Find your way here

CSC 250: Foundations of Computer Science I

Fall 2023 - Lecture 15

Amitabha Dey

Department of Computer Science
University of North Carolina at Greensboro

Compiler vs Interpreter

Compilers and interpreters are used to convert a high-level language into machine code.

- **Compiler:** A compiler translates code from a high-level programming language (like Python, JavaScript or Go) into machine code before the program runs. A compiler translates code written in a high-level programming language into a lower-level language like assembly language, object code and machine code (binary 1 and 0 bits). It converts the code ahead of time before the program runs.
  Example - Java, Microsoft Visual C++, Python

- **Interpreter:** An interpreter translates code written in a high-level programming language into machine code line-by-line as the code runs. An interpreter translates the code line-by-line when the program is running.
  Example - Python, Ruby, PHP

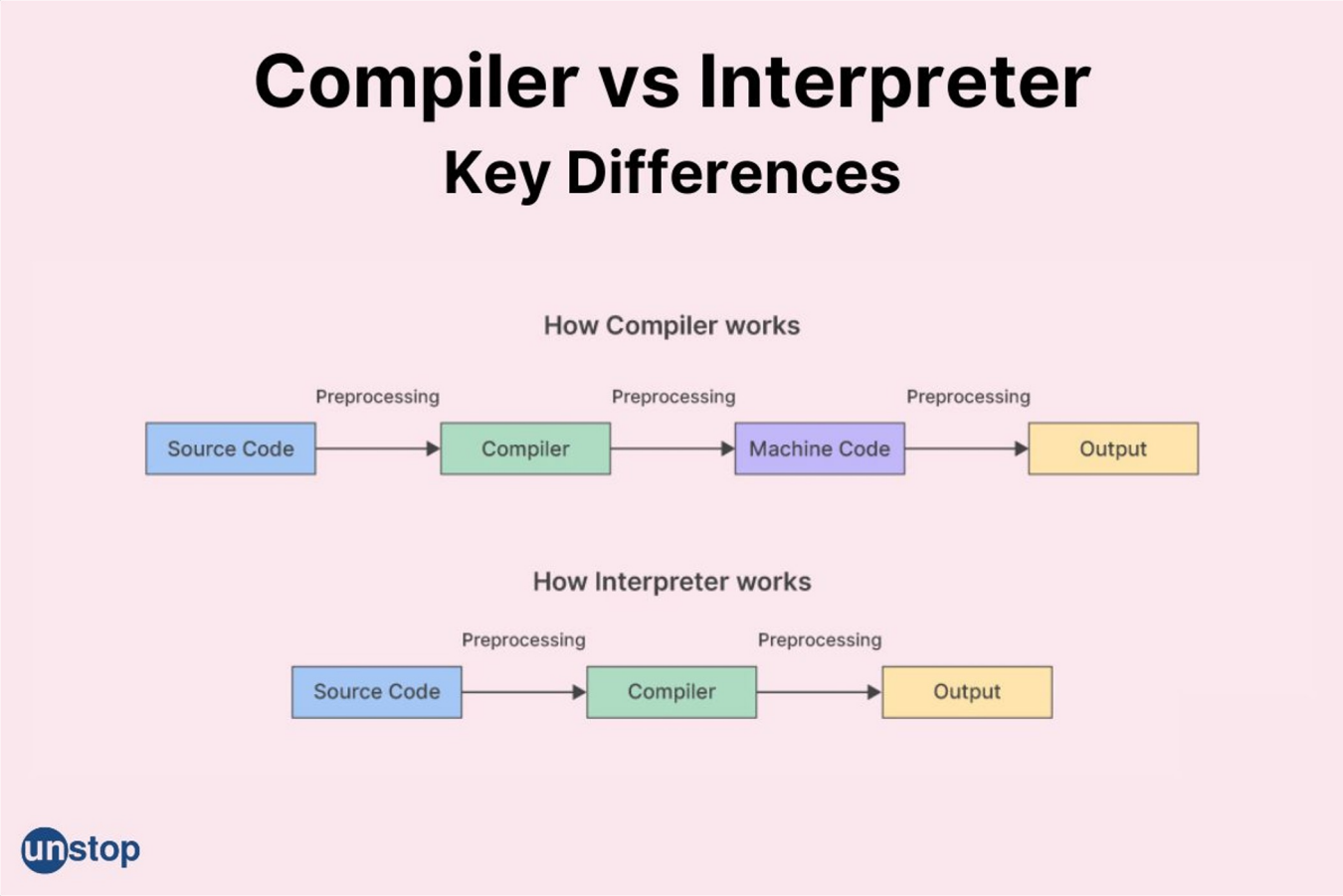


Image credit - https://unstop.com/

Differences between Compilers and Interpreters

- **Translation Process:** Compilers translate the entire source code from a high-level programming language into machine code or an intermediate representation in a single pass. Interpreters work by translating and executing the source code line by line or statement by statement without producing a separate intermediate or machine code.
- **Output:** Compilers generate a separate executable file or object file. This file can be executed independently without the need for the original source code or the compiler itself. Interpreters do not generate a separate executable file. They directly execute the code from the source files or source text.
