

```

/* -- Mode: C --
*
* Cforall Grammar Version 1.0, Copyright (C) Peter A. Buhr 2001 – Permission is granted to copy this
*   grammar and to use it within software systems. THIS GRAMMAR IS PROVIDED “AS IS” AND WITHOUT
*   ANY EXPRESS OR IMPLIED WARRANTIES.
*
* cfa.y –
*
* Author : Peter A. Buhr
* Created On : Sat Sep 1 20:22:55 2001
* Last Modified By : Peter A. Buhr
* Last Modified On : Fri Dec 12 14:09:35 2003
* Update Count : 877
*/

/* This grammar is based on the ANSI99 C grammar, specifically parts of EXPRESSION and STATEMENTS, and on the
   C grammar by James A. Roskind, specifically parts of DECLARATIONS and EXTERNAL DEFINITIONS. While parts
   have been copied, important changes have been made in all sections; these changes are sufficient to
   constitute a new grammar. In particular, this grammar attempts to be more syntactically precise, i.e., it
   parses less incorrect language syntax that must be subsequently rejected by semantic checks. Nevertheless,
   there are still several semantic checks required and many are noted in the grammar. Finally, the grammar
   is extended with GCC and CFA language extensions. */

/* Acknowledgments to Richard Bilson, Glen Ditchfield, and Rodolfo Gabriel Esteves who all helped when I got
   stuck with the grammar. */

/* The root language for this grammar is ANSI99 C. All of ANSI99 is parsed, except for:

   1. designation with ‘=’ (use ‘.’ instead)

   Most of the syntactic extensions from ANSI90 to ANSI99 C are marked with the comment “ANSI99”. This grammar
   also has two levels of extensions. The first extensions cover most of the GCC C extensions, except for:

   1. nested functions
   2. generalized lvalues
   3. designation with and without ‘=’ (use ‘.’ instead)
   4. attributes not allowed in parenthesis of declarator

   All of the syntactic extensions for GCC C are marked with the comment “GCC”. The second extensions are for
   Cforall (CFA), which fixes several of C’s outstanding problems and extends C with many modern language
   concepts. All of the syntactic extensions for CFA C are marked with the comment “CFA”. As noted above,
   there is one unreconcilable parsing problem between ANSI99 and CFA with respect to designators; this is
   discussed in detail before the “designation” grammar rule. */

/***** TERMINAL TOKENS *****/

/* keywords */
%token TYPEDEF
%token AUTO EXTERN REGISTER STATIC
%token INLINE /* ANSI99 */
%token FORTRAN /* ANSI99, extension ISO/IEC 9899:1999 Section J.5.9(1) */
%token CONST VOLATILE
%token RESTRICT /* ANSI99 */
%token FORALL LVALUE /* CFA */
%token VOID CHAR SHORT INT LONG FLOAT DOUBLE SIGNED UNSIGNED
%token BOOL COMPLEX IMAGINARY /* ANSI99 */
%token TYPEOF LABEL /* GCC */
%token ENUM STRUCT UNION

```

```

%token TYPE FTYPE DTYPE CONTEXT /* CFA */
%token SIZEOF
%token ALIGNOF ATTRIBUTE EXTENSION /* GCC */
%token IF ELSE SWITCH CASE DEFAULT DO WHILE FOR BREAK CONTINUE GOTO RETURN
%token CHOOSE FALLTHRU TRY CATCH FINALLY THROW /* CFA */
%token ASM /* ANSI99, extension ISO/IEC 9899:1999 Section J.5.10(1) */

```

*/\* names and constants: lexer differentiates between identifier and typedef names \*/*

```

%token<tok> IDENTIFIER TYPEDEFname TYPEGENname
%token<tok> ATTR_IDENTIFIER ATTR_TYPEDEFname ATTR_TYPEGENname
%token<tok> INTEGERconstant FLOATINGconstant CHARACTERconstant STRINGliteral
%token<tok> ZERO ONE /* CFA */

```

*/\* multi-character operators \*/*

```

%token ARROW /* -> */
%token ICR DECR /* ++ - */
%token LS RS /* << >> */
%token LE GE EQ NE /* <= >= == != */
%token ANDAND OROR /* && || */
%token ELLIPSIS /* ... */

%token MULTassign DIVassign MODassign /* *= /= %= */
%token PLUSassign MINUSassign /* += -= */
%token LSassign RSassign /* <<= >>= */
%token ANDassign ERassign ORassign /* &= ^= |= */

```

*/\* Types declaration \*/*

*/\* Handle single shift/reduce conflict for dangling else by shifting the ELSE token. For example, this string is ambiguous:*

```

. _____ matches IF '( comma_expression )' statement
if ( C ) S1 else S2
` _____ matches IF '( comma_expression )' statement ELSE statement */

```

*%nonassoc THEN /\* rule precedence for IF '( comma\_expression )' statement \*/*

*%nonassoc ELSE /\* token precedence for start of else clause in IF statement \*/*

*%start translation\_unit /\* parse-tree root \*/*

*%%*

*/\* \*\*\*\*\* Namespace Management \*\*\*\*\* \*/*

*/\* The grammar in the ANSI C standard is not strictly context-free, since it relies upon the distinct terminal symbols "identifier" and "TYPEDEFname" that are lexically identical. While it is possible to write a purely context-free grammar, such a grammar would obscure the relationship between syntactic and semantic constructs. Hence, this grammar uses the ANSI style.*

*Cforall compounds this problem by introducing type names local to the scope of a declaration (for instance, those introduced through "forall" qualifiers), and by introducing "type generators" – parametrized types. This latter type name creates a third class of identifiers that must be distinguished by the scanner.*

*Since the scanner cannot distinguish among the different classes of identifiers without some context information, it accesses a data structure (the TypedefTable) to allow classification of an identifier that it has just read. Semantic actions during the parser update this data structure when the class of identifiers change.*

*Because the Cforall language is block-scoped, there is the possibility that an identifier can change its*

class in a local scope; it must revert to its original class at the end of the block. Since type names can be local to a particular declaration, each declaration is itself a scope. This requires distinguishing between type names that are local to the current declaration scope and those that persist past the end of the declaration (i.e., names defined in “typedef” or “type” declarations).

The non-terminals “push” and “pop” derive the empty string; their only use is to denote the opening and closing of scopes. Every push must have a matching pop, although it is regrettable the matching pairs do not always occur within the same rule. These non-terminals may appear in more contexts than strictly necessary from a semantic point of view. Unfortunately, these extra rules are necessary to prevent parsing conflicts – the parser may not have enough context and look-ahead information to decide whether a new scope is necessary, so the effect of these extra rules is to open a new scope unconditionally. As the grammar evolves, it may be necessary to add or move around “push” and “pop” nonterminals to resolve conflicts of this sort. \*/

push:

;

pop:

;

/\*\*\*\*\*\* CONSTANTS \*\*\*\*\*/

constant:

/\* ENUMERATION constant is not included here; it is treated as a variable with type “enumeration constant”. \*/

INTEGERconstant  
| FLOATINGconstant  
| CHARACTERconstant  
;

identifier:

IDENTIFIER  
| ATTR\_IDENTIFIER /\* CFA \*/  
| zero\_one /\* CFA \*/  
;

no\_01\_identifier:

IDENTIFIER  
| ATTR\_IDENTIFIER /\* CFA \*/  
;

no\_attr\_identifier:

IDENTIFIER  
;

zero\_one:

/\* CFA \*/

ZERO  
| ONE  
;

string\_literal\_list:

/\* juxtaposed strings are concatenated \*/

STRINGliteral  
| string\_literal\_list STRINGliteral  
;

