
Computer Programming: Skills & Concepts (CP1)

Syntax of Programming Languages

17th November 2009



- $S_1 S_2 \dots S_n$ (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)

Backus-Naur Form (BNF)

A way of describing syntactically correct programs. 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 syntactic category and the “right hand side” is an expression built up from tokens and syntactic categories by any of the following constructs:

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 the corresponding right hand side.

Example: letters and digits

```
digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
letter ::= lowercase_letter | uppercase_letter
lowercase_letter ::= a | ... | z
uppercase_letter ::= A | ... | Z
underscore ::= _
```

$digit \rightsquigarrow 3$, so the string 3 belongs to the syntactic category *digit*.

$letter \rightsquigarrow lowercase_letter \rightsquigarrow k$, so k belongs to the syntactic category *letter*.

N.B. The ... is *not* proper BNF – it's a human abbreviation of something too long and obvious to write down.

Example: identifiers

```
identifier ::= {letter | underscore}_1 {letter | underscore | digit}_{0+}
```

$identifier \rightsquigarrow letter \rightsquigarrow lowercase_letter \rightsquigarrow i$

$identifier \rightsquigarrow underscore \ letter \ letter \rightsquigarrow$
 $underscore \ uppercase_letter \ lowercase_letter \rightsquigarrow$
 $_Io$

This is (arguably) more understandable and accurate than:

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

Real number constants

These are given in C by the following rules.

```
floating_constant ::=
    fractional_constant {exponential_part}_{opt} {floating_suffix}_{opt}
    | digit_sequence exponential_part {floating_suffix}_{opt}
fractional_constant ::=
    {digit_sequence}_{opt} . digit_sequence
    | digit_sequence .
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
exponential_part ::= {e | E}_1 {+ | -}_{opt} digit_sequence
floating_suffix ::= f | F | l | L
digit_sequence ::= {digit}_{1+}
```

Statements

```
statement ::= ...
            | expression_statement
            | selection_statement
            | { statement_list }
expression_statement ::= expression_{opt} ;
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!