)_(0|1|2|3|4|5|6|7|8|9)*_._(0|1|2|3|4|5|6|7|8|9)*_(((e|E)_((\+|-)|\e)_(0|1|2|3|4|5|6|7|8|9)_(0|1|2|3|4|5|6|7|8|9)*)|\e)

e. 关键字:
a_u_t_o|b_r_e_a_k|c_a_s_e|c_h_a_r|c_o_n_s_t|c_o_n_t_i_n_u_e|d_e_f_a_u_l_t|d_o|d_o_u_b_l_e|e_l_s_e|e_n_u_m|e_x_t_e_r_n|f_l_o_a_t|f_o_r|g_o_t_o|i_f|i_n_t|l_o_n_g|r_e_g_i_s_t_e_r|r_e_t_u_r_n|s_h_o_r_t|s_i_g_n_e_d|s_i_z_e_o_f|s_t_a_t_i_c|s_t_r_u_c_t|s_w_i_t_c_h|t_y_p_e_d_e_f|u_n_i_o_n|u_n_s_i_g_n_e_d|v_o_i_d|v_o_l_a_t_i_l_e|w_h_i_l_e|i_n_l_i_n_e|r_e_s_t_r_i_c_t|__B_o_o_l|__C_o_m_p_l_e_x|__I_m_a_g_i_n_a_r_y|__A_l_i_g_n_a_s|__A_l_i_g_n_o_f|__A_t_o_m_i_c|__S_t_a_t_i_c___a_s_s_e_r_t|__N_o_r_e_t_u_r_n|__T_h_r_e_a_d___l_o_c_a_l|__G_e_n_e_r_i_c

f. 分界符：,|;|:|{|}|'|"|#

g. 运算符：\(|\)|\+|-|*|/|.|=|&|\||^|%|<|>|~|!|[|]|-_>|\+_\+|-_-|<_<|>_>|<_=|>_=|=_=|!_=|&_&|\|_\||?_:|\+_=|-_=|*_=|/_=|%_=|&_=|^_=|\|_=|<_<_=|>_>_=

3. 词法编码表及说明

单词	词类编码
auto	1
break	2
case	3
char	4
const	5
continue	6
default	7
do	8
double	9
else	10

enum	11
extern	12
float	13
for	14
goto	15
if	16
int	17
long	18
register	19
return	20
short	21
signed	22
sizeof	23
static	24
struct	25
switch	26
typedef	27
union	28
unsigned	29
void	30
volatile	31
while	32
inline	33
restrict	34

_Bool	35
_Complex	36
_Imaginary	37
_Alignas	38
_Alignof	39
_Atomic	40
_Static_assert	41
_Noreturn	42
_Thread_local	43
_Generic	44
,	45
;	46
:	47
{	48
}	49
\a	50
\b	51
\f	52
\n	53
\r	54
\t	55
\v	56
\\	57
\’	58

\”	59
\?	60
\0	61
\ddd	62
\xhh	63
(	64
)	65
+	66
-	67
*	68
/	69
.	70
=	71
&	72
|	73
^	74
%	75
<	76
>	77
~	78
!	79
[	80
]	81
->	82

++	83
--	84
<<	85
>>	86
<=	87
>=	88
==	89
!=	90
&&	91
||	92
?:	93
+=	94
-=	95
*=	96
/=	97
%=	98
&=	99
^=	100
|=	101
<<=	102
>>=	103
‘	104
“	105
id	106

decimal	107
float_const	108
hex	109
#	110

说明：id 表示标识符，decimal 表示十进制整数常量，float_const 表示浮点数常量，hex 表示十六进制整数常量。界符和运算符采用一符一码，关键字采用一字一码，常数采用一类型一码，标识符采用一类一码。

4. 符号表接口及说明

因为符号表中的记录与具体测试用例相关，这里给出符号表表头。

词素	类型	首次出现位置

该符号表仅用来记录标识符，词素即为标识符的名字，首次出现位置记录该标识符首次出现的行和列，它应该是二元组的形式。类型属性在词法分析阶段尚不能确定，因此这里仅定义该属性，词法分析器向符号表中添加条目时将忽略类型属性。实际上一个标识符是否应该被添加到符号表中也无法由词法分析器完全决定，这里简单的将检测到的所有标识符（包括重复的）添加到符号表中。

5. 错误处理说明

本词法分析器共处理了四种类型的错误，分别为

a. 以数字开头的标识符，如“1a”。

b. 十六进制数中出现非法字符，如“0xfg”。

c. 本词法分析器不支持的非法字符或词法规则，如“`aw_1”。

d. 浮点数非正常结束，如“2.3aa”。

6. 程序说明：

(1) 调度场算法

```
string ShuntingYard(string& re){
    string rep;
    stack<char> st;
    set<char> op={'(',')','_','*','|'};
    int escape=0;
    for(int i=0;i<re.size();++i){
        if(escape){
            if(re[i]=='e'){
                rep.pop_back();
                rep.push_back(eps);
            }else{
                rep.push_back(re[i]);
            }
```

```
            escape=0;
        }else{
            if(re[i]=='\\'){
                rep.push_back(re[i]);
                escape=1;
                continue;
            }
            if(op.count(re[i])){
                if(re[i]==')'){
                    char top='\0';
                    while(!st.empty()){
                        top=st.top();
                        st.pop();
                        if(top=='(')break;
                        rep.push_back(top);
                    }
                    if(top!='('){
                        printf("Error! No compared '(' to ')' in regular
expression!\n");
                    }
                }else{
                    char top;
                    while(!st.empty()){
                        top=st.top();
                        if(compre(top,re[i])){
                            st.pop();
                            rep.push_back(top);
                        }else break;
                    }
                    st.push(re[i]);
                }
            }else{
                rep.push_back(re[i]);
            }
        }
    }
    while(!st.empty()){
        rep.push_back(st.top());
        st.pop();
    }
    return rep;
}
```

