CO523 – Programming Languages
Assignment 01: Execution Models & Language Processing

Section A – Short Answer Questions (10 × 4 = 40 marks)

1. Define an execution model in the context of programming languages.
An execution model in the context of programming languages refers to the method or system used to execute programs written in a particular language. It defines how a program's code is translated into a form that can be executed on a machine, such as through direct execution by a CPU or via an intermediary process like interpretation or compilation.

2. Differentiate between compiled, interpreted, and hybrid execution models with one example each.

- **Compiled**: In a compiled execution model, the entire source code is translated into machine code by a compiler before execution. Example: **C**.
- **Interpreted**: In an interpreted execution model, the source code is read and executed line-by-line by an interpreter at runtime. Example: **Python**.
- **Hybrid**: A hybrid execution model uses both compilation and interpretation. The source code is first compiled into an intermediate form, which is then interpreted at runtime. Example: **Java** (compiled to bytecode and interpreted by the Java Virtual Machine).

3. What is a Virtual Machine (VM)? State two advantages of using a VM.
A Virtual Machine (VM) is an abstraction layer that enables the execution of programs in an environment independent of the underlying hardware. It simulates a computer system, allowing software to run as if it were on its own machine.
Advantages of using a VM:

- **Portability**: Programs can run on any platform with the same VM, making them platform-independent.
- **Isolation**: VMs provide an isolated environment for programs, which can enhance security and prevent interference between processes.

4. List and briefly explain the components of a runtime environment of a program.
The runtime environment is the environment in which a program executes and typically includes:

- **Memory Management**: Handling memory allocation and deallocation.

- **Execution Engine**: The core component that runs the program, including the CPU's execution cycle.
- **I/O Handling**: Manages input and output operations.
- **Runtime Libraries**: Standard functions or routines available to the program during execution.

5. Define lexical analysis. What are the input and output of a lexical analyzer?
Lexical analysis is the process of breaking down a program's source code into tokens, which are meaningful units such as keywords, identifiers, operators, and literals.

- **Input**: The raw source code.
- **Output**: A sequence of tokens (e.g., keywords, operators, identifiers, constants).

6. What is a token? List any four types of tokens with examples.
A token is a basic unit of meaningful data generated by a lexical analyzer from the source code.

- **Keyword**: if, for, while.
- **Identifier**: variableName, functionName.
- **Operator**: +, -, *.
- **Literal**: 10, "Hello World".

7. Explain the difference between a lexeme and a token.
A **lexeme** is the actual sequence of characters in the source code that represents a token, whereas a **token** is the category or classification assigned to the lexeme by the lexical analyzer. For example, in the expression x = 5, x is the lexeme, and the token would be the identifier type.

8. What is syntax analysis? What kind of errors are detected during this phase?
Syntax analysis is the phase of compilation where the program's structure is checked to ensure it conforms to the rules (grammar) of the programming language. During this phase, **syntax errors** like mismatched parentheses, incorrect operators, and improper expression formatting are detected.

9. Define a symbol table and mention two types of information stored in it.
A symbol table is a data structure used by a compiler to store information about variables, functions, objects, and other entities in the program.
Two types of information stored are:

- **Variable Name**: The identifier of the variable.
- **Type Information**: The type of the variable, such as integer, float, or string.

10. Explain the analogy between lexical analysis & syntax analysis and natural language processing.
In natural language processing (NLP), lexical analysis is similar to the process of breaking down sentences into words or phrases (tokens). Syntax analysis in NLP is analogous to parsing these words into grammatically correct structures (e.g., sentence trees), which is similar to how syntax analysis checks the program's structure for correctness.

Section B – Descriptive Questions (4 × 10 = 40 marks)

11. Explain the program execution process from source code to execution, clearly describing the role of:

- **Compiler / Interpreter**: A **compiler** translates the entire source code into machine code or bytecode. An **interpreter**, on the other hand, executes the source code directly, line by line.
- **Virtual Machine**: In languages like Java, the compiled code (bytecode) is executed on a **virtual machine** (e.g., JVM), which provides a platform-independent execution environment.
- **Runtime Environment**: During execution, the runtime environment manages memory, performs garbage collection, handles I/O operations, and ensures that the program runs correctly.

12. Describe the lexical analysis phase in detail. Explain its responsibilities and illustrate the process using the following statement:
float average = total / count;
The lexical analysis phase is responsible for dividing the source code into tokens, such as keywords, identifiers, literals, and operators. In the statement above:

- float is a keyword.
- average is an identifier.
- = is an assignment operator.
- total and count are identifiers.
- / is an operator.
- ; is a statement terminator.

