**Q2) Explain two Code functions Lexer and Parser Phases**

**Lexer and Parser Functions:**

**1. Lexer Function:**

**Purpose**

The Lexer (also known as a scanner or tokenizer) is responsible for breaking the source code into smaller units called tokens. Each token represents a meaningful component of the source code, such as keywords, identifiers, literals, operators, or punctuation marks.

**Key Components of the Lexer**

* **Input:** Source code as a string.
* **Output:** A list of tokens, where each token is represented by:
  + **Type:** Specifies the category of the token (e.g., Keyword, Identifier, Literal, Operator).
  + **Value:** Stores the actual value of the token (e.g., class, int, +).
  + **Position Information:** Line and column numbers for error reporting.

**Implementation Highlights**

1. The Lexer reads the source code character by character.
2. It groups characters into meaningful tokens based on predefined rules.
3. Tokens are categorized into types like Keyword, Identifier, Literal, etc.
4. Whitespace and comments are ignored unless significant (e.g., for indentation-based languages).

**Example Tokenization**

For the input code:

class Example {

int x = 10;

}

The Lexer produces the following tokens:

| **Token Type** | **Value** | **Line** | **Column** |
| --- | --- | --- | --- |
| Keyword | class | 1 | 1 |
| Identifier | Example | 1 | 7 |
| Punctuation | { | 1 | 15 |
| Keyword | int | 2 | 5 |
| Identifier | x | 2 | 9 |
| Operator | = | 2 | 11 |
| Literal | 10 | 2 | 13 |
| Punctuation | ; | 2 | 15 |
| Punctuation | } | 3 | 1 |

**Significance**

* The Lexer ensures that the Parser operates on structured data instead of raw source code.
* By categorizing tokens, it simplifies the job of the Parser and ensures accurate syntax analysis.

**2. Parser Function:**

**Context-Free Grammar (CFG)**

The implemented Parser adheres to the following CFG, which defines the structure of the source code:

**CompilationUnit → ClassDeclaration (TypeDeclaration)\***

**ClassDeclaration → "class" Identifier "{" (TypeDeclaration)\* "}"**

**TypeDeclaration → AccessModifier? Modifier\* DataType Identifier (MethodDeclaration | LocalVariableDeclaration)**

**MethodDeclaration → AccessModifier? Modifier\* ReturnType Identifier "(" (Parameter ("," Parameter)\*)? ")" Block**

**LocalVariableDeclaration → DataType Identifier "=" Literal ";"**

**Block → "{" (Statement)\* "}"**

**Statement → LocalVariableDeclaration | ExpressionStatement | MethodDeclaration**

**ExpressionStatement → Identifier (MethodCall | Expression)\* ";"**

**MethodCall → "." Identifier "(" (Expression ("," Expression)\*)? ")"**

**Parameter → DataType Identifier**

**AccessModifier → "public" | "private" | "protected" | "internal"**

**Modifier → "static" | "const" | "readonly" | "abstract" | "sealed"**

**DataType → "int" | "double" | "string" | "bool" | "char" | "long" | "void"**

**ReturnType → DataType**

**Literal → IntegerLiteral | StringLiteral**

**IntegerLiteral → [0-9]+**

**StringLiteral → "\"" [^"]\* "\""**

**Identifier → [a-zA-Z\_][a-zA-Z0-9\_]\***

**Explanation:**

1. **CompilationUnit**: Represents the entire unit of code being parsed. It starts with a ClassDeclaration and can have multiple TypeDeclarations (classes, structs, etc.).
2. **ClassDeclaration**: Represents a class definition, which starts with the class keyword, followed by an Identifier (class name), and contains a block of code with one or more TypeDeclarations.
3. **TypeDeclaration**: Represents method or local variable declarations inside a class. This includes optional access modifiers and modifiers (static, readonly, etc.), followed by a data type and an identifier (for the method name or variable name).
4. **MethodDeclaration**: Represents a method, which includes a return type, method name, parameters, and a method body (Block).
5. **LocalVariableDeclaration**: Represents a local variable inside a method, including the data type, variable name, and an optional initialization.
6. **Block**: Represents a block of code enclosed in curly braces ({ }), which can contain multiple statements.
7. **ExpressionStatement**: Represents a statement that contains an expression, such as an identifier (a variable or method) or method calls.
8. **MethodCall**: Represents a method call with arguments.
9. **Parameter**: Represents a parameter in a method's signature, which has a DataType and an Identifier (parameter name).
10. **AccessModifier**: Represents the access modifiers (e.g., public, private, etc.).
11. **Modifier**: Represents various modifiers for methods or variables (e.g., static, const, readonly, etc.).
12. **DataType**: Represents the data types for methods, variables, and return types.
13. **Literal**: Represents literals like integers or strings in the code.
14. **Identifier**: Represents identifiers, typically variables, method names, class names, or parameters.

**Purpose**

The Parser processes the list of tokens generated by the Lexer to construct an **Abstract Syntax Tree (AST)**. The AST is a hierarchical, tree-like representation of the program's structure, which preserves the syntactic relationships between various elements of the code.

**Key Components of the Parser**

* **Input:** A list of tokens generated by the Lexer.
* **Output:** An AST that represents the program's structure.

**Implementation Highlights**

1. **Top-Level Parsing:**
   * The entry point is the Parse method, which calls ParseCompilationUnit to process the entire source code.
2. **Grammar Rules:**
   * The Parser follows a set of grammar rules to recognize constructs like class declarations, method declarations, and variable assignments.
3. **Error Handling:**
   * The Parser identifies syntax errors, such as unexpected tokens, and provides meaningful error messages with line and column numbers.

**Major Methods**

1. **ParseCompilationUnit:** Handles the entire program and recognizes multiple class or struct declarations.
2. **ParseClassDeclaration:** Parses a class definition, including its name and body.
3. **ParseMethodDeclaration:** Processes method signatures and their bodies.
4. **ParseBlock:** Handles code blocks enclosed in {}.
5. **ParseStatement:** Recognizes individual statements, such as variable declarations or method calls.

**Example of Grammar Rules**

For a class definition:

class Identifier { ClassBody }

The Parser processes:

1. The class keyword (ensures the token matches class).
2. The class name (expects an Identifier).
3. The opening { (ensures the token matches {).
4. The class body, which includes methods, fields, etc.
5. The closing } (ensures the token matches }).

**Abstract Syntax Tree**

For the input:

class Example {

int x = 10;

void Print() {

Console.WriteLine(x);

}

}

The AST looks like this:

CompilationUnit

└─ ClassDeclaration: Example

├─ Field: int x = 10

└─ Method: void Print()

└─ Block

└─ ExpressionStatement

└─ MethodCall: Console.WriteLine

└─ Identifier: x

**Significance**

* The AST serves as the foundation for further compiler phases, such as semantic analysis, optimization, and code generation.
* It represents the program in a structured format that is easier to analyze and manipulate.

**3. Error Handling in the Parser**

The Parser provides robust error detection by verifying token sequences against the grammar rules. If a mismatch occurs:

* An error is thrown with a message specifying the expected token, actual token, and its position in the source code.

For example, encountering:

class Example (

Produces the error:

Syntax error: Expected '{', found '(' at Line: 1, Column: 15