**1. Constants**

**Design Considerations:**

* **Consistency**: Constants should be used to represent fixed values throughout the language to avoid magic numbers and string literals. This ensures consistency across the codebase.
* **Ease of Maintenance**: Centralizing constants makes it easier to update or change values without affecting multiple parts of the codebase.

**Assumptions:**

* Constants will primarily be used for error messages, token types, and other fixed values that do not change during execution.
* These constants are defined at the beginning of the implementation file or in a separate module.

**2. Errors**

**Design Considerations:**

* **Clear Communication**: Errors should provide clear, concise, and informative messages to help developers quickly identify and resolve issues.
* **Categorization**: Errors should be categorized (e.g., syntax errors, runtime errors) to differentiate between issues encountered at different stages of execution.

**Assumptions:**

* The language assumes that errors will be user-friendly and provide information about the location and nature of the problem.
* Error handling will be implemented at various stages, including during lexing, parsing, and execution.

**3. Position**

**Design Considerations:**

* **Precision**: The position component should precisely track the line number, column number, and index within the source code to assist with error reporting and debugging.
* **Efficiency**: Position tracking should not add significant overhead to the lexing or parsing process.

**Assumptions:**

* Position information will be included in error messages to indicate where in the source code the issue occurred.
* The position is tracked incrementally as the lexer processes each character of the input.

**4. Tokens**

**Design Considerations:**

* **Simplicity**: Tokens should represent the smallest units of meaning in the language, such as keywords, operators, identifiers, and literals.
* **Clarity**: Each token should be clearly distinguishable and carry sufficient information for the parser to process.

**Assumptions:**

* The lexer will generate tokens based on predefined patterns and the tokens will be passed to the parser in a linear sequence.
* Token types will be represented as constants to ensure consistency.

**5. Nodes**

**Design Considerations:**

* **Structure**: Nodes represent the abstract syntax tree (AST) of the program, with each node corresponding to a syntactic construct like expressions, statements, or program blocks.
* **Modularity**: Nodes should be modular, allowing for easy extension of the language with new syntax or features.

**Assumptions:**

* The AST will be traversed by the interpreter or compiler to execute or translate the program.
* Each node will carry information about its type, position, and children.

**6. Parse Result**

**Design Considerations:**

* **Flexibility**: The parse result should accommodate both successful parses and errors, providing detailed feedback about what went wrong if parsing fails.
* **Encapsulation**: Parse results should encapsulate the state of parsing at any given point, allowing the parser to backtrack or recover from errors.

**Assumptions:**

* Parsing will produce a single parse result object that includes the root of the AST and any errors encountered during parsing.

**7. Parser**

**Design Considerations:**

* **Accuracy**: The parser must accurately interpret the sequence of tokens to produce a valid AST.
* **Error Recovery**: The parser should attempt to recover from errors and continue parsing where possible.

**Assumptions:**

* The parser will