Design Considerations and assumptions – Yahav Yehoshua Bariah

ERROR -

Design Considerations

1. **Error Hierarchy and Specificity**:
    - The system uses a base Error class with specialized subclasses (IllegalCharError, ExpectedCharError, InvalidSyntaxError, and RTError) to handle different types of errors. This design allows for specific error handling and detailed error reporting, improving clarity and debugging capabilities.
2. **Error Reporting and Contextual Information**:
    - The Error class and its subclasses are designed to provide comprehensive error messages that include file name, line number, and visual indicators of the error's location. The RTError subclass further includes a traceback to help trace the error's origin, offering detailed context for runtime issues.

Assumptions

1. **Position Objects**:
    - The pos_start and pos_end attributes are assumed to be objects that encapsulate position information, such as file name, line number, and possibly column number, facilitating precise error location within the source code.
2. **Context Management**:
    - For RTError, it is assumed that the context object provides a meaningful traceback structure, including details like function names and parent contexts, enabling detailed runtime error reporting and debugging.

Position -

Design Considerations

1. **Position Tracking**:
    - The Position class is designed to keep track of the current position in a source text, including index, line, and column number. This enables precise location tracking of characters in the text, which is crucial for error reporting and text processing.
2. **Advancement and Copying**:
    - The advance() method updates the position based on the current character, handling line breaks appropriately. This method is essential for iterating through the text and updating the position during parsing or tokenization. The copy() method allows creating a duplicate of the current position, which is useful for maintaining original positions while processing or error reporting.

Assumptions

1. **Character Tracking**:
    - The Position class assumes that the current_char parameter in the advance() method will be used to determine if a newline character is encountered, affecting the line and column counters. This assumes that line breaks are denoted by '\n' and resets the column index upon encountering a newline.
2. **Text Consistency**:
    - It is assumed that fn (filename) and ftxt (file text) provided to the Position class are consistent with the text being processed. The ftxt is used for generating error messages and visual indicators, so its integrity and accuracy are crucial for proper error reporting.

TOKENS -

Design Considerations

1. **Token Types and Constants**:
    - A set of predefined token types (TT_INT, TT_IDENTIFIER, etc.) is used to classify various elements of the language. This design helps in identifying and differentiating between different types of tokens during lexical analysis. Constants for each token type ensure consistency and readability in token handling.
2. **Token Structure and Positioning**:
    - The Token class includes attributes for type, value, and positional information (pos_start and pos_end). This structure allows tokens to be associated with specific locations in the source code, aiding in accurate error reporting and debugging. The matches() method helps in token comparison, while __repr__() provides a clear representation of the token for debugging and logging purposes.

Assumptions

1. **Token Positioning**:
    - The Token class assumes that the pos_start and pos_end attributes are used to track the position of the token within the source text. This includes copying the position and advancing it for accurate location tracking. It also assumes that tokens generally span a single character unless pos_end is explicitly provided.
2. **Keyword List**:
    - The KEYWORDS list contains specific keywords used in the language. It is assumed that this list will be used to identify keywords during tokenization. The provided keywords like 'Lambd' and 'Defun' suggest that the language might include special constructs or functions associated with these keywords.

LEXER -

Design Considerations

1. **Tokenization Process**:
    - The Lexer class is designed to convert a source text into a list of tokens by iterating through the characters and matching them to predefined token types. This process involves handling whitespace, comments, and different characters, ensuring that tokens are generated accurately. Special handling is included for multi-character tokens like -> and !=, allowing for the recognition of more complex syntax.
2. **Error Handling and Position Tracking**:
    - The lexer includes detailed error handling for illegal characters and unexpected sequences (e.g., ExpectedCharError). It maintains precise position tracking using the Position class, which is crucial for accurate error reporting and debugging. The advance() method updates both the position and the current character, ensuring that tokens are correctly identified and errors are reported with context.

Assumptions

1. **Character and Token Definitions**:
    - It is assumed that token types and special characters (such as DIGITS, LETTERS, LETTERS_DIGITS, etc.) are defined elsewhere and correctly represent the set of valid characters in the source text. This allows the lexer to identify and handle various token types and characters appropriately.
2. **Comment Handling**:
    - The lexer assumes that comments start with # and continue to the end of the line. It processes comments by advancing through the text until the end of the comment, ensuring comments do not interfere with token generation. If comments are present, they are flagged but not included in the token list.