/\*\*\*\*\*\* EXPRESSIONS \*\*\*\*\*/

primary\_expression:

```

IDENTIFIER                                     /* typedef name cannot be used as a variable name */
| zero_one
| constant
| string_literal_list
| '(' comma_expression ')'
| '(' compound_statement ')'                  /* GCC, lambda expression */
;

postfix_expression:
primary_expression
| postfix_expression '[' assignment_expression ']'
      /* CFA, comma_expression disallowed in the context because it results in a common user error:
      subscripting a matrix with x[i,j] instead of x[i][j]. While this change is not backwards
      compatible, there seems to be little advantage to this feature and many disadvantages. It
      is possible to write x[(i,j)] in CFA, which is equivalent to the old x[i,j]. */
| postfix_expression '(' argument_expression_list ')'
| postfix_expression '.' no_attr_identifier
| postfix_expression '.' '[' field_list ']'          /* CFA, tuple field selector */
| postfix_expression ARROW no_attr_identifier
| postfix_expression ARROW '[' field_list ']'        /* CFA, tuple field selector */
| postfix_expression ICR
| postfix_expression DECR
      /* GCC has priority: cast_expression */
| '(' type_name_no_function ')' '{' initializer_list comma_opt '}' /* ANSI99 */
;

argument_expression_list:
argument_expression
| argument_expression_list ',' argument_expression
;

argument_expression:
/* empty */                                     /* use default argument */
| assignment_expression
| no_attr_identifier ':' assignment_expression
      /* Only a list of no_attr_identifier_or_typedef_name is allowed in this context. However, there
      is insufficient look ahead to distinguish between this list of parameter names and a tuple,
      so the tuple form must be used with an appropriate semantic check. */
| '[' assignment_expression ']' ':' assignment_expression
| '[' assignment_expression ',' tuple_expression_list ']' ':' assignment_expression
;

field_list:                                     /* CFA, tuple field selector */
field
| field_list ',' field
;

field:                                          /* CFA, tuple field selector */
no_attr_identifier
| no_attr_identifier '.' field
| no_attr_identifier '.' '[' field_list ']'
| no_attr_identifier ARROW field
| no_attr_identifier ARROW '[' field_list ']'
;

unary_expression:
postfix_expression
| ICR unary_expression

```

```

| DECR unary_expression
| EXTENSION cast_expression /* GCC */
| unary_operator cast_expression
| '!' cast_expression
| '*' cast_expression /* CFA */
    /* '*' is separated from unary_operator because of shift/reduce conflict in:
       { *X; } // dereference X
       { *int X; } // CFA declaration of pointer to int
       '&' must be moved here if C++ reference variables are supported. */
| SIZEOF unary_expression
| SIZEOF '(' type_name_no_function ')'
| ATTR_IDENTIFIER
| ATTR_IDENTIFIER '(' type_name ')'
| ATTR_IDENTIFIER '(' argument_expression ')'
| ALIGNOF unary_expression /* GCC, variable alignment */
| ALIGNOF '(' type_name_no_function ')' /* GCC, type alignment */
| ANDAND no_attr_identifier /* GCC, address of label */
;

```

unary\_operator:

```

'&'
| '+'
| '-'
| '~'
;

```

cast\_expression:

```

unary_expression
| '(' type_name_no_function ')' cast_expression
| '(' type_name_no_function ')' tuple
;

```

multiplicative\_expression:

```

cast_expression
| multiplicative_expression '*' cast_expression
| multiplicative_expression '/' cast_expression
| multiplicative_expression '%' cast_expression
;

```

additive\_expression:

```

multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;

```

shift\_expression:

```

additive_expression
| shift_expression LS additive_expression
| shift_expression RS additive_expression
;

```

relational\_expression:

```

shift_expression
| relational_expression '<' shift_expression
| relational_expression '>' shift_expression
| relational_expression LE shift_expression
| relational_expression GE shift_expression
;

```

```

equality_expression:
    relational_expression
    | equality_expression EQ relational_expression
    | equality_expression NE relational_expression
    ;

AND_expression:
    equality_expression
    | AND_expression '&' equality_expression
    ;

exclusive_OR_expression:
    AND_expression
    | exclusive_OR_expression '^' AND_expression
    ;

inclusive_OR_expression:
    exclusive_OR_expression
    | inclusive_OR_expression '|' exclusive_OR_expression
    ;

logical_AND_expression:
    inclusive_OR_expression
    | logical_AND_expression ANDAND inclusive_OR_expression
    ;

logical_OR_expression:
    logical_AND_expression
    | logical_OR_expression OROR logical_AND_expression
    ;

conditional_expression:
    logical_OR_expression
    | logical_OR_expression '?' comma_expression ':' conditional_expression
    | logical_OR_expression '?' /* empty */ ':' conditional_expression /* GCC, omitted first operand */
    | logical_OR_expression '?' comma_expression ':' tuple /* CFA, tuple expression */
    ;

constant_expression:
    conditional_expression
    ;

assignment_expression:
    /* CFA, assignment is separated from assignment_operator to ensure no assignment operations
    for tuples */
    conditional_expression
    | unary_expression '=' assignment_expression
    | unary_expression assignment_operator assignment_expression
    | tuple assignment_opt /* CFA, tuple expression */
    ;

assignment_expression_opt:
    /* empty */
    | assignment_expression
    ;

tuple:
    /* CFA, tuple */

```

```

        /* CFA, one assignment_expression is factored out of comma_expression to eliminate a
        shift/reduce conflict with comma_expression in new_identifier_parameter_array and
        new_abstract_array */
    '[' '' ]'
    | '[' assignment_expression ']'
    | '[' '' ',' tuple_expression_list ']'
    | '[' assignment_expression ',' tuple_expression_list ']'
    ;

tuple_expression_list:
    assignment_expression_opt
    | tuple_expression_list ',' assignment_expression_opt
    ;

assignment_operator:
    MULTassign
    | DIVassign
    | MODassign
    | PLUSassign
    | MINUSassign
    | LSassign
    | RSassign
    | ANDassign
    | ERassign
    | ORassign
    ;

comma_expression:
    assignment_expression
    | comma_expression ',' assignment_expression
    ;

comma_expression_opt:
    /* empty */
    | comma_expression
    ;

/***** STATEMENTS *****/

statement:
    labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | exception_statement
    | asm_statement
    ;

labeled_statement:
    no_attr_identifier ':' attribute_list_opt statement
    ;

compound_statement:
    '{' '}'
    | '{' label_declaration_opt                               /* GCC, local labels */
      block_item_list '}'                                     /* ANSI99, intermix declarations and statements */
    ;

```

```

;

block_item_list:                               /* ANS199 */
    block_item
    | block_item_list block_item
;

block_item:
    declaration                               /* CFA, new & old style declarations */
    | EXTENSION declaration                  /* GCC */
    | statement
;

statement_list:
    statement
    | statement_list statement
;

expression_statement:
    comma_expression_opt ';'
;

selection_statement:
    IF '(' comma_expression ')' statement      %prec THEN
        /* explicitly deal with the shift/reduce conflict on if/else */
    | IF '(' comma_expression ')' statement ELSE statement
    | SWITCH '(' comma_expression ')' case_clause /* CFA */
    | SWITCH '(' comma_expression ')' '{' declaration_list_opt switch_clause_list_opt '}' /* CFA */
        /* The semantics of the declaration list is changed to include any associated initialization,
           which is performed *before* the transfer to the appropriate case clause. Statements after
           the initial declaration list can never be executed, and therefore, are removed from the
           grammar even though C allows it. */
    | CHOOSE '(' comma_expression ')' case_clause /* CFA */
    | CHOOSE '(' comma_expression ')' '{' declaration_list_opt choose_clause_list_opt '}' /* CFA */
;

/* CASE and DEFAULT clauses are only allowed in the SWITCH statement, precluding Duff's device. In addition, a
   case clause allows a list of values and subranges. */

case_value:
    constant_expression                       /* CFA */
    | constant_expression ELLIPSIS constant_expression /* GCC, subrange */
    | subrange                               /* CFA, subrange */
;

case_value_list:
    case_value                               /* CFA */
    | case_value_list ',' case_value
;

case_label:
    CASE case_value_list ':'                 /* CFA */
    | DEFAULT ':'
        /* A semantic check is required to ensure only one default clause per switch/choose
           statement. */
;

case_label_list:
    /* CFA */

