What is a compiler?

A compiler is a program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer.

The compiler goes through five steps (one is optional) before arriving at a conclusion:
1.Lexical Analysis
2.Syntactic Analysis (ie Parsing)
3.Semantic Analysis
4.Code Generation

1. Lexical Analysis

Key Functionalities:

- **Breaking down source code into tokens:** The lexer takes a string of code as input and segments it into meaningful units called tokens, such as keywords, variables, operators, and numbers.
- **Providing tokens for parsing:** The parsed tokens are then utilized by a parser during the syntax analysis stage of compiler construction.

Code Structure:

- **Lexer class:**
    - Contains the core functionality of the lexer.
    - Encapsulates the code to be analyzed (code), maintains a pointer to the current position (currentIndex), and stores the current and previous tokens (currentToken, previousToken).

Key Methods:

- **nextToken():**
    - Identifies and returns the next valid token from the input code.
    - Iterates through the code, handling different token types:
        - Whitespace (' ', \r, \t, \n) is skipped using skipWhiteSpace().
        - Equals operator (=) creates an EQUALS_OPERATOR token.
        - Numbers (0-9) are read using readNumber() and construct a NUMBER token.
        - Variables (letters) are read using readVariable() and create either a SHOW token (for "show") or a VARIABLE token.
        - Unrecognized characters throw a LexerException.
- **readNumber():** Reads a sequence of digits and returns the corresponding string representing the integer value.
- **readVariable():** Reads a sequence of letters and returns the string representing the variable name.
- **skipWhiteSpace():** Skips any whitespace characters in the code.
- **isEndOfCode():** Checks if the end of the code has been reached.
- **getPreviousToken():** Returns the previous token for potential use in more complex grammars.
- **getCurrentToken():** Returns the current token.

2. Syntatic Analysis

Key Functionalities:

- **Constructing a parse tree:** The primary purpose of this code is to analyze a stream of tokens (generated by a lexer) and create an Abstract Syntax Tree (AST) that represents the program's structure.
- **Enforcing grammatical rules:** It ensures that the token sequence adheres to the grammatical rules of the programming language being parsed.
- **Detecting syntax errors:** If it encounters invalid token sequences, it throws parser exceptions to signal syntax errors.

Code Structure:

- **Parser class:**
    - Contains the core parsing functionality.
    - Collaborates with a Lexer to obtain tokens for parsing.

Key Methods:

- **parseProgram():**
    - Orchestrates the overall parsing process.
    - Repetitively calls parseStatement() to parse individual statements.
    - Constructs a ProgramContext containing parsed statements.
- **parseStatement():**
    - Parses a single statement, creating either a LetContext (for assignment statements) or a ShowContext (for output statements).
- **parseLet():**
    - Parses the syntax of a LET statement, handling variable declaration and assignment.
- **parseShow():**
    - Parses the syntax of a SHOW statement, which outputs a value or variable.
- **parseTerminalNode():**
    - Constructs a TerminalNode representing a single parsed token.

Parsing Approach:

- **Deterministic Finite Automata (DFA):** Although not explicitly employing a DFA data structure, the parsing logic relies on a similar state-based approach using a series of if-else statements to validate token sequences and create appropriate AST nodes.

An Explanation on Parsing

ParseTree Interface:

- **Purpose:** Defines the core structure and behavior of nodes in a parse tree.
- **Key Elements:**
    - getParent() and setParent(): Manage hierarchy within the tree.
    - getText(): Retrieves concatenated text content from children or token value for terminals.
    - getPayload(): Returns the underlying payload (this object or token).

- addChild(): Adds a child node to the tree.
- getChild(i): Accesses a child node by index.
- getChildCount(): Counts the number of child nodes.
- toStringTree(): Formats the tree as a readable string.
- accept(Visitor): Enables visitor pattern for custom operations on the tree.

ParserRuleContext Class:

- **Purpose:** Concrete implementation of ParseTree, serving as the foundation for representing grammar elements in the parsed code.
- **Structure:**
    - Inherits from ParseTree, providing access to core tree-related methods.
    - Contains parent and children fields for managing relationships between nodes.
- **Key Methods:**
    - Overloads methods from ParseTree to handle specific node content and child management.

