**0.**＜加法运算符＞ ::= +｜-  
＜乘法运算符＞  ::= \*｜/  
＜关系运算符＞  ::=  <｜<=｜>｜>=｜!=｜==  
＜字母＞   ::= ＿｜a｜．．．｜z｜A｜．．．｜Z  
＜数字＞   ::= ０｜＜非零数字＞  
＜非零数字＞  ::= １｜．．．｜９  
＜字符＞    ::=  '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'  
＜字符串＞   ::=  "｛十进制编码为32,33,35-126的ASCII字符｝"

分析：  
 在这一部分为文法的绝大部分终结符号

**1**.＜程序＞    ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞  
＜主函数＞    ::= void main‘(’‘)’ ‘{’＜复合语句＞‘}’

范例：  
 void main() //主函数  
 {  
 int a; //主函数中的复合语句，同时是变量定义  
 }

分析：  
 根据程序的文法得知，每一个程序按顺序由常量说明，变量说明，函数定义（可以是有返回值的或者无返回值的），主函数组成。  
 其中，常量说明、变量说明、函数定义、主函数之间有严格的先后顺序，不能打乱。而相邻的两个函数定义之间是可以相互打乱的。  
 对程序而言，主函数外的常量说明、变量说明以及函数定义是可有可无的，而其中常量说明和变量说明只有一个，并严格按照顺序，不能有[常量说明 变量说明 常量说明]这样的序列出现，而常量说明和变量说明内可以分别有能够打乱顺序的常量定义和变量定义，且数量没有限制。而函数定义的数量也没有限制。（除非超过了系统限制，而对于文法本身来说是没有限制的） 。而只有主函数这一个部分是一个程序所不可或缺的。

**2.**＜常量说明＞ ::=  const＜常量定义＞;{ const＜常量定义＞;}  
＜常量定义＞   ::=   int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}  
                            | char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}  
＜无符号整数＞  ::= ＜非零数字＞｛＜数字＞｝  
＜整数＞        ::= ［＋｜－］＜无符号整数＞｜０  
＜标识符＞    ::=  ＜字母＞｛＜字母＞｜＜数字＞｝

范例：  
 const int a = -1,b = +2,c = 0; //整型常量的定义  
 const char ch1 = ‘+’, ch2 = ‘\*’ , ch3 = ‘d’, ch4 = ‘6’; //字符型常量的定义  
 分析：  
 根据常量说明的文法，常量定义可以有1~n个，各个常量定义间没有顺序要求。  
 常量定义可以是整型常量定义，也可以是字符型常量定义，而在一个定义结束前，不能出现其他种类的定义，例如不能在整型变量定义中定义字符型变量。  
 标识符以字母开头且至少得有一个字母，而<字母>中包含下划线，所以也能以下划线开头。  
 整型变量定义中，整数可以为零，也可以为开头不为零的整数，而+0和-0的文法也是允许的，但不允许011这样以0开头的多位整数。  
 字符型变量定义中，字符可以为加法运算符、乘法运算符、字母、数字，范例都已给出样例。

**3.** ＜变量说明＞  ::= ＜变量定义＞;{＜变量定义＞;}  
＜变量定义＞  ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’) }  
＜类型标识符＞      ::=  int | char

范例：  
 int a,arr[5]; //整型变量定义，包括数组的示例  
 char c, carr[5]; //字符型变量定义，包括字符型数组的示例  
 分析：  
 变量说明中可以有多个变量定义，但至少得有一个，而变量定义分为整型变量定义和字符型变量定义。  
 变量定义中可以出现数组定义。

4.＜声明头部＞   ::=  int＜标识符＞|char＜标识符＞  
＜有返回值函数定义＞  ::=  ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’  
＜无返回值函数定义＞  ::= void＜标识符＞‘(’＜参数表＞‘)’‘{’＜复合语句＞‘}’  
＜复合语句＞   ::=  ［＜常量说明＞］［＜变量说明＞］＜语句列＞  
＜参数表＞    ::=  ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

范例：  
 int fun1(int num) //有声明头部和有返回值的函数定义，参数表不为空  
 {   
 return num+1; //返回语句  
 }  
 int fun2() //有声明头部和返回值的函数定义，参数表为空  
 {  
 return 0;  
 }  
 void fun3(char ch , int a) //无返回函数值的函数定义，参数表不为空  
 {  
 printf(“yes”); //写语句  
 }  
 分析：  
 函数定义有两种类型：有返回值的函数定义，无返回值的函数定义。其中，有返回值的函数定义可以返回的值的类型由声明头部中的int或char决定。而函数参数表可以为空，也可以有一或多个参数，参数前必须要有类型标识符。范例给出了典型的情况。

**5.**＜表达式＞    ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}  
＜项＞     ::= ＜因子＞{＜乘法运算符＞＜因子＞}  
＜因子＞    ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’

