
Computer Programming: Skills & Concepts (CP1)

Syntax of programming languages

John Longley

20th November 2004



Backus-Naur Form (BNF)

A way of describing syntactically correct (i.e. legal) programs in some language.

Example: legal forms of *identifiers* (such as variable names) in C:

$$\textit{identifier} ::= \{ \textit{letter} \mid \textit{underscore} \}_1 \{ \textit{letter} \mid \textit{underscore} \mid \textit{digit} \}_{0+}$$
$$\textit{digit} ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$$
$$\textit{letter} ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$$
$$\textit{underscore} ::= _$$

This is (arguably) more understandable and precise than:

“An identifier is a non-empty sequence of letters, digits, and underscores which begins with a letter or an underscore, but not with a digit.”

BNF Conventions

- Syntactic categories or nonterminals are written in *italic font*.
- Tokens or terminals in typewriter font.

A grammar is a sequence of productions or rules which take the form

“left hand side” ::= “right hand side”

where “left hand side” is a single syntactic category, and “right hand side” is an expression built up from tokens and syntactic categories by any of the following constructs: *[Dramatic pause. . .]*

- $S_1 S_2 \dots S_n$ (juxtaposition or sequencing)
- $\dots | \dots | \dots | \dots$ (alternatives)
- $\{\dots\}$ (bracket expression for other operation)
- \dots_1 (exactly one)
- \dots_{opt} (zero or one)
- \dots_{0+} (zero or more)
- \dots_{1+} (one or more but not zero)

Meaning of BNF

A **string** is a sequence of **tokens**.

For every **syntactic category**, the grammar determines a set of strings which belong to that syntactic category.

A string belongs to a syntactic category if it can be obtained from the syntactic category by repeatedly applying productions until only tokens remain. “Applying a production” means replacing an occurrence of the left hand side by an instance of the corresponding right hand side.

Identifiers again

$$\textit{identifier} ::= \{ \textit{letter} \mid \textit{underscore} \}_1 \{ \textit{letter} \mid \textit{underscore} \mid \textit{digit} \}_{0+}$$
$$\textit{identifier} \rightsquigarrow \textit{letter} \rightsquigarrow \textit{i}$$
$$\textit{identifier} \rightsquigarrow \textit{underscore} \textit{letter} \textit{letter} \rightsquigarrow _ \textit{letter} \textit{letter} \rightsquigarrow _ \textit{I} \textit{letter} \rightsquigarrow _ \textit{Io}$$

Bigger example: Real number constants

These are given in C by the following rules.

$$\begin{aligned} \textit{floating_constant} & ::= \\ & \quad \textit{fractional_constant} \{ \textit{exponential_part} \}_{\text{opt}} \{ \textit{floating_suffix} \}_{\text{opt}} \\ & \quad | \textit{digit_sequence} \textit{exponential_part} \{ \textit{floating_suffix} \}_{\text{opt}} \\ \textit{fractional_constant} & ::= \\ & \quad \{ \textit{digit_sequence} \}_{\text{opt}} . \textit{digit_sequence} \\ & \quad | \textit{digit_sequence} . \\ \textit{digit} & ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 \\ \textit{exponential_part} & ::= \{ \textit{e} | \textit{E} \}_1 \{ + | - \}_{\text{opt}} \textit{digit_sequence} \\ \textit{floating_suffix} & ::= \textit{f} | \textit{F} | \textit{l} | \textit{L} \\ \textit{digit_sequence} & ::= \{ \textit{digit} \}_{1+} \end{aligned}$$

Statements

```
statement ::= ...  
            | expression_statement  
            | selection_statement  
            | { statement_list }  
  
expression_statement ::= expressionopt ;  
statement_list ::= { statement }0+  
selection_statement ::= if ( expression ) statement  
                        | if ( expression ) statement else statement  
                        | ...
```

NB. This represents only a fragment!

Note that the curly brackets appear in two guises, distinguished by font: as part of the BNF notation (e.g., { *statement* }₀₊), and as terminal symbols (e.g., { *statement_list* }).

Example

```
if (x == 0) x = 1; else if (x == 1) {x = 0;}
```

Tokens in C

For the grammar describing C-programs the tokens are:

- identifiers (these seem generally to be treated as tokens, even though we saw earlier that they can be broken down further);
- keywords (such as `if`, `return`, `void`);
- punctuation symbols (`,`, `;`, `...`);
- operators (`*`, `+`, `==`, `...`).

Non self-terminating tokens such as keywords and identifiers are separated from one another by [separators](#) newline, blank, tab.

Parsing

It can be effectively decided whether a given string belongs to a syntactic category. In the positive case the derivation leading to it is called the **syntax tree** or **parse tree**.

The process of finding a syntax tree or rejecting a string is called *parsing*. Grammars for programming language are such that parsing can be done quickly.

The syntax tree is then passed on to subsequent compilation stages.

Ambiguity

Sometimes a string admits more than one derivation. This is called ambiguity.

- $x + y + z$
- $x + y * z$
- `if (x == 0) if (y == 0) z = 0; else z = 1;`

Special conventions such as operator precedence and associativity or the “dangling else convention” are used to resolve ambiguities.

Semantics (= meaning)?

Suppose that the syntactic category *natural_number* is defined by

$$\textit{natural_number} ::= \textit{digit} \mid \textit{natural_number} \textit{digit}$$

where *digit* is as before. Strings in the syntactic category *natural_number* are intended to represent non-negative integer constants.

Try to define a function V that assigns an integer value to such strings:

$$V(0) = 0,$$

$$\vdots$$

$$V(9) = 9;$$

and

$$V(\alpha 0) = 10 V(\alpha) + 0,$$

$$\vdots$$

$$V(\alpha 9) = 10 V(\alpha) + 9.$$

The symbol α stands for any string which is a *natural_number*; note the crucial distinction between mathematics (012...9) and typewriter (012...9) font here!

OK, but imagine extending this program to the rest of the language!

Summary

- Syntactically correct programs can be described using BNF notation.
- This idea was illustrated using examples from the C programming language.

Appendix B of Kelley and Pohl claims to give the BNF grammar for C, but some is left out!