NODES -

Design Considerations

1. **Node Structure for Abstract Syntax Tree (AST)**:
    - Each node class (NumberNode, VarAccessNode, BinOpNode, UnaryOpNode, FuncDefNode, CallNode) represents a specific type of construct in the abstract syntax tree (AST) for the language. These nodes encapsulate both the syntactic elements (such as operators and variables) and their positions in the source code, facilitating structured representation and manipulation of the code during interpretation or compilation.
2. **Position Tracking**:
    - Each node includes pos_start and pos_end attributes to track the position of the corresponding source code segment. This design supports accurate error reporting and debugging by linking AST nodes to their source locations. Position information is propagated from tokens or nodes to ensure that the entire AST can be traced back to the original source text.

Assumptions

1. **Token Consistency**:
    - It is assumed that tokens used to create nodes (e.g., tok in NumberNode and op_tok in BinOpNode) are properly initialized with accurate position data (pos_start and pos_end). This consistency ensures that the position tracking in the AST nodes reflects the correct locations in the source code.
2. **Node Relationships**:
    - The design assumes that nodes are correctly connected according to the language's grammar rules. For instance, BinOpNode expects a left_node, op_tok, and right_node, while FuncDefNode expects var_name_tok, arg_name_toks, and body_node. Proper initialization and relationships among nodes are critical for correct AST construction and subsequent evaluation or execution.

PARSE RESULT -

Design Considerations

1. **Result Handling for Parsing**:
    - The ParseResult class is designed to encapsulate the outcome of a parsing operation, including whether it was successful or failed. It provides methods to handle both success and failure scenarios, ensuring that parsing results are managed consistently. The register() method facilitates chaining by checking and propagating errors from other ParseResult instances, enabling streamlined error handling and result aggregation.
2. **Success and Failure States**:
    - The class distinguishes between successful parse operations (success()) and failures (failure()). On success, it stores the resulting node and allows method chaining by returning self. On failure, it records the error, which can be used to diagnose parsing issues. This design ensures that parsing functions can return meaningful results or errors without interrupting the parsing flow.

Assumptions

1. **Result Types**:
    - It is assumed that methods like register() and success() receive valid ParseResult instances or appropriate node types. This requires that the parsing functions properly return ParseResult objects or node results, maintaining consistency in result handling.
2. **Error Propagation**:
    - The ParseResult class assumes that errors are instances of error classes (like IllegalCharError, ExpectedCharError, etc.). The register() method relies on these errors being properly set in ParseResult objects, ensuring that errors are correctly propagated and managed during parsing.

Parser -

Design Considerations

1. **Recursive Descent Parsing**:
    - The Parser class uses recursive descent parsing techniques, where each method (e.g., expr, comp_expr, arith_expr, etc.) handles a specific part of the grammar. This approach allows for straightforward implementation of complex parsing rules by breaking down the grammar into manageable functions. Each parsing method returns a ParseResult that includes either a successful node or an error, facilitating error propagation and result handling.

2. **Error Reporting and Token Management**:
    - The parser handles errors by creating InvalidSyntaxError instances when unexpected tokens are encountered. It also manages tokens by advancing (advance()), deleting (delete()), or checking them. This allows for precise control over token processing and ensures that the parser can detect and report syntax errors accurately, including issues with function definitions, lambda expressions, and operator usage.

Assumptions

1. **Token Stream Integrity**:
    - The Parser assumes that the token stream provided during initialization is valid and that tokens are properly generated by the lexer. This includes correct types, values, and positions. The parser relies on the integrity of the token stream for accurate parsing and error handling.
2. **Consistent Token Types and Grammar Rules**:
    - The parser assumes that tokens conform to the predefined types (TT_INT, TT_IDENTIFIER, etc.) and that grammar rules are consistently followed. This includes handling operators, function definitions, and lambda expressions. The parser's methods are designed to process tokens based on these assumptions, so any deviation or error in token types or grammar can lead to parsing failures.

