

# Syntax

## 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](#).
- There are other grammars that do not follow Chomsky’s hierarchy, such as L-systems, because they [apply one or more rules in parallel to all instances of a symbol](#) in the string. However, L-systems are not suited for compilation of a program; they are intended as grammars to generate an output string.
- Chomsky’s grammar categories are intended to support [deconstruction](#) of a [string into nonterminal symbols](#) to [evaluate](#) whether it is a [correct](#) program and enable [deduction](#) of its [semantics](#).
- [Regular](#) - corresponds to a standard [finite-state automata](#); its rules are *all of the form*  $A \rightarrow \omega B$  or *all of the form*  $A \rightarrow B\omega$ , where  $A$  and  $B$  are [single nonterminal symbols](#), and  $\omega$

is a **valid string of terminal symbols**. In both cases,  $B$  can be null if  $\omega$  is not null. The **ordering of  $B$  and  $\omega$  determine** whether the language is a **right regular** grammar or a **left regular** grammar.

- **Context-free** - corresponds to a **pushdown automata**; its rules are of the form  $A \rightarrow \omega$ , where  $A$  is a **single nonterminal symbol** and  $\omega$  is a **valid string** and can be *mix of terminal and nonterminal symbols, including  $A$* . Context-free grammars allow **recursive syntactic** structures.
- **Context-sensitive** - has rules of the  $\alpha \rightarrow \beta$ , where both  $\alpha$  and  $\beta$  can be **strings containing terminals and nonterminal symbols**; the **length of  $\beta$**  must be **greater than or equal to  $\alpha$**  so that the *program cannot shrink*. A context-sensitive grammar is **undecidable**, in the sense that we *cannot necessarily decide if a program is valid* given the grammar.
- **Unrestricted** - **identical to a context-sensitive** grammar, but **removes the length restriction** on  $\beta$ .
- Programming languages use **context-free** grammars because they are the **most powerful** grammars that are **decidable**.
  - A regular grammar, for example, cannot represent the need for opening and closing brackets in an expression. (cannot match any level of nested balanced parentheses)
  - An arbitrary context-sensitive grammar is unlikely to be compilable in reasonable time and may not have an unambiguous interpretation.

## Context-free Grammar

- **LL(k)**
  - Most programming languages are, in fact, part of a **subset of context-free** grammars called LL(k) grammars, which means the **grammar can be parsed in a single pass** through the code **by looking ahead k symbols**.
  - LL(k) **parses** the input **from Left to right**, performing the **Leftmost derivation** of the string.
  - Not all context-free grammars are LL(k) grammars, but all **LL(k) grammars** can be **converted into a parse tree** in a linear time in a single pass.
  - An LL(k) grammar **cannot contain left recursion** (e.g.  $A \rightarrow A + B$ )
  - An LL(k) parser begins with the start, or program symbol and **builds the parse tree from the top down**, ending with the leaves as rules that convert to the terminal symbols of input string.
- **LR(k)**
  - An **alternative to LL(k)** grammars. A **subset of context-free** grammars that **allow left recursion, but not right recursion** (e.g.  $A \rightarrow B + A$ ).
  - An LR(k) parser **looks ahead k symbols**, **parses** the input **from Left to right**, performing the **Rightmost derivation** of the string, and starts with the leaves of the parse tree and **builds up the tree from the bottom**.
- Most production compilers, such as gcc, are recursive descent LL(1) compilers.
- 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  $N$ ,  $\omega$  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,  $\omega$ , is not less than one ( $|\omega| \geq 1$ ).

- **Grammars are used to define two steps of compilation (or interpretation)**
  - **Lexical Analysis**: separates the stream of characters into separate tokens (e.g. string literals, identifiers such as function and variable names, numbers, and operators).
  - **Syntax Analysis**: determines whether or not the tokens form a legal program. This step creates the Abstract Syntax Tree, which is a helpful representation of how the grammar rules determine it is a legal program
- A **syntactically correct** program **cannot necessarily** be converted into an **executable** program. There may still be a semantic error, for example, suppose you have a function named `my_func` that has 2 parameters, and the code has the line  
`b = my_func( 1, 2, 3 )`  
 This is syntactically correct, but there are too many arguments, so the program should not be compiled (at least for a programming language like C or Java - Python may let this go until runtime).

## 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, such as `Program`. Some languages permit libraries or packages to sit at the top level.
- Metasyms 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

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

- 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.