```

```

        case_label
        | case_label_list case_label
        ;

case_clause:                                     /* CFA */
        case_label_list statement
        ;

switch_clause_list_opt:                         /* CFA */
        /* empty */
        | switch_clause_list
        ;

switch_clause_list:                             /* CFA */
        case_label_list statement_list
        | switch_clause_list case_label_list statement_list
        ;

choose_clause_list_opt:                         /* CFA */
        /* empty */
        | choose_clause_list
        ;

choose_clause_list:                             /* CFA */
        case_label_list fall_through
        | case_label_list statement_list fall_through_opt
        | choose_clause_list case_label_list fall_through
        | choose_clause_list case_label_list statement_list fall_through_opt
        ;

fall_through_opt:                               /* CFA */
        /* empty */
        | fall_through
        ;

fall_through:                                   /* CFA */
        FALLTHRU
        | FALLTHRU ';'
        ;

iteration_statement:
        WHILE '(' comma_expression ')' statement
        | DO statement WHILE '(' comma_expression ')' ';'
        | FOR '(' for_control_expression ')' statement
        ;

for_control_expression:
        comma_expression_opt ';' comma_expression_opt ';' comma_expression_opt
        | declaration comma_expression_opt ';' comma_expression_opt /* ANS199 */
        /* Like C++, the loop index can be declared local to the loop. */
        ;

jump_statement:
        GOTO no_attr_identifier ';'
        | GOTO '*' comma_expression ';'                                     /* GCC, computed goto */
        /* The syntax for the GCC computed goto violates normal expression precedence, e.g.,
           goto *i+3; => goto *(i+3); whereas normal operator precedence yields goto (*i)+3; */
        | CONTINUE ';'

```

```

        /* A semantic check is required to ensure this statement appears only in the body of an
        iteration statement. */
| CONTINUE no_attr_identifier ';' /* CFA, multi-level continue */
        /* A semantic check is required to ensure this statement appears only in the body of an
        iteration statement, and the target of the transfer appears only at the start of an
        iteration statement. */
| BREAK ';'
        /* A semantic check is required to ensure this statement appears only in the body of an
        iteration statement. */
| BREAK no_attr_identifier ';' /* CFA, multi-level exit */
        /* A semantic check is required to ensure this statement appears only in the body of an
        iteration statement, and the target of the transfer appears only at the start of an
        iteration statement. */
| RETURN comma_expression_opt ';'
| THROW assignment_expression ';'
| THROW ';'
;

```

exception\_statement:

```

TRY compound_statement handler_list
| TRY compound_statement finally_clause
| TRY compound_statement handler_list finally_clause
;

```

handler\_list:

```

        /* There must be at least one catch clause */
handler_clause
        /* ISO/IEC 9899:1999 Section 15.3(6) If present, a “...” handler shall be the last handler for
        its try block. */
| CATCH '(' ELLIPSIS ')' compound_statement
| handler_clause CATCH '(' ELLIPSIS ')' compound_statement
;

```

handler\_clause:

```

CATCH '(' exception_declaration ')' compound_statement
| handler_clause CATCH '(' exception_declaration ')' compound_statement
;

```

finally\_clause:

```

FINALLY compound_statement
;

```

exception\_declaration:

*/\* A semantic check is required to ensure type\_specifier does not create a new type, e.g.:*

```

        catch ( struct { int i; } x ) ...

```

*This new type cannot catch any thrown type because of name equivalence among types. \*/*

```

type_specifier
| type_specifier declarator
| type_specifier variable_abstract_declarator
| new_abstract_declarator_tuple no_attr_identifier /* CFA */
| new_abstract_declarator_tuple /* CFA */
;

```

asm\_statement:

```

ASM type_qualifier_list_opt '(' constant_expression ')' ';'
| ASM type_qualifier_list_opt '(' constant_expression ':' asm_operands_opt ')' ';' /* remaining GCC */

```

```

        | ASM type_qualifier_list_opt '(' constant_expression ':' asm_operands_opt ':' asm_operands_opt ')' ';'
        | ASM type_qualifier_list_opt '(' constant_expression ':' asm_operands_opt ':' asm_operands_opt ':'
          asm_clobbers_list ')' ';'
    ;

asm_operands_opt:                               /* GCC */
    /* empty */
    | asm_operands_list
    ;

asm_operands_list:                              /* GCC */
    asm_operand
    | asm_operands_list ',' asm_operand
    ;

asm_operand:                                    /* GCC */
    STRINGliteral '(' constant_expression ')'
    ;

asm_clobbers_list:                              /* GCC */
    STRINGliteral
    | asm_clobbers_list ',' STRINGliteral
    ;

/***** DECLARATIONS *****/

declaration_list_opt:                           /* used at beginning of switch statement */
    | declaration_list
    ;

declaration_list:
    declaration
    | declaration_list declaration
    ;

old_declaration_list_opt:                       /* used to declare parameter types in K&R style functions */
    | old_declaration_list
    ;

old_declaration_list:
    old_declaration
    | old_declaration_list old_declaration
    ;

label_declaration_opt:                          /* GCC, local label */
    /* empty */
    | label_declaration_list
    ;

label_declaration_list:                         /* GCC, local label */
    LABEL label_list ';'
    | label_declaration_list LABEL label_list ';'
    ;

label_list:                                     /* GCC, local label */
    no_attr_identifier_or_typedef_name
    | label_list ',' no_attr_identifier_or_typedef_name
    ;

```

```

declaration:                                     /* CFA, new & old style declarations */
    new_declaration
    | old_declaration
    ;

```

*/\* C declaration syntax is notoriously confusing and error prone. Cforall provides its own type, variable and function declarations. CFA declarations use the same declaration tokens as in C; however, CFA places declaration modifiers to the left of the base type, while C declarations place modifiers to the right of the base type. CFA declaration modifiers are interpreted from left to right and the entire type specification is distributed across all variables in the declaration list (as in Pascal). ANSI C and the new CFA declarations may appear together in the same program block, but cannot be mixed within a specific declaration.*

```

        CFA      C
    [ 10 ] int x;   int x[ 10];   // array of 10 integers
    [ 10 ] * char y; char *y[ 10]; // array of 10 pointers to char
*/

```

```

new_declaration:                                 /* CFA */
    new_variable_declaration ';'
    | new_typedef_declaration ';'
    | new_function_declaration ';'
    | type_declaring_list ';'
    | context_specifier ';'
    ;

```

```

new_variable_declaration:                       /* CFA */
    new_variable_specifier initializer_opt
    | declaration_qualifier_list new_variable_specifier initializer_opt
      /* declaration_qualifier_list also includes type_qualifier_list, so a semantic check is
      necessary to preclude them as a type_qualifier cannot appear in that context. */
    | new_variable_declaration ';' identifier_or_typedef_name initializer_opt
    ;

```

```

new_variable_specifier:                         /* CFA */
      /* A semantic check is required to ensure asm_name only appears on declarations with implicit
      or explicit static storage-class */
    new_abstract_declarator_no_tuple identifier_or_typedef_name asm_name_opt
    | new_abstract_tuple identifier_or_typedef_name asm_name_opt
    | type_qualifier_list new_abstract_tuple identifier_or_typedef_name asm_name_opt
    ;

```

```

new_function_declaration:                       /* CFA */
    new_function_specifier
    | declaration_qualifier_list new_function_specifier
      /* declaration_qualifier_list also includes type_qualifier_list, so a semantic check is
      necessary to preclude them as a type_qualifier cannot appear in this context. */
    | new_function_declaration ';' identifier_or_typedef_name
    ;

```

```

new_function_specifier:                         /* CFA */
    '[' ']' identifier '(' new_parameter_type_list_opt ')'
    | '[' ']' TYPEDEFname '(' new_parameter_type_list_opt ')'
      /* identifier_or_typedef_name must be broken apart because of the sequence:

    '[' ']' identifier_or_typedef_name '(' new_parameter_type_list_opt ')'
    '[' ']' type_specifier