ShuntingYard 函数实现了调度场算法，接收正则表达式作为参数，返回该正则表达式的逆波兰式。建立一个字符栈 st，用于转换正则表达式中运算

符和操作数的顺序。把标准正则表达式中所有运算符存到一个字符集合 op 中，通过判断字符是否在该集合中的方式来判断字符是否是运算符。从左到右遍历正则表达式，按照调度场算法进行操作。其中转义字符保留反斜杠(空字符ε除外)，从而确保能够区分出逆波兰式中的转义字符和运算符。对于用\e 表示的空字符ε，则转换为 ASC II 码中编码为 127 的字符，且不再需要保留反斜杠。

compre 函数用于比较两个运算符的优先级，参数为两个运算符 a 和 b，返回 a>=b 的布尔值，这里>=表示前者优先级不低于后者。代码如下：

```
bool compre(char a,char b){
    char pre[5]={'*','_','|','('};
    if(b=='('){
        return false;
    }
    for(int i=0;pre[i]!=b;++i){
        if(pre[i]==a)return true;
    }
    if(a==b)return true;
    return false;
}
```

(2) Thompson 算法

用图来表示 NFA。需要两种结构，一是节点，二是子图。

节点包含节点 id，访问标记(用于在遍历 NFA 时判断该节点是否已经被访问)，以及一个后继节点数组。后继节点以边的结构表示，包含一个目标节点 id 和一个转移字符。

边结构定义：

```
struct edge{
    int dst;
    char c;
};
```

NFA 节点结构定义：

```
struct NFAnode{
    int id,v=0;
    vector<edge> nxt;
};
```

NFA 可以用 NFA 节点的数组表示：

```
vector<NFAnode> NFA;
```

NFA 子图实际上只需要提供 start 节点和 end 节点的 id 即可，其它信息可在 NFA 图中得到，NFA 子图结构定义：

```
struct subNFA{
    int start,end;
};
```

使用 Thompson 算法构建 NFA 可分为 4 部分，分别为构建字符的 NFA，构建运算符'*'，'_'，'\'的 NFA。

构建字符的 NFA 只需要向 NFA 中添加两个节点，并使前一个节点通过该字符指向后一个节点即可：

```
subNFA fc(char a){
    edge edg;
    edg.c=a;
    edg.dst=NFA.size()+1;
    NFAnode start,end;
    start.id=NFA.size();
    end.id=NFA.size()+1;
    start.nxt.push_back(edg);
    subNFA sub;
    sub.start=start.id;
    sub.end = end.id;
    NFA.push_back(start);
    NFA.push_back(end);
    return sub;
}
```

构建运算符的 NFA 则要分别根据每种运算符，对子图做相应操作。

采用一个栈来存放子图，根据当前运算符的需要，弹出相应数量的子图，再将该运算符得到的子图压入栈中，对于字符构建的 NFA 则直接压栈。最终栈中将只有正则式对应的完整的 NFA。Thompson 算法实现如下：

```
subNFA Thompson(string& re){
    set<char> op={'_','*','|'};
    stack<subNFA> st;
    int escape=0;
    //cout << re << endl;
    //printf("rep.size()=%d\n",re.size());
    for(int i=0;i<re.size();++i){
        if(escape){
            st.push(fc(re[i]));
            escape=0;
```

```
        }else{
            if(re[i]=='\\'){
                escape=1;
                continue;
            }
            if(op.count(re[i])){
                switch(re[i]){
                    case '_':{
                        if(st.size()<2){
                            printf("Error! There are less than two opera
nds for operator _.\n");
                        }
                        subNFA a,b,nsb;
                        b=st.top();
                        st.pop();
                        a=st.top();
                        st.pop();
                        nsb.start = a.start;
                        nsb.end = b.end;
                        NFA[a.end].nxt.insert(NFA[a.end].nxt.end(),NFA[b
.start].nxt.begin(),NFA[b.start].nxt.end());
                        st.push(nsb);
                        break;
                        }
                    case '|':{
                        if(st.size()<2){
                            printf("Error! There are less than two opera
nds for operator |.\n");
                        }
                        subNFA a,b,nsb;
                        b=st.top();
                        st.pop();
                        a=st.top();
                        st.pop();
                        NFAnode start,end;
                        start.id=NFA.size();
                        end.id=NFA.size()+1;
                        edge sa,ae,sb,be;
                        sa.c=eps;
                        sa.dst=a.start;
                        sb.c=eps;
                        sb.dst=b.start;
                        ae.c=eps;
                        ae.dst=end.id;
```

```
                    be.c=eps;
                    be.dst=end.id;
                    start.nxt.push_back(sa);
                    start.nxt.push_back(sb);
                    NFA[a.end].nxt.push_back(ae);
                    NFA[b.end].nxt.push_back(be);
                    NFA.push_back(start);
                    NFA.push_back(end);
                    nsb.start=start.id;
                    nsb.end=end.id;
                    st.push(nsb);
                    break;
                    }
                case '*':{
                    if(st.size()<1){
                        printf("Error! There are no operand for oper
ator *.\n");
                    }
                    subNFA a,nsb;
                    a=st.top();
                    st.pop();
                    NFAnode start,end;
                    start.id=NFA.size();
                    end.id=NFA.size()+1;
                    edge se,ses,sa,ae;// start to end, sub end to st
art
                    se.c=ses.c=sa.c=ae.c=eps;
                    se.dst=end.id;
                    ses.dst=a.start;
                    sa.dst=a.start;
                    ae.dst=end.id;
                    start.nxt.push_back(sa);
                    start.nxt.push_back(se);
                    NFA[a.end].nxt.push_back(ses);
                    NFA[a.end].nxt.push_back(ae);
                    NFA.push_back(start);
                    NFA.push_back(end);
                    nsb.start=start.id;
                    nsb.end=end.id;
                    st.push(nsb);
                    break;
                    }
                default:printf("Wrong operator!\n");break;
            }
```

```
            }else{
                st.push(fc(re[i]));
            }
        }
    }
    return st.top();
}
```