范例：

int a ,b ,c ,d; //变量说明

int fun1(int num) //有声明头部和有返回值的函数定义，参数表不为空  
 {   
 return num+1; //返回语句  
 }  
   
 a = 1,b = 1; //赋值语句  
 c = a ; // a 为表达式  
 d = - a + b \* (a / b) \* ‘q’ + a \* 7 + fun1(b); // - a + b \* a / b + a \* 7 + fun1(b)为表达式  
 分析：  
 从表达式的语法可知，表达式是用于计算的，表达式的字部分因子可以为标识符，整数，字符，有返回值的函数调用语句，(表达式)。而因子包括表达式说明表达式理论上是不限长度的。范例中对c的赋值的 a 为最简单的表达式之一，表达式最简单可以只包括一个项，而一个项可以只包括一个因子。而范例中对d的赋值的- a + b \*( a / b) + a \* 7 + fun1(b)为较为复杂的表达式，其中包含了各种表达式可能组成的样例。  
 注意一： - -a这样的表达式是不可行的，但 - -1是可行的

**6.**＜语句＞    ::= ＜条件语句＞｜＜循环语句＞｜<情况语句>|‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;   
                      |＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;｜＜返回语句＞;  
＜赋值语句＞   ::=  ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞  
＜条件语句＞  ::=  if ‘(’＜条件＞‘)’＜语句＞  
＜条件＞    ::=  ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为0条件为假，否则为真  
＜循环语句＞   ::=  do＜语句＞while ‘(’＜条件＞‘)’

＜常量＞   ::=  ＜整数＞|＜字符＞

＜情况语句＞  ::=  switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞ ‘}’  
＜情况表＞   ::=  ＜情况子语句＞{＜情况子语句＞}  
＜情况子语句＞  ::=  case＜常量＞：＜语句＞  
＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’  
＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’  
＜值参数表＞   ::= ＜表达式＞{,＜表达式＞}｜＜空＞  
＜语句列＞::=｛＜语句＞｝  
＜读语句＞    ::=  scanf ‘(’＜标识符＞{,＜标识符＞}‘)’  
＜写语句＞    ::=  printf‘(’＜字符串＞,＜表达式＞‘)’|printf ‘(’＜字符串＞‘)’|printf ‘(’＜表达式＞‘)’  
＜返回语句＞   ::=  return[‘(’＜表达式＞‘)’]  

范例：

int a , b, c , d[5], n ;  
  
 int fun1() //有声明头部和返回值的函数定义，参数表为空  
 {  
 return 0; //返回语句  
 }  
 void fun2(char ch , int a) //无返回函数值的函数定义，参数表不为空  
 {  
 printf(“yes”); //写语句  
 }  
  
 a = 0; //赋值语句  
 d[0] = 0; //赋值语句  
 n = 100; //赋值语句  
 { // 语句列  
 if( a < 5 ) //条件语句，()内为条件，表达式为0条件为假，否则为真   
 {  
 do  
 {  
 a = a + 1;  
 d[a] = a; //赋值语句

} while( a <= 5) //循环语句，()内为条件，表达式为0条件为假，否则为真

}  
scanf(b,c); //读语句  
swich ( b ) // 情况语句   
{ //情况表  
 case 1: //情况子语句，case后跟常量  
 printf(“case 1”,a+b ); //写语句的第一种形式，字符串加表达式  
 case 2:  
 printf(“case 2”); //写语句的第二种形式，单字符串  
 case 3:  
 printf(a+b); //写语句的第三种形式，单表达式

}   
n = fun1() //在赋值语句的表达式部分嵌入了有返回值函数调用语句，其中的值参数表为空  
fun2(“d”,d[0]+d[1]+d[2]+d[3]); //无返回值函数调用语句，值参数表不为空  
; //空语句

}  
分析：  
 语句部分的语法可以说是编译文法部分的核心了。在范例中已经给出了语句所能推出的各种形式。  
 语句能推出‘{’＜语句列＞‘}’，语句列中可以有一或无限个语句（理论上）。  
 赋值语句可以对变量进行赋值，也能对数组变量的单个元素进行赋值。  
 条件语句中条件为0条件为假，否则为真。后跟的语句可以是‘{’＜语句列＞‘}’。  
 循环语句中条件为0条件为假，否则为真。后跟的语句可以是‘{’＜语句列＞‘}’，且一定会执行一次。  
 情况语句的情况表中至少得有一个情况子语句，即至少出现一个case，case后面的常量只允许出现int和char类型，每个情况子语句执行完毕后，不继续执行后面的情况子语句。  
 在语句列中能够直接有＜有返回值函数调用语句＞和＜无返回值函数调用语句＞，也可以将＜有返回值函数调用语句＞放在赋值语句中。  
 写语句有三种形式。  
 返回语句能直接出现在语句列中，但一般都是在函数中。  
 复合语句中［＜常量说明＞］［＜变量说明＞］＜语句列＞不能调换顺序，需要严格遵守，即所有常量变量，都需要在语句之前被说明定义。

附加说明：

（1）char类型的表达式，用字符的ASCII码对应的整数参加运算，在写语句中输出字符

（2）标识符不区分大小写字母

（3）写语句中的字符串原样输出

（4）数组的下标从0开始

（5）情况语句中，switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句