**Name: Shah Sarmad Hamdani**

**Reg. No: fa21-bcs-021**

**1. Lexical Analysis**

**Purpose**: Transform the raw source code into a sequence of meaningful tokens.

**Description**:

* This phase is performed by the **Lexer** (also called a scanner or lexical analyzer).
* The input to this phase is the **source code** as a plain text file.
* The output is a stream of **tokens**. Tokens are the smallest meaningful units of the program, such as keywords, identifiers, operators, and literals.

**How it works**:

1. **Input Processing**: The lexer reads the source code character by character.
2. **Pattern Matching**: It matches character sequences against predefined patterns (regular expressions) for known token types.
   * Example patterns:
     + if|else|while → Keyword
     + [a-zA-Z\_][a-zA-Z0-9\_]\* → Identifier
     + \d+ → Numeric literal
     + \+|-|\\*|/ → Operator
3. **Error Handling**: If an unrecognized pattern is encountered, the lexer raises a **lexical error**.
4. **Token Stream Creation**: The lexer outputs a list of tokens, which includes the token type and sometimes its value or location in the source code.

**Example**: Input:

c

Copy code

int x = 5 + 3;

Output (tokens):

scss

Copy code

Token(Type=Keyword, Value=int)

Token(Type=Identifier, Value=x)

Token(Type=Operator, Value==)

Token(Type=Number, Value=5)

Token(Type=Operator, Value=+)

Token(Type=Number, Value=3)

Token(Type=Symbol, Value=;)

**Significance**:

* Simplifies the input for subsequent phases by abstracting away raw text.
* Ensures that only syntactically correct symbols proceed further.

**2. Syntax Analysis**

**Purpose**: Construct an **Abstract Syntax Tree (AST)** that represents the grammatical structure of the program.

**Description**:

* This phase is performed by the **Parser** (also called a syntactic analyzer).
* The input to this phase is the **stream of tokens** produced by the lexer.
* The output is an **Abstract Syntax Tree (AST)**, a tree-like representation of the code's syntactic structure.

**How it works**:

1. **Grammar Rules**: The parser uses a predefined set of grammar rules (e.g., context-free grammar) that define the programming language's syntax.
   * Example:

mathematica

Copy code

Expression → Term '+' Term

Term → Factor '\*' Factor

Factor → Number | Identifier

1. **Parsing Methods**:
   * **Top-Down Parsing**: Starts with the highest-level rule and recursively breaks it down.
   * **Bottom-Up Parsing**: Starts with the input tokens and works backwards to construct the tree.
2. **AST Construction**: The parser builds an AST that reflects the hierarchy of the program’s structure.
   * Example AST for 5 + 3:

markdown

Copy code

+

/ \

5 3

1. **Error Detection and Recovery**:
   * If a token sequence violates the grammar, the parser raises a **syntax error** and attempts to recover (e.g., skip invalid tokens).

**Example**: Input Tokens:

arduino

Copy code

int x = 5 + 3;

Output AST:

markdown

Copy code

=

/ \

x +

/ \

5 3

**Significance**:

* Ensures that the program adheres to the language's grammatical rules.
* Provides a structured representation (AST) for later phases, such as semantic analysis and code generation.

**Comparison of Functions**

| **Feature** | **Lexical Analysis** | **Syntax Analysis** |
| --- | --- | --- |
| **Input** | Raw source code | Stream of tokens |
| **Output** | Stream of tokens | Abstract Syntax Tree (AST) |
| **Main Focus** | Identifying token patterns | Validating grammatical structure |
| **Error Type** | Lexical errors | Syntax errors |
| **Tools/Techniques** | Regular expressions | Context-free grammar, parsers |

Together, these two functions form the backbone of any compiler. Lexical analysis simplifies the raw input, while syntax analysis provides a structured representation, ensuring the program is syntactically correct and ready for further processing.