(3) 子集构造算法

用图来表示 DFA，DFA 节点应包含 id，结束标记 end，后继节点组，对应 NFA 中的状态集。结束标记用于判断是否可从当前节点跳出 DFA。后继节点组用边向量实现，边的定义与 NFA 中相同，状态集用整型数集合实现。DFA 节点结构定义如下：

```
struct DFAnode{
    int id,end=0;
    vector<edge> nxt;
    set<int> states;
};
```

引入三个辅助函数，set<int> closures(int id)，set<int> closureT(set<int>& states)和 set<int> moveTa(set<int> states,char a)，分别用于求 NFA 中某个状态 id 的闭包，某个状态集 states 的闭包和从状态集 states 中某个状态出发，通过输入 a 能转换到达的 NFA 状态的集合，对应代码如下：

```
set<int> closures(int id){
    set<int> states;
    states.insert(id);
    cssdfs(states,id);
    return states;
}
set<int> closureT(set<int>& states){
    set<int> Tstates;
    for(auto it=states.begin();it!=states.end();++it){
        set<int> clo=closures(*it);
        set_union(clo.begin(),clo.end(),Tstates.begin(),Tstates.end(),in
serter(Tstates,Tstates.begin()));
    }
    return Tstates;
}
set<int> moveTa(set<int> states,char a){
    set<int> nstates;
    for(auto it=states.begin();it!=states.end();++it){
```

```
        for(auto itt=NFA[*it].nxt.begin();itt!=NFA[*it].nxt.end();++itt)
{
            if(itt->c==a){
                nstates.insert(itt->dst);
            }
        }
    }
    return nstates;
}
```

子集构造算法实现如下:

```
void subsetConstruction(int s0,subNFA nfa){
    set<int> clos0 = closures(s0);
    D[clos0]=0;
    int ti=0;// tag index
    DFAnode cs0;
    cs0.id=0;
    if(clos0.count(nfa.end)){
        cs0.end=1;
    }
    cs0.states=clos0;
    DFA.push_back(cs0);
    while(ti<DFA.size()){
        set<int> U;
        for(auto it=chars.begin();it!=chars.end();++it){
            set<int> moveti=moveTa(DFA[ti].states,*it);
            set<int> cmta=closureT(moveti);
            DFAnode nn;
            if(cmta.size()){
                if(!D.count(cmta)){
                    nn.id=DFA.size();
                    nn.states=cmta;
                    D[cmta]=DFA.size();
                    if(nn.states.count(nfa.end)){
                        nn.end=1;
                    }
                    DFA.push_back(nn);
                }else{
                    nn=DFA[D[cmta]];
                }
                edge edg;
                edg.c=*it;
                edg.dst=nn.id;
                DFA[ti].nxt.push_back(edg);
```

```
            }
        }
        ++ti;
    }
}
```

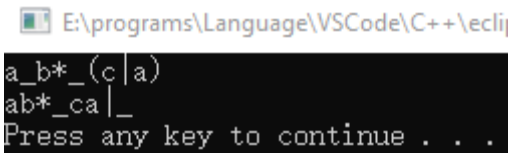
因为采用向量 vector 结构存储 DFA，每次添加节点都添加在向量的末尾，可通过顺序遍历的方式执行算法，而不必再通过添加标记来判断某个节点是否已被访问过。

三、测试结果

调度场算法测试：

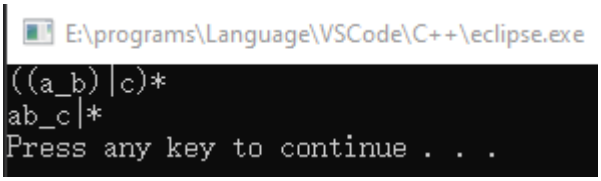
用例 1：a_b*_(c|a)

结果：



用例 2：((a_b)|c)*

结果：



用例 3：

(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|_)_(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|_|0|1|2|3|4|5|6|7|8|9)*

结果：

用例 4：((\+|-)|\e)_(1|2|3|4|5|6|7|8|9)_(0|1|2|3|4|5|6|7|8|9)*

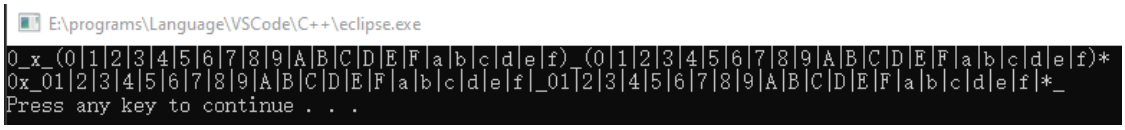
结果：

注：这里因为空字符\e 的存在，其转化后的 127 号字符无法显示，因此对应位置为一个空格。后面的测试结果中也是同样。

用例 5:

0_x_(0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|a|b|c|d|e|f)_(0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|a|b|c|d|e|f)*

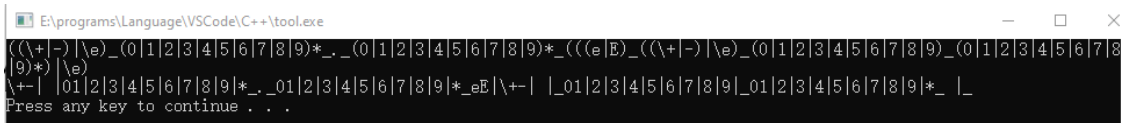
结果:



