C Syntax

This is an adaptation of the grammars for the versions of C listed in the ISO Standards. The syntax is significantly simplified – but still embodies the 1989 ANSI Standard (or the 1990 ISO Standard derived from it), as well as the revisions made in the 1999 and 2010 ISO Standards. The main differences are as follows:

- Abstract declarators can now be empty. This introduces an ambiguity Dc → Dc() → (), for a function declarator with an empty parameter list, versus Dc → (Dc) → (), for a parenthesized declarator. The ambiguity is resolved in favor of the former in the ISO grammars and, here, by explicit stipulation.
- Both declarators and expressions are associated with precedence levels in the Standards' grammars. The precedence levels are not associated with the operators but with the declarators and expressions themselves. Here, this is made explicit. The expression syntax has 17 levels, while the declarator syntax has 2. In both cases, a simplified syntax is listed alongside in which the precedence levels are removed.
- The "not part of C, but ought to be" rules are my preferred extensions to the language.
- Other constraints that the Standards wove into the syntax are stipulated separately and kept out of the syntax, since they don't belong there.

There are two other ambiguities that are inherited from the original grammar, which are left unchanged here:

- The function declarator Dc → Dc() with empty parameter list is ordinarily K&R style (meaning: unspecified parameter list). However, in a function definition, it may also be read as a function prototype, indicating the function has an empty parameter list. In C one may specify an empty parameter list for a function definition with "void". In all places other than function definitions, one *must* use "void" to indicate an empty parameter list, or else it is treated as K&R style. The 1989, 1999 and 2011 Standards are all clear on this matter. For function definitions, the ambiguity is harmless, since both readings (K&R style vs. function prototype) mean the same thing – a function with an empty parameter list. But the Standards are phasing out K&R style.
- The if-then-else ambiguity is a common feature to most Algol-derived languages and is always resolved in the fashion "if (A) if (B) S else T" = "if (A) { if (B) S else T }" rather than as "if (A) { if (B) S } else T". The latter is equivalent to "if (!(A)) T else if (B) S", while the former cannot be so easily transformed.

Additions made since the 1989/1990 standard are highlighted like so

- those present in C99, the 1999 C standard
- those present in C1X, the 2011 C standard

Other changes are noted in the adjoining commentary highlighted like so.

Finally: this account is based solely on whatever portions of existing or previous standards are freely available. None of the standards are published as open standards, and there will be no attempt here to recapitulate any information that is not freely available.

1. Lexicon

The morphology of the lexicon is left unspecified here. In the main syntax, actual morphemes are indicated in colored boldface. In several cases, the boldface does not indicate an actual item, but a class of items. This includes the following:

- **X**: Name. Names (or "identifiers") are used for variables, functions, function parameters, **goto** labels, user-defined types names (or **typedef** names), tags attached to **struct**, **union** and **enum** types, members in **struct** and **union** types and the constants in **enum** types.
- **C**: Literal constants. Includes: character strings and characters; base 8, 10 and 16 integer numerals, base 10 and 16 rational numerals.
- **qual**: Type qualifiers (**volatile**, **const**, **restrict**, **_Atomic**),
- **store**: Storage class specifiers (**auto**, **register**, **static**, **extern**, **typedef**, **_Thread_local**),
- **func_sp**: Function specifiers (**inline**, **_Noreturn**).
- **scalar**: Empty or scalar type specifiers (**void**, **char**, **int**, **short**, **long**, **float**, **double**, **signed**, **unsigned**, **_Bool**, **_Complex**).

The detailed composition of the identifiers and constants is left unspecified here. Instead, the abbreviations **X** and **C** are used for them, here and below. The interpretation of the identifiers depends on context, and these details are not specified here either.

In addition, the major operator classes include the following:

- **pref**: Prefix operators – includes the subclasses: **un**, **inc**.
- **inf**: Infix operators – includes the subclasses: **as**, **eq**, **rel**, **sh**, **add**, **mul**, as well as: **,**, ||, **&&**, |, ^, **&**.
- **acc**: Infix operators for structure/union member access: **.**, **->**.
- **postf**: Postfix operators – includes only the subclass: **inc**.

The detailed composition of the operator subclasses is

- **as**: Assignment operators – =,*****=, /=, **%**=, +=, –=, <<=, >>=, **&**=, ^=, |=.
- **eq**: Equality comparison operators – **==**, **!=** .
- **rel**: Relational comparison operators – <, >, <=, >=.
- **sh**: Bit-wise shift operators <<, >>.
- **add**: Arithmetic and pointer additive operators +, –.
- **mul**: Arithmetic multiplicative operators *****, /, **%**.
- **un**: Prefix unary operators **&**, *****, +, –, ~, **!** .
- **inc**: Increment/decrement operators ++, —.

