### **Design Considerations and Assumptions for the Language's Runtime and Values Section**

#### **1. Runtime Result Handling (RTResult Class)**

* **Consideration:** The runtime result class is designed to track the outcome of any operation in the language, whether it be a success or failure.
* **Assumptions:**
  + All operations might result in an error, and this needs to be captured and propagated through the system.
  + The register method assumes that the operations it is tracking are part of a larger sequence, where errors need to be passed upwards if encountered.

#### **2. Value Representation (Value and Number Classes)**

* **Consideration:** The Value class serves as the base class for all types of values in the language, allowing for flexibility in handling different types. The Number class, derived from Value, handles numerical operations specifically.
* **Assumptions:**
  + All values share common behaviors, such as being able to set their position in the source code and their context.
  + The Number class is expected to handle basic arithmetic and comparison operations, with the assumption that these operations are always between numbers.
  + Division by zero is a handled case, throwing a specific runtime error.

#### **3. Function Handling (Function Class)**

* **Consideration:** The Function class is designed to encapsulate the behavior and execution of functions within the language, including handling of arguments and context.
* **Assumptions:**
  + Functions can have a name or be anonymous (e.g., lambdas).
  + The context in which a function is defined is crucial for resolving variables and symbols during execution.
  + The number of arguments provided during a function call must match the number of arguments expected by the function.

#### **4. Context Management (Context Class)**

* **Consideration:** The Context class is crucial for maintaining the scope and variable bindings during the execution of code blocks.
* **Assumptions:**
  + Each code block (like a function or conditional) may have its own context, which can inherit from a parent context.
  + The context holds a symbol table, ensuring that variable lookups and assignments are properly scoped.

#### **5. Interpreter Design (Interpreter Class)**

* **Consideration:** The interpreter is designed to visit nodes of the abstract syntax tree (AST) and execute the corresponding operations.
* **Assumptions:**
  + Each node type in the AST corresponds to a specific visit method in the interpreter, ensuring modular and clear handling of different language constructs.
  + Short-circuiting is assumed to be the correct behavior for logical AND and OR operations, optimizing performance and preventing unnecessary evaluations.

#### **6. Symbol Table Management (SymbolTable Class)**

* **Consideration:** The symbol table class is designed to handle variable storage and lookup, ensuring that variables are correctly scoped and managed.
* **Assumptions:**
  + Variables can be nested in different scopes, and the symbol table needs to handle this hierarchical structure.
  + The global symbol table holds default values like NULL, TRUE, and FALSE, ensuring that these are always available in the language environment.

#### **7. Error Handling**

* **Consideration:** Errors are expected and should be gracefully handled, providing meaningful messages to help in debugging.
* **Assumptions:**
  + Errors have a clear association with positions in the source code, aiding in pinpointing issues.
  + The system should fail early and provide as much context as possible to the developer.

#### **8. Execution Flow (run Function)**

* **Consideration:** The run function orchestrates the process of lexing, parsing, and interpreting the code.
* **Assumptions:**
  + The process is linear, with each stage dependent on the successful completion of the previous one.
  + Global symbols and context are set up before execution begins, ensuring that all operations have the necessary environment to run.

#### **Lexer**

**Considerations:**

* The lexer is responsible for converting raw source code into a sequence of tokens. This process involves recognizing keywords, identifiers, operators, and literals.
* The lexer handles whitespace, comments, and other non-essential characters by ignoring them unless they are contextually significant.
* Error detection is minimal, focused primarily on unrecognized characters or incomplete tokens.

**Assumptions:**

* The input source code adheres to the language's lexical structure.
* All valid tokens are distinguishable through patterns or regular expressions.
* Whitespace is only relevant when separating tokens and does not affect the structure of the code.

#### **Parser**

**Considerations:**

* The parser converts the list of tokens produced by the lexer into an Abstract Syntax Tree (AST), reflecting the hierarchical structure of the code.
* It uses a recursive descent parsing method, with each grammar rule implemented as a corresponding method in the parser class.
* The parser prioritizes clear error reporting to help users identify and correct syntax errors early.

**Assumptions:**

* The input tokens are correctly ordered and formed according to the grammar of the language.
* The parser operates under the assumption that the language grammar is context-free, allowing straightforward parsing without ambiguity.
* The AST nodes accurately represent all constructs of the language, enabling seamless traversal during interpretation.

#### **Nodes**

**Considerations:**

* Each node in the AST corresponds to a specific element of the language, such as expressions, operations, or function calls.
* Nodes are designed to be immutable, ensuring that the tree structure remains consistent throughout interpretation.
* Each node class includes methods for execution, making it easy to evaluate the tree during runtime.

**Assumptions:**

* The structure and behavior of nodes directly reflect the language's design, ensuring that they cover all possible constructs.
* Nodes are context-aware, carrying necessary information like position and context to facilitate error reporting and debugging
* The AST is complete and accurately represents the entire source code, enabling efficient and correct interpretation.

#### **Position**

**Considerations:**

* The Position class tracks the location of each token within the source code, including the line number, column number, and the index in the overall text.
* Position tracking is essential for precise error reporting, allowing users to quickly locate and fix issues in their code.
* The position data is attached to tokens and nodes to maintain context throughout the lexing, parsing, and interpreting stages.

**Assumptions:**

* Source code is read sequentially from start to finish, making it possible to increment the position linearly.
* The position information is sufficient to uniquely identify any token or node in the source code.
* The position is primarily used for error reporting and does not influence the logic of parsing or interpreting.

#### **Tokens**

**Considerations:**

* Tokens are the smallest meaningful units of the source code, classified by their type (e.g., identifier, keyword, operator, number).
* Each token stores its type, value, and the position where it was found, making it easy to reconstruct the source code or provide detailed error messages

.

* Token types are designed to cover all the constructs of the language, ensuring that every part of the source code can be tokenized.

**Assumptions:**

* The source code is free of any ambiguities that could lead to a token being misclassified.
* Each token type has a clear and distinct role in the language, ensuring that the parser can unambiguously interpret them.
* Tokens are immutable once created, providing stability during the parsing and interpretation processes.