RUNTIME RESULT -

Design Considerations

1. **Result Management in Runtime Execution**:
    - The RTResult class is designed to manage the outcome of runtime execution, similar to ParseResult in parsing. It encapsulates the result of runtime operations, allowing for clear separation of successful values and errors. This design pattern ensures that the execution flow can handle results and errors consistently, facilitating error propagation and result handling during the interpretation or execution phase.
2. **Consistency with Parsing Results**:
    - The RTResult class mirrors the design of ParseResult, allowing for a consistent approach to handling results and errors across different stages of the interpreter. Methods like register(), success(), and failure() support method chaining and error propagation, which aligns with the parsing phase's approach to managing outcomes and errors.

Assumptions

1. **Error and Value Handling**:
    - It is assumed that any runtime error or value returned by operations is encapsulated in an RTResult instance. The class assumes that methods interacting with RTResult are designed to handle its value and error attributes correctly, ensuring proper result and error reporting during runtime execution.

2. **Consistency with Runtime Operations**:
    - The class assumes that the runtime operations and functions interacting with RTResult adhere to its structure and behavior. This includes providing appropriate values or errors and utilizing the register() method to handle results from other runtime functions. Consistency in how results and errors are managed is crucial for smooth runtime execution.

VALUES -

Design Considerations

1. **Base Value Class for Polymorphism**:
    - The Value class is designed as a base class for representing values in the interpreter. It provides a common interface for different value types (e.g., integers, strings, booleans) to handle operations and comparisons. This design supports polymorphism, allowing different types of values to implement their specific behavior while maintaining a consistent API for operations and comparisons.
2. **Error Handling for Unsupported Operations**:
    - Methods in the Value class, such as added_to() and subbed_by(), are designed to return an error when an unsupported operation is attempted. This approach ensures that the interpreter can handle errors gracefully and provides a mechanism for extending the class with specific implementations for different value types. The illegal_operation() method centralizes error reporting for unsupported operations, which simplifies error management across various value implementations.

Assumptions

1. **Standard Operation Handling**:
    - It is assumed that subclasses of Value will override methods like added_to() and subbed_by() to provide specific implementations for different value types. The base Value class provides default behavior that indicates unsupported operations, but subclasses are expected to implement the actual logic for operations and comparisons relevant to their type.
2. **Error Propagation**:
    - The class assumes that any instance of Value will have a pos_start and context set when operations are performed, ensuring that errors can be reported accurately. This setup is crucial for debugging and tracing errors in runtime operations. The illegal_operation() method relies on these attributes to generate meaningful error messages, which presumes that subclasses will correctly set pos_start and context when performing operations.

Number -

Design Considerations

1. **Arithmetic and Comparison Operations**:
    - **Handling Type Safety**: The Number class ensures that arithmetic and comparison operations are only performed with other Number instances. This design prevents operations with incompatible types and uses the illegal_operation method to handle errors gracefully. This approach helps maintain type safety and avoid runtime errors due to invalid operations.
    - **Error Management**: Division and modulo operations explicitly handle division by zero to prevent runtime exceptions. This is crucial for avoiding crashes and providing meaningful error messages, such as "Division by zero" and "Modulo by zero", which improve debugging and user experience.
2. **Context and Copy Management**:
    - **Context Preservation**: Each Number instance keeps track of its context and position through the set_context and set_pos methods. This design choice ensures that the context in which the number was created is preserved, aiding in error reporting and debugging.
    - **Copying Instances**: The copy method allows for the creation of duplicate Number instances while preserving context and position information. This feature supports scenarios where numbers need to be reused or manipulated without losing their original tracking details.

Assumptions

1. **Consistent Context Handling**:
    - It is assumed that the context and pos_start/pos_end attributes are correctly set and used throughout the Number class. These attributes provide necessary information for accurate error reporting and debugging, such as the origin of an operation or error within the code.
2. **Error Reporting Mechanism**:
    - The illegal_operation method in the Number class assumes that errors are reported with sufficient context to diagnose issues effectively. The design relies on the RTError class or similar error-handling mechanisms to provide detailed feedback on why an operation failed, facilitating easier debugging and understanding of runtime issues.

Function -

Design Considerations

1. **Function Execution and Argument Handling**:
    - **Argument Validation**: The execute method performs validation checks to ensure the number of arguments passed matches the function's expected number. It returns detailed error messages if there are too many or too few arguments, which improves error clarity and user feedback.

