ANSI C Yacc grammar

(This Yacc file is accompanied by a matching Lex file.)

In 1985, Jeff Lee published his Yacc grammar based on a draft version of the ANSI C standard, along with a supporting Lex specification. Tom Stockfisch reposted those files to net.sources in 1987; as mentioned in the answer to question 17.25 of the comp.lang.c FAQ, they used to be available from ftp.uu.net as usenet/net.sources/ansi.c.grammar.Z.

The version you see here has been updated based on the 2011 ISO C standard. (The previous version's Lex and Yacc files for ANSI C9X still exist as archived copies.)

This grammar assumes that translation phases 1..5 have already been completed, including preprocessing and `_Pragma` processing. The Lex rule for string literals will perform concatenation (translation phase 6). Transliteration of universal character names (`\u`*HHHH* or `\U`*HHHHHHHH*) must have been done by either the preprocessor or a replacement for the `input()` macro used by Lex (or the YY_INPUT function used by Flex) to read characters. Although comments should have been changed to space characters during translation phase 3, there are Lex rules for them anyway.

I want to keep this version as close to the current C Standard grammar as possible; please let me know if you discover discrepancies.
(There is an FAQ for this grammar that you might want to read first.)

jutta@pobox.com, 2012

Last edit: 2012-12-18 DAGwyn@aol.com

Note: There are two shift/reduce conflicts, correctly resolved by default:

```
  IF '(' expression ')' statement _ ELSE statement
```

