Problem Description

Develop the following modules of the compiler for implementing the given language.

Lexical Analyzer: This module takes as input the file containing the source code in the given language and produces the tokens. The lexical analyzer module scans the input only once and collects all relevant information required by other modules of the compiler. The lexical analyzer ignores the comments and white spaces, while recognizes the useful lexemes as valid tokens The lexical errors are reported by this module when it sees any symbol or pattern not belonging to the language. Your lexical analyzer must

- **Tokenize lexemes appropriately**
- **Maintain all information collected during a single pass of the source code**
- **Be efficient with respect to time and space complexity**
- **Report lexical errors**

Syntax Analyzer: This module takes as input the token stream from the lexical analyzer module and verifies the syntactic correctness of the source code. This uses predictive parser (using parsing table) to establish the syntactic structure of the source code. As the parser sees next token, verifies its correctness, it uses the token information to build a tree node and inserts appropriately in the parse tree corresponding to the input source code. If the source code (in given language) is syntactically correct, a corresponding parse tree is produced as the output. If the input is syntactically wrong, errors are reported appropriately. Your syntax analyzer (Parser) must

i. **Ensure a single pass of the token stream**
ii. **Use predictive parser using parsing table**
iii. **Produce as output the parse tree, if the source code is syntactically correct**
iv. **Produce a list of syntax errors with appropriate messages and line numbers**

Abstract Syntax Tree (AST): This module takes as input the parse tree . The abstract syntax tree is generated by eliminating unnecessary details such as semicolon, colon, comma, parenthesis, square brackets, range operator, assignment operator etc. Any node in the abstract Syntax Tree retains the information about the non terminal symbol that would have derived the corresponding subtree (which is essential to keep the syntactic structure intact with you throughout during front end). The AST retains only those children that are essential later for semantic analysis, while those appearing as a linear chain, are collapsed.

Implement a AST with the following general rule

(parent)(children list)

where parent is the same non-terminal symbol that exists in the parse tree, and the children list contains the nodes which are meaningful. The leaf nodes of the AST

still continue to contain the tokens and other relevant information extracted. The AST later helps in traversing the tree faster than the parse tree traversing the meaningful nodes only. You must

i. **Prepare the rules to derive the abstract syntax tree structure**
ii. **Modify the structure of the parse tree node (at least a link to the ST can be added) to use for AST.**
iii. **Ensure a single pass of the parse tree.**
iv. **Produce as output the Abstract Syntax Tree, if the source code is syntactically correct**