- **Context Management**: A new Context is created for each function execution to encapsulate local variables and function-specific settings. This design isolates the function's scope, preventing side effects on the global or parent contexts and ensuring proper variable management within the function.

2. **Copying Functions**:
    - **Preserving Function State**: The copy method creates a new instance of Function with the same name, body, and argument names. It also preserves the context and position information, which is essential for accurate debugging and error tracking. This design supports scenarios where functions need to be duplicated or manipulated while retaining their original properties.

Assumptions

1. **Consistent Context Handling**:
    - **Context Setup**: It is assumed that the Context class correctly manages the function's local scope and inherits from the parent context. This allows for effective variable management and error reporting specific to the function's execution environment.
2. **Error Reporting Structure**:
    - **Error Handling**: The RTError class is used to provide detailed error messages, including the number of mismatched arguments. The design assumes that RTError effectively captures and reports errors with relevant context, such as position and function name, to aid in debugging and issue resolution.

CONTEXT -

Design Considerations

1. **Context Hierarchy and Scope Management**:
    - **Hierarchical Structure**: The Context class is designed to manage hierarchical scopes. Each Context object can have a parent, allowing nested scopes to inherit from parent contexts. This design supports complex scoping rules where variables and functions can be resolved based on their scope levels.
    - **Display Name**: The display_name attribute provides a meaningful label for each context, which is useful for debugging and error reporting. It helps identify which context is active or causing issues, particularly in nested or complex function calls.
2. **Symbol Table Management**:
    - **Symbol Table Assignment**: The symbol_table attribute is used to store and manage variables and functions specific to the context. This design separates variable storage and retrieval from other context attributes, allowing for efficient variable lookups and updates.

Assumptions

1. **Context Initialization**:
    - **Parent Context**: It is assumed that the parent parameter in the Context constructor can be None, which indicates the top-level context (e.g., the global scope). If not None, the parent attribute should correctly reference a parent Context, enabling the inheritance of symbols and scope resolution.
    - **Symbol Table Initialization**: The symbol_table attribute is assumed to be properly initialized and populated within the context's lifecycle. The design assumes that it will be set to an appropriate SymbolTable instance, either directly or through context-specific operations.
2. **Error Reporting**:
    - **Parent Entry Position**: The parent_entry_pos attribute is included to provide the position information of the parent context entry, which is useful for tracing where a particular context was created. This attribute is assumed to aid in error reporting and debugging by providing context on where the context was instantiated.

SYMBOL TABLE -

Design Considerations

1. **Symbol Table Hierarchy**:
    - **Parent-Child Relationships**: The SymbolTable class supports hierarchical symbol resolution through the parent attribute. If a symbol is not found in the current table, the class will recursively search in the parent table. This allows for nested scopes (e.g., functions or blocks) to have access to symbols defined in outer scopes, facilitating lexical scoping.
    - **Symbol Storage**: Symbols are stored in a dictionary (self.symbols), providing efficient lookups, insertions, and deletions. This design choice ensures that symbol management within each context is both straightforward and performant.
2. **Symbol Management Operations**:
    - **Symbol Lookup**: The get method allows for hierarchical symbol lookup, enabling the retrieval of symbols from both the current context and its parent contexts. This is critical for resolving variable references and function calls that span multiple scopes.
    - **Symbol Assignment and Removal**: The set and remove methods provide straightforward operations for managing symbols within the current context. This design ensures that symbols can be dynamically added, updated, or deleted as needed during runtime.

Assumptions

1. **Symbol Table Initialization**:
    - **Parent Symbol Table**: It is assumed that the parent parameter in the SymbolTable constructor can be None, indicating the top-level or global symbol table. When not None, parent should be an instance of SymbolTable that represents the parent scope.
    - **Symbol Presence**: The get method assumes that if a symbol is not found in the current table, it should attempt to retrieve it from the parent table if one exists. This hierarchy is crucial for maintaining correct scope resolution.
2. **Symbol Table Operations**:
    - **Symbol Removal**: The remove method assumes that the symbol to be removed is present in the current table. There are no safeguards for attempting to remove a non-existent symbol, which implies that the symbol should exist before removal operations.
    - **Symbol Overwriting**: The set method allows overwriting existing symbols in the current table. This behavior assumes that updates to symbols are acceptable and intended, reflecting changes in variable values or function definitions dynamically.