```

```

        type_specifier can resolve to just TYPEDEFname (e.g. typedef int T; int f( T );). Therefore
        this must be flattened to allow lookahead to the '(' without having to reduce
        identifier_or_typedef_name. */
| new_abstract_tuple identifier_or_typedef_name '(' new_parameter_type_list_opt ')'
  /* To obtain LR(1), this rule must be factored out from function return type (see
  new_abstract_declarator). */
| new_function_return identifier_or_typedef_name '(' new_parameter_type_list_opt ')'
;

new_function_return:
                                /* CFA */
  '[' new_parameter_list ']'
| '[' ' new_parameter_list ', ' new_abstract_parameter_list ']'
  /* To obtain LR(1), the last new_abstract_parameter_list is added into this flattened rule to
  lookahead to the ']'. */
;

new_typedef_declaration:
                                /* CFA */
  TYPEDEF new_variable_specifier
| TYPEDEF new_function_specifier
| new_typedef_declaration ', ' no_attr_identifier
;

/* Traditionally typedef is part of storage-class specifier for syntactic convenience only. Here, it is
factored out as a separate form of declaration, which syntactically precludes storage-class specifiers and
initialization. */

typedef_declaration:
  TYPEDEF type_specifier declarator
| typedef_declaration ', ' declarator
| type_qualifier_list TYPEDEF type_specifier declarator /* remaining OBSOLESCE (see 2) */
| type_specifier TYPEDEF declarator
| type_specifier TYPEDEF type_qualifier_list declarator
;

typedef_expression:
                                /* GCC, naming expression type */
  TYPEDEF no_attr_identifier '=' assignment_expression
| typedef_expression ', ' no_attr_identifier '=' assignment_expression
;

old_declaration:
  declaring_list ';'
| typedef_declaration ';'
| typedef_expression ';'
| sue_declaration_specifier ';'
;

declaring_list:
  /* A semantic check is required to ensure asm_name only appears on declarations with implicit
or explicit static storage-class */
  declaration_specifier declarator asm_name_opt initializer_opt
| declaring_list ', ' attribute_list_opt declarator asm_name_opt initializer_opt
;

declaration_specifier:
                                /* type specifier + storage class */
  basic_declaration_specifier
| sue_declaration_specifier
| typedef_declaration_specifier

```

```

    | typegen_declaration_specifier
    ;

type_specifier:                               /* declaration specifier - storage class */
    basic_type_specifier
    | sue_type_specifier
    | typedef_type_specifier
    | typegen_type_specifier
    ;

type_qualifier_list_opt:                     /* GCC, used in asm_statement */
    /* empty */
    | type_qualifier_list
    ;

type_qualifier_list:
    /* A semantic check is necessary to ensure a type qualifier is appropriate for the kind of
       declaration.

       ISO/IEC 9899:1999 Section 6.7.3(4) : If the same qualifier appears more than once in the
       same specifier-qualifier-list, either directly or via one or more typedefs, the behavior is
       the same as if it appeared only once. */

    type_qualifier
    | type_qualifier_list type_qualifier
    ;

type_qualifier:
    type_qualifier_name
    | attribute
    ;

type_qualifier_name:
    CONST
    | RESTRICT
    | VOLATILE
    | LVALUE                               /* CFA */
    | FORALL '(' type_parameter_list ')'   /* CFA */
    ;

declaration_qualifier_list:
    storage_class_list
    | type_qualifier_list storage_class_list /* remaining OBSOLESCENT (see 2) */
    | declaration_qualifier_list type_qualifier_list storage_class_list
    ;

storage_class_list:
    /* A semantic check is necessary to ensure a storage class is appropriate for the kind of
       declaration and that only one of each is specified, except for inline, which can appear
       with the others.

       ISO/IEC 9899:1999 Section 6.7.1(2) : At most, one storage-class specifier may be given in
       the declaration specifiers in a declaration. */

    storage_class
    | storage_class_list storage_class
    ;

storage_class:
    storage_class_name

```

```

;

storage_class_name:
    AUTO
    | EXTERN
    | REGISTER
    | STATIC
    | INLINE /* ANSI99 */
    /* INLINE is essentially a storage class specifier for functions, and hence, belongs here. */
    | FORTRAN /* ANSI99 */
;

basic_type_name:
    CHAR
    | DOUBLE
    | FLOAT
    | INT
    | LONG
    | SHORT
    | SIGNED
    | UNSIGNED
    | VOID
    | BOOL /* ANSI99 */
    | COMPLEX /* ANSI99 */
    | IMAGINARY /* ANSI99 */
;

basic_declaration_specifier:
    /* A semantic check is necessary for conflicting storage classes. */
    basic_type_specifier
    | declaration_qualifier_list basic_type_specifier
    | basic_declaration_specifier storage_class /* remaining OBSOLESCE (see 2) */
    | basic_declaration_specifier storage_class type_qualifier_list
    | basic_declaration_specifier storage_class basic_type_specifier
;

basic_type_specifier:
    direct_type_name
    | type_qualifier_list_opt indirect_type_name type_qualifier_list_opt
;

direct_type_name:
    /* A semantic check is necessary for conflicting type qualifiers. */
    basic_type_name
    | type_qualifier_list basic_type_name
    | direct_type_name type_qualifier
    | direct_type_name basic_type_name
;

indirect_type_name:
    TYPEOF '(' type_name ')' /* GCC: typeof(x) y; */
    | TYPEOF '(' comma_expression ')' /* GCC: typeof(a+b) y; */
    | ATTR_TYPEGENname '(' type_name ')' /* CFA: e.g., @type(x) y; */
    | ATTR_TYPEGENname '(' comma_expression ')' /* CFA: e.g., @type(a+b) y; */
;

sue_declaration_specifier:
    sue_type_specifier

```

```

| declaration_qualifier_list sue_type_specifier
| sue_declaration_specifier storage_class          /* remaining OBSOLESCE (see 2) */
| sue_declaration_specifier storage_class type_qualifier_list
;

sue_type_specifier:
elaborated_type_name                               /* struct, union, enum */
| type_qualifier_list elaborated_type_name
| sue_type_specifier type_qualifier
;

typedef_declaration_specifier:
typedef_type_specifier
| declaration_qualifier_list typedef_type_specifier
| typedef_declaration_specifier storage_class      /* remaining OBSOLESCE (see 2) */
| typedef_declaration_specifier storage_class type_qualifier_list
;

typedef_type_specifier:                               /* typedef types */
TYPEDEFname
| type_qualifier_list TYPEDEFname
| typedef_type_specifier type_qualifier
;

elaborated_type_name:
aggregate_name
| enum_name
;

aggregate_name:
aggregate_key '{' field_declaration_list '}'
| aggregate_key no_attr_identifier_or_typedef_name
| aggregate_key '(' type_parameter_list ')' '{' field_declaration_list '}'
| aggregate_key '(' type_parameter_list ')' no_attr_identifier_or_typedef_name /* CFA */
| aggregate_key '(' type_parameter_list ')' no_attr_identifier_or_typedef_name '{' field_declaration_list '}' /* CFA */
| aggregate_key '(' type_parameter_list ')' '(' type_name_list ')' '{' field_declaration_list '}' /* CFA */
| aggregate_key '(' type_name_list ')' no_attr_identifier_or_typedef_name /* CFA */
/* push and pop are only to prevent S/R conflicts */
| aggregate_key '(' type_parameter_list ')' '(' type_name_list ')' no_attr_identifier_or_typedef_name '{'
field_declaration_list '}' /* CFA */
;

aggregate_key:
STRUCT attribute_list_opt
| UNION attribute_list_opt
;

field_declaration_list:
field_declaration
| field_declaration_list field_declaration
;

field_declaration:
new_field_declarating_list ';' /* CFA, new style field declaration */
| EXTENSION new_field_declarating_list ';' /* GCC */
| field_declarating_list ';'
| EXTENSION field_declarating_list ';' /* GCC */