- **Efficiency:** Compiled programs are generally more efficient in terms of execution speed because the code has been optimized and translated into machine code specific to the target hardware. Interpreted programs are generally slower in terms of execution speed because the code is translated and executed in real-time, without the same level of optimization as compilers.
- **Error Detection:** Compilers perform a complete analysis of the source code, detecting errors like syntax and type errors before generating the executable code. This can lead to a longer feedback loop for developers. Interpreters can detect errors as they execute the code, which can lead to faster feedback for developers. They may halt execution on the first encountered error.
- **Debugging:** Debugging compiled code can be more challenging because there may not be a one-to-one mapping between source code and machine code. Debugging tools are needed to aid in finding issues. Debugging interpreted code is often easier because the interpreter can provide more direct feedback on the location of errors. Developers can interactively test and debug code.

How does a compiler work?

A compiler is a complex piece of software that translates high-level programming code into low-level machine code or another intermediate representation. It performs this translation in a series of steps.

- **Lexical Analysis (Scanning):** The process begins with lexical analysis, where the source code is broken down into a sequence of tokens. A token is a meaningful unit, like a keyword, identifier, operator, or literal. The compiler reads the source code character by character, identifying tokens and their types. It also discards comments and unnecessary white spaces.

- **Syntax Analysis (Parsing):** In this phase, the compiler checks the syntax of the source code to ensure it adheres to the language's grammar rules. It builds an abstract syntax tree (AST) or a similar data structure, representing the program's structure and relationships between elements.