Leveraging ParseTree and ParserRuleContext in the Parser Class:

1. **Building the Tree:**
    - As the Parser parses tokens, it creates instances of ParserRuleContext subclasses that represent specific grammar elements in the code (e.g., LetContext, ShowContext, etc.). These subclasses inherit from ParserRuleContext which implements the ParseTree interface.
2. **Establishing Parent-Child Relationships:**
    - When the Parser encounters production rules in the grammar (e.g., LET variable = expression;), it creates the appropriate nodes (e.g., LetContext, TerminalNode for variables and expressions) and sets their parent-child relationships using methods like setParent() and addChild().
    - The ParserRuleContext class automatically establishes these relationships as it manages the parent and children fields.
3. **Accessing Node Information:**
    - During parsing, the Parser might need to access information from child nodes. Methods like getText() and getChild(i) from the ParseTree interface allow the parser to extract text content from child nodes (e.g., variable names, expressions) or navigate the tree structure.
4. **Custom Operations with Visitor (Optional):**
    - The accept(Visitor) method in ParseTree is a placeholder for the visitor pattern. If needed, the compiler might implement a Visitor class to perform specific tasks on the parse tree, such as semantic analysis or code generation. The Parser could then call accept() on the root node, triggering the visitor's logic on the entire tree structure.

Example:

Consider parsing the following line: LET x = 10;

1. The Parser would create a LetContext node as the root.

2. It would then create TerminalNode instances for x and 10.
3. Using addChild(), the parser would make x and 10 children of the LetContext node.
4. By calling getText() on the root (LetContext), the parser could retrieve the textual representation of the entire statement (LET x = 10;).

3. Semantic Analysis

Purpose:

- This SemanticAnalyzer class performs semantic analysis on a parsed program's AST (Abstract Syntax Tree), checking for type errors and logical inconsistencies beyond pure syntax.

Key Elements:

- **Inheritance:** It extends SimplerLangBaseVisitor, inheriting a framework for traversing a parse tree and performing actions on specific nodes.
- **Variable Map:** It maintains a Map<String, String> called variableMap to track declared variables and their types for semantic checks.

Key Methods:

- **visitStatement(StatementContext):**
  - Validates that a statement is either a LET or a SHOW statement, not both.
- **visitLet(LetContext):**
  - Enforces rules for LET statements:
    - Verifies that variable names and values are non-empty.
    - Ensures variable values are integers.
    - Updates variableMap with declared variables.
- **visitShow(ShowContext):**
  - Enforces rules for SHOW statements:
    - Ensures SHOW has either a variable or an integer argument, not both.
    - Verifies that integer arguments are valid integers (not strictly necessary due to tokenizer).
    - Checks that variable arguments have been previously declared using LET.

Role in Semantic Analysis:

- The compiler invokes this class to traverse the AST, calling appropriate visitor methods based on node types.
- These methods check for semantic errors and throw SemanticException if issues are found.
- The variableMap enables tracking variable usage and preventing undefined variable errors.

Example:

- Consider the code SHOW x without a prior LET declaration of x.

- The visitShow() method would detect this using variableMap and throw a SemanticException.

3. Code generation

The phase where the compiler translates the **intermediate representation (IR)** of the source code into **machine code** (or another lower-level language) that can be directly executed by the target system. This machine code is often referred to as **target code**.

InterpreterVisitor.java:

Purpose:

- Executes SimplerLang code directly as it's parsed, without creating a separate executable file.

Key Features:

- Uses a HashMap to store variable values in memory.
- Handles let statements by storing assignments in the variable map.
- Handles show statements by printing values from the variable map or constant literals.

Key Code Explanation:

- **variableMap:** A HashMap to store variable names and their associated values.
- **visitLet(LetContext context):**
    - Stores the variable name and value in the variableMap.
- **visitShow(ShowContext context):**
    - Prints the value of a variable or constant literal using System.out.println().

CodeGeneratorVisitor.java:

Purpose:

- Creates a Java bytecode file (.class) that can be executed on a Java Virtual Machine (JVM).

Key Features:

- Uses ASM to generate bytecode instructions.
- Manages variable storage using a variableIndexMap.
- Emits bytecode for control flow and variable management.

Key Code Explanation:

- **classWriter:** An ASM ClassWriter to create the bytecode class.
- **variableIndexMap:** Tracks variable names and their corresponding bytecode indices.
- **mainMethodVisitor:** An ASM MethodVisitor to generate code for the main method.
- **visitProgram(ProgramContext context):**
    - Initializes the class structure and main method.
- **visitLet(LetContext context):**
    - Generates bytecode to store the variable's value in JVM memory.
- **visitShow(ShowContext context):**
    - Generates bytecode to print the value of a variable or constant literal.

- **writeToFile(byte[] code, String filePath):**
    - Writes the generated bytecode to a .class file.

Key Points:

- Both classes are part of the code generation phase of a compiler for the SimplerLang language.
- The interpreter directly executes code during parsing, while the code generator creates a separate executable file.
- The choice between them depends on factors like portability, performance requirements, and distribution needs.