## Syntax Analysis (parsing)

Syntax analysis or parsing is the second phase of a compiler. In this phase, we shall learn the basic concepts used in the construction of a parser.
We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses **context-free grammar (CFG)**, which is recognized by push-down automata.
CFG, on the other hand, is a superset of Regular Grammar, as depicted below:
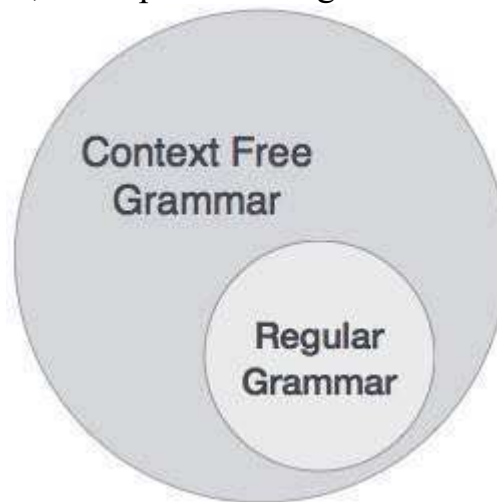
It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar
In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.
A context-free grammar has four components:
• **A set of non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
• **A set of tokens, known as terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
• **A set of productions (P)**. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the

production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.

• **One of the non-terminals is designated as the start symbol** (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, $L = \{ w \mid w = w^R \}$ is not a regular language. But it can be described by means of CFG, as illustrated below:

$G = (V, \Sigma, P, S)$

Where:

$V = \{Q, Z, N\}$
$\Sigma = \{0, 1\}$
$P = \{ Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \varepsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1\}$
$S = \{ Q \}$

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code *token stream* against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.
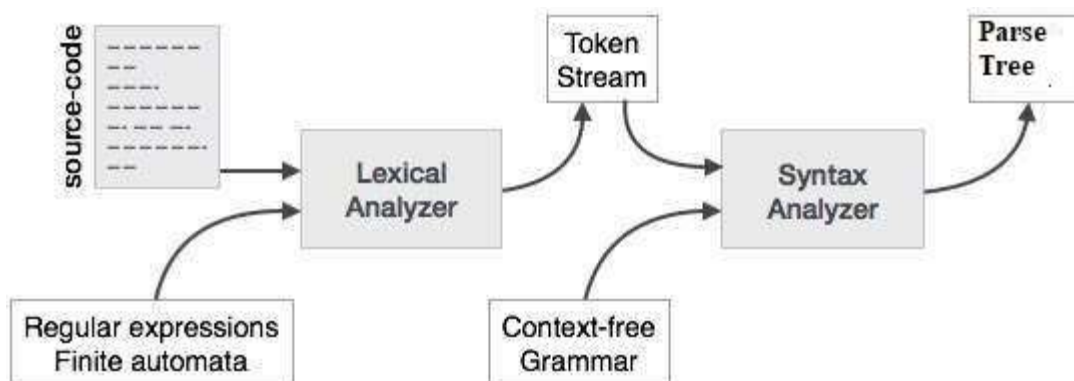
This way, the parser accomplishes **two tasks**, i.e., **parsing the code**, **looking for errors** and **generating a parse tree as the output of the phase**.
Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later.

Derivation

A derivation is **basically a sequence of production rules**, in order to get the input string. During parsing, we take **two decisions** for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from **left to right**, it is called **left-most derivation**. The sentential form derived by the left-most derivation is called the **left-sentential form**.

Right-most Derivation

If we scan and replace the input with production rules, from **right to left**, it is known as **right-most derivation**. The sentential form derived from the right-most derivation is called the **right-sentential form**.

Example
Production rules:

E → E + E
E → E * E
E → id
Input string: id + id * id
The left-most derivation is:
E → **E** * E
E → **E + E** * E
E → **id** + E * E
E → id + **id** * E
E → **id + id * id**

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:
E → E + E
E → E + **E * E**
E → E + E * **id**
E → E + **id** * id
E → **id** + id * id