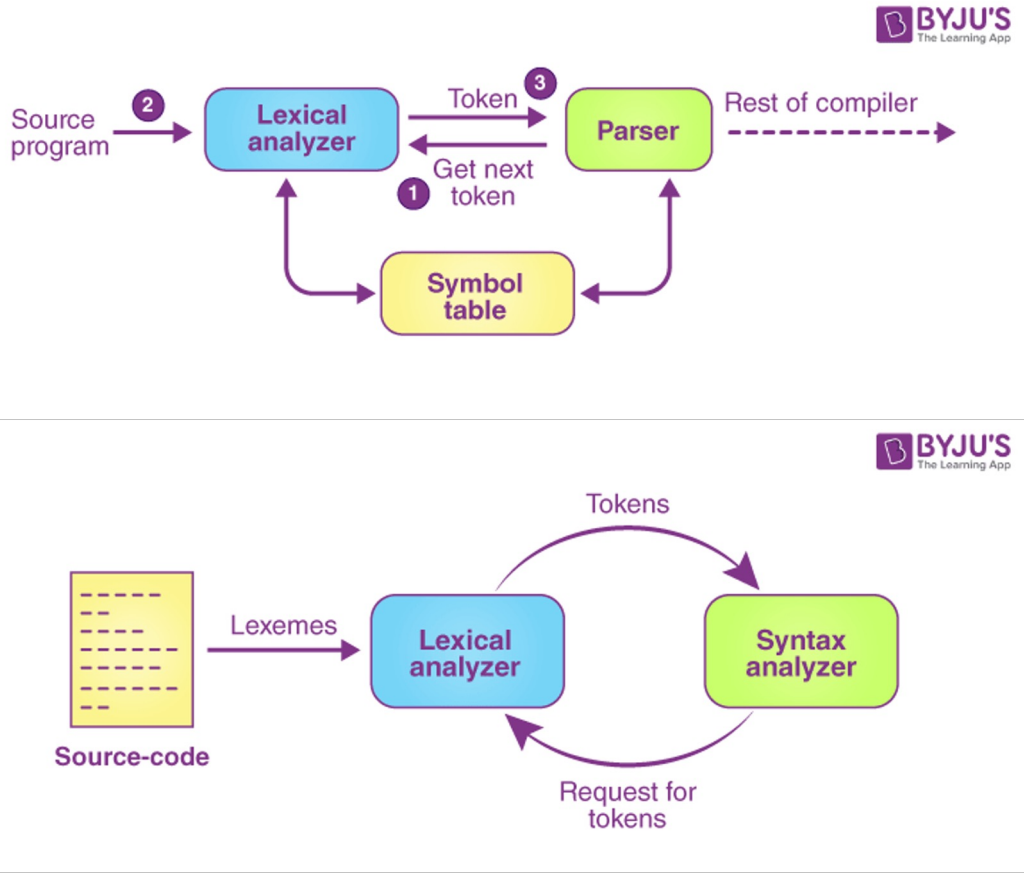


Image credit - https://byjus.com/

How does a compiler work?

- **Semantic Analysis:** The compiler performs semantic analysis to check for correctness beyond syntax. It verifies that variables are declared before use, enforces type checking, and ensures other language-specific rules are followed. This phase also resolves ambiguities and generates symbol tables for variable and function management.

- **Intermediate Code Generation:** At this stage, the compiler may generate intermediate code, which is an abstraction that is closer to machine code but not tied to a specific hardware architecture. Intermediate code simplifies later optimization and target code generation phases.

- **Optimization:** The compiler performs various optimization techniques to improve the efficiency of the generated code. Common optimizations include constant folding, common subexpression elimination, and dead code elimination. These optimizations aim to reduce execution time and memory usage while maintaining the program's functionality.

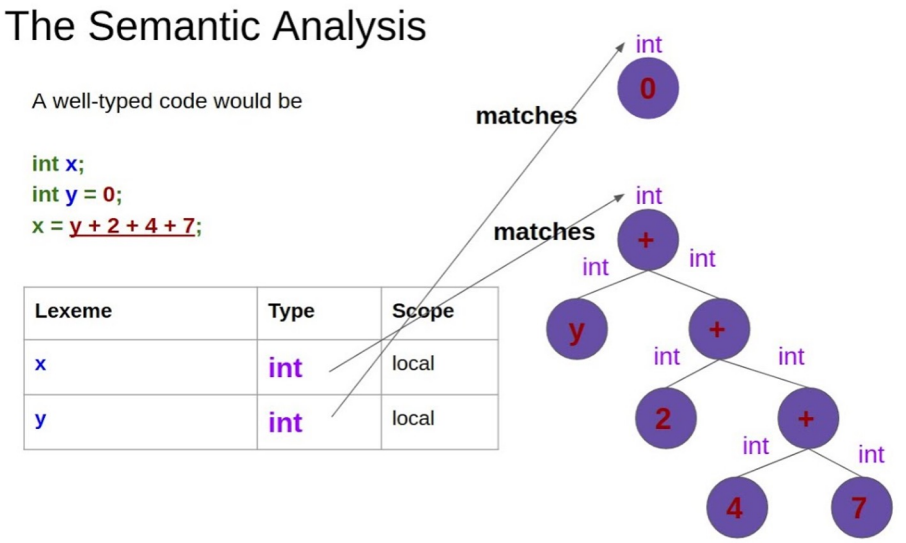


Image credit - The Semantic Analysis! (Demystifying Compilers, lesson 4) @ YouTube.com

