```
//vector<string>
//vector<int>

//属性定义

struct Ǐ
{
    vector<int> place;
};
struct E
{
    string place,code;
};
struct Ě
{
    vector<string> place,code;
};
struct B
{
    string tc,fc,code;
};
struct S
{
    string code;
};
struct Š
{
    vector<string> code;
};
struct T
{
    string type;
    int width;
};
struct D
{
    Tab tab;
    string place;
    bool func_or_arr;
};
struct Ď
{
    vector<string> place;
    bool func_or_arr;
};
struct d
{
    string name;
```

```
};
struct i
{
    int value;
};
struct r
{
    string name;
};


//符号表Tab及其操作

struct Tab;
struct Item
{
    string name,type;
    int offset; Tab* mytab;
    string etype; int dims,dim[5],base; //至多6维数组
};
struct Tab
{
    Tab* outer;
    int width;
    string code;
    string rtype;
    int args; vector<string> arglist;
    vector<Item> list;
    Tab(){outer=NULL,width=0,code="",args=0;}
};
void bind(Tab &tb,string name,string type)
{
    Item tp;
    tp.name=name,tp.type=type;
    tb.list.push_back(tp);
}
void lookup_c(Tab &tb,string name,string seg_name,string str="",int it=0,Tab* add=NULL)
//修改
{
    for(auto J:tb.list)
        if(J.name==name)
        {
            if(seg_name=="name")tb.name=str;
            else if(seg_name=="type")tb.type=str;
            else if(seg_name=="offset")tb.offset=it;
            else if(seg_name=="mytab")tb.mytab=add;
            else if(seg_name=="etype")tb.etype=str;
            else if(seg_name=="dims")tb.dims=it;
            else if(seg_name=="dim[0]")tb.dim[0]=it;
```

```
            else if(seg_name=="dim[1]")tb.dim[1]=it;
            else if(seg_name=="dim[2]")tb.dim[2]=it;
            else if(seg_name=="dim[3]")tb.dim[3]=it;
            else if(seg_name=="dim[4]")tb.dim[4]=it;
            else if(seg_name=="dim[5]")tb.dim[5]=it;
            else if(seg_name=="base")tb.base=it;
        }
}
string lookup_t(string name) //在symtab的头两个符号表里查type
{
    Tab* tp=symtab.top();
    for(auto J:tp->list)
        if(J.name==name)return J.type;
    if(symtab.size()>1)
    {
        symtab.pop();
        for(auto J:symtab.top()->list)
            if(J.name==name)
            {
                sysmtab.push(tp);
                return J.type;
            }
        sysmtab.push(tp);
    }
    return "UNBOUND";
}
int lookup_d(string name,int i=-1) //在symtab的头两个符号表里查dims,dim[i]
{
    Tab* tp=symtab.top();
    for(auto J:tp->list)
        if(J.name==name)
        {
            if(i==-1)return J.dims;
            else return J.dim[i];
        }
    if(symtab.size()>1)
    {
        symtab.pop();
        for(auto J:symtab.top()->list)
            if(J.name==name)
            {
                sysmtab.push(tp);
                if(i==-1)return J.dims;
                else return J.dim[i];
            }
        sysmtab.push(tp);
    }
    return "UNBOUND";
}
```

```
string lookup_b(string name) //在symtab的头两个符号表里查base
{
    Tab* tp=symtab.top();
    for(auto J:tp->list)
        if(J.name==name)return J.base;
    if(symtab.size()>1)
    {
        symtab.pop();
        for(auto J:symtab.top()->list)
            if(J.name==name)
            {
                sysmtab.push(tp);
                return J.base;
            }
        sysmtab.push(tp);
    }
    return "UNBOUND";
}
void merge(Tab &A,Tab &B)
{
    for(auto J:B.list)
    {
        A.list.push_back(J);
        if(J.type=="ARRAY")A.list[(int)A.list.size()-1].base+=A.width;
        else A.list[(int)A.list.size()-1].offset+=A.width;
    }
    A.width+=B.width;
}
stack<Tab*> symtab;
vector<Tab*> tablist;

//辅助函数
void error(string msg)
{
    cout<<"错误："+msg;
    while(!symtab.empty())
        delete symtab.top(),symtab.pop();
    while(!tablist.empty())
        delete tablist.top(),tablist.pop();
    exit(0);
}
string to_str(int x)
{
    char a[20];
    int w=0,flg=0;
    if(x<0)flg=1,x=-x;
    while(x)
    {
        a[w++]=x%10+'0';
```

```
        x=x/10;
    }
    if(!w)a[w++]='0';
    if(flg)a[w++]='-'
    reverse(a,a+w);
    a[w++]=0;
    return string(a);
}
string gen(string a1,string a2="",string a3="",string a4="",string a5="",string
a6="",string a7="",string a8="")
{
    string c=a1;
    if(a2!="")c=c+" "+a2;
    if(a3!="")c=c+" "+a3;
    if(a4!="")c=c+" "+a4;
    if(a5!="")c=c+" "+a5;
    if(a6!="")c=c+" "+a6;
    if(a7!="")c=c+" "+a7;
    if(a8!="")c=c+" "+a8;
    return c+"\n";
}
int tot_var;
string newvar()
{
    return "t"+to_str(tot_var++);
}
int tot_label;
string newlabel()
{
    return "t"+to_str(tot_label++);
}
```