Parse Tree
A parse tree is a **graphical depiction of a derivation**. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.
a, b and c are id
We take the **left-most derivation** of   **a + b * c**
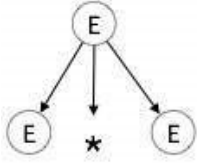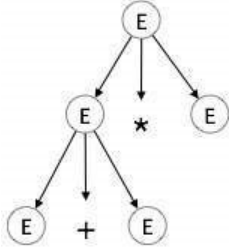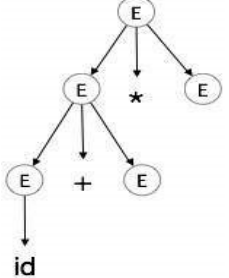**The left-most derivation is:**
E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → a + b * c
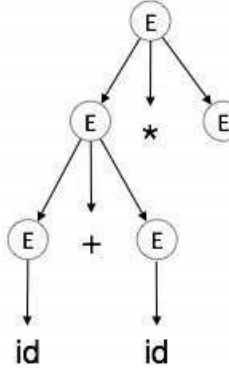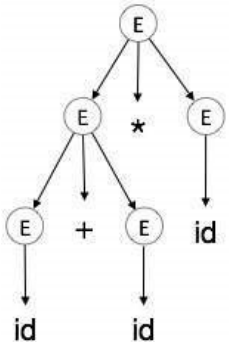
Step 1:	E → E * E	
Step 2:	E → E + E * E	

Step 3:	E → id + E * E	
Step 4:	E → id + id * E	
Step 5:	E → id + id * id	

In a parse tree:
- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- **In-order traversal gives original input string**.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Ambiguity

A grammar G is said to be ambiguous **if it has more than one parse tree (left or right derivation)** for at **least one string**.
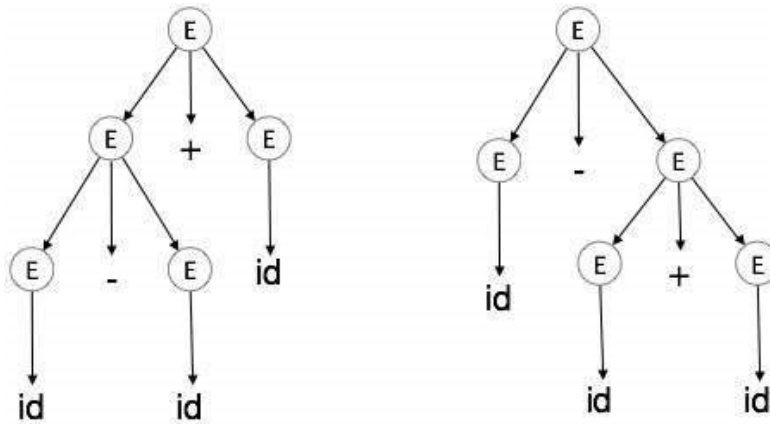
Example
E → E + E
E → E − E
E → id
For the string **id - id + id**, the above grammar generates two parse trees:

The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following **associativity and precedence** constraints.

Associativity
If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example
Operations such as **Addition, Multiplication, Subtraction, and Division are left associative**. If the expression contains:
id op id op id
it will be evaluated as:
(id op id) op id
For example, (id + id) + id
Operations like **Exponentiation are right associative**, i.e., the order of evaluation in the same expression will be:
id op (id op id)

For example, id ^ (id ^ id)

Precedence

If **two different operators share a common operand**, the precedence of operators decides which will take the operand. That is, **2+3*4** can have **two different parse trees**, one corresponding to **(2+3)*4** and another corresponding to **2+(3*4)**. By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically *** (multiplication) has precedence over + (addition)**, so the expression 2+3*4 will always be interpreted as:

2 + (3 * 4)

These methods decrease the chances of ambiguity in a language or its grammar.

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.
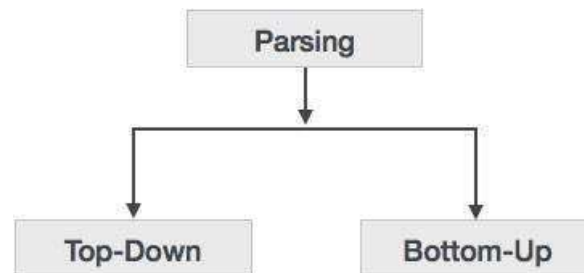
Top-down Parsing

When the parser starts constructing the parse tree from the **start symbol** and then tries to transform the start symbol to the input, it is called top-down parsing.

- **Recursive descent parsing**: It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- **Backtracking**: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, **bottom-up parsing starts with the input symbols** and tries to construct the parse tree up to the **start symbol**.