用例 6:

((\+|-)|\e)_(0|1|2|3|4|5|6|7|8|9)*_._(0|1|2|3|4|5|6|7|8|9)*_(((e|E)_((\+|-)|\e)_(0|1|2|3|4|5|6|7|8|9)_(0|1|2|3|4|5|6|7|8|9)*)|\e)

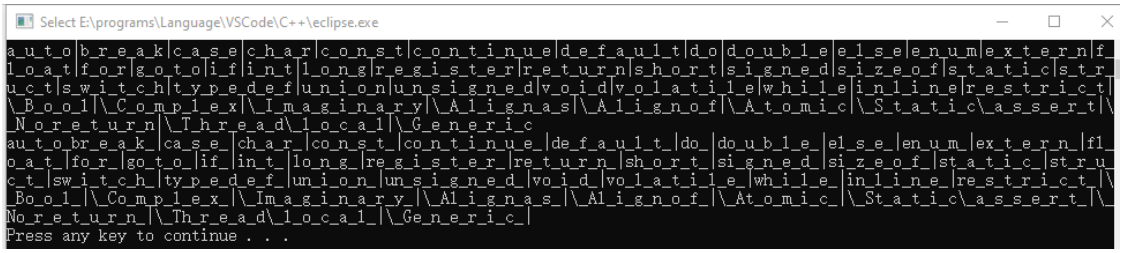
结果:



用例 7:

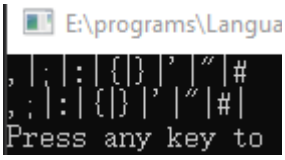
a_u_t_o|b_r_e_a_k|c_a_s_e|c_h_a_r|c_o_n_s_t|c_o_n_t_i_n_u_e|d_e_f_a_u_l_t|d_o|d_o_u_b_l_e|e_l_s_e|e_n_u_m|e_x_t_e_r_n|f_l_o_a_t|f_o_r|g_o_t_o|i_f|i_n_t|l_o_n_g|r_e_g_i_s_t_e_r|r_e_t_u_r_n|s_h_o_r_t|s_i_g_n_e_d|s_i_z_e_o_f|s_t_a_t_i_c|s_t_r_u_c_t|s_w_i_t_c_h|t_y_p_e_d_e_f|u_n_i_o_n|u_n_s_i_g_n_e_d|v_o_i_d|v_o_l_a_t_i_l_e|w_h_i_l_e|i_n_l_i_n_e|r_e_s_t_r_i_c_t|_B_o_o_l|_C_o_m_p_l_e_x|_I_m_a_g_i_n_a_r_y|_A_l_i_g_n_a_s|_A_l_i_g_n_o_f|_A_t_o_m_i_c|_S_t_a_t_i_c_a_s_s_e_r_t|_N_o_r_e_t_u_r_n|_T_h_r_e_a_d_l_o_c_a_l|_G_e_n_e_r_i_c

结果:

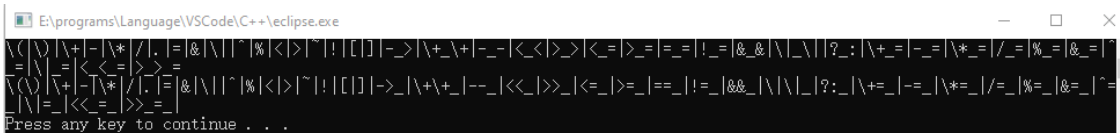


用例 8: ,|;|:|{|}|'|"|#

结果:



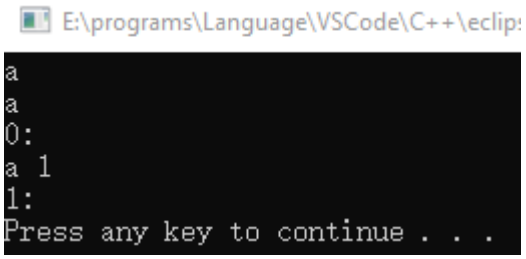
用例 9：\(|\)|\+|-|*|/|.|=|&|\||^|%|<|>|~|!|[|]|-_>|\+_\+|-_-
|<_<|>_>|<_=|>_=|=_=|!_=|&_&|\|_\||?_:|\+_=|-
_=|*_=|/_=|%_=|&_=|^_=|\|_=|<_<_=|>_>_=
[()+-*/.=&|^%<>~!]|\[|\]|->|\+\+|--|<<|>>|<=|>=|==|!=|&&|\|\||\?:|\+=|-
=|*=|/=|%=|&=|\^=|\|=|<<=|>>=
结果：



Thompson 算法测试：

用例 1：a

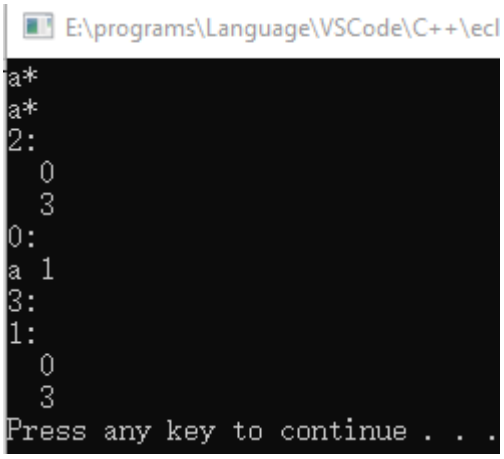
结果：



结果的显示格式为：第一行是输入的正则表达式，第二行是该正则表达式的逆波兰式，后面按照广度优先搜索的顺序显示节点信息，对于每个结点，首先输出节点号:，接下来每一行输出一个该结点的后继节点的 id 以及到该后继节点的字符，格式为字符 id。后面的用例的结果采取同样的格式。

