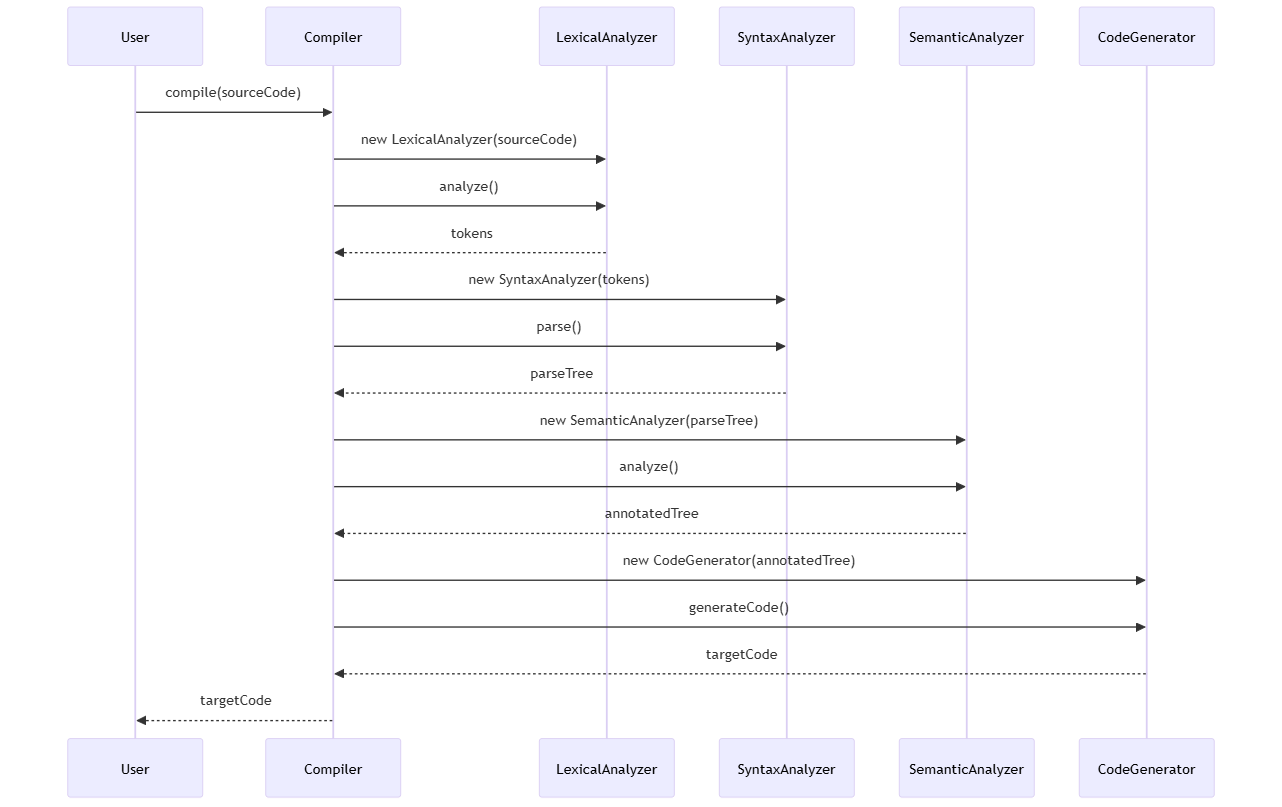
**Repository URL:**

https://github.com/Abdullah-Malik-39/MiniJavaCompiler

**Question: 1**



**Question: 2**

analyze() in LexicalAnalyzer:

This function is responsible for the lexical analysis phase, where the source code is read and broken down into a sequence of tokens. These tokens represent the smallest units of meaning, such as keywords, identifiers, operators, and literals. The analyze() function processes the input source code character by character, identifies patterns that match the defined token types, and generates a list of tokens to be used in subsequent compilation stages. This step is crucial as it transforms raw source code into a structured format that the parser can work with.

parse() in SyntaxAnalyzer:

Following lexical analysis, the parse() function in the SyntaxAnalyzer takes the list of tokens produced by the LexicalAnalyzer and performs syntax analysis. This involves checking the token sequence against the grammatical rules of the MiniJava language to ensure that the code's structure is syntactically correct. The parse() function constructs a parse tree (or abstract syntax tree) that represents the hierarchical syntactic structure of the source code. This tree serves as a foundation for further analysis, such as semantic checking and code generation.

**Question: 3**

1. Constant Folding
2. Dead Code Elimination
3. Common Subexpression Elimination
4. Inline Expansion

**Question: 4**

**Input:**



**Output:**

****

**Question: 5**

Key Responsibilities of the Semantic Analysis Function:

1. Symbol Table Construction:

Purpose: Collects information about variable and method declarations, their types, and scopes.

Process:

Traverses the AST to identify declarations.

Records each symbol's name, type, and scope in the symbol table.

Example:

For a variable declaration int x;, it records x with type int in the current scope.

1. Type Checking:

Purpose: Ensures that operations in the code are semantically correct with respect to data types.

Process:

Examines expressions and statements to verify type compatibility.

Checks assignments, arithmetic operations, method calls, etc., for type correctness.

Example:

In an assignment x = y + z;, it verifies that y and z are of compatible types, and x can accept the result.

1. Scope Resolution:

Purpose: Determines the visibility and lifetime of variables and methods to prevent illegal references.

Process:

Maintains information about nested scopes (e.g., method scopes within class scopes).

Ensures that references to variables and methods are valid within the current scope.

Example:

Detects if a variable x is used outside its declared scope, which would result in an error.

1. Inheritance and Overriding Checks:

Purpose: Validates correct usage of inheritance, ensuring that subclass relationships and method overriding adhere to language rules.

Process:

Checks that subclasses properly extend base classes.

Ensures overridden methods maintain consistent signatures with their base class counterparts.