The classes overlap in the following places:

- **un** and **inf** both contain **&**, an operator ***** of **mul**, and two operators + and – of **add**, and
- **pref** and **postf** both contain both operators ++ and — of **inc**.

This part of the list is going to be shortly expanded to cover the details of both the morphology and the preprocessor.

2. Phrase Structure Rules, Notation

The syntax is listed as a sequence of *Phrase Structure Rules* all of the form

PhraseType → *Pattern*

followed by the main, top-level, structure of the language – listed as a *Pattern*.

Indicating the constituency of the phrase types and the main structure, a pattern is a Kleene-algebraic expression composed of morphemes/lexical classes and phrase types, with the following notation:

A + B	Alternatives	A or B
AB	Juxtaposition	A then B
[A]	Optional	0 or 1 of A.
	Empty phrase	This occurs with declarators: Dc ␣ .
A^*	Optional iteration	0, 1, 2 or more of A: $A^* = [A^+]$
A^+	Iteration	1, 2 or more of A: $A^+ = AA^*$
<A>	Comma-separated list	<A> = (A (**,** A)*)
(A)	Grouping	
K	Literal	Morpheme or lexical class
	This includes the literals for **( ) * < > + |**.	

A rule of the form A → B + C indicating alternatives is equivalent to the combination of rules A → B and A → C separately stated for each alternate. Therefore, this notation is only used sparingly. It's common to use the notation A | B to denote alternatives, instead of A + B, but the former is more difficult to see, so we adopt the latter notation. Grouping is understood as AB + C = (AB) + C, not A (B + C). Note also the distinction between $B^+ C = (B^+) C$ and B + C.

This part of the list will be expanded shortly to discuss the details of the algebraic formalism used to derive the parser from the grammar, and the computations involved in doing so.

3. Declarative Level

3.1. Declarations and Definitions

Declarations, Definitions and Types

		Phrase Structure Rule	Comment
D_F	→	$Sp^+\ Dc_F\ D^*\ S$	Function definitions (only compound statements allowed) Change made in 1999: $Sp^*\ Dc_F\ D^*\ S$ → $Sp^+\ Dc_F\ D^*\ S$
D_M	→	Sp^+ [<Dc_M>] **;** + *Assert*	Component members of structure and union types Change made in 2010: Sp^+ <Dc_M> **;** → Sp^+ [<Dc_M>] **;**
D	→	Sp^+ <Dc_I>**;** + *Assert*	Top-level and block-level declarations
T	→	$Sp^+\ Dc_T$	Types
T	→	**typeof** E	*(Not part of C, but ought to be)*
D_P	→	$Sp^+\ Dc_P$	Parameters
D_E	→	<Dc_E> **[,]**	Enumerations
Assert	→	**_Static_assert (** E_C **,** E_C **) ;**	Static assertion. The second E_C may only be a string literal.

3.2. Specifiers for Basic and Composite Types

Type Specifiers

		Phrase Structure Rule	Comment
Sp	→	**qual**	
Sp	→	**store**	Not allowed in D_M or T.
Sp	→	**func_sp**	Not allowed in D_M or T.
Sp	→	**_Alignas (** (T + E_C) **)**	Not allowed in D_M or T.
Sp	→	**scalar**	
Sp	→	**_Atomic (** T **)**	
Sp	→	**X**	
Sp	→	**struct X** + **struct** [**X**] **{** D_M^+ **}**	
Sp	→	**union X** + **union** [**X**] **{** D_M^+ **}**	
Sp	→	**enum X** + **enum** [**X**] **{** D_E **}**	
Sp	→	**(** T **)**	*(Not part of C, but ought to be)*

3.3. Declarators for Function, Pointer and Array Types

Type Declarator Contexts

Phrase Structure Rule			Comment
Dc_E	→	**X** [**=** E_C]	Enumeration type members
Dc_I	→	Dc_0 [**=** *Init*]	Top-level and block-level declarations
Dc_M	→	Dc_0	Structure and union members
Dc_M	→	[Dc_0] **:** E_C	Bit-Fields structure members
Dc_P	→	Dc_0	Parameters
Dc_T	→	Dc_0	Types
Dc_F	→	Dc_0	Function definitions

Type Declarators

		Phrase Structure Rule	Comment	
Dc_0	→	Dc_1	(2 precedence levels)	
Dc_0	→	*** qual*** Dc_0	Pointers	Dc → *** qual*** Dc
Dc_1	→	**X**	Variable name. Not allowed in Dc_T.	Dc → [**X**]
Dc_1	→		Empty declarator. Only in Dc_P, Dc_T.	
Dc_1	→	**(** Dc_0 **)**	Dc_1 → **( )** is not allowed.	Dc → **(** Dc **)**
Dc_1	→	Dc_1 [*Dim*]	Arrays	Dc → Dc [*Dim*]
Dc_1	→	Dc_1 **(** [<D_P> [**, ...**]] **)**	Function prototype.	Dc → Dc **(** [<D_P> [**, ...**]] **)**
Dc_1	→	Dc_1 **(** [<**X**>] **)**	K&R prototype. Not allowed in Dc_T.	Dc → Dc **(** [<**X**>] **)**
Dim	→	**qual*** [E_A + *****]	Array dimensions. **qual**'s not allowed for array dimensions in Dc_T. Change made in 1999: [E_A] → **qual*** [E_A + *****]	
Dim	→	**qual*** **static qual*** E_A	Static array dimensions. **qual**'s may not occur both before and after **static**.	