INTERPRETER -

Design Considerations

1. **Visitor Pattern Implementation**:
    - **Dynamic Method Dispatch**: The visit method in the Interpreter class dynamically dispatches the call to the appropriate method based on the node type. This approach leverages Python's dynamic nature to handle different node types without hardcoding specific methods for each type, making the interpreter extensible and easy to maintain.
    - **Error Handling**: The no_visit_method provides a fallback mechanism for cases where no specific visitor method is defined for a node type. This ensures that missing visitor methods are caught early, preventing runtime errors and aiding in debugging.
2. **Evaluation and Execution**:
    - **Expression Evaluation**: The visit_BinOpNode method handles binary operations, distinguishing between different operators and delegating the operation to the appropriate method in the Number class. This modular design allows for clear separation between different operation types and their implementations.
    - **Function Definition and Calls**: The visit_FuncDefNode method creates and stores function definitions in the current context, while the visit_CallNode method executes function calls. This design ensures that function definitions and calls are managed efficiently, supporting features like closures and nested function calls.

Assumptions

1. **Node Types and Tokens**:
    - **Node Structure**: It is assumed that each node type (NumberNode, BinOpNode, VarAccessNode, etc.) has attributes relevant to its function, such as tok for tokens or node for child nodes. The interpreter relies on these attributes to correctly process and evaluate each node.
    - **Token Types**: The visit_BinOpNode method assumes that the operation tokens (op_tok) have a type attribute that specifies the operation (e.g., TT_PLUS, TT_MINUS). The interpreter uses these types to determine which operation to perform.
2. **Context Management**:
    - **Symbol Table**: The visit_VarAccessNode method assumes that variables are stored in the symbol_table of the context and that the table supports operations like get and set. It also assumes that variables are copied when returned, preserving their original state.
    - **Function Execution**: The visit_CallNode method assumes that functions have an execute method that takes a list of arguments and returns a result. This implies that functions are first-class objects in the language and can be dynamically executed with varying arguments.

This design encapsulates functionality for evaluating and executing nodes in the abstract syntax tree, providing a flexible and modular approach to interpretation while handling errors and context-specific operations gracefully.

RUN -

Design Considerations

1. **Modular Design**:
    - **Separation of Concerns**: The run function orchestrates the entire process from token generation to AST parsing and execution. By separating these concerns into distinct components (lexer, parser, interpreter), the design promotes modularity and maintainability. Each component handles a specific aspect of the language processing pipeline, making it easier to test and extend.
    - **Error Handling**: Each stage in the pipeline (token generation, AST parsing, and execution) includes error handling. If an error occurs at any stage, it is returned immediately, halting further processing. This design ensures that errors are managed and reported effectively, preventing propagation of incorrect results.

2. **Global Symbol Table**:
    - **Predefined Symbols**: The global_symbol_table is initialized with predefined symbols (NULL, FALSE, TRUE). This approach provides a base set of constants that are available globally across the program, which can be useful for built-in language features and standard operations.
    - **Context Management**: The global symbol table is assigned to the context used by the interpreter. This design choice allows for easy access to global symbols and ensures that they are available throughout the execution of the program.

Assumptions

1. **Lexer and Parser Behavior**:
    - **Tokenization**: It is assumed that the Lexer class correctly processes the input text and generates a list of tokens. The make_tokens method should either return a tuple of tokens and an error or a boolean True if no tokens are generated, indicating an end-of-file or similar condition.
    - **Parsing**: The Parser class is assumed to correctly parse the list of tokens into an abstract syntax tree (AST). The parse method should return an AST node and an error if parsing fails.
2. **Interpreter Execution**:
    - **AST Nodes**: It is assumed that the Interpreter can handle the AST nodes produced by the parser. The visit method processes these nodes and evaluates the program based on the context provided.
    - **Context and Symbol Table**: The Context class and SymbolTable are assumed to manage symbol resolution correctly. The global symbol table should be accessible and modifiable during the execution of the program, allowing for correct interpretation of global variables and functions.