MCA Part III

Paper- XIX

Topic: Syntax Analysis

Prepared by: Dr. Kiran Pandey

School of Computer science

Email-Id: kiranpandey.nou@gmail.com

Introduction

We have already discussed lexical analysis. In this unit we will discuss syntax analysis. Syntax analysis or parsing is the second phase of a compiler. We will also discuss different types of grammars and their applications. We will mainly focus on Context Free Grammar (CFG) and learn how to draw a parse tree, derivation trees and the problems with CFG like left recursion, left factoring, ambiguous grammar and how to simplify grammar.

Formal grammar and their application to Syntax Analysis

The structure of a sentence is defined with grammars in any language. Grammar is a set of rules and examples dealing with the syntax and word structures of a language, usually intended as an aid to the learning of that language.

A grammar G is defined as a set of 4-tuple (**V, ∑ , R , S**),

Where, V is set of non-terminal symbols (variables)

$\sum$ is a set of terminal symbols.

R is set of production rules.

S is a start symbol with which the strings in the grammar are derived.

Language defined by a grammar G is denoted by L(G). Here L(G) represents a set of strings w derived from G. Start symbol S in one or more steps derives the strings or sentences of grammar that is represented by:

$S \rightarrow w$.

The sentential form of grammar is a combination of terminals and non-represented by **$S \rightarrow {}^{+}\alpha$.** Two grammars G1 and G2 are equivalent if **L(G1)=L(G2)**.

Some of the notations used to represent the grammar are:

1. **For terminal symbols:**

- Lower case letters of alphabet like **a, b, c...z.**
- Operator symbols like **+, -,** etc.
- Punctuation symbols like (, ! ,),..etc.
- Digits like 0, 1,…9, etc.

2. **For non-terminal symbols:**

- Upper case letters of alphabet like A**, B, C..., Z.**
- Start symbol **S**.
- Lower case strings like **expr, stmt, etc.**

Example:

