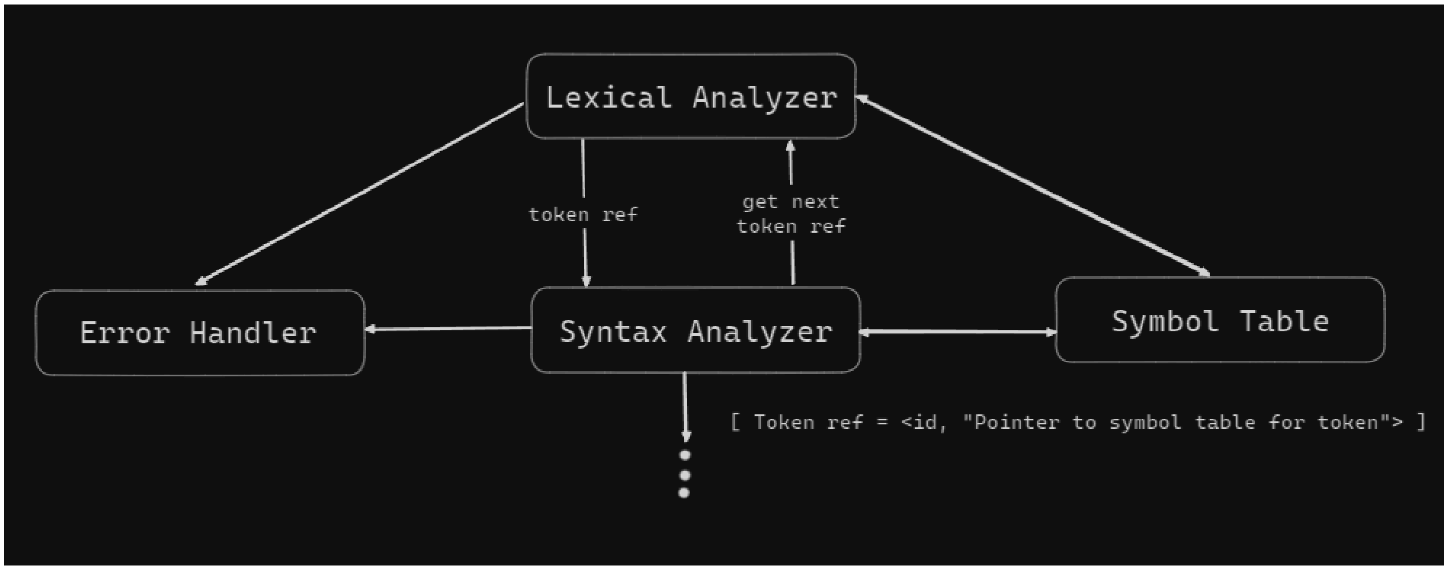
Intro:

- As the name suggests, It takes the tokens provided by lexical analyzer from the symbol table and verifies whether the tokens are in correct syntax or not, according to the Grammar of the programming language.
- As the output, the Syntax Analyzer generates a parse tree. That is, the syntax is checked by generating a parse tree for each token.
- A parse tree is a hierarchical structure of a lexeme built by referencing the grammar rules. (production rules)
- If a parse tree for a token is fully constructed, it means that the syntax is correct (token satisfies the grammar), else the Syntax analyzer informs about the syntax error to error handler, with line number and position at which the syntax is mismatched.
- The parsing methods are of two types, based on the direction of parsing:

a. Top-Down Parsing (ex: LL Parser)
b. Bottom-Up Parsing (ex: LR Parser)

- The grammar considered by the parser of a compiler, is often in the form of CFG (Context-Free-Grammer).
- So, Inshort, a syntax-analyzer (parser) functions are:

- Ensuring Syntax Validity: Checks if the syntaxes of lexemes are correct.
- Constructing a parse tree: Hierarchical Representation of a lexeme w.r.t the grammar.
- Syntax Error Detection: Identifying Syntax Errors and reports it to the Error Handler of Compiler.



CFG: (Notes)

Top-Down Parsing

- **Approach**: Top-down parsing starts from the root (the start symbol) of the parse tree and attempts to construct the parse tree down to the leaves (the input symbols).
- **Procedure**:
    - Begin with the start symbol.
    - Expand the start symbol into production rules, choosing one production at each step.
    - Attempt to match the expanded symbols with the input tokens.

    - Backtrack if a wrong production rule is chosen (used in backtracking parsers like recursive descent parsers).
- **Types**:
    - **Recursive Descent Parsing**: A top-down parser that uses a set of recursive procedures to process the input.
    - **Non-Recursive Descent / LL(1) (Left-to-right, Leftmost derivation) / Predictive Parsing**: A type of top-down parsing that reads the input from left to right and constructs a leftmost derivation.
- **Advantages**:
    - Simple and intuitive.
    - Can be implemented easily with recursive functions (as in recursive descent parsing).
- **Disadvantages**:
    - Inefficient for grammars with a lot of backtracking.
    - Cannot handle left-recursive grammars directly.

Bottom-Up Parsing

- **Approach**: Bottom-up parsing starts from the leaves (the input symbols) and attempts to construct the parse tree up to the root (the start symbol).
- **Procedure**:
    - Begin with the input tokens.
    - Attempt to reduce the tokens into non-terminals using the production rules in reverse.
    - Continue reducing until the start symbol is derived.
- **Types**:
    - **Shift-Reduce Parsing**: A type of bottom-up parsing where tokens are shifted onto a stack and then reduced based on production rules.

Working

1. **Stack**: The parser uses a stack to hold symbols (terminals and non-terminals) as it processes the input string.
2. **Input Buffer**: The input string is stored in a buffer, and the parser reads symbols from this buffer one at a time.

Key Operations

- **Shift**:
    - The parser reads the next input symbol from the buffer and pushes it onto the stack.
    - This operation shifts the focus to the next symbol in the input.
- **Reduce**:
    - The parser looks at the symbols on the top of the stack and tries to match them with the right-hand side of a production rule.
    - If a match is found, the symbols are popped from the stack and replaced by the non-terminal on the left-hand side of the production rule.
    - This operation effectively reduces the stack's contents to a single non-terminal that represents a completed sub-tree of the parse tree.

    - **LR Parsing (Left-to-right, Rightmost derivation)**: A powerful bottom-up parsing strategy that reads the input from left to right and constructs a rightmost derivation in reverse.
        - Type of LR Parsers:
            1. Simple LR(SLR)
            2. Canonical LR(CLR)
            3. Look Ahead LR(LALR)
- **Advantages**:
    - More powerful than top-down parsers; can handle a broader range of grammars.
    - Can efficiently parse left-recursive grammars.
    - Provides better error detection.
- **Disadvantages**:
    - More complex to implement.
    - Requires more sophisticated algorithms and data structures.