$E \to E + E$

```
E[0].place=newvar();
E[0].code=E[1].code+E[2].code+gen(E[0].place,"=",E[1].place,"+",E[2].place);
```

$E \to E * E$

```
E[0].place=newvar();
E[0].code=E[1].code+E[2].code+gen(E[0].place,"=",E[1].place,"*",E[2].place);
```

$E \to i$

```
string t=newvar();
E.place=t;
E.code=gen(t,"=",to_str(i.value));
```

$\hat{E} \rightarrow E$

```
Ě.place.push_back(E.place);
Ě.code.push_back(E.code);
```

$\hat{E} \rightarrow \hat{E}, E$

```
Ě[0].place=Ě[1].place;
Ě[0].place.push_back(E.place);
Ě[0].code=Ě[1].code;
Ě[0].code.push_back(E.code);
```

$\hat{I} \rightarrow i$

```
Ĭ.place.push_back(i.value);
```

$\hat{I} \rightarrow \hat{I}, i$

```
Ĭ[0].place=Ĭ[1].place;
Ĭ[0].place.push_back(i.value);
```

$B \rightarrow E$

```
string l1=newlabel(),l2=newlabel();
B.tc=l1;
B.fc=l2;
B.code=E.code+gen("if",E.place,"!=","0","then",l1,"else",l2);
```

$B \rightarrow E\ r\ E$

```
string l1=newlabel(),l2=newlabel();
B.tc=l1;
B.fc=l2;
B.code=E[1].code+E[2].code+gen("if",E[1].place,r.name,E[2].place,"then",l1,"else",l2);
```

$T \rightarrow int$

```
T.type="INT";
T.width=2;
```

$T \rightarrow float$

```
T.type="FLO";
T.width=2;
```

$T \to void$

```
T.type="VOID";
T.width=0;
```

根据文法，d可以定义为float类型，但其值都只能是整数，所以不用存和判断类型

$D \to T\ d$

```
D.tab=Tab();
bind(D.tab,d.name,T.type);
lookup_c(D.tab,d.name,"offset",0);
D.tab.width+=T.width;
D.place=d.name;
D.func_or_arr=false;
```

$D \to T\ d[\hat{I}]$

```
if(Ĭ.place.size()>6)error("数组维数不能超过6\n");
if(Ĭ.place.size()<1)error("数组维数不能小于1\n");
D.tab=Tab();
bind(D.tab,d.name,"ARRAY");
lookup_c(D.tab,d.name,"dims",Ĭ.place.size());
int size=1;
char ch[2]={'0',0};
for(auto J:Ĭ.place)
{
    lookup_c(D.tab,d.name,"dim["+string(ch)+"]",J);
    size*=c;
    ++ch[0];
}
lookup_c(D.tab,d.name,"etype",T.type);
lookup_c(D.tab,d.name,"base",0);
D.tab.width+=size*T.width;
D.place=d.name;
D.func_or_arr=true;
```

$\hat{D} \to \varepsilon$

```
symtab.push(new Tab());
Ď.func_or_arr=false;
```

$\hat{D} \to \hat{D}D;$

```
merge(*symtab.top(),D.tab);
Ď[0].place=Ď[1].place;
Ď[0].place.push_back(D.place);
if(D.func_or_arr||Ď[1].func_or_arr)
    Ď[0].func_or_arr=true;
else Ď[0].func_or_arr=false;
```

