

Syntax (I)

Syntax in English vs. in Programming Languages

- A sentence in English can be composed of an adjective followed by a noun followed by a verb followed by an adverb.
- There are many other forms of sentences in English. But for now, we just use this simple form.

sentence $\rightarrow$ *adjective noun verb adverb*

adjective $\rightarrow$ *good|delicious|green|beautiful*

noun $\rightarrow$ *pens|rice|sun|ideas*

verb $\rightarrow$ *eats|sleep|write|jumps*

adverb $\rightarrow$ *deeply|well|furiously|calmly*

Green ideas sleep furiously.

- Good, delicious, green, and beautiful are adjective; Pen, rice, sun, and ideas are nouns; verb can be chosen from eat, sleep, write, and jump; Adverb can be selected from deeply, well, furiously, and calmly.
- Here we define a simple grammar of sentences in English. Using this grammar, we can compose a sentence like “Green ideas sleep furiously.” *This sentence is grammatically correct, but does not make any sense.*
- In [programming languages](#), we can think [syntax as the grammar](#) in English, and [semantics are the meaning](#) in English.
- The [syntax](#) of a programming language is [defined by a grammar](#).
- The [syntax](#) of a programming language is a [precise description](#) of [all](#) its [grammatically correct programs](#).

Categories of grammars

- **Noam Chomsky** defined [four categories](#) of grammars: [regular](#), [context-free](#), [context-sensitive](#), and [unrestricted](#).
- In all of these grammars, the process of [analyzing a string](#) involves [applying a single rule to a single location](#) in [serial order](#).
- To analyze programming languages, the grammar should be [unambiguous](#) and be able to [match any level of](#) nested balanced [parentheses](#), also it must be [compilable in reasonable time](#). Therefore, [context-free](#) grammars are used by programming languages.

Context-free Grammar

- A context-free grammar consists of **four components**:
 - **T** : A set of **terminal** symbols, which defines the **alphabet** of the language. It includes **keyword** strings and the set of **legal characters** for creating symbols.
 - **N** : A set of **nonterminal** symbols, which defines the **abstract concepts** in the programming language, such as **expression** or **functions**.
 - **P** : A set of **productions**, which defines the **relationships** between **nonterminal** symbols and **terminal** symbols.
 - **S** : A **start symbol**, which identifies the **highest level concept** of the language.
- A production in a context-free grammar has the form
$$A \rightarrow \omega$$
- where the capital A is a single symbol from the set of nonterminal symbols N , ω is a sequence of symbols from the union of the set of terminal symbols T and the set of nonterminal symbols N ($T \cup N$). The length of the sequence, ω , is not less than one ($|\omega| \geq 1$).
- **Grammars define the lexical processing step of compilation** or interpretation and allow the computer (or programmer) to determine if the program has the **potential** to be converted into an executable program.
- A **syntactically correct** program **cannot necessarily** be converted into an **executable** program.
- **LL(k)**
 - A type of context-free grammar **parses the code in a single pass by looking ahead k symbols**.
 - LL(k) **parses** the input **from Left to right**, performing the **Leftmost derivation** of the string.
 - Most production compilers, such as gcc, are recursive descent LL(1) compilers.

BNF

- Most programming languages are formally described using variations of a notation called **Backus-Naur Form [BNF]**.
- A BNF grammar is a **context-free grammar**.
- The **terminal** and **nonterminal** symbols are **disjoint**.
- **Terminal** symbols are generally limited to those that can be **entered on a keyboard**. **Non-terminal** symbols represent **organizational concepts** in a programming language.
- The **start symbol** for a programming language is the **highest level concept** in the language that is not a component of any other concepts.
- Metasymbols used by BNF:
 - Vertical bar **|** : **or**
 - Dots **...** : **a series of values**
 - Right arrow **→** : **imply**

- **Example:** Use BNF describe integer formally.
 - An integer is a sequence of digits. To describe integer, we need to define digit first.
 - A digit can be any number between 0 and 9. So, we define digit as

$$Digit \rightarrow 0 | \dots | 9$$

- An integer can be a single digit or an integer followed by a digit. So, we defined integer as

$$\begin{aligned} Integer &\rightarrow Digit | IntegerDigit \text{ (left recursion, leftmost of } \omega \text{ is } Integer) \\ Integer &\rightarrow Digit | DigitInteger \text{ (right recursion, rightmost of } \omega \text{ is } Integer) \end{aligned}$$

- What are T, N, P, S in this example?
 - T is 0 ... 9
 - N is Digit and Integer
 - P are the two products
 - S is Integer.

EBNF

- To **increase the clarity and brevity** of syntax descriptions, **Extended BNF (EBNF)** was introduced.
- Based on BNF, EBNF has a few more metasymbols other than those from BNF:
 - Curly braces $\{\}$: include the enclosed symbols **0 or more times**
 - Parentheses $()$: include the enclosed symbols **1 or more times**
 - Square brackets $[]$: indicate an **optional sequence of symbols**
- **Example:** Describe the integer formally in EBNF

$$Integer \rightarrow Digit\{Digit\}$$

- The above production cannot avoid the situation where an integer with multiple digits starts with 0. How to rewrite the production to avoid the situation?

$$\begin{aligned} Digit &\rightarrow 0 | \dots | 9 \\ nonzeroDigit &\rightarrow 1 | \dots | 9 \\ Integer &\rightarrow Digit | nonzeroDigit(Digit) \end{aligned}$$

or

$$\begin{aligned} zero &\rightarrow 0 \\ nonzeroDigit &\rightarrow 1 | \dots | 9 \\ Integer &\rightarrow zero | nonzeroDigit\{[zero, nonzeroDigit]\} \end{aligned}$$