The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.1 Notation

In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**.

Commentary

A *terminal* is a token that can appear in the source code. A *nonterminal* is the name of a syntax rule used to group together zero or more terminals and other nonterminals. The nonterminals can be viewed as a tree. The root is the nonterminal *translation-unit*. The terminals are the leaves of this tree.

Syntax analysis is the processing of a sequence of terminals (as written in the source) via various nonterminals until the nonterminal *translation-unit* is reached. Failure to reach this final nonterminal, or encountering an unexpected sequence of tokens, is a violation of syntax.

The syntax notation used in the C Standard is not overly formal; it is often supported by text in the semantics clause. The C syntax can be written in LALR(1) form. (Although some reorganization of the productions listed in the standard is needed), assuming the **typedef** issue is fudged (the only way to know whether an identifier is a typedef name or not is to look it up in a symbol table, which introduces a context dependency; the alternative of syntactically treating a typedef name as an identifier requires more than one token lookahead.) This also happens to be the class of grammars that can be processed by yacc and many other parser generators.

```
1    A(B) /* Declare B to have type A, or call function A with argument B? */
```

The syntax specified in the C Standard effectively describes four different grammars:

preprocess-
ing token
syntax
transla-
tion phase
2
preprocessor
directives
syntax
transla-
tion phase
3
token
syntax

1. A grammar whose start symbol is *preprocessing-token*; the input stream processed by this grammar contains the source characters output by translation phase 2.

2. A grammar whose start symbol is *preprocessing-file*; the input stream processed by this grammar contains the *preprocessing-token*s output by translation phase 3.

3. A grammar whose start symbol is *token*; the input to this grammar is a single *preprocessing-token*. The syntax of the characters forming the *preprocessing-token* need to form a valid parse of the *token* syntax.

transla-
tion unit
syntax
transla-
tion phase
6

4. A grammar whose start symbol is *translation-unit*; the input stream processed by this grammar contains the *token*s output by translation phase 6.

The *preprocessor-token* and *token* syntax is sometimes known as the *lexical grammar* of C.

There are many factors that affect the decision of whether to specify language constructs using syntax or English prose in a Constraints clause. The C Standard took the approach of having a relatively simple, general syntax specification and using wording in constraints clauses to handle the special cases. There are techniques available (e.g., two-level grammars) for specifying the requirements (including the type rules) through syntax. This approach was famously taken, because of its impenetrability, in the specification of Algol 68[4]

The first published standard for syntax notation was BS 6154:1981, *Method of Defining— Syntactic metalanguage*. Although a British rather than an International standard this document was widely circulated. It was also used as the base document for ISO/IEC 14977,[3] which specified an extended BNF. Most compiler books limit there discussion to LR and LL related methods. For a good introduction to a variety of parsing methods see Grune and Jacobs,[2] which is now freely available online.

C++

1.6p1

In the syntax notation used in this International Standard, syntactic categories are indicated by italic type, and literal words and characters in `constant width` *type.*

The C++ grammar contains significantly more syntactic ambiguities than C. Some implementations have used mainly syntactic approaches to resolving these,[6] while others make use of semantic information to guide the parse.[1] For instance, knowing what an identifier has been declared as, simplifies the parsing of the following:

```
1  template_name     < a , b > - 5  // equivalent to (template_name < a , b >) - 5)
2  non_template_name < a , b > - 5  // equivalent to (non_template_name < a) , (b > - 5)
```

Other Languages

Most computer languages definitions use some form of semiformal notation to define their syntax. The availability of parser generators is an incentive to try to ensure that the syntax is, or can be, rewritten in LALR(1) form.

Some languages are line-based, for instance Fortran (prior to Fortran 90). Each statement or declaration has its syntax and sequences of them can be used to build functions; there is no nesting of constructs over multiple statements.

The method used to analyze the syntax of a language can be influenced by several factors. Many Pascal translators used a handwritten recursive descent parser; the original, widely available, implementation of the language uses just this technique. Many Fortran translators use ad hoc techniques to process each statement or declaration as it is encountered, the full power of a parser generator not being necessary (also, it is easier to perform error recovery in an ad hoc approach).

Common Implementations

Most implementations use an automated tool-based approach to analyzing some aspects of the C syntax. Although tools do exist for automating the lexical grammar (e.g., `lex`) a handwritten lexer offers greater flexibility for handling erroneous input. The tool almost universally used to handle the language syntax is `yacc`, or one of its derivatives. The preprocessor syntax is usually handled in a handwritten ad hoc manner, although a few implementations use an automated tool approach.

Tools that do not require the input to be consolidated into tokens, but parsed at the character sequence level, are available.[5]

Coding Guidelines

It is unlikely that a coding guideline recommendation would specify language syntax unless an extension was being discussed. Coding guideline documents that place restrictions on the syntax that can be used are rare.

---

385 A colon (`:`) following a nonterminal introduces its definition.

Commentary

This is a simple notation. Some syntax notations have been known to be almost as complex as the languages they describe.

C++

The C++ Standard does not go into this level of detail (although it does use this notation).

Other Languages

The notation *::=* is used in some language definitions and more formal specifications. The notation = is specified by the ISO Standard for extended BNF.[3]

---

386 Alternative definitions are listed on separate lines, except when prefaced by the words "one of".

Commentary

The Syntax clauses in the standard are written in an informal notation. As typified by this simple, nonformal rule.

Other Languages

The character | often used to indicate alternative definitions. This character is used by ISO/IEC 14977.

An optional symbol is indicated by the subscript "opt", so that

387

$$\{ \; expression_{opt} \; \}$$

indicates an optional expression enclosed in braces.

Commentary

This is a more informal notation. In this example *expression* is optional; the braces are not optional.

Other Languages

Some languages use a more formal syntax notation where an empty alternative indicates that it is possible for a nonterminal to match against nothing (i.e., the symbol is optional).

When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

388

Commentary

There are some places in the main text where words are separated by hyphens instead of spaces.

C90

This convention was not explicitly specified in the C90 Standard.

C++

The C++ Standard does not explicitly specify the conventions used. However, based on the examples given in clause 1.6 and usage within the standard, the conventions used appear to be the reverse of those used in C (i.e., syntactic categories are italicized and words are separated by hyphens).

Coding Guidelines

Companies may have typographical conventions for their documents which differ from those used by ISO. The issue of which typographical conventions to use in a company's coding guideline document is outside the scope of these coding guidelines.

A summary of the language syntax is given in annex A.

389

C90

The summary appeared in Annex B of the C90 Standard, and this fact was not pointed out in the normative text.

References

1. T. H. Gibbs. *The design and implementation of a parser and front-end for the ISO C++ language and validation of the parser*. PhD thesis, Clemson University, May 2003.

2. D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

3. ISO. *ISO/IEC 14977:1996 Information technology —Syntactic meta-language —Extended BNF*. ISO, 1996.

4. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. *Algol 68*. Springer–Verlag, 1976.

5. E. Visser. Scannerless generalised-LR parsing. Technical Report Report P9707, University of Amsterdam, Aug. 1997.

6. E. D. Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, June 2001.