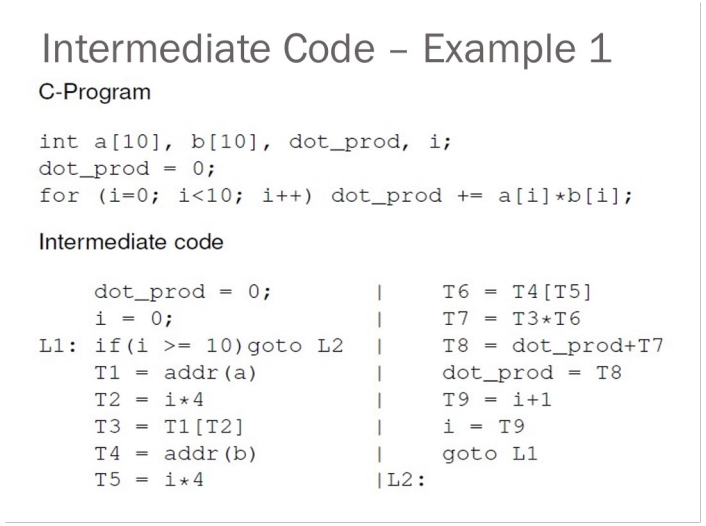


Image credit - Intermediate Code Generation, Dr. ShaukatAli, University of Peshawar

How does a compiler work?

- **Code Generation:** The compiler translates the optimized intermediate code into target machine code. This involves selecting appropriate machine instructions and generating assembly code or binary code. The generated code is tailored to the target hardware architecture, addressing issues like memory management and register allocation.

- **Linking (for multi-source programs):** In the case of multi-source programs, where code is divided into multiple source files, the compiler may need to link these files together. This involves resolving references to functions and variables across different source files. Linking may also involve the inclusion of libraries and external dependencies.

- **Output:** The final output of the compiler is an executable program or an object file, depending on the language and the compilation process. In some cases, an interpreter may be used to execute the program immediately, but traditional compilers produce standalone executable files that can be run independently.

- **Error Handling:** Throughout the compilation process, the compiler detects and reports errors, such as syntax errors or type mismatches. It provides detailed error messages to help programmers correct their code.

