# Experiment No. 4: Chomsky Classification of Grammar

## Objective:

To study and classify grammars based on the Chomsky hierarchy, which divides grammars into four types: Type 0 (Recursively Enumerable), Type 1 (Context-Sensitive), Type 2 (Context-Free), and Type 3 (Regular Grammars).

## Theory: Chomsky's classification system organizes grammars into four types based on the complexity of the rules used to generate languages. The hierarchy is critical for understanding computational linguistics and automata theory. 1. Type 0 - Recursively Enumerable Grammars: - These grammars generate the most general class of languages known as recursively enumerable languages. - There are no restrictions on the production rules, allowing the form α → β, where α and β are any strings of terminal and non-terminal symbols. - A Turing Machine can recognize languages generated by Type 0 grammars, meaning any computation that can be performed algorithmically can be represented by these grammars.

**Example**:   
 Consider the grammar:   
 - S → AB  
 - A → aA | a  
 - B → bB | b  
 - This generates the language L = { a^n b^n | n ≥ 1 }, where the number of 'a's equals the number of 'b's.

**2. Type 1 - Context-Sensitive Grammars:**  
 - Context-sensitive grammars generate languages that require the context of surrounding symbols to determine the production rule.  
 - Production rules are of the form αAβ → αγβ, where γ ≠ ε (i.e., the right-hand side of the production is at least as long as the left-hand side).  
 - Context-sensitive grammars are more restrictive than Type 0 but still powerful enough to generate many natural languages.  
 - The languages generated by these grammars are recognized by linear-bounded automata, a more limited computational model than the Turing Machine.

Example:   
 Consider the grammar:  
 - S → aSbc | abc  
 - This generates the language L = { a^n b^n c^n | n ≥ 1 }, where the number of 'a's, 'b's, and 'c's must be the same.

**3. Type 2 - Context-Free Grammars:**  
 - These grammars generate context-free languages, where each production rule is of the form A → γ, with A being a single non-terminal and γ being a string of terminals and/or non-terminals.  
 - Context-free grammars are widely used in programming languages and natural language processing, as they can describe a large class of syntactic constructs.  
 - Pushdown automata, which use a stack to keep track of intermediate computations, can recognize context-free languages.

Example:  
 - Consider the grammar:  
 - S → aSb | ε  
 - This generates the language L = { a^n b^n | n ≥ 0 }, where the number of 'a's is equal to the number of 'b's.

**4. Type 3 - Regular Grammars:**  
 - Regular grammars generate the simplest class of languages, known as regular languages.  
 - The production rules are of the form A → aB or A → a, where A, B are non-terminals and a is a terminal symbol.  
 - Regular languages can be recognized by finite automata, which do not require memory or a stack, making them computationally efficient.  
 - Regular expressions, commonly used in search algorithms and text processing, describe regular languages.

Example:  
 - Consider the grammar:  
 - S → aS | a  
 - This generates the language L = { a^n | n ≥ 1 }, which is a simple set of repeated 'a's.

**Applications:**  
- Regular Grammars: Used in lexical analysis, simple pattern matching, and search algorithms (e.g., regex in programming).  
- Context-Free Grammars: Widely used in the design of programming languages and parsers (e.g., parsing expressions in C, Java).  
- Context-Sensitive Grammars: Useful in compilers for checking type consistency, scope rules, and more.  
- Recursively Enumerable Grammars: Used in describing the syntax of the most complex computational systems.

**Conclusion:**The Chomsky classification of grammars provides a foundational framework for understanding the complexity of formal languages and their computational requirements. This hierarchy not only helps in designing parsers for programming languages but also in understanding the limits of machine computation.

# Experiment No. 5: To show all operations of a Stack in C++

## Objective:

To implement and demonstrate all basic operations of a stack, including push, pop, peek, and check if the stack is empty or full, using C++.

## Code:

#include<iostream>  
#define MAX 5   
using namespace std;  
class Stack {  
 int top;  
 int arr[MAX];  
public:  
 Stack() {   
 top = -1;   
 }  
void push(int val) {  
 if (top >= (MAX - 1)) {  
 cout << "Stack Overflow" << endl;  
 } else {  
 arr[++top] = val;  
 cout << val << " pushed into stack" << endl;  
 }  
 }  
 void pop() {  
 if (top < 0) {  
 cout << "Stack Underflow" << endl;  
 } else {  
 int val = arr[top--];  
 cout << val << " popped from stack" << endl;  
 }  
 }  
 int peek() {  
 if (top < 0) {  
 cout << "Stack is Empty" << endl;  
 return -1;  
 } else return arr[top];  
   
 }  
 bool isEmpty() {  
 return (top < 0);  
 }  
 bool isFull() {  
 return (top >= (MAX - 1));  
 }  
};  
int main() {  
 Stack stack;  
 stack.push(10);  
 stack.push(20);  
 stack.push(30);  
 stack.push(40);  
 stack.push(50);  
 cout << "Top element is: " << stack.peek() << endl;  
 stack.pop();  
 stack.pop();  
 cout << "Top element after popping two elements: " << stack.peek() << endl;  
 return 0;  
}

**Output:**

A screenshot of a computer

Description automatically generated

**Explanation**:

1. Push Operation: Adds an element to the top of the stack.  
2. Pop Operation: Removes the top element from the stack.  
3. Peek Operation: Retrieves the top element without removing it.  
4. isEmpty: Checks if the stack is empty.  
5. isFull: Checks if the stack is full.

## Conclusion:

This experiment demonstrates how to implement and use stack operations in C++ effectively.