＜加法运算符＞ ::= +|-

范例：

const int x = -1, y = 1, a = 2;

int r;

r = a - x + y;

分析：

普通的加号和减号，注意到由于加号减号运算的优先级一致，计算时的操作也十分相似，因此归到一类来处理。注意若运算结果超过了char的取值范围，注意容错时进行截断操作。

＜乘法运算符＞ ::= \*|/

范例：

const int x = -1, a = 2;

int r;

r = a / x \* a;

分析：

普通的乘号和除号，同样由于二者运算优先级一致，计算时的操作也十分相似，也归到一类来处理。同样注意char的容错。

＜关系运算符＞ ::= < | <= | > | >= | != | ==

范例：

const int x = -1, a = 2;

int r;

r = 0;

if (a > 0)

{

if(x <= 0)

r = a / x \* a;

}

if (r != 0)

{

if(r == -4)

r = r + 1;

}

if (r < 0)

{

if(a >= 0)

r = r + a;

}

分析：

数字大小比较判断符号。关系运算符与C语言中的保持一致，char按照存的ASCII码进行比较。

＜字母＞ ::= \_|a| ... |z|A| ... |Z

范例：

int rA\_, int \_ss;

分析：

本文法中的字母。与C语言一致，在标识符定义时不允许数字打头，允许下划线’\_’，为方便起见，将下划线与字母归到一类。

＜数字＞ ::= 0|1| ... |9

范例：

const int x = -1, a = 2;

int r;

r = a / x \* a;

分析：

阿拉伯数字0-9，定义了本文法使用的数字。

＜字符＞ ::= '＜加法运算符＞'|'＜乘法运算符＞'|'＜字母＞'|'＜数字＞'

范例：

const char a1 = 'a', b1 = 'B', mul = '\*', udl = '\_', zero = '0', sub = '-';

分析：

字符在常量定义时作为常量值使用。常量定义时和表达式中的字符只能是以上的范围，范围比字符串允许的范围小（不允许!@#$%^&\*(){}[];'"<>,.?\|~）

＜字符串＞ ::= "｛十进制编码为32,33,35-126的ASCII字符｝"

范例：

printf("!@#$%abcdABCD963287~;: ");

分析：

声明了本文法可以识别的字符串。不允许双引号、控制字符，允许空格。

＜有返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况  
＜无返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况  
＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}  
＜返回语句＞ ::= return['('＜表达式＞')']

＜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＞

范例：

const int a = 1;

char b;

int fun1(int x)

{

Return(x);

}

void fun2

{

printf("114514");

}

void main()

{

int aa;

aa = 8;

printf("!@#$%abcdABCD963287~;: ");

printf(fun1(a));

fun2;

}

分析：

说明了函数的调用结构，以及返回语句的组成，以及程序体本身的基本结构。程序开头先声明/定义变量常量，定义函数；之后在主函数内也有可能存在的变量/常量的声明/定义，以及函数的调用，主函数内不能定义函数，主函数外部不能进行表达式的计算。函数调用时，有参数情况和C语言的一致，无参数情况则无需括号。返回语句中返回值可以不指定，返回语句末尾的分号在语句中有指定，注意必须用用小括号括起返回值。

＜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＞}

范例：

const char a1 = 'a', b1 = 'B';

const int m = 4, n = 9;

分析：

描述了常量的说明定义方式。一个分号前只能定义一种常量，不能混合int和char。常量定义不能用表达式或是有返回值函数来赋值，只能用字面常量。

＜无符号整数＞ ::= ＜数字＞｛＜数字＞｝  
＜整数＞ ::= [+|-]＜无符号整数＞

范例：

const int m = +04, n = 9, p = -003;

分析：

交代了整数及其符号的具体情况。带符号的整数由无符号整数带上可能存在的+/-符号得来。注意数字允许前导0。

＜标识符＞ ::= ＜字母＞｛＜字母＞|＜数字＞｝

＜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＞']' )}

//＜无符号整数＞表示数组元素个数，值需大于0

＜类型标识符＞ ::= int | char

范例：

int m1[8], \_2p;

char a[6], c[9], q;

分析：

交代了标识符和变量的声明情况。变量定义不能带初始赋值。支持数组定义。由于字母可以推出下划线符号'\_'，标识符与C语言的规定保持一致。

＜声明头部＞ ::= int＜标识符＞ | char＜标识符＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