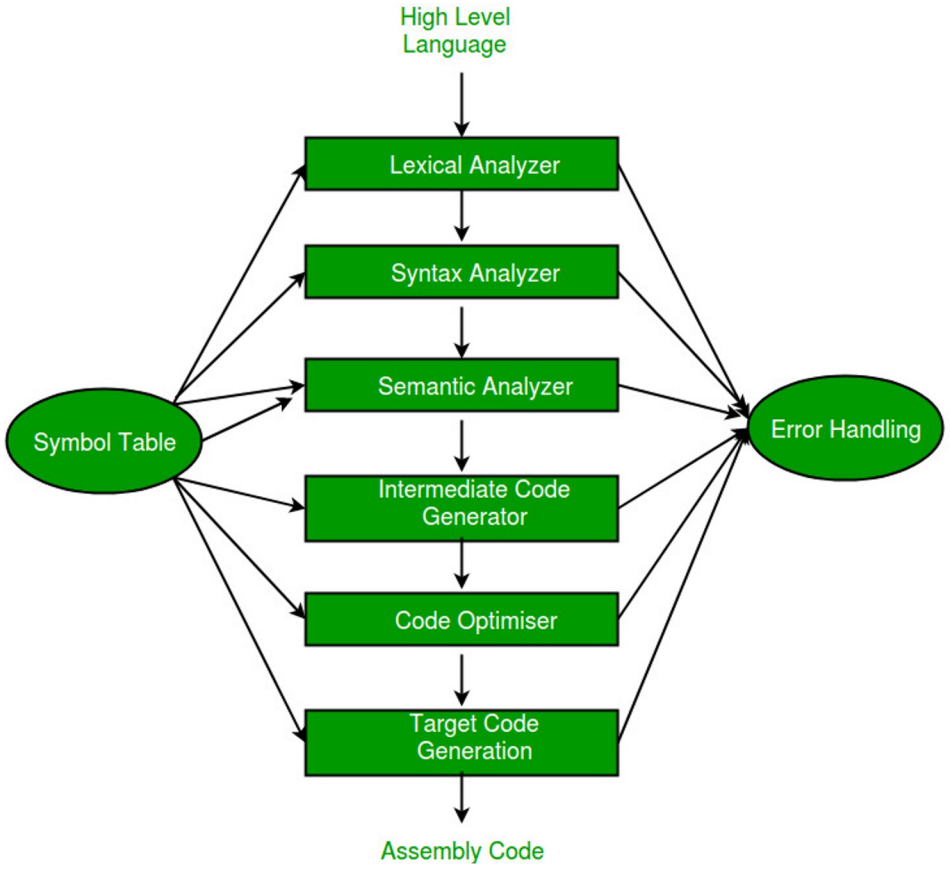


Image credit - https://www.geeksforgeeks.org/

Context Free Grammars (CFGs)

A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

Context-free grammars are studied in fields of theoretical computer science, compiler design, and linguistics. CFG's are used to describe programming languages and parser programs in compilers can be generated automatically from context-free grammars.

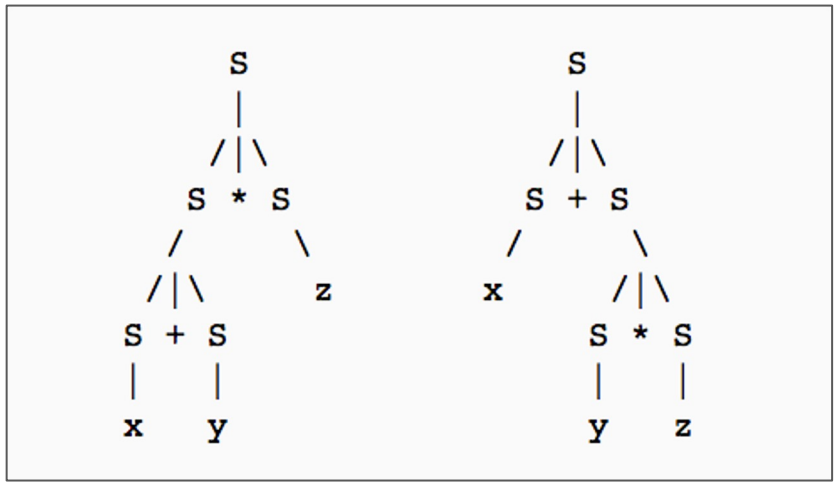


Image credit - https://brilliant.org

Smaller and inside ovals present the associated language classes that are subsets of the languages associated with the container ovals.