用例 2：a*

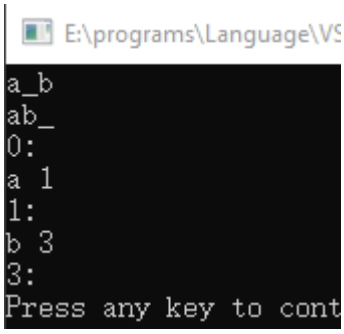
结果：



空字符 127 显示为一个空格。后面的结果中也是同样。

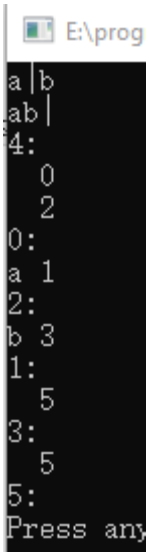
用例 3：a_b

结果：



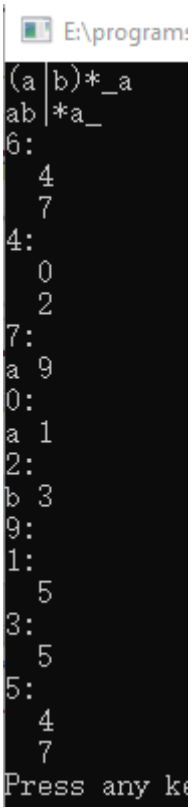
用例 4：a|b

结果：



用例 5：(a|b)*_a

结果：


E:\program
(a|b)*_a
ab|*a_
6:
4
7
4:
0
2
7:
a 9
0:
a 1
2:
b 3
9:
1:
5
3:
5
5:
4
7
Press any k

用例 6：0_1*_(0|1)

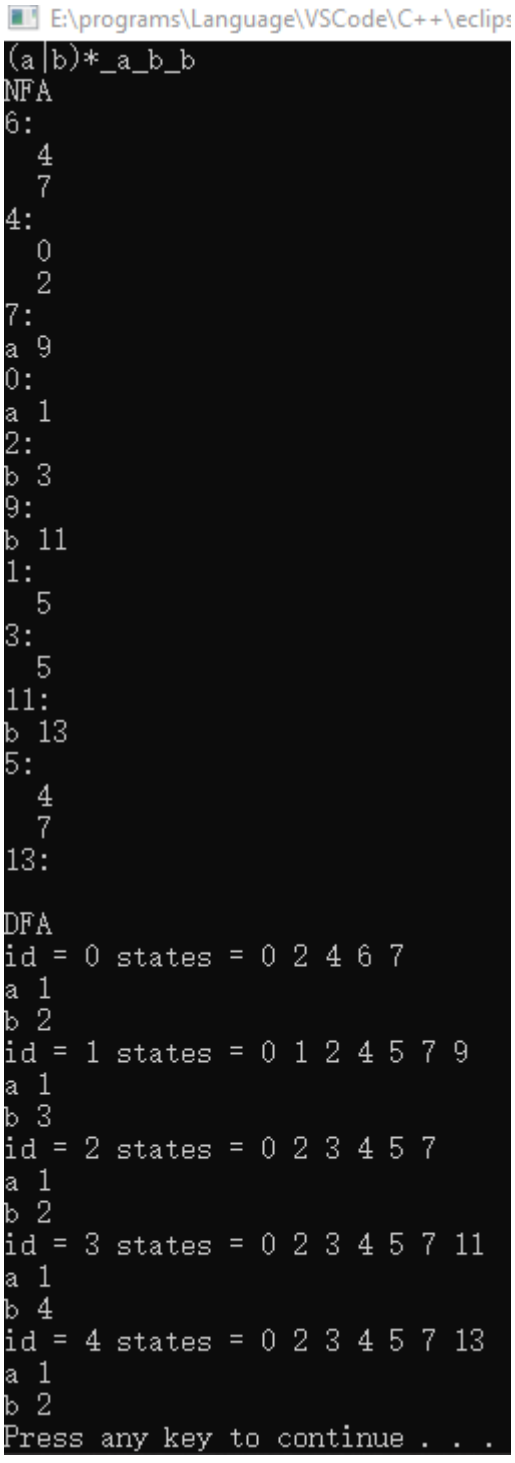
结果：


0_1*_(0|1)
01*_01|_
0:
0 1
1:
2
5
2:
1 3
5:
6
8
3:
2
5
6:
0 7
8:
1 9
7:
11
9:
11
11:
Press any ke

子集构造算法测试：

用例 1：(a|b)*_a_b_b

结果：



NFA 的输出格式前面已经描述过。对于 DFA 的每一个节点，首先输出节点号 id，在同一行紧跟着输出该节点对应的 NFA 中的状态集，接下来每一行输出一个该节点的后继节点，格式为“转移到后继节点的字符 后继节点 id”。顺序输出 DFA 中的节点。