＜有返回值函数定义＞ ::=

＜声明头部＞'('＜参数表＞')' '{'＜复合语句＞'}'|＜声明头部＞'{'＜复合语句＞'}'

//第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数定义＞ ::=

void＜标识符＞('＜参数表＞')''{'＜复合语句＞'}'| void＜标识符＞{'＜复合语句＞'}' //第一种选择为有参数的情况，第二种选择为无参数的情况

范例：

int m(int a, int b, int c)

{

//省略

}

char n(char a, int b, int c)

{

//省略

}

int p

{

//省略

}

char a

{

//省略

}

void pp(int x)

{

//省略

}

void nu

{

//省略

}

分析：

描述了函数的定义结构。有参数的列表的函数跟C语言的比较相似，无参数的要注意参数列表为空时不需要括号。注意标识符不分大小写，实现中可以将读到的标识符统一处理成小写。有返回值的要有”return(x);”，而无返回值的不能有，可以不写返回语句或只写return;

＜复合语句＞ ::= [＜常量说明＞][＜变量说明＞]＜语句列＞  
＜主函数＞ ::= void main'('')''{'＜复合语句＞'}'

范例：

void main()

{

int a;

a = 1;

}

分析：

描述了主函数的基本结构。主函数限定void，其他部分和C语言较为相似。常量说明和变量说明必须保持顺序。void和main间的空格/制表符可以不止一个。

＜表达式＞    ::= [+|-]＜项＞{＜加法运算符＞＜项＞}  //[+|-]只作用于第一个<项>  
＜项＞     ::= ＜因子＞{＜乘法运算符＞＜因子＞}  
＜因子＞    ::= ＜标识符＞|＜标识符＞'['＜表达式＞']'|'('＜表达式＞')'|＜整数＞|＜字符＞|＜有返回值函数调用语句＞  
范例：

int a;

const int b = 1, c = 2, d = 3, e = 4;

char ar[2];

ar[1] = '0';

a = -b \* ar[1] + (c - e \* 9) + 2 \* 'm';

分析：

文法的表达式的结构。和C语言较为相似，注意开头的+/-是首项的符号。

＜语句＞ ::= ＜条件语句＞|＜循环语句＞|'{'＜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＞;  
＜语句列＞ ::= ｛＜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＞[else＜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＞ ::= while '('＜条件＞')'＜语句＞ |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞  
＜步长＞::= ＜无符号整数＞

范例：

const int b = 1, c = 2, d = 3, e = 4;

int a, x;

char ar[2];

a[1] = '0';

a = -b \* ar[1] + (c - e \* 9) + 2 \* 'm'

if(a > 0)

{

a = a \* 8;

}

else

{

a = a \* -2;

}

if (a)

{

a = a + 1;

}

x = 0;

for(a = -2; a<d; a = a+1)

{

x = x + e;

}

while (x > 0)

{

x = x - c;

}

分析：

文法允许出现的语句。循环步长不能是表达式，不能带有+/-号，因为之前部分已经带有，且循环指示只能是加减变化，不可用函数调用的返回值，不可乘除。for的流程为：指示变量初始化，执行循环体，指示变量加或减步长，判断指示变量是否满足条件。语句中不能有常量变量的声明定义，只能进行表达式运算、赋值、读写、if/循环、函数调用。注意区分复合语句和语句列，区别在于是否允许常量变量的说明。空语句的情况，则兼容了分号后紧跟着仍为分号的情况。

＜读语句＞ ::= scanf '('＜标识符＞{,＜标识符＞}')'  
范例：

int x, y, z;

scanf(x,y,z);

分析：

文法定义的程序接受输入的方式。按照变量声明的类型对输入流中的数据进行读取与存储。

＜写语句＞ ::= printf '('＜字符串＞,＜表达式＞')'| printf '('＜字符串＞')'| printf '('＜表达式＞')'

范例：

printf("114514");

printf(114514+1919\*810);

printf("114514", 114514+1919\*810);

分析：

文法定义的输出信息的方式。表达式中，int按照数值输出，char按照字符输出，注意字符串和表达式都有的情况下，只能是先字符串后表达式，且不支持多字符串输出，不支持多表达式输出。   
  
附加说明：  
（1）char类型的变量或常量，用字符的ASCII码对应的整数参加运算  
（2）标识符不区分大小写字母  
（3）写语句中，字符串原样输出，单个字符类型的变量或常量输出字符，其他表达式整型输出   
（4）数组的下标从0开始  
（5）for语句先执行一次循环体中的语句再进行循环变量是否越界的测试