(i)	S→ a	{a}
(ii)	S→ aS →aa	{aa}
(iii)	S→ aS →aaS	{aaa}

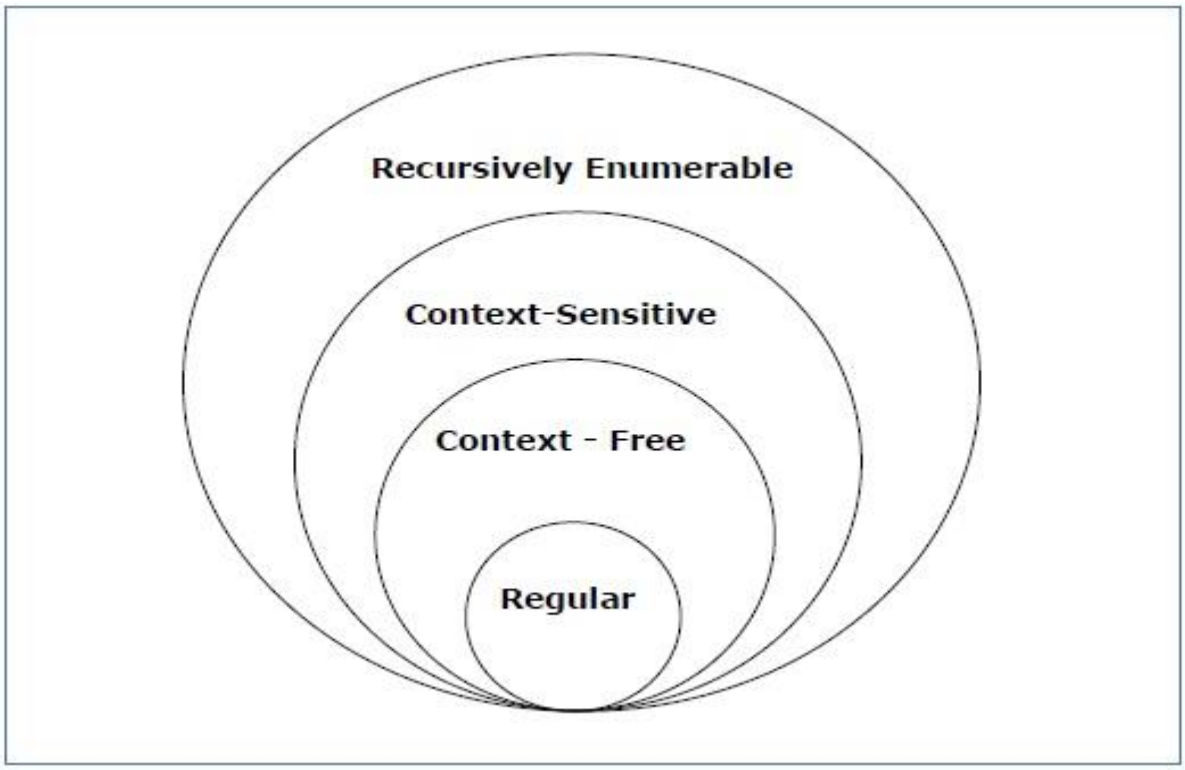
Chomsky Classification of Grammar

Noam Chomsky defined a hierarchy of grammar in terms of complexity. According to him there are four types of grammars – Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other –

Table 1: Chomsky classification of grammar.

Grammar type	Grammar Name	Language Accepted	Machine defined
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite automaton

The diagram below shows the scope of each type of grammar –



Type – 3 Grammar generate regular languages. This grammars has a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or

single terminal followed by a single non-terminal. The productions must be in the form **X → a or X → aY**

Where, **X, Y ∈ N** (Non terminal) and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

X→ ε

X→ a| aA

A→ b

Type-2 grammars generate context-free languages. The productions must be in the form **A → γ**

Where, A **∈ N** (Non terminal) and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals). These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example:

S→ Aa

A→ a

A→ aA

A→ abc

A→ ε

Type-1 grammars generate context-sensitive languages. The productions must be in the form

αAβ → αγβ where **A ∈ N** (Non-terminal) and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals). The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example:

AB → AbBc

A→ bcA

B→ b

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars. They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α → β** where **α** is a string of terminals and nonterminal with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Context Free Grammar

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 is a superset of Regular Grammar, as shown below in the diagram:

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.

A context-free grammar is defined by 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 rules** (R). 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 | w = w^R } is not a regular language. But it can be described by means of CFG, as illustrated below:

Given G = (V, ∑, R, S)

Where,

V = {Q, Z, N}

∑ = {0, 1}

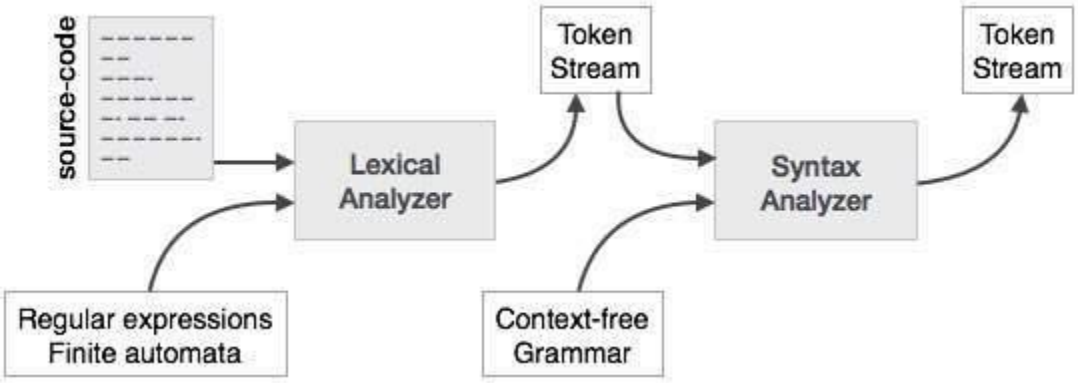
P = {Q → Z | Q → N | Q →ε | Z → 0Q0 | N → 1Q1}

S = {Q}

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

PARSER

A parser or syntax analyzer 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 in this chapter.

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.

(i) 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.