$D \to T\ d(\hat{D})\{\hat{D}\hat{S}\}$

```
if(Ď[0].func_or_arr)error("过程和数组不能作为其他过程的参数\n");
Tab* tab_inner=symtab.top();
symtab.pop();
Tab* tab=symtab.top();
symtab.pop();
merg_tab(*tab,*tab_inner);
delete tab_inner;
if(symtab.empty())tab->outer=NULL;
else tab->outer=symtab.top();
tab->args=Ď[0].place.size();
tab->arglist=Ď[0].place;
for(auto J:Š.code)
    tab->code=tab->code+J;
tab->rtype=T.type;
tablist.push_back(tab);

D.tab=Tab();
bind(D.tab,d.name,"FUNC");
lookup_c(D.tab,d.name,"offset",0);
lookup_c(D.tab,d.name,"mytab",tab);
D.tab.width+=8;
D.place=d.name;
D.func_or_arr=true;
```

$E \to d$

```
string type=lookup_t(d.name);
if(type!="INT"&&type!="FLO")error(d.name+"未声明或声明类型与调用方式不匹配\n");
E.place=d.name;
E.code="";
```

$E \to d[\hat{E}]$

```
string type=lookup_t(d.name);
if(type!="ARRAY")error(d.name+"未声明或声明类型与调用方式不匹配\n");
int dims=lookup_d(d.name,-1);
if(dims!=Ě.place.size())error(d.name+"数组下标个数错误\n");
```

```
for(auto J:Ě.code)E.code=E.code+J;
string t1=newvar();
E.code=E.code+gen(t1,"=",Ě.place[0]);
for(int i=1;i<dims;i++)
{
  string t=newvar();
  int dim=lookup(x,i);
  E.code=E.code+gen(t,"=",t1,"*",to_str(dim));
  t1=newvar();
  E.code=E.code+gen(t1,"=",t,"+",Ě.place[i]);
}
string t=newvar();
E.place=t;
string base=lookup_b(x);
E.code=E.code+gen(t,"=",t1,"[",base,"]")
```

$E \to d(\hat{E})$

```
string type=lookup_t(d.name);
if(type!="FUNC")error(d.name+"未声明或声明类型与调用方式不匹配\n");
string cd="";
for(auto J:Ě.code)cd=cd+J;
for(auto J:Ě.place)E.code=gen("par",J)+E.code;
E.code=E.code+gen("call",x,",",to_str(Ě.place.size()));
E.code=cd+E.code;
E.place="REG0";
```

$S \to d(\hat{E})$

```
string type=lookup_t(d.name);
if(type!="FUNC")error(d.name+"未声明或声明类型与调用方式不匹配\n");
string cd="";
for(auto J:Ě.code)cd=cd+J;
for(auto J:Ě.place)S.code=gen("par",J)+S.code;
S.code=S.code+gen("call",x,",",to_str(Ě.place.size()));
S.code=cd+S.code;
```

$S \to return\ E$

```
S.code=E.code+gen("return",E.place);
```

$S \to d = E$

```
string tp=lookup_t(d.name);
if(tp!="INT"&&tp!="FLO")error(d.name+"未声明或声明类型与调用方式不匹配\n");
S.code=E.code+gen(d.name,"=",E.place);
```

$S \rightarrow if\ (B)\ S$

```
S[0].code=B.code+gen("label",B.tc)+S[1].code+gen("label",B.fc);
```

$S \rightarrow if\ (B)\ S\ else\ S$

```
string l=newlabel();
S[0].code=B.code+gen("label",B.tc)+S[1].code+gen("goto",l)+gen("label",B.fc)+S[2].code+
gen("label",l);
```

$S \longrightarrow while(B)\ S$

```
string l=newlabel();
S[0].code=gen("label",l)+B.code+gen("label",B.tc)+S[1].code+gen("goto",l)+gen("label",B
.fc);
```

$S \rightarrow \{\hat{S}\}$

```
for(auto J:Š.code)S.code=S.code+J;
```

$\hat{S} \rightarrow S$

```
Š.code.push_back(S.code);
```

$\hat{S} \rightarrow \hat{S}; S$

```
Š[0].code=Š[1].code;
Š[0].code.push_back(S.code);
```