```

```

;

new_field_declaring_list:                               /* CFA, new style field declaration */
    new_abstract_declarator_tuple                       /* CFA, no field name */
    | new_abstract_declarator_tuple no_attr_identifier_or_typedef_name
    | new_field_declaring_list ',' no_attr_identifier_or_typedef_name
    | new_field_declaring_list ',' ' ' /* CFA, no field name */
;

field_declaring_list:
    type_specifier field_declarator
    | field_declaring_list ',' attribute_list_opt field_declarator
;

field_declarator:
    /* empty */ /* CFA, no field name */
    | bit_subrange_size /* no field name */
    | variable_declarator bit_subrange_size_opt
        /* A semantic check is required to ensure bit_subrange only appears on base type int. */
    | typedef_redeclarator bit_subrange_size_opt
        /* A semantic check is required to ensure bit_subrange only appears on base type int. */
    | variable_abstract_declarator /* CFA, no field name */
;

bit_subrange_size_opt:
    /* empty */
    | bit_subrange_size
;

bit_subrange_size:
    ':' constant_expression
;

enum_key:
    ENUM attribute_list_opt
;

enum_name:
    enum_key '{' enumerator_list comma_opt '}'
    | enum_key no_attr_identifier_or_typedef_name '{' enumerator_list comma_opt '}'
    | enum_key no_attr_identifier_or_typedef_name
;

enumerator_list:
    no_attr_identifier_or_typedef_name enumerator_value_opt
    | enumerator_list ',' no_attr_identifier_or_typedef_name enumerator_value_opt
;

enumerator_value_opt:
    /* empty */
    | '=' constant_expression
;

/* Minimum of one parameter after which ellipsis is allowed only at the end. */

new_parameter_type_list_opt:                          /* CFA */
    /* empty */
    | new_parameter_type_list

```

```

;

new_parameter_type_list:                               /* CFA, abstract + real */
  new_abstract_parameter_list
  | new_parameter_list
  | new_parameter_list ',' new_abstract_parameter_list
  | new_abstract_parameter_list ',' ELLIPSIS
  | new_parameter_list ',' ELLIPSIS
;

new_parameter_list:                                   /* CFA */
  /* To obtain LR(1) between new_parameter_list and new_abstract_tuple, the last
   new_abstract_parameter_list is factored out from new_parameter_list, flattening the rules
   to get lookahead to the ']'. */
  new_parameter_declaration
  | new_abstract_parameter_list ',' new_parameter_declaration
  | new_parameter_list ',' new_parameter_declaration
  | new_parameter_list ',' new_abstract_parameter_list ',' new_parameter_declaration
;

new_abstract_parameter_list:                          /* CFA, new & old style abstract */
  new_abstract_parameter_declaration
  | new_abstract_parameter_list ',' new_abstract_parameter_declaration
;

parameter_type_list_opt:
  /* empty */
  | parameter_type_list
;

parameter_type_list:
  parameter_list
  | parameter_list ',' ELLIPSIS
;

parameter_list:                                       /* abstract + real */
  abstract_parameter_declaration
  | parameter_declaration
  | parameter_list ',' abstract_parameter_declaration
  | parameter_list ',' parameter_declaration
;

/* Provides optional identifier names (abstract_declarator/variable_declarator), no initialization, different
 semantics for typedef name by using typedef_parameter_redeclarator instead of typedef_redeclarator, and
 function prototypes. */

new_parameter_declaration:                             /* CFA, new & old style parameter declaration */
  parameter_declaration
  | new_identifier_parameter_declarator_no_tuple identifier_or_typedef_name assignment_opt
  | new_abstract_tuple identifier_or_typedef_name assignment_opt
  /* To obtain LR(1), these rules must be duplicated here (see new_abstract_declarator). */
  | type_qualifier_list new_abstract_tuple identifier_or_typedef_name assignment_opt
  | new_function_specifier
;

new_abstract_parameter_declaration:                   /* CFA, new & old style parameter declaration */
  abstract_parameter_declaration
  | new_identifier_parameter_declarator_no_tuple

```

```

    | new_abstract_tuple
      /* To obtain LR(1), these rules must be duplicated here (see new_abstract_declarator). */
    | type_qualifier_list new_abstract_tuple
    | new_abstract_function
    ;

parameter_declaration:
    declaration_specifier identifier_parameter_declarator assignment_opt
    | declaration_specifier typedef_parameter_redeclarator assignment_opt
    ;

abstract_parameter_declaration:
    declaration_specifier
    | declaration_specifier abstract_parameter_declarator
    ;

/* ISO/IEC 9899:1999 Section 6.9.1(6) : "An identifier declared as a typedef name shall not be redeclared as a
parameter." Because the scope of the K&R-style parameter-list sees the typedef first, the following is
based only on identifiers. The ANSI-style parameter-list can redefine a typedef name. */

identifier_list:
    /* K&R-style parameter list => no types */
    no_attr_identifier
    | identifier_list ',' no_attr_identifier
    ;

identifier_or_typedef_name:
    identifier
    | TYPEDEFname
    | TYPEGENname
    ;

no_01_identifier_or_typedef_name:
    no_01_identifier
    | TYPEDEFname
    | TYPEGENname
    ;

no_attr_identifier_or_typedef_name:
    no_attr_identifier
    | TYPEDEFname
    | TYPEGENname
    ;

type_name_no_function:
    /* sizeof, alignof, cast (constructor) */
    /* CFA */
    new_abstract_declarator_tuple
    | type_specifier
    | type_specifier variable_abstract_declarator
    ;

type_name:
    /* typeof, assertion */
    /* CFA */
    /* CFA */
    new_abstract_declarator_tuple
    | new_abstract_function
    | type_specifier
    | type_specifier abstract_declarator
    ;

initializer_opt:
    /* empty */

```

```
| '=' initializer
;
```

initializer:

```
assignment_expression
| '{' initializer_list comma_opt '}'
;
```

initializer\_list:

```
initializer
| designation initializer
| initializer_list ',' initializer
| initializer_list ',' designation initializer
;
```

*/\* There is an unreconcilable parsing problem between ANSI99 and CFA with respect to designators. The problem is use of '=' to separator the designator from the initializer value, as in:*

```
int x[10] = { [1] = 3 };
```

*The string "[1] = 3" can be parsed as a designator assignment or a tuple assignment. To disambiguate this case, CFA changes the syntax from "=" to ":" as the separator between the designator and initializer. GCC does uses ":" for field selection. The optional use of the "=" in GCC, or in this case ":", cannot be supported either due to shift/reduce conflicts \*/*

designation:

```
designator_list ':' '          /* ANSI99, CFA uses ":" instead of "=" */
| no_attr_identifier_or_typedef_name ':' ' /* GCC, field name */
;
```

designator\_list:

```
designator
| designator_list designator
;
```

*/\* ANSI99 \*/*

designator:

```
'.' no_attr_identifier_or_typedef_name /* ANSI99, field name */
| '[' assignment_expression ']' /* ANSI99, single array element */
/* assignment_expression used instead of constant_expression because of shift/reduce conflicts
with tuple. */
| '[' subrange ']' /* CFA, multiple array elements */
| '[' constant_expression ELLIPSIS constant_expression ']' /* GCC, multiple array elements */
| '.' '[' field_list ']' /* CFA, tuple field selector */
;
```

*/\* The CFA type system is based on parametric polymorphism, the ability to declare functions with type parameters, rather than an object-oriented type system. This required four groups of extensions:*

*Overloading: function, data, and operator identifiers may be overloaded.*

*Type declarations: "type" is used to generate new types for declaring objects. Similarly, "dtype" is used for object and incomplete types, and "ftype" is used for function types. Type declarations with initializers provide definitions of new types. Type declarations with storage class "extern" provide opaque types.*

*Polymorphic functions: A forall clause declares a type parameter. The corresponding argument is inferred at the call site. A polymorphic function is not a template; it is a function, with an address and a type.*