Example:

Input string : **a + b * c**

Production rules:

S → E

E → E + T

E → E * T

E → T

T → **id**

Let us start bottom-up parsing

a + b * c

Read the input and check if any production matches with the input:

a + b * c

T + b * c

E + **b** * c

E + T * c

E * c

E * T

E

S

Example: Consider the grammar:-

(1) S → i C t S

(2) S → i C t S e S ------ (1)

(3) S → a

(4) C → b

Here i, t and e stand for "if", "then" and "else" respectively and C for "conditional", S for "statement".
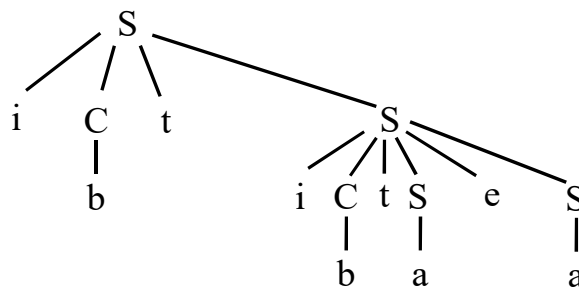
Parse Tree T

The **left-most derivation** corresponding to this parse tree is given by:-

$$S \underset{lm}{\Longrightarrow} i\,C\,t\,S$$

$$\underset{lm}{\Longrightarrow} i\,b\,t\,\mathbf{S}$$

$$\underset{lm}{\Longrightarrow} i\,b\,t\,\mathbf{i\,C\,t\,S\,e\,S}$$

$$\underset{lm}{\Longrightarrow} i\,b\,t\,i\,\mathbf{b}\,t\,S\,e\,S$$

$$\underset{lm}{\Longrightarrow} i\,b\,t\,i\,b\,t\,\mathbf{a}\,e\,S$$

$$\underset{lm}{\Longrightarrow} i\,b\,t\,i\,b\,t\,a\,e\,\mathbf{a}$$

A right most derivation can be constructed from a parse tree analogously. At each step we replace the right-most non-terminal by the tables of its children. For example, the first two steps of a right most derivation constructed from Figure would be

$$S \Longrightarrow i\,C\,t\,S \Longrightarrow i\,C\,t\,i\,C\,t\,S\,e\,S$$

Examples

E → E + E
E → E * E
E → (E)
E → a | b | c | ...

The string is (a + b) * c

Here is a sequence of replacements that leads to a sequence of terminal symbols.

E → E * E
  → (E) * E
  → (E + E) * E
  → (a + E) * E
  → (a + b) * E
  → (a + b) * c

Parse Tree: (a + b) * c

```
                                  E
                                 /|\
                                / | \
            E                  / | \
           /|\                / | \
          E * E              / | \
         /|\ \             E  *   E
        ( E ) c           /|\ |   |
        / | \            /|\ |   |
       E + E            / E \ |   |
       |  |           / /|\ \ |   |
       a  b          | / | \ |   |
                     | E + E |   |
                     | || |      |
                     ( a + b )*   c
```
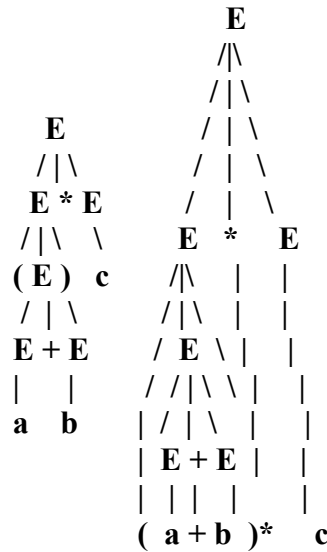
Consider the context-free grammar
S→ AB | CD
A→0A1 | 01
B →2B | 2
C →0C | 0
D→1D2 | 12
and the string **012**. Is the grammar ambiguous or unambiguous? Justify your answer.

S ⇒**AB**⇒**01**B⇒**012**

S⇒CD⇒0D⇒012
the grammar is ambiguous, two derivation from one side

$E \rightarrow I \mid C \mid E +E \mid E *E$
$I \rightarrow y \mid z$
$C \rightarrow 4$

Where **E** is **start symbol**, and the string **y + 4 * z**, is the grammar ambiguous or unambiguous? Justify your answer.