(ii) 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:

$E \rightarrow E + E$

$E \rightarrow E * E$

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

(iii) 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. We take the left-most derivation of a + b * c

The left-most derivation is:

E → E * E

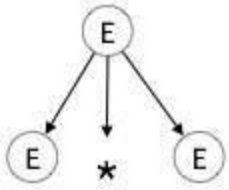
E → E + E * E

E → id + E * E

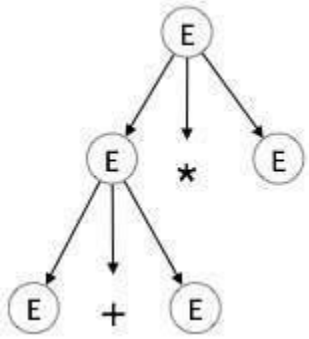
E → id + id * E

E → id + id * id

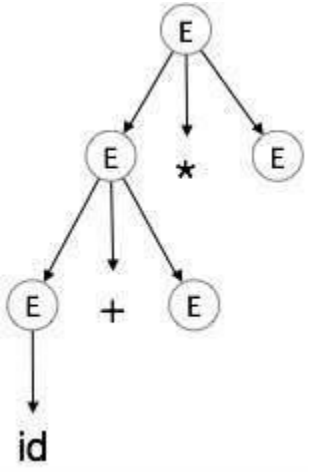
Step 1:

E → E * E	

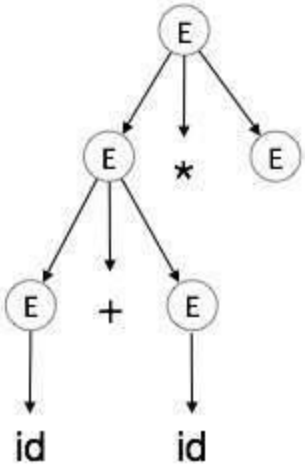
Step 2:

E → E + E * E	

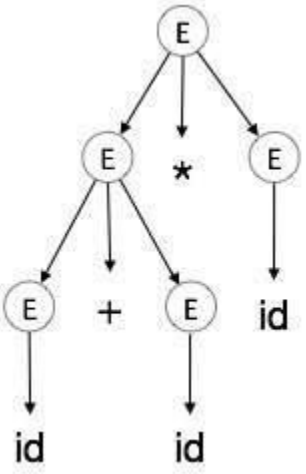
Step 3:

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.

(iv) 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.

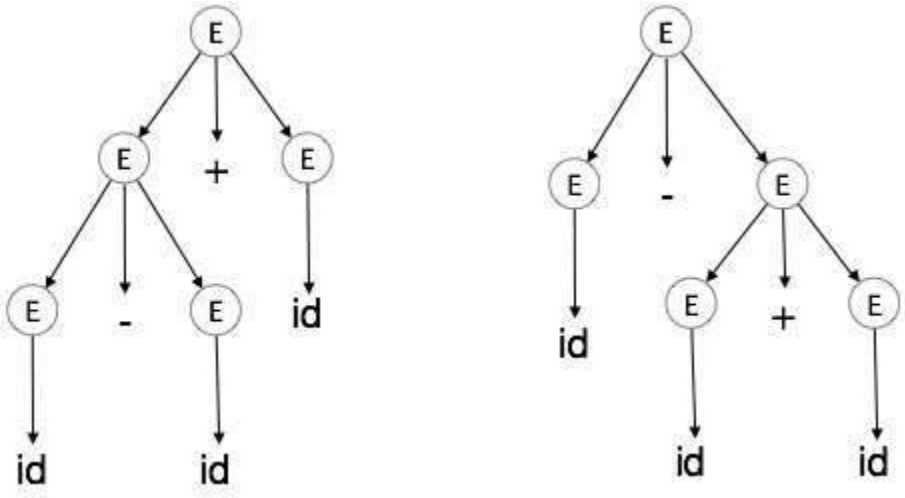
Example:

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.

SYNTAX ANALYSER AND CAPABILITIES OF CFG

Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- it cannot determine if an operation performed on a token type is valid or not.

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficient parser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

QUESTIONS

1. Find the number of parse tree for an input string “aaa” in S → Sa | aS | a.
2. Explain Chomsky classification of grammar with examples.
3. What is CFG? Give an example of CFG.
4. What is a parse tree?
5. Explain the following concepts:
   (a) Left recursive
   (b) Right recursive
   (c) Ambiguity
6. Left factor the given grammar:
   S→ aSSbS|aSaSb|abb|b