*Specifications and Assertions: Specifications are collections of declarations parameterized by one or more types. They serve many of the purposes of abstract classes, and specification hierarchies resemble subclass hierarchies. Unlike classes, they can define relationships between types. Assertions declare that a type or types provide the operations declared by a specification. Assertions are normally used to declare requirements on type arguments of polymorphic functions. \*/*

```

typegen_declaration_specifier:                               /* CFA */
  typegen_type_specifier
  | declaration_qualifier_list typegen_type_specifier
  | typegen_declaration_specifier storage_class /* remaining OBSOLESCE (see 2) */
  | typegen_declaration_specifier storage_class type_qualifier_list
  ;

typegen_type_specifier:                                     /* CFA */
  TYPEGENname '(' type_name_list ')'
  | type_qualifier_list TYPEGENname '(' type_name_list ')'
  | typegen_type_specifier type_qualifier
  ;

type_parameter_list:                                       /* CFA */
  type_parameter assignment_opt
  | type_parameter_list ',' type_parameter assignment_opt
  ;

type_parameter:                                           /* CFA */
  type_class no_attr_identifier_or_typedef_name assertion_list_opt
  | type_specifier identifier_parameter_declarator
  ;

type_class:                                                /* CFA */
  TYPE
  | DTYPE
  | FTYPE
  ;

assertion_list_opt:                                        /* CFA */
  /* empty */
  | assertion_list_opt assertion
  ;

assertion:                                                 /* CFA */
  '|' no_attr_identifier_or_typedef_name '(' type_name_list ')'
  | '|' '{ context_declaration_list }'
  | '|' '(' type_parameter_list ')' '{ context_declaration_list }' '(' type_name_list ')'
  ;

type_name_list:                                           /* CFA */
  type_name
  | assignment_expression
  | type_name_list ',' type_name
  | type_name_list ',' assignment_expression
  ;

type_declaring_list:                                       /* CFA */
  TYPE type_declarator
  | storage_class_list TYPE type_declarator
  | type_declaring_list ',' type_declarator
  ;

```

```

type_declarator:
    type_declarator_name assertion_list_opt
    | type_declarator_name assertion_list_opt '=' type_name
    ;

type_declarator_name:
    no_attr_identifier_or_typedef_name
    | no_01_identifier_or_typedef_name '(' type_parameter_list ')'
    ;

context_specifier:
    /* CFA */
    CONTEXT no_attr_identifier_or_typedef_name '(' type_parameter_list ')' '{' '}'
    | CONTEXT no_attr_identifier_or_typedef_name '(' type_parameter_list ')' '{' context_declaration_list '}'
    ;

context_declaration_list:
    /* CFA */
    context_declaration
    | context_declaration_list context_declaration
    ;

context_declaration:
    /* CFA */
    new_context_declaring_list ';'
    | context_declaring_list ';'
    ;

new_context_declaring_list:
    /* CFA */
    new_variable_specifier
    | new_function_specifier
    | new_context_declaring_list ',' identifier_or_typedef_name
    ;

context_declaring_list:
    /* CFA */
    type_specifier declarator
    | context_declaring_list ',' declarator
    ;

/***** EXTERNAL DEFINITIONS *****/

translation_unit:
    /* empty */
    /* empty input file */
    | external_definition_list
    ;

external_definition_list:
    external_definition
    | external_definition_list external_definition
    ;

external_definition_list_opt:
    /* empty */
    | external_definition_list
    ;

external_definition:
    declaration
    | function_definition
    | asm_statement
    /* GCC, global assembler statement */

```

```

| EXTERN STRINGliteral '{' external_definition_list_opt '}' /* C++-style linkage specifier */
| EXTENSION external_definition
;

function_definition:
new_function_specifier compound_statement /* CFA */
| declaration_qualifier_list new_function_specifier compound_statement /* CFA */
/* declaration_qualifier_list also includes type_qualifier_list, so a semantic check is
necessary to preclude them as a type_qualifier cannot appear in this context. */

| declaration_specifier function_declarator compound_statement

/* These rules are a concession to the "implicit int" type_specifier because there is a
significant amount of code with functions missing a type-specifier on the return type.
Parsing is possible because function_definition does not appear in the context of an
expression (nested functions would preclude this concession). A function prototype
declaration must still have a type_specifier. OBSOLESCENT (see 1) */
| function_declarator compound_statement
| type_qualifier_list function_declarator compound_statement
| declaration_qualifier_list function_declarator compound_statement
| declaration_qualifier_list type_qualifier_list function_declarator compound_statement

/* Old-style K&R function definition, OBSOLESCENT (see 4) */
| declaration_specifier old_function_declarator old_declaration_list_opt compound_statement
| old_function_declarator old_declaration_list_opt compound_statement
| type_qualifier_list old_function_declarator old_declaration_list_opt compound_statement

/* Old-style K&R function definition with "implicit int" type_specifier, OBSOLESCENT (see 4) */
| declaration_qualifier_list old_function_declarator old_declaration_list_opt compound_statement
| declaration_qualifier_list type_qualifier_list old_function_declarator old_declaration_list_opt
compound_statement
;

declarator:
variable_declarator
| function_declarator
| typedef_redeclarator
;

subrange:
constant_expression '~' constant_expression /* CFA, integer subrange */
;

asm_name_opt: /* GCC */
/* empty */
| ASM '(' string_literal_list ')' attribute_list_opt
;

attribute_list_opt: /* GCC */
/* empty */
| attribute_list
;

attribute_list: /* GCC */
attribute
| attribute_list attribute
;

```

```

attribute:                                     /* GCC */
    ATTRIBUTE '(' '(' attribute_parameter_list ')' ')'
    ;

attribute_parameter_list:                     /* GCC */
    attrib
    | attribute_parameter_list ',' attrib
    ;

attrib:                                       /* GCC */
    /* empty */
    | any_word
    | any_word '(' comma_expression_opt ')'
    ;

any_word:                                     /* GCC */
    identifier_or_typedef_name
    | storage_class_name
    | basic_type_name
    | type_qualifier
    ;

```

=====  
*The following sections are a series of grammar patterns used to parse declarators. Multiple patterns are necessary because the type of an identifier is wrapped around the identifier in the same form as its usage in an expression, as in:*

```

    int (*f())[10] { ... };
    ... (*f())[3] += 1;      // definition mimics usage

```

*Because these patterns are highly recursive, changes at a lower level in the recursion require copying some or all of the pattern. Each of these patterns has some subtle variation to ensure correct syntax in a particular context.*

=====  
*The set of valid declarators before a compound statement for defining a function is less than the set of declarators to define a variable or function prototype, e.g.:*

<u>valid declaration</u>	<u>invalid definition</u>
<i>int f;</i>	<i>int f {}</i>
<i>int *f;</i>	<i>int *f {}</i>
<i>int f[10];</i>	<i>int f[10] {}</i>
<i>int (*f)(int);</i>	<i>int (*f)(int) {}</i>

*To preclude this syntactic anomaly requires separating the grammar rules for variable and function declarators, hence `variable_declarator` and `function_declarator`.*

=====  
*This pattern parses a declaration of a variable that is not redefining a typedef name. The pattern precludes declaring an array of functions versus a pointer to an array of functions. \*/*

```

variable_declarator:
    paren_identifier attribute_list_opt
    | variable_ptr
    | variable_array attribute_list_opt
    | variable_function attribute_list_opt

```

```

;

paren_identifier:
    identifier
    | '(' paren_identifier ')' /* redundant parenthesis */
;

variable_ptr:
    '*' variable_declarator
    | '*' type_qualifier_list variable_declarator
    | '(' variable_ptr ')'
;

variable_array:
    paren_identifier array_dimension
    | '(' variable_ptr ')' array_dimension
    | '(' variable_array ')' multi_array_dimension /* redundant parenthesis */
    | '(' variable_array ')' /* redundant parenthesis */
;

variable_function:
    '(' variable_ptr ')' '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    | '(' variable_function ')' /* redundant parenthesis */
;