4. Functional Level

Expression Contexts			
Phrase Structure Rule			**Comment**
E	→	E_0	General expressions
E_A	→	E_1	Array dimensions
			Change made in 1999: E_C → E_1
Init	→	E_1 + E_S	Scalar and structured initializers
E_S	→	**{** <[*Sub*] *Init*> [**,**] **}**	Structured expressions with initialization
E_C	→	E_2	Constant expressions (in enumerator initializers, bit-fields and case labels)
Sub	→	(**[** E_C **]** + **.** **X**)$^+$ **=**	Subobject designator (for structured expressions)

Expressions				
	Phrase Structure Rule		**Comment**	
E_i	→	E_{i+1}	(i = 0,…,16) (17 Precedence Levels)	
E_2	→	E_3 **?** E_0 **:** E_2	Conditional	*E* → *E* **?** *E* **:** *E*
E_0	→	E_0 **,** E_1	Sequence	*E* → *E* **inf** *E*
E_1	→	E_{14} **as** E_1	Assignment	
E_3	→	E_3 **||** E_4	Logical OR	
E_4	→	E_4 **&&** E_5	Logical AND	
E_5	→	E_5 **|** E_6	Bit-wise OR	
E_6	→	E_6 **^** E_7	Bit-wise XOR	
E_7	→	E_7 **&** E_8	Bit-wise AND	
E_8	→	E_8 **eq** E_9	Equality	
E_9	→	E_9 **rel** E_{10}	Relational	
E_{10}	→	E_{10} **sh** E_{11}	Bit-shift	
E_{11}	→	E_{11} **add** E_{12}	Additive	
E_{12}	→	E_{12} **mul** E_{13}	Multiplicative	
E_{13}	→	**(** *T* **)** E_{13}	Type-casting	*E* → **(** *T* **)** *E*
E_{14}	→	**un** E_{13}	Prefix operators	*E* → **pref** *E*
E_{14}	→	**inc** E_{14}	Prefix increment	
E_{14}	→	**sizeof** E_{14}	Expression/type size	*E* → **sizeof** *E*
E_{14}	→	**sizeof (** *T* **)**		*E* → **sizeof (** *T* **)**
E_{14}	→	**alignof (** *T* **)**	Expression type alignment	*E* → **alignof (** *T* **)**
E_{15}	→	E_{15} **inc**	Postfix increment	*E* → *E* **postf**
E_{15}	→	E_{15} **acc X**	Structure and union access	*E* → *E* **acc X**
E_{15}	→	E_{15} **[** E_0 **]**	Array access	*E* → *E* **[** *E* **]**
E_{15}	→	E_{15} **(** [<E_1>] **)**	Function call	*E* → *E* **(** [<*E*>] **)**
E_{15}	→	**(** *T* **)** E_S	Structured expressions	*E* → **(** *T* **)** E_S
E_{16}	→	**(** E_0 **)**	Sub-expressions	*E* → **(** *E* **)**
E_{16}	→	**_Generic (** E_1 **,** <*G*> **)**	Generic selection	*E* → **_Generic (** *E* **,** <*G*> **)**
E_{16}	→	**C**	Literal constants	*E* → **C**
E_{16}	→	**X**	Variables	*E* → **X**
G	→	(*T* + **default) :** E_1	Generic association (used with generic selections)	

5. Procedural Level

Statements			
		Phrase Structure Rule	**Comment**
S	→	**X :** *S*	Labeled statements
S	→	**case** E_C **:** *S*	
S	→	**default :** *S*	
S	→	**{** $(D + S)^*$ **}**	Compound statement Change made in 1999: **{** D^* S^* **}** → **{** $(D + S)^*$ **}**
S	→	**if (** *E* **)** *S* [**else** *S*]	Branch statements
S	→	**switch (** *E* **)** *S*	
S	→	**while (** *E* **)** *S*	Loop statements
S	→	**do** *S* **while (** *E* **) ;**	
S	→	**for (** ([*E*]**;** + *D*) [*E*] **;** [*E*] **)** *S*	
S	→	[*E*] **;**	Expression & empty statement
S	→	**goto X ;**	Jump statements
S	→	**continue ;**	
S	→	**break ;**	
S	→	**return** [*E*] **;**	

6. Top Level

Top Level
Phase Structure
$(D_F + D)^+$