13. Explain syntax analysis with the help of a suitable example. Describe how a parse tree is generated and how syntax errors are detected.
Syntax analysis involves checking whether the sequence of tokens generated by lexical analysis follows the grammatical structure of the language. For example, in the expression x = y + z, the

parser would generate a parse tree where the root node represents the assignment, and the child nodes represent the variables and the addition operation. If the expression had a missing operator, the parser would flag a syntax error.

14. Compare static (lexical) scoping and dynamic scoping. Use code examples to justify your answer.

Static Scoping: In static scoping, the scope of a variable is determined at compile time based on the program's structure. Example:

```
int x = 10;
void func() {
    int x = 20;
    printf("%d", x);  // Prints 20
}
```

Dynamic Scoping: In dynamic scoping, the scope of a variable is determined at runtime based on the call stack. Example:

```
x = 10
def func():
    print(x)  # Prints 10, from the dynamic environment
func()
```

15. Explain different variable lifetimes (static, stack-dynamic, heap-dynamic) and relate them to memory regions (stack, heap, static memory).

- **Static Lifetime**: Variables with static lifetime are allocated once and exist throughout the program’s execution (e.g., global variables). They are stored in **static memory**.
- **Stack-Dynamic Lifetime**: Variables with stack-dynamic lifetime are created when a function is called and destroyed when the function exits (e.g., local variables in a function). They are stored in the **stack**.
- **Heap-Dynamic Lifetime**: Variables with heap-dynamic lifetime are allocated and deallocated manually during execution (e.g., objects created using new or malloc). They are stored in the **heap**.
-

16. Discuss memory management techniques in programming languages. Compare manual memory management and automatic garbage collection with examples.

- **Manual Memory Management**: The programmer manually allocates and deallocates memory using functions like malloc() and free() in C. This provides more control but is prone to errors like memory leaks.
- **Automatic Garbage Collection**: The programming language runtime automatically handles memory allocation and deallocation, as seen in languages like Java and Python, where the garbage collector reclaims unused memory.

Section C – Application / Analytical Questions (2 × 10 = 20 marks)

17. Given the following code snippet:

int x = 5;

x = x + 1;

a) **List all tokens generated by the lexical analyzer.**

- int (Keyword)
- x (Identifier)
- = (Operator)
- 5 (Literal)
- + (Operator)
- 1 (Literal)
- ; (Statement terminator)

b) **Identify the token type for each token.**

- int: Keyword
- x: Identifier
- =: Assignment operator
- 5: Integer literal
- +: Arithmetic operator
- 1: Integer literal
- ;: Statement terminator

c) **State whether any lexical error exists. Justify your answer.**
There are no lexical errors in this code snippet. All tokens are valid in C programming.

18. Consider the following incorrect statement:

int = value 20;

a) **Will the lexical analyzer detect an error? Why?**
Yes, the lexical analyzer will detect an error because the variable name int is a reserved keyword, and an assignment is missing.

b) **Will the syntax analyzer detect an error? Explain clearly.**
Yes, the syntax analyzer will detect an error because the statement does not follow the correct syntax for variable declaration and assignment in C.

c) **Identify the phase responsible for reporting this error.**
The lexical analyzer will initially detect issues related to invalid tokenization (e.g., use of the reserved word int). The syntax analyzer will later catch the overall syntax error in the declaration.

19. Explain how a symbol table is used during program execution. Illustrate your answer by showing sample symbol table entries for a simple function.
The symbol table stores information about variables, functions, and other symbols during compilation and execution. For a simple function:

```
int sum(int a, int b) {

  return a + b;

}
```

Sample Symbol Table:

Name	**Type**	**Scope**	**Value**
sum	Function	Global	Address of function
a	Parameter	sum	-
b	Parameter	sum	-

20. Analyze how modern programming languages (e.g., Python, Rust, Go) improve safety and reliability using concepts such as:

- **Memory Safety**:
  - **Rust**: Rust uses ownership and borrowing rules to ensure memory safety without a garbage collector.

- **Type Safety**:
    - **Go**: Go has strong type safety, preventing type mismatches at compile-time.
- **Concurrency Safety**:
    - **Python**: Python uses the Global Interpreter Lock (GIL) to ensure that only one thread executes Python bytecodes at a time, avoiding issues with shared memory in multithreaded programs.