Unit- V

Parsing

In Natural Language Processing (NLP), parsing is the process of analyzing a sentence to determine its grammatical structure and the relationships between its words and phrases. It involves breaking down text into smaller components and representing them in a structured format, such as a parse tree, to identify elements like nouns, verbs, and their dependencies. This syntactic analysis is crucial for machines to understand human language and is applied in tasks such as machine translation, information extraction, and question answering.

How Parsing Works

Tokenization: The input text is first divided into individual words or tokens.

Grammar Rules: A parser uses a set of formal grammar rules to analyze the sequence of tokens and determine if the sentence is syntactically valid.

Structure Generation: The analysis results in a structured representation of the sentence, often a parse tree or dependency tree, which shows the hierarchical and syntactic relationships between words and phrases.

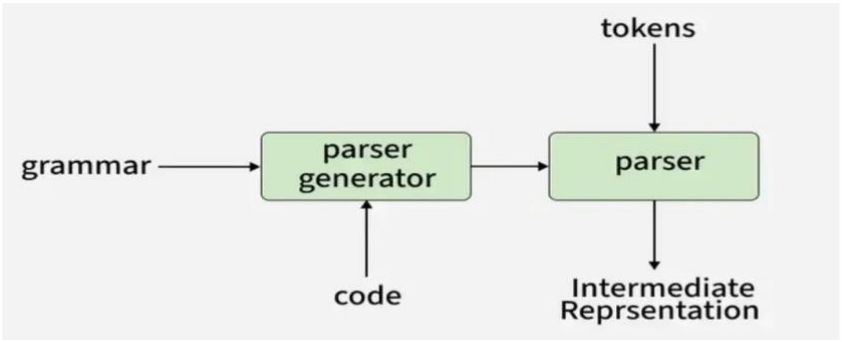
Types of Parsing in NLP

Syntactic Parsing: This broad category aims to uncover the grammatical structure of a sentence, focusing on how words form phrases and how these phrases relate to each other.

Dependency Parsing: A form of syntactic parsing that focuses on the relationships between individual words in a sentence, identifying which words depend on others (e.g., subject-verb, verb-object relationships) to form a dependency tree.

Constituency Parsing: This method breaks down a sentence into its constituent phrases, such as noun phrases and verb phrases, representing them in a phrase structure tree.

The parse tree visually represents how the tokens fit together according to the rules of the language's syntax. This tree structure is crucial for understanding the program's structure and helps in the next stages of processing, such as code generation or execution. Additionally, parsing ensures that the sequence of tokens follows the syntactic rules of the programming language, making the program valid and ready for further analysis or execution.



What is the Role of Parser?

A parser performs syntactic and semantic analysis of source code, converting it into an intermediate representation while detecting and handling errors.

Context-free syntax analysis: The parser checks if the structure of the code follows the basic rules of the programming language (like grammar rules). It looks at how words and symbols are arranged.

Guides context-sensitive analysis: It helps with deeper checks that depend on the meaning of the code, like making sure variables are used correctly. For example, it ensures that a variable used in a mathematical operation, like x + 2, is a number and not text.

Constructs an intermediate representation: The parser creates a simpler version of your code that's easier for the computer to understand and work with.

Produces meaningful error messages: If there's something wrong in your code, the parser tries to explain the problem clearly so you can fix it.

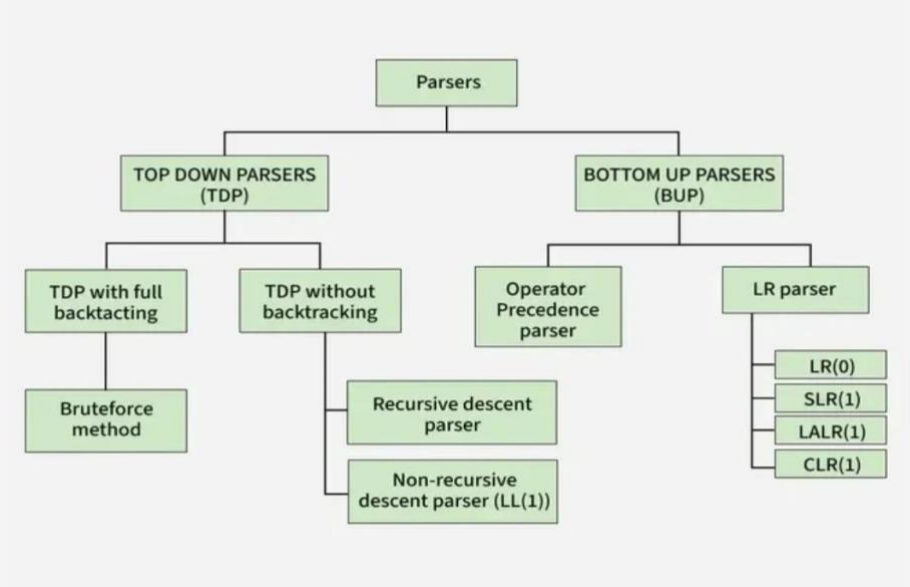
Attempts error correction: Sometimes, the parser tries to fix small mistakes in your code so it can keep working without breaking completely.