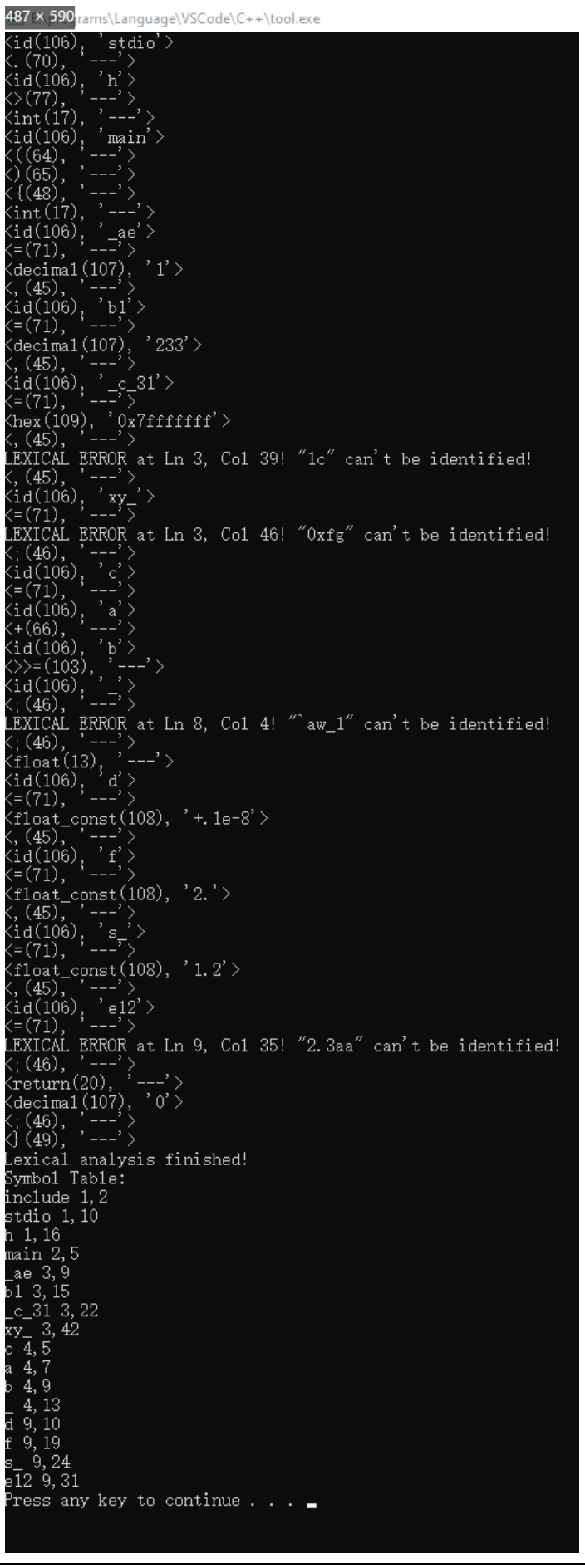
最终结果测试：

测试代码：

```
#include<stdio.h>
int main(){
    int _ae=1,b1=233,_c_31=0x7fffffff,1c,xy_=0xfg;//*///e+f*h
    c=a+b>>=_;/*

    asda;;22 wlkfj
    */
  `aw_1;
  float d=+.1e-8,f=2.,s_=1.2,e12=2.3aa;
    return 0;
}
```

测试结果：


487 × 590
rams\Language\VSCode\C++\tool.exe
<id(106), 'stdio'>
<,(70), '---'>
<id(106), 'h'>
<>(77), '---'>
<int(17), '---'>
<id(106), 'main'>
<((64), '---'>
<)(65), '---'>
<{(48), '---'>
<int(17), '---'>
<id(106), '_ae'>
<=(71), '---'>
<decimal(107), '1'>
<,(45), '---'>
<id(106), 'b1'>
<=(71), '---'>
<decimal(107), '233'>
<,(45), '---'>
<id(106), '_c_31'>
<=(71), '---'>
<hex(109), '0x7fffffff'>
<,(45), '---'>
LEXICAL ERROR at Ln 3, Col 39! "1c" can't be identified!
<,(45), '---'>
<id(106), 'xy_'>
<=(71), '---'>
LEXICAL ERROR at Ln 3, Col 46! "0xfg" can't be identified!
<;(46), '---'>
<id(106), 'c'>
<=(71), '---'>
<id(106), 'a'>
<+(66), '---'>
<id(106), 'b'>
<>>=(103), '---'>
<id(106), '_'>
<;(46), '---'>
LEXICAL ERROR at Ln 8, Col 4! "`aw_1" can't be identified!
<;(46), '---'>
<float(13), '---'>
<id(106), 'd'>
<=(71), '---'>
<float_const(108), '+.1e-8'>
<,(45), '---'>
<id(106), 'f'>
<=(71), '---'>
<float_const(108), '2.'>
<,(45), '---'>
<id(106), 's_'>
<=(71), '---'>
<float_const(108), '1.2'>
<,(45), '---'>
<id(106), 'e12'>
<=(71), '---'>
LEXICAL ERROR at Ln 9, Col 35! "2.3aa" can't be identified!
<;(46), '---'>
<return(20), '---'>
<decimal(107), '0'>
<;(46), '---'>
<}(49), '---'>
Lexical analysis finished!
Symbol Table:
include 1,2
stdio 1,10
h 1,16
main 2,5
_ae 3,9
b1 3,15
_c_31 3,22
xy_ 3,42
c 4,5
a 4,7
b 4,9
_ 4,13
d 9,10
f 9,19
s_ 9,24
e12 9,31
Press any key to continue . . .

