1. Tokenization

The first step in interpreting the code involves tokenizing the input string. The language defines several token types:

- **Arithmetic Operators**: PLUS (+), MINUS (-), MUL (*), DIV (/), MOD (%)
- **Boolean Operators**: AND (&&), OR (||), NOT (!)
- **Comparison Operators**: EQ (==), NEQ (!=), GT (>), LT (<), GEQ (>=), LEQ (<=)
- **Other Tokens**:
    - **Parentheses**: LPAREN, RPAREN for grouping expressions.
    - **Keywords**: IF, DEFUN (function definition), LAMBDA (lambda functions), ELSE (for conditional branching).
    - **Identifiers and Constants**: ID for variable names, INTEGER for numbers, and BOOLEAN for true/false values.

Design Choices:

- **Integer Representation**: Integers are assumed to be non-decimal, parsed directly from the input without allowing floating-point values.
- **Boolean Values**: The language includes True and False as built-in values for boolean operations.
- **Error Handling for Unknown Tokens**: Any character not explicitly handled (such as unrecognized symbols) will throw an error.

Assumptions:

- Input is well-formed and will only contain supported symbols, numbers, and keywords.
- The lexer will always skip whitespace and recognize multi-character tokens such as >= and != through lookahead (e.g., peek method).

---

2. Grammar and Parsing

The language supports a variety of expressions, including arithmetic, boolean logic, function definitions, and lambda expressions. The grammar rules are handled in the Parser class, which builds an Abstract Syntax Tree (AST) from the list of tokens.

Key Grammar Elements:

1. **Arithmetic Expressions**:
    - Support standard precedence rules for operators (+, -, *, /, %).
    - Parentheses are used for grouping expressions to enforce specific evaluation orders.
2. **Boolean Expressions**:
    - Support logical operations (&&, ||, !) and comparison operators (==, !=, >, <, >=, <=).

3. **Functions**:
    - **Defun**: Functions are defined using the defun keyword. The function body can contain a block of expressions enclosed in braces {}.
    - **Lambda**: Anonymous functions are supported via the lambd keyword. These lambda functions can be immediately invoked and can accept parameters.
4. **Conditionals**:
    - The if keyword supports conditional branching. Optionally, an else clause is used for fallback behavior.

Assumptions:

- Function arguments and lambda parameters are always passed correctly by the user.
- There is no support for floating-point arithmetic, string manipulation, or advanced data structures like arrays.
- The parser assumes a linear, left-to-right evaluation order without advanced optimizations.

---

3. Abstract Syntax Tree (AST)

The language uses several custom AST node types:

- **BinOp**: Represents binary operations such as a + b, x > y.
- **UnaryOp**: Handles unary operations such as -x or !True.
- **Num**: Holds integer values.
- **Bool**: Holds boolean values.
- **Var**: Represents variables.
- **Function**: Stores function definitions with their parameters and bodies.
- **Lambda**: Represents anonymous functions with parameters and a body.
- **Call**: Represents function or lambda invocations.
- **If**: Holds conditional statements with an optional else branch.

Design Choices:

- The AST follows a node-based approach to ensure flexibility and extensibility.
- Unary operations are treated distinctly from binary operations for clarity.
- Function calls and lambda invocations are modeled uniformly, where both are treated as first-class citizens.

Assumptions:

- The AST generation follows strict grammar rules, and syntax errors are detected early in the parsing phase.
- Each function or lambda stores its own scope, ensuring proper variable resolution during execution.

---

4. Interpreter and Execution

The interpreter walks the AST and evaluates expressions according to their type. The Interpreter class follows a visitor pattern to handle each node type (visit_Num, visit_BinOp, etc.).

Key Features:

- **Global Scope**: The interpreter maintains a global scope dictionary for storing variable and function values.
- **Function Invocation**: Functions and lambdas are stored as callable objects. The interpreter checks for correct argument counts and evaluates the body in the function's scope.
- **Conditional Execution**: The if node evaluates the condition and selects the appropriate branch.

Design Choices:

- **Integer Division**: Division operations always return integer results (e.g., 5 / 2 == 2).
- **No Side-Effects**: Functions and lambdas are pure, meaning they don't modify the global scope unless explicitly returning a value.
- **Strict Argument Matching**: Functions and lambdas must be called with exactly the number of arguments they expect.

Assumptions:

- Recursive function calls are supported, but there is no tail-call optimization.
- The language does not support stateful constructs like loops, mutability, or exceptions beyond basic error handling for undefined variables and functions.
- The interpreter assumes the user will handle function scoping correctly without requiring deep nested scopes.

---

5. Error Handling

Throughout the lexer, parser, and interpreter, several types of errors are anticipated:

- **Lexer Errors**: Unrecognized tokens trigger errors, halting execution.
- **Parser Errors**: Syntax errors such as mismatched parentheses or incomplete expressions are flagged during parsing.
- **Runtime Errors**: Division by zero, calling undefined functions, and argument count mismatches lead to runtime errors.

Assumptions:

- Error messages are not designed for full user-friendliness but are intended to assist in debugging by providing specific token or AST-related feedback.
- Runtime errors are not recoverable within the current session (i.e., REPL commands fail and require a new input).