```

*/\* This pattern parses a function declarator that is not redefining a typedef name. Because functions cannot be nested, there is no context where a function definition can redefine a typedef name. To allow nested functions requires further separation of variable and function pointers in typedef\_redeclarator. The pattern precludes returning arrays and functions versus pointers to arrays and functions. \*/*

```

function_declarator:
    function_no_ptr attribute_list_opt
    | function_ptr
    | function_array attribute_list_opt
;

function_no_ptr:
    paren_identifier '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    | '(' function_ptr ')' '(' parameter_type_list_opt ')'
    | '(' function_no_ptr ')' /* redundant parenthesis */
;

function_ptr:
    '*' function_declarator
    | '*' type_qualifier_list function_declarator
    | '(' function_ptr ')'
;

function_array:
    '(' function_ptr ')' array_dimension
    | '(' function_array ')' multi_array_dimension /* redundant parenthesis */
    | '(' function_array ')' /* redundant parenthesis */
;

```

*/\* This pattern parses an old-style K&R function declarator (OBSOLESCE, see 4) that is not redefining a typedef name (see function\_declarator for additional comments). The pattern precludes returning arrays and functions versus pointers to arrays and functions. \*/*

```

old_function_declarator:
    old_function_no_ptr
    | old_function_ptr
    | old_function_array
    ;

old_function_no_ptr:
    paren_identifier '(' identifier_list ')' /* function_declarator handles empty parameter */
    | '(' old_function_ptr ')' '(' identifier_list ')' /* redundant parenthesis */
    | '(' old_function_no_ptr ')'
    ;

old_function_ptr:
    '*' old_function_declarator
    | '*' type_qualifier_list old_function_declarator
    | '(' old_function_ptr ')'
    ;

old_function_array:
    '(' old_function_ptr ')' array_dimension
    | '(' old_function_array ')' multi_array_dimension /* redundant parenthesis */
    | '(' old_function_array ')' /* redundant parenthesis */
    ;

```

*/\* This pattern parses a declaration for a variable or function prototype that redefines a typedef name, e.g.:*

```

typedef int foo;
{
    int foo; // redefine typedef name in new scope
}

```

*The pattern precludes declaring an array of functions versus a pointer to an array of functions, and returning arrays and functions versus pointers to arrays and functions. \*/*

```

typedef_redeclarator:
    paren_typedef attribute_list_opt
    | typedef_ptr
    | typedef_array attribute_list_opt
    | typedef_function attribute_list_opt
    ;

paren_typedef:
    TYPEDEFname
    | '(' paren_typedef ')'
    ;

typedef_ptr:
    '*' typedef_redeclarator
    | '*' type_qualifier_list typedef_redeclarator
    | '(' typedef_ptr ')'
    ;

typedef_array:
    paren_typedef array_dimension
    | '(' typedef_ptr ')' array_dimension
    | '(' typedef_array ')' multi_array_dimension /* redundant parenthesis */
    | '(' typedef_array ')' /* redundant parenthesis */
    ;

```

```

typedef_function:
    paren_typedef '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    | '(' typedef_ptr ')' '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    | '(' typedef_function ')' /* redundant parenthesis */
    ;

```

*/\* This pattern parses a declaration for a parameter variable or function prototype that is not redefining a typedef name and allows the ANSI99 array options, which can only appear in a parameter list. The pattern precludes declaring an array of functions versus a pointer to an array of functions, and returning arrays and functions versus pointers to arrays and functions. \*/*

```

identifier_parameter_declarator:
    paren_identifier attribute_list_opt
    | identifier_parameter_ptr
    | identifier_parameter_array attribute_list_opt
    | identifier_parameter_function attribute_list_opt
    ;

```

```

identifier_parameter_ptr:
    '*' identifier_parameter_declarator
    | '*' type_qualifier_list identifier_parameter_declarator
    | '(' identifier_parameter_ptr ')'
    ;

```

```

identifier_parameter_array:
    paren_identifier array_parameter_dimension
    | '(' identifier_parameter_ptr ')' array_dimension
    | '(' identifier_parameter_array ')' multi_array_dimension /* redundant parenthesis */
    | '(' identifier_parameter_array ')' /* redundant parenthesis */
    ;

```

```

identifier_parameter_function:
    paren_identifier '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    | '(' identifier_parameter_ptr ')' '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    | '(' identifier_parameter_function ')' /* redundant parenthesis */
    ;

```

*/\* This pattern parses a declaration for a parameter variable or function prototype that is redefining a typedef name, e.g.:*

```

typedef int foo;
int f( int foo ); // redefine typedef name in new scope

```

*and allows the ANSI99 array options, which can only appear in a parameter list. In addition, the pattern handles the special meaning of parenthesis around a typedef name:*

*ISO/IEC 9899:1999 Section 6.7.5.3(11) : "In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier."*

*which precludes the following cases:*

```

typedef float T;
int f( int ( T [5] ) ); // see abstract_parameter_declarator
int g( int ( T ( int ) ) ); // see abstract_parameter_declarator
int f( int f1( T a[5] ) ); // see identifier_parameter_declarator
int g( int g1( T g2( int p ) ) ); // see identifier_parameter_declarator

```

*In essence, a '(' immediately to the left of typedef name, T, is interpreted as starting a parameter type list, and not as redundant parentheses around a redeclaration of T. Finally, the pattern also precludes declaring an array of functions versus a pointer to an array of functions, and returning arrays and functions versus pointers to arrays and functions. \*/*

```
typedef_parameter_redeclarator:
    typedef attribute_list_opt
    | typedef_parameter_ptr
    | typedef_parameter_array attribute_list_opt
    | typedef_parameter_function attribute_list_opt
    ;
```

```
typedef:
    TYPEDEFname
    ;
```

```
typedef_parameter_ptr:
    '*' typedef_parameter_redeclarator
    | '*' type_qualifier_list typedef_parameter_redeclarator
    | '(' typedef_parameter_ptr ')'
    ;
```

```
typedef_parameter_array:
    typedef_array_parameter_dimension
    | '(' typedef_parameter_ptr ')' array_parameter_dimension
    ;
```

```
typedef_parameter_function:
    typedef '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    | '(' typedef_parameter_ptr ')' '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
    ;
```

*/\* This pattern parses a declaration of an abstract variable or function prototype, i.e., there is no identifier to which the type applies, e.g.:*

```
sizeof( int );
sizeof( int [ 10 ] );
```

*The pattern precludes declaring an array of functions versus a pointer to an array of functions, and returning arrays and functions versus pointers to arrays and functions. \*/*

```
abstract_declarator:
    abstract_ptr
    | abstract_array attribute_list_opt
    | abstract_function attribute_list_opt
    ;
```

```
abstract_ptr:
    '*'
    | '*' type_qualifier_list
    | '*' abstract_declarator
    | '*' type_qualifier_list abstract_declarator
    | '(' abstract_ptr ')'
    ;
```

```
abstract_array:
    array_dimension
```

```

| '(' abstract_ptr ')' array_dimension
| '(' abstract_array ')' multi_array_dimension /* redundant parenthesis */
| '(' abstract_array ')' /* redundant parenthesis */
;

```

abstract\_function:

```

 '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
| '(' abstract_ptr ')' '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
| '(' abstract_function ')' /* redundant parenthesis */
;

```

array\_dimension:

```

/* Only the first dimension can be empty. */
 '[' '' ']'
| '[' '' ']' multi_array_dimension
| multi_array_dimension
;

```

multi\_array\_dimension:

```

 '[' assignment_expression ']'
| '[' '*' ']' /* ANS199 */
| multi_array_dimension '[' assignment_expression ']'
| multi_array_dimension '[' '*' ']' /* ANS199 */
;

```

*/\* This pattern parses a declaration of a parameter abstract variable or function prototype, i.e., there is no identifier to which the type applies, e.g.:*

```

int f( int ); /* abstract variable parameter; no parameter name specified
int f( int (int) ); /* abstract function-prototype parameter; no parameter name specified

```

*The pattern precludes declaring an array of functions versus a pointer to an array of functions, and returning arrays and functions versus pointers to arrays and functions. \*/*

abstract\_parameter\_declarator:

```

abstract_parameter_ptr
| abstract_parameter_array attribute_list_opt
| abstract_parameter_function attribute_list_opt
;