分析：测试代码中覆盖了本词法分析器支持的所有词法规则(标识符、关键字、十进制整型常量、十六进制整型常量、浮点型常量、分界符、运算符)，同时覆盖了所有四种错误类型。对于正确匹配到的词素，按照词类编码表输出其类型以及在编码表中的编号，如果是关键字和常量还会输出该词素。遇到错误则输出包含错误字符串以及其在源程序中的位置的错误信息，然后跳过该字符串继续检测。源程序检测结束后输出结束提示，然后输出符号表中的内容，因为词法分析器无法确定标识符类型，所以只输出标识符和它在源程序中的位置(以行，列形式表示)。

四、总结

遇到的问题及解决办法

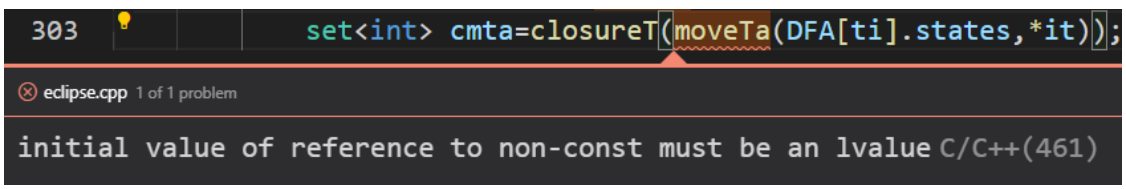
问题 1：标准正则表达式中，'(',')','+','*','|'五个字符既可能作为正则表达式的运算符出现，也可能作为在其上定义了该正则表达式的字符表中的字符出现，存在无法区分两者的问题。

解决方案：采用转义字符'\'对这些运算符进行转义，以表示这些字符不作为正则表达式的运算符出现。用"\\"表示'\'本身。

问题 2：正则表达式中，程序无法读入空字符ε。

解决方案：C++语言支持标准 ASC II 码中的字符，而ε不在该表中，解决思路是选取一个 ASC II 码中的字符来替代，且该字符能被成功输入。这样的字符都在 C 语言的字符集中，选取字母 e 表示ε，为和字母 e 本身区分，当表示ε时，在其前面添加转义字符'\'，即采用"\e"表示ε。

问题 3：求 move(T,a)的闭包时报错(如下图)。



解决方案：经研究发现，C++函数在传递占用内存较大的类型的参数时，默认采用传递引用的方式，取引用运算只能作用于左值，而函数返回值是个右值，因此出错。通过声明中间变量暂存 moveTa 的返回值使问题得到解决(如下图)。

```
set<int> moveti=moveTa(DFA[ti].states,*it);
set<int> cmta=closureT(moveti);
```

问题 4：在实现子集构造算法时，需要快速判断一个状态集是否已经出现在 DFA 中并找到其在 DFA 中所属的节点(若出现)。

解决方案：采用 map 结构，将状态集映射到 DFA 节点号。使用 count 方法判断状态集是否在 DFA 中。

问题 5：C 语言中相邻词素之间不必有空格，如何判断当前词素已经结束或是产生异常？

解决方案：将那些状态集中包含 NFA 结束状态的 DFA 节点标记为结束节点，只有在结束节点跳出 DFA 时当前标识符才算识别成功，否则产生异常。

问题 6：不同词法单元对应不同的 DFA，为了识别出词素所匹配的词法单元，需要依次使用这些 DFA，因此需要能够为每个输入的正则式生成 DFA，并将它们保存起来。

解决方案：将根据正则表达式生成 DFA 的完整过程封装起来，放到头文件中，并提供对外的接口 getDFA 函数。该函数根据传入的正则表达式返回 DFA。由于使用全局变量保存 NFA 和 DFA，因此该接口每次被调用时需要先把 NFA 和 DFA 清空。

问题 7：关键字的识别问题。关键字符合标识符的模式，如何将关键字从标识符中区分出来。

解决方案：可以通过识别顺序进行区分，优先识别关键字的模式，如果符合关键字的模式，就不再继续尝试匹配标识符的模式。

问题 8：识别不出 return 关键字，对于 return 关键字的识别结果如下图。

```
<retur(0), '---'>
<id(106), 'n'>
```

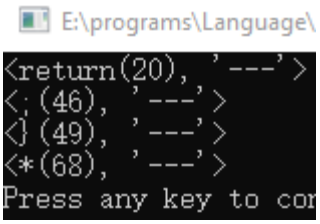
解决方案：经检查发现，判断跳出 DFA 的逻辑实现存在问题。判断跳出 DFA 的基本逻辑为，若当前节点没有任何后继节点是可以通过当前字符到达的，则跳出。如下图，第 24 行判断是否所有后继节点都已遍历的条件出错。

```
17      for(auto it=DFAs[k][cur].nxt.begin();it!=DFAs[k][cur].nxt.end();++it){
18          if(it->c==cod[i].c){
19              cur=it->dst;
20              break;
21          }
22          ++j;
23      }
24      if(j==DFAs[k][cur].nxt.size()){
25          if(DFAs[k][cur].end)
26              return pair<int,int>(0,i);
27          else return pair<int,int>(1,i);
28      }
```

因为如果当前字符可以使当前节点跳转到其后继节点，则当前节点在第 19 行被跳转到的后继节点替换，此时 24 行的判断条件等号右边成为了后继节点的后继节点数，因此出错。引入跳出标记 f，默认为跳出，如果在当前符号下有后继节点可以转移(执行第 18 行的 if 分支)，则改变 f 的状态，以此判断是否跳出。修改后的代码如下图。

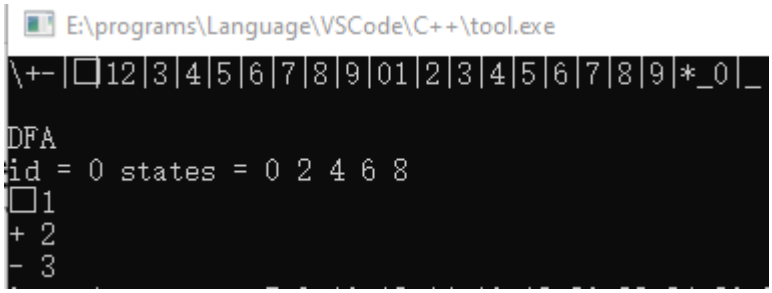
```
16      int j=0,f=0;
17      for(auto it=DFAs[k][cur].nxt.begin();it!=DFAs[k][cur].nxt.end();++it){
18          if(it->c==cod[i].c){
19              cur=it->dst;
20              f=1;
21              break;
22          }
23          ++j;
24      }
25      if(!f){
26          if(DFAs[k][cur].end)
27              return pair<int,int>(0,i);
28          else return pair<int,int>(1,i);
29      }
```