Techniques of Parsing

The parsing is divided into two types, which are as follows:

Top-down Parsing

Bottom-up Parsing



Top-Down Parsing

Top-down parsing is a method of building a parse tree from the start symbol (root) down to the leaves (end symbols). The parser begins with the highest-level rule and works its way down, trying to match the input string step by step.

Process: The parser starts with the start symbol and looks for rules that can help it rewrite this symbol. It keeps breaking down the symbols (non-terminals) into smaller parts until it matches the input string.

Leftmost Derivation: In top-down parsing, the parser always chooses the leftmost non-terminal to expand first, following what is called leftmost derivation. This means the parser works on the left side of the string before moving to the right.

Other Names: Top-down parsing is sometimes called recursive parsing or predictive parsing. It is called recursive because it often uses recursive functions to process the symbols.

Top-down parsing is useful for simple languages and is often easier to implement. However, it can have trouble with more complex or ambiguous grammars.

Top-down parsers can be classified into two types based on whether they use backtracking or not:

1. Top-down Parsing with Backtracking

In this approach, the parser tries different possibilities when it encounters a choice, If one possibility doesn't work (i.e., it doesn't match the input string), the parser backtracks to the previous decision point and tries another possibility.

Example: If the parser chooses a rule to expand a non-terminal, and it doesn't work, it will go back, undo the choice, and try a different rule.

Advantage: It can handle grammars where there are multiple possible ways to expand a non-terminal.

Disadvantage: Backtracking can be slow and inefficient because the parser might have to try many possibilities before finding the correct one.

2. Top-down Parsing without Backtracking

In this approach, the parser does not backtrack. It tries to find a match with the input using only the first choice it makes, If it doesn't match the input, it fails immediately instead of going back to try another option.

Example: The parser will always stick with its first decision and will not reconsider other rules once it starts parsing.

Advantage: It is faster because it doesn't waste time going back to previous steps.

Disadvantage: It can only handle simpler grammars that don't require trying multiple choices.

Bottom-Up Parsing

Bottom-up parsing is a method of building a parse tree starting from the leaf nodes (the input symbols) and working towards the root node (the start symbol). The goal is to reduce the input string step by step until we reach the start symbol, which represents the entire language.

Process: The parser begins with the input symbols and looks for patterns that can be reduced to non-terminals based on the grammar rules. It keeps reducing parts of the string until it forms the start symbol.

Rightmost Derivation in Reverse: In bottom-up parsing, the parser traces the rightmost derivation of the string but works backwards, starting from the input string and moving towards the start symbol.

Shift-Reduce Parsing: Bottom-up parsers are often called shift-reduce parsers because they shift (move symbols) and reduce (apply rules to replace symbols) to build the parse tree.

Bottom-up parsing is efficient for handling more complex grammars and is commonly used in compilers. However, it can be more challenging to implement compared to top-down parsing.

Generally, bottom-up parsing is categorized into the following types:

1. LR parsing/Shift Reduce Parsing: Shift reduce Parsing is a process of parsing a string to obtain the start symbol of the grammar.

   LR(0)

   SLR(1)

   LALR

   CLR

2. Operator Precedence Parsing: The grammar defined using operator grammar is known as operator precedence parsing. In operator precedence parsing there should be no null production and two non-terminals should not be adjacent to each other.

Difference Between Bottom-Up and Top-Down Parser

Feature	Top-down Parsing	Bottom-up Parsing
Direction	Builds tree from root to leaves.	Builds tree from leaves to root.
Derivation	Uses leftmost derivation	Uses rightmost derivation in reverse
Efficiency	Can be slower, especially with backtracking.	More efficient for complex grammars.
Example Parsers	Recursive descent, LL parser.	Shift-reduce, LR parser.