```

abstract\_parameter\_ptr:

```

 '*'
| '*' type_qualifier_list
| '*' abstract_parameter_declarator
| '*' type_qualifier_list abstract_parameter_declarator
| '(' abstract_parameter_ptr ')'
;

```

abstract\_parameter\_array:

```

array_parameter_dimension
| '(' abstract_parameter_ptr ')' array_parameter_dimension
| '(' abstract_parameter_array ')' multi_array_dimension /* redundant parenthesis */
| '(' abstract_parameter_array ')' /* redundant parenthesis */
;

```

abstract\_parameter\_function:

```

 '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */
| '(' abstract_parameter_ptr ')' '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCE (see 3) */

```

```

| '(' abstract_parameter_function ')' /* redundant parenthesis */
;

array_parameter_dimension:
    /* Only the first dimension can be empty or have qualifiers. */
    array_parameter_1st_dimension
    | array_parameter_1st_dimension multi_array_dimension
    | multi_array_dimension
;

/* The declaration of an array parameter has additional syntax over arrays in normal variable declarations:

    ISO/IEC 9899:1999 Section 6.7.5.2(1) : "The optional type qualifiers and the keyword static shall
    appear only in a declaration of a function parameter with an array type, and then only in the
    outermost array type derivation."

*/

array_parameter_1st_dimension:
    '[' '['
    | '[' type_qualifier_list '*' '[' /* remaining ANSI99 */
    | '[' type_qualifier_list assignment_expression '['
    | '[' STATIC assignment_expression '['
    | '[' STATIC type_qualifier_list assignment_expression '['
;

/* This pattern parses a declaration of an abstract variable, i.e., there is no identifier to which the type
applies, e.g.:

    sizeof( int ); // abstract variable; no identifier name specified

The pattern precludes declaring an array of functions versus a pointer to an array of functions, and
returning arrays and functions versus pointers to arrays and functions. */

variable_abstract_declarator:
    variable_abstract_ptr
    | variable_abstract_array attribute_list_opt
    | variable_abstract_function attribute_list_opt
;

variable_abstract_ptr:
    '*'
    | '*' type_qualifier_list
    | '*' variable_abstract_declarator
    | '*' type_qualifier_list variable_abstract_declarator
    | '(' variable_abstract_ptr ')'
;

variable_abstract_array:
    array_dimension
    | '(' variable_abstract_ptr ')' array_dimension
    | '(' variable_abstract_array ')' multi_array_dimension /* redundant parenthesis */
    | '(' variable_abstract_array ')' /* redundant parenthesis */
;

variable_abstract_function:
    '(' variable_abstract_ptr ')' '(' parameter_type_list_opt ')' /* empty parameter list OBSOLESCENT (see 3) */
    | '(' variable_abstract_function ')' /* redundant parenthesis */
;

```

*/\* This pattern parses a new-style declaration for a parameter variable or function prototype that is either an identifier or typedef name and allows the ANSI99 array options, which can only appear in a parameter list. \*/*

```
new_identifier_parameter_declarator_tuple:          /* CFA */
  new_identifier_parameter_declarator_no_tuple
  | new_abstract_tuple
  | type_qualifier_list new_abstract_tuple
  ;
```

```
new_identifier_parameter_declarator_no_tuple:      /* CFA */
  new_identifier_parameter_ptr
  | new_identifier_parameter_array
  ;
```

```
new_identifier_parameter_ptr:                     /* CFA */
  '*' type_specifier
  | type_qualifier_list '*' type_specifier
  | '*' new_abstract_function
  | type_qualifier_list '*' new_abstract_function
  | '*' new_identifier_parameter_declarator_tuple
  | type_qualifier_list '*' new_identifier_parameter_declarator_tuple
  ;
```

```
new_identifier_parameter_array:                   /* CFA */
  /* Only the first dimension can be empty or have qualifiers. Empty dimension must be factored
  out due to shift/reduce conflict with new-style empty (void) function return type. */
  '[' '[' type_specifier
  | new_array_parameter_1st_dimension type_specifier
  | '[' '[' multi_array_dimension type_specifier
  | new_array_parameter_1st_dimension multi_array_dimension type_specifier
  | multi_array_dimension type_specifier
  | '[' '[' new_identifier_parameter_ptr
  | new_array_parameter_1st_dimension new_identifier_parameter_ptr
  | '[' '[' multi_array_dimension new_identifier_parameter_ptr
  | new_array_parameter_1st_dimension multi_array_dimension new_identifier_parameter_ptr
  | multi_array_dimension new_identifier_parameter_ptr
  ;
```

```
new_array_parameter_1st_dimension:
  '[' type_qualifier_list '*' '[' /* remaining ANSI99 */
  | '[' type_qualifier_list assignment_expression '['
  | '[' declaration_qualifier_list assignment_expression '['
  /* declaration_qualifier_list must be used because of shift/reduce conflict with
  assignment_expression, so a semantic check is necessary to preclude them as a
  type_qualifier cannot appear in this context. */
  | '[' declaration_qualifier_list type_qualifier_list assignment_expression '['
  ;
```

*/\* This pattern parses a new-style declaration of an abstract variable or function prototype, i.e., there is no identifier to which the type applies, e.g.:*

```
[int] f( int );           // abstract variable parameter; no parameter name specified
[int] f( [int] (int) );   // abstract function-prototype parameter; no parameter name specified
```

*These rules need LR(3):*

```

new_abstract_tuple identifier_or_typedef_name
[' new_parameter_list ' ] identifier_or_typedef_name '(' new_parameter_type_list_opt ')'

```

since a function return type can be syntactically identical to a tuple type:

```

[int, int] t;
[int, int] f(int);

```

Therefore, it is necessary to look at the token after identifier\_or\_typedef\_name to know when to reduce new\_abstract\_tuple. To make this LR(1), several rules have to be flattened (lengthened) to allow the necessary lookahead. To accomplish this, new\_abstract\_declarator has an entry point without tuple, and tuple declarations are duplicated when appearing with new\_function\_specifier. \*/

```

new_abstract_declarator_tuple:                               /* CFA */
    new_abstract_tuple
    | type_qualifier_list new_abstract_tuple
    | new_abstract_declarator_no_tuple
    ;

new_abstract_declarator_no_tuple:                           /* CFA */
    new_abstract_ptr
    | new_abstract_array
    ;

new_abstract_ptr:                                          /* CFA */
    '*' type_specifier
    | type_qualifier_list '*' type_specifier
    | '*' new_abstract_function
    | type_qualifier_list '*' new_abstract_function
    | '*' new_abstract_declarator_tuple
    | type_qualifier_list '*' new_abstract_declarator_tuple
    ;

new_abstract_array:                                       /* CFA */
    /* Only the first dimension can be empty. Empty dimension must be factored out due to
    shift/reduce conflict with empty (void) function return type. */
    '[' ']' type_specifier
    | '[' ']' multi_array_dimension type_specifier
    | multi_array_dimension type_specifier
    | '[' ']' new_abstract_ptr
    | '[' ']' multi_array_dimension new_abstract_ptr
    | multi_array_dimension new_abstract_ptr
    ;

new_abstract_tuple:                                       /* CFA */
    '[' new_abstract_parameter_list ']'
    ;

new_abstract_function:                                    /* CFA */
    '[' ']' '(' new_parameter_type_list_opt ')'
    | new_abstract_tuple '(' new_parameter_type_list_opt ')'
    | new_function_return '(' new_parameter_type_list_opt ')'
    ;

```

/\* 1) ISO/IEC 9899:1999 Section 6.7.2(2) : "At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each structure declaration and type name."

- 2) ISO/IEC 9899:1999 Section 6.11.5(1) : *“The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.”*
- 3) ISO/IEC 9899:1999 Section 6.11.6(1) : *“The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.”*
- 4) ISO/IEC 9899:1999 Section 6.11.7(1) : *“The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.” \*/*

*/\*\*\*\*\*\* MISCELLANEOUS \*\*\*\*\*/*

```

comma_opt:                                     /* redundant comma */
    /* empty */
    | ' '
    ;

assignment_opt:
    /* empty */
    | '=' assignment_expression
    ;

```