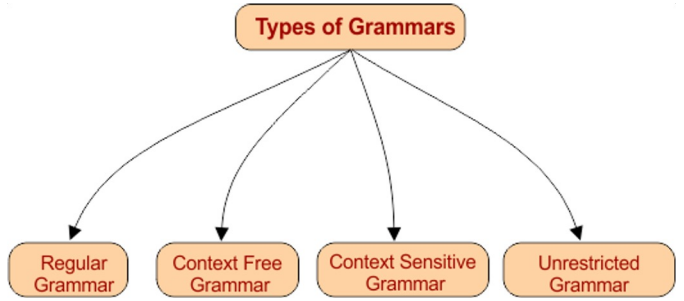


Image credit - https://cstaleem.com/

Formal Language Classes

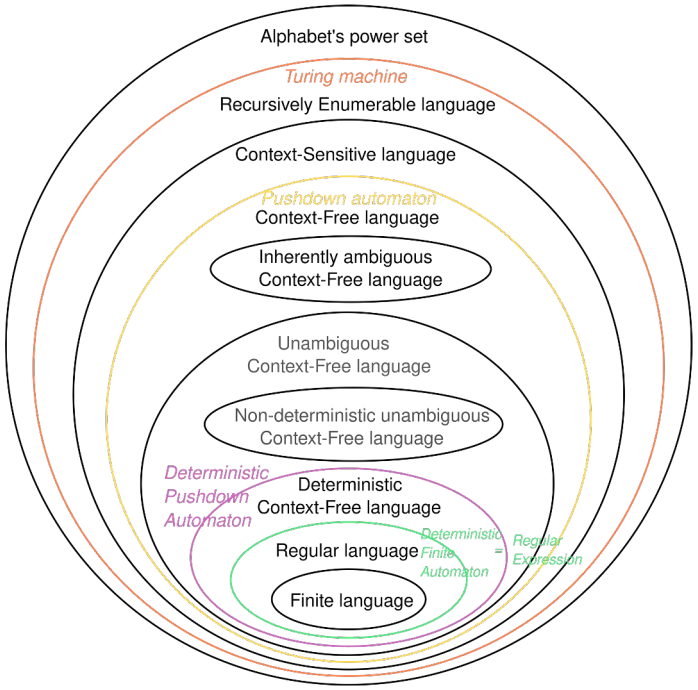


Image credit - https://transang.me/

Components of Context Free Grammars

- A set of **terminal** symbols which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.
- A set of **nonterminal** symbols (or variables) which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.
- A set of **production rules** which are the rules for replacing nonterminal symbols.
  Production rules have the following form:
  variable → string of variables and terminals.
- A **start symbol** which is a special nonterminal symbol that appears in the initial string generated by the grammar.

- **Non-terminal symbols**: denoted by uppercase letters.

  Example: Q_1, Q_2, A, P, S denote non-terminal symbols
- **Terminal symbols**: denoted by lowercase letters.

  Example: a, b, c denote terminal symbols

Terminal versus non-terminal symbols

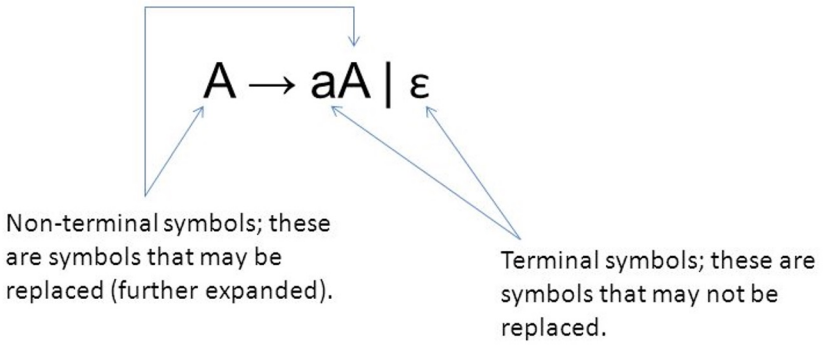


Image credit - Clyde Robbins

Constructing a Parse Tree

A parse tree is an entity which represents the structure of the derivation of a terminal string from some non-terminal (not necessarily the start symbol).

- Parse tree is the hierarchical representation of terminals or non-terminals.
- These symbols (terminals or non-terminals) represent the derivation of the grammar to yield input strings.
- In parsing, the string springs using the beginning symbol.
- The starting symbol of the grammar must be used as the root of the Parse Tree.
- Leaves of parse tree represent terminals.
- Each interior node represents productions of a grammar.

Rules to construct a Parse Tree

- All leaf nodes need to be terminals.
- All interior nodes need to be non-terminals.
- In-order traversal gives the original input string.

