*红色标示的是我们对 BNF 的改动

```
program         ->      external_decls
external_decls  ->      declaration external_decls | function_list
declaration     ->      EXTERN type_name var_list ;
                        | REGISTER type_name var_list ;
                        | type_name var_list ;
function_list   ->      function_list function_def | function_def
type_name       ->      VOID | INT | CHAR
var_list        ->      var_list , var_item | var_item
var_item        ->      array_var | scalar_var | * scalar_var
array_var       ->      IDENT [ ICONSTANT ]
scalar_var      ->      IDENT | IDENT ( parm_type_list )
function_def    ->      function_hdr { function_body }
function_hdr    ->      type_name IDENT ( parm_type_list )
                        | type_name * IDENT ( parm_type_list )
                        | IDENT ( parm_type_list )
parm_type_list          ->      EPSILON | VOID | parm_list
parm_list       ->      parm_list , parm_decl | parm_decl
parm_decl       ->      type_name IDENT | type_name * IDENT
function_body   ->      internal_decls statement_list
internal_decls  ->      EPSILON | declaration internal_decls
statement_list  ->      EPSILON | statement statement_list
statement       ->      compoundstmt
                        | nullstmt
                        | expression_stmt
                        | ifstmt
                        | for_stmt
                        | while_stmt
                        | return_stmt
compoundstmt            ->      { internal_decls statement_list }
nullstmt        ->      ;
expression_stmt         ->      expression ;
ifstmt          ->      IF ( expression ) statement
                        | IF ( expression ) statement ELSE statement
for_stmt        ->      FOR ( expression ; expression ; expression ) statement
while_stmt      ->      WHILE ( expression ) statement
return_stmt     ->      RETURN expression ; | RETURN ;
expression      ->      rvalue | assignment_expression
assignment_expression
                ->      lvalue = assignment_expression | lvalue = rvalue
lvalue          ->      * rvalue | IDENT | IDENT [ expression ]
rvalue          ->      lvalue
                        | rvalue + rvalue
                        | rvalue - rvalue
                        | rvalue * rvalue
                        | rvalue op rvalue
                        | ( rvalue )
                        | + rvalue
                        | - rvalue
                        | ! rvalue
                        | & lvalue
```

| DOUBLE_OP lvalue
|lvalue DOUBLE_OP
| constant
| IDENT (argument_list)
|IDENT ( )

op -> BOOLEAN_OP | REL_OP

constant -> ICONSTANT | CHAR_CONSTANT | STRING_CONSTANT

argument_list -> argument_list , expression | expression

*以下“兼容”指该产生式生成的文法类是原文法同一条产生式产生的文法类的超集。

改动：

1.

compoundstmt -> { internal_decls statement_list }

为了支持在 scope 中声明变量或函数（C 支持这一特性）——兼容原来的文法，我们的符号表也支持在这种嵌套格式下的查询

2.

expression -> rvalue | assignment_expression

为了区分表达式和赋值语句：因为赋值语句是“左值 = 表达式”的形式（expression 是广义的表达式，包含赋值语句，但下面所说的“表达式”均指 rvalue）

3.

lvalue -> * rvalue | IDENT | IDENT [expression]

rvalue -> lvalue

|...

rvalue 和 lvalue 的定义：

rvalue 是任何能计算出值的表达式

lvalue 是某一内存空间的标识，比如：变量名、数组元素或对某一指针取参考(*rvalue)

例：lvalue 不能是如下表示，因为它们均不能被赋值：

++a, a--, +a, !a, &a，a+b

我们的表达式定义避免了上述的不合理的语句，并且支持如下的 C 语法规范（均通过 gcc 编译）：

例如：

a=b=a+b （连续赋值）

a + - + - + - + b （正负号在表达式中同加减法连用）

c[a=b] （赋值语句的值就是等号一端表达式的值）