修改后问题解决，结果如下图。

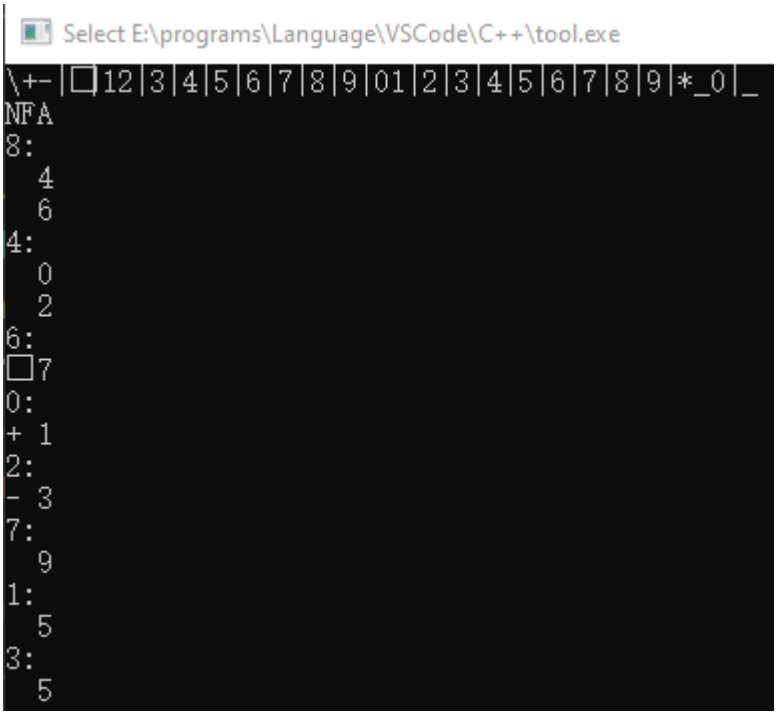


问题 9：数字 0 没有被检查出来，如上图。

解决方案：经检查，匹配十进制整数的正则表达式出错，原来的正则式 ((\+|-)|\e)_(1|2|3|4|5|6|7|8|9)_(0|1|2|3|4|5|6|7|8|9)* 不能匹配数字 0。将其修正为 ((\+|-)|\e)_((1|2|3|4|5|6|7|8|9)_(0|1|2|3|4|5|6|7|8|9)*|0)，问题仍未解决。经检查，发现生成的 DFA 中存在空转移，如下图。

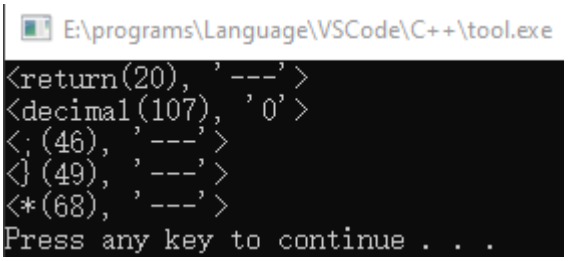


进一步检查发现 NFA 中输入的空字符和 Thompson 算法内部生成的空字符不同，如下图。



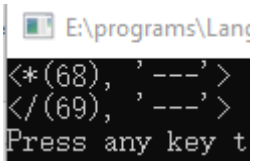
图中 6：状态下 7 前面的方框为输入的空字符，而 Thompson 算法生成的空字符表现为一个空格。

单步调试发现输入的空字符被转换为'\033'，而调度场算法中根本没有检测到'\e'，这是因为'\e'被当作了一个转义字符。将其改为"\\e"后，问题解决。这是在内部测试，正则式直接由代码中的字符串常量提供而非用户输入时才会出现的问题。用户输入时，仍然只需输入'\e'即可。结果如下图。



问题 10：文件结尾处的'/'没有被检测出来。如上图。

解决方案：检查发现这是设计读源代码文件的代码时的一处疏漏，原设计分了三种状态，0 状态是没有注释的读入状态，1 状态是看到单行注释"//"后的读入状态，2 状态是看到多行注释"/*"后的读入状态。记录上一个读入的字符，0 状态时若检测到当前读入字符和上一个读入字符都是'/'则跳转到 1 状态，如果检测到当前读入字符是'/'而上一个读入的字符是'*'，则跳转到 2 状态，否则如果当前读入字符是'/'，则不做任何处理，否则如果当前读入字符不是'/'，则在上一个读入的字符'/'时向读入后生成的序列追加上一个读入的字符'/'。这个逻辑的问题在于，'/'字符的读入依赖于其下一个字符，若'/'字符出现在文件结尾处，则其无法被读入。可以将逻辑修改为遇到'/'直接追加到生成的字符串上，在看到注释开始符号"//"或"/*"时，再将生成的字符串末尾的'/'移除，这样'/'的读入就不再依赖于下一个字符了。修改后成功读出了文件结尾处的'/'字符，如下图。



问题 11:从 DFA 中的结束节点跳出时不一定能匹配出词素，如"1a"，会识别出整数"1"和标识符"a"。也就是说，存在如何正确分割相邻词素的问题。

解决方案：通过预先设定一组字符集合来对相邻词素进行粗略分割，这组字符集应包含分界符、空格和运算符中的字符，当跳出常量的 DFA 时看到的是这组字符集中的字符时认为成功跳出，否则失败。当然并不是所有词法单元后面都可以紧跟这组字符集中

的任意字符，因此这种方法只能粗略检测出部分错误，剩下无法检测的错误则交由后续语法分析处理。