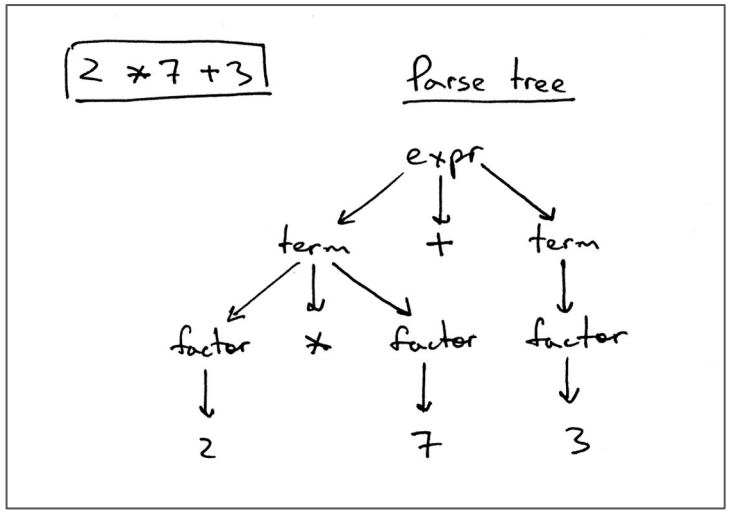


Image credit - https://ruslanspivak.com

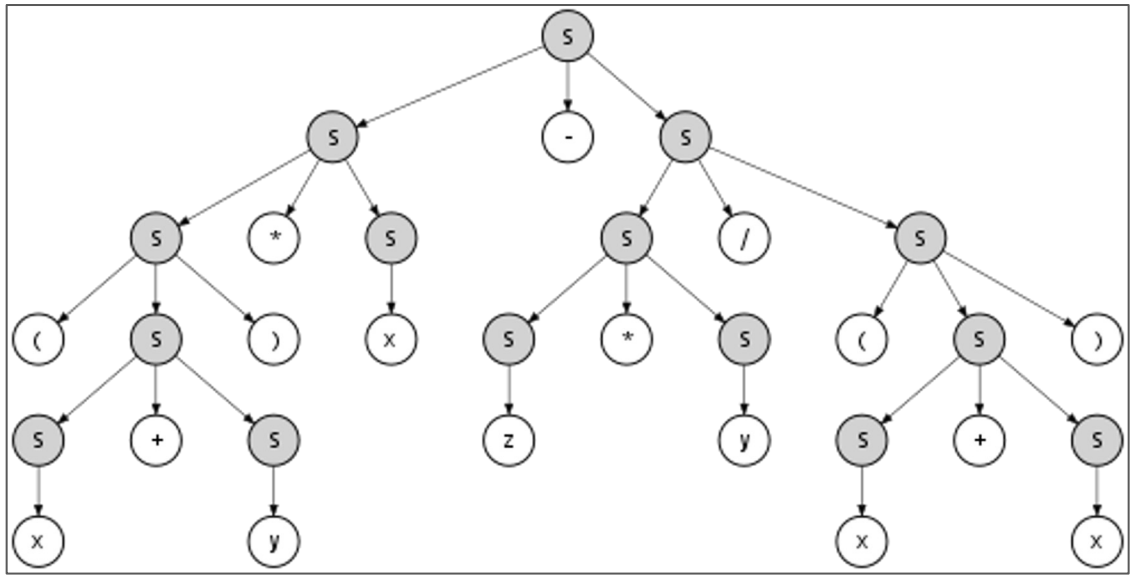


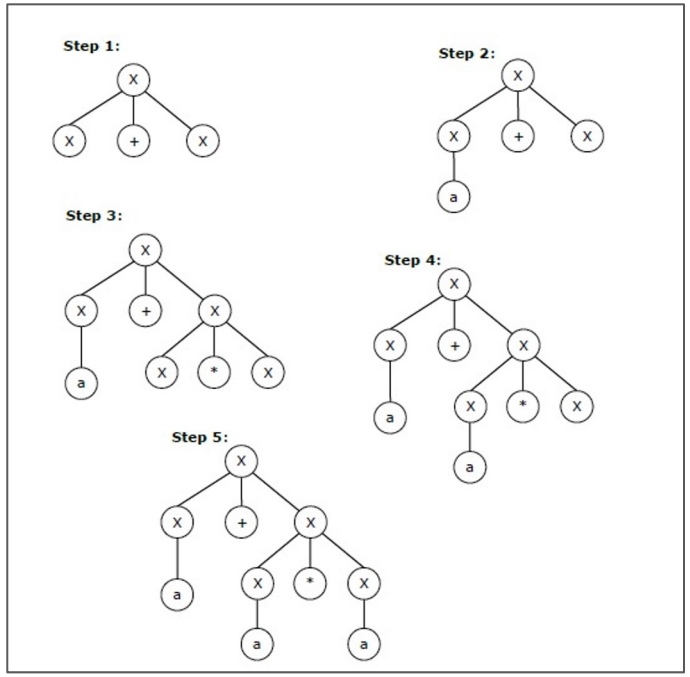
Image credit - https://en.wikipedia.org/

Leftmost and Rightmost Derivation

- **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Let any set of production rules in a CFG be
$X \rightarrow X+X \mid X*X \mid X \mid a$
over an alphabet {a}. Target string **"a+a*a"**

Leftmost Derivation



Rightmost Derivation

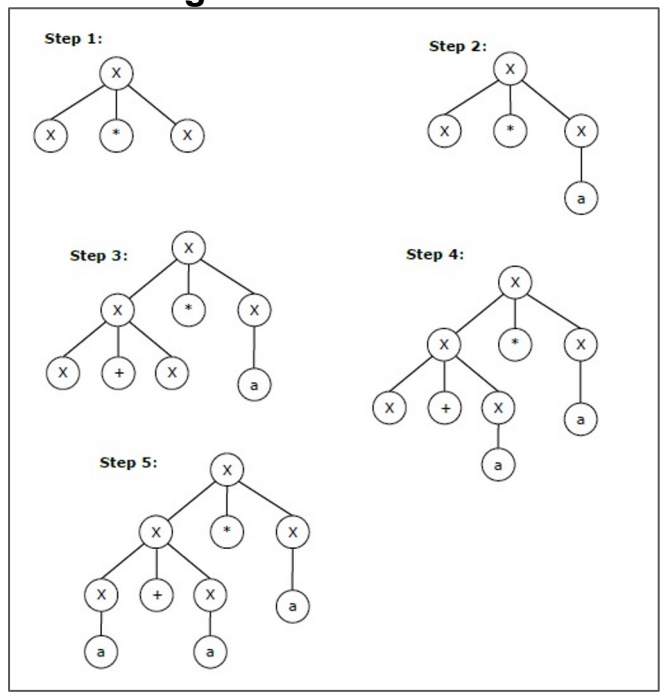


Image credit - https://www.tutorialspoint.com/

Ambiguity in Grammar

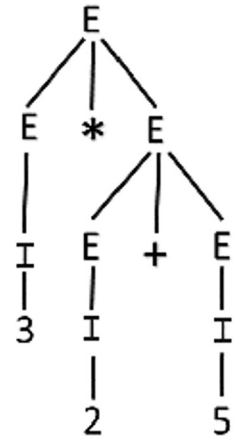
A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

Let us consider a grammar G with the production rule:

E → I
E → E + E
E → E * E
E → (E)
I → ε | 0 | 1 | 2 | ... | 9

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:



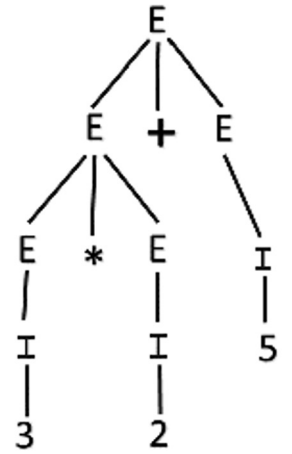


Image credit - https://www.tutorialspoint.com/

Since there are two parse trees for a single string **"3 * 2 + 5"**, the grammar G is ambiguous.