and

```
  ATOMIC _ '(' type_name ')'
```

where "_" has been used to flag the points of ambiguity.

```
%token  IDENTIFIER I_CONSTANT F_CONSTANT STRING_LITERAL FUNC_NAME SIZEOF
%token  PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token  AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token  SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token  XOR_ASSIGN OR_ASSIGN
%token  TYPEDEF_NAME ENUMERATION_CONSTANT

%token  TYPEDEF EXTERN STATIC AUTO REGISTER INLINE
%token  CONST RESTRICT VOLATILE
%token  BOOL CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE VOID
%token  COMPLEX IMAGINARY
%token  STRUCT UNION ENUM ELLIPSIS

%token  CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%token  ALIGNAS ALIGNOF ATOMIC GENERIC NORETURN STATIC_ASSERT THREAD_LOCAL

%start translation_unit
%%
```

```
primary_expression
        : IDENTIFIER
        | constant
        | string
        | '(' expression ')'
        | generic_selection
        ;

constant
        : I_CONSTANT            /* includes character_constant */
        | F_CONSTANT
        | ENUMERATION_CONSTANT  /* after it has been defined as such */
        ;

enumeration_constant            /* before it has been defined as such */
        : IDENTIFIER
        ;

string
        : STRING_LITERAL
        | FUNC_NAME
        ;

generic_selection
        : GENERIC '(' assignment_expression ',' generic_assoc_list ')'
        ;

generic_assoc_list
        : generic_association
        | generic_assoc_list ',' generic_association
        ;

generic_association
        : type_name ':' assignment_expression
        | DEFAULT ':' assignment_expression
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        | '(' type_name ')' '{' initializer_list '}'
        | '(' type_name ')' '{' initializer_list ',' '}'
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        | ALIGNOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
```

```
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;

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 LEFT_OP additive_expression
        | shift_expression RIGHT_OP additive_expression
        ;

relational_expression
        : shift_expression
        | relational_expression '<' shift_expression
        | relational_expression '>' shift_expression
        | relational_expression LE_OP shift_expression
        | relational_expression GE_OP shift_expression
        ;

equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
        | equality_expression NE_OP 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 AND_OP inclusive_or_expression
        ;

logical_or_expression
        : logical_and_expression
        | logical_or_expression OR_OP logical_and_expression
        ;

conditional_expression
```

```
        : logical_or_expression
        | logical_or_expression '?' expression ':' conditional_expression
        ;

assignment_expression
        : conditional_expression
        | unary_expression assignment_operator assignment_expression
        ;

assignment_operator
        : '='
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | LEFT_ASSIGN
        | RIGHT_ASSIGN
        | AND_ASSIGN
        | XOR_ASSIGN
        | OR_ASSIGN
        ;

expression
        : assignment_expression
        | expression ',' assignment_expression
        ;

constant_expression
        : conditional_expression        /* with constraints */
        ;

declaration
        : declaration_specifiers ';'
        | declaration_specifiers init_declarator_list ';'
        | static_assert_declaration
        ;

declaration_specifiers
        : storage_class_specifier declaration_specifiers
        | storage_class_specifier
        | type_specifier declaration_specifiers
        | type_specifier
        | type_qualifier declaration_specifiers
        | type_qualifier
        | function_specifier declaration_specifiers
        | function_specifier
        | alignment_specifier declaration_specifiers
        | alignment_specifier
        ;

init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;

init_declarator
        : declarator '=' initializer
        | declarator
        ;

storage_class_specifier
        : TYPEDEF       /* identifiers must be flagged as TYPEDEF_NAME */
        | EXTERN
        | STATIC
        | THREAD_LOCAL
        | AUTO
        | REGISTER
        ;
```

```
type_specifier
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        | BOOL
        | COMPLEX
        | IMAGINARY             /* non-mandated extension */
        | atomic_type_specifier
        | struct_or_union_specifier
        | enum_specifier
        | TYPEDEF_NAME          /* after it has been defined as such */
        ;

struct_or_union_specifier
        : struct_or_union '{' struct_declaration_list '}'
        | struct_or_union IDENTIFIER '{' struct_declaration_list '}'
        | struct_or_union IDENTIFIER
        ;

struct_or_union
        : STRUCT
        | UNION
        ;

struct_declaration_list
        : struct_declaration
        | struct_declaration_list struct_declaration
        ;

struct_declaration
        : specifier_qualifier_list ';'  /* for anonymous struct/union */
        | specifier_qualifier_list struct_declarator_list ';'
        | static_assert_declaration
        ;

specifier_qualifier_list
        : type_specifier specifier_qualifier_list
        | type_specifier
        | type_qualifier specifier_qualifier_list
        | type_qualifier
        ;

struct_declarator_list
        : struct_declarator
        | struct_declarator_list ',' struct_declarator
        ;

struct_declarator
        : ':' constant_expression
        | declarator ':' constant_expression
        | declarator
        ;

enum_specifier
        : ENUM '{' enumerator_list '}'
        | ENUM '{' enumerator_list ',' '}'
        | ENUM IDENTIFIER '{' enumerator_list '}'
        | ENUM IDENTIFIER '{' enumerator_list ',' '}'
        | ENUM IDENTIFIER
        ;

enumerator_list
```

```
        : enumerator
        | enumerator_list ',' enumerator
        ;

enumerator      /* identifiers must be flagged as ENUMERATION_CONSTANT */
        : enumeration_constant '=' constant_expression
        | enumeration_constant
        ;

atomic_type_specifier
        : ATOMIC '(' type_name ')'
        ;

type_qualifier
        : CONST
        | RESTRICT
        | VOLATILE
        | ATOMIC
        ;

function_specifier
        : INLINE
        | NORETURN
        ;

alignment_specifier
        : ALIGNAS '(' type_name ')'
        | ALIGNAS '(' constant_expression ')'
        ;

declarator
        : pointer direct_declarator
        | direct_declarator
        ;

direct_declarator
        : IDENTIFIER
        | '(' declarator ')'
        | direct_declarator '[' ']'
        | direct_declarator '[' '*' ']'
        | direct_declarator '[' STATIC type_qualifier_list assignment_expression ']'
        | direct_declarator '[' STATIC assignment_expression ']'
        | direct_declarator '[' type_qualifier_list '*' ']'
        | direct_declarator '[' type_qualifier_list STATIC assignment_expression ']'
        | direct_declarator '[' type_qualifier_list assignment_expression ']'
        | direct_declarator '[' type_qualifier_list ']'
        | direct_declarator '[' assignment_expression ']'
        | direct_declarator '(' parameter_type_list ')'
        | direct_declarator '(' ')'
        | direct_declarator '(' identifier_list ')'
        ;

pointer
        : '*' type_qualifier_list pointer
        | '*' type_qualifier_list
        | '*' pointer
        | '*'
        ;

type_qualifier_list
        : type_qualifier
        | type_qualifier_list type_qualifier
        ;


parameter_type_list
        : parameter_list ',' ELLIPSIS
        | parameter_list
        ;
```

```
parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration
        ;

parameter_declaration
        : declaration_specifiers declarator
        | declaration_specifiers abstract_declarator
        | declaration_specifiers
        ;

identifier_list
        : IDENTIFIER
        | identifier_list ',' IDENTIFIER
        ;

type_name
        : specifier_qualifier_list abstract_declarator
        | specifier_qualifier_list
        ;

abstract_declarator
        : pointer direct_abstract_declarator
        | pointer
        | direct_abstract_declarator
        ;

direct_abstract_declarator
        : '(' abstract_declarator ')'
        | '[' ']'
        | '[' '*' ']'
        | '[' STATIC type_qualifier_list assignment_expression ']'
        | '[' STATIC assignment_expression ']'
        | '[' type_qualifier_list STATIC assignment_expression ']'
        | '[' type_qualifier_list assignment_expression ']'
        | '[' type_qualifier_list ']'
        | '[' assignment_expression ']'
        | direct_abstract_declarator '[' ']'
        | direct_abstract_declarator '[' '*' ']'
        | direct_abstract_declarator '[' STATIC type_qualifier_list assignment_expression ']'
        | direct_abstract_declarator '[' STATIC assignment_expression ']'
        | direct_abstract_declarator '[' type_qualifier_list assignment_expression ']'
        | direct_abstract_declarator '[' type_qualifier_list STATIC assignment_expression ']'
        | direct_abstract_declarator '[' type_qualifier_list ']'
        | direct_abstract_declarator '[' assignment_expression ']'
        | '(' ')'
        | '(' parameter_type_list ')'
        | direct_abstract_declarator '(' ')'
        | direct_abstract_declarator '(' parameter_type_list ')'
        ;

initializer
        : '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        | assignment_expression
        ;

initializer_list
        : designation initializer
        | initializer
        | initializer_list ',' designation initializer
        | initializer_list ',' initializer
        ;

designation
        : designator_list '='
        ;
```

```
designator_list
        : designator
        | designator_list designator
        ;

designator
        : '[' constant_expression ']'
        | '.' IDENTIFIER
        ;

static_assert_declaration
        : STATIC_ASSERT '(' constant_expression ',' STRING_LITERAL ')' ';'
        ;

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

labeled_statement
        : IDENTIFIER ':' statement
        | CASE constant_expression ':' statement
        | DEFAULT ':' statement
        ;

compound_statement
        : '{' '}'
        | '{'  block_item_list '}'
        ;

block_item_list
        : block_item
        | block_item_list block_item
        ;

block_item
        : declaration
        | statement
        ;

expression_statement
        : ';'
        | expression ';'
        ;

selection_statement
        : IF '(' expression ')' statement ELSE statement
        | IF '(' expression ')' statement
        | SWITCH '(' expression ')' statement
        ;

iteration_statement
        : WHILE '(' expression ')' statement
        | DO statement WHILE '(' expression ')' ';'
        | FOR '(' expression_statement expression_statement ')' statement
        | FOR '(' expression_statement expression_statement expression ')' statement
        | FOR '(' declaration expression_statement ')' statement
        | FOR '(' declaration expression_statement expression ')' statement
        ;

jump_statement
        : GOTO IDENTIFIER ';'
        | CONTINUE ';'
        | BREAK ';'
        | RETURN ';'
```

```
        | RETURN expression ';'
        ;

translation_unit
        : external_declaration
        | translation_unit external_declaration
        ;

external_declaration
        : function_definition
        | declaration
        ;

function_definition
        : declaration_specifiers declarator declaration_list compound_statement
        | declaration_specifiers declarator compound_statement
        ;

declaration_list
        : declaration
        | declaration_list declaration
        ;

%%
#include <stdio.h>

void yyerror(const char *s)
{
        fflush(stdout);
        fprintf(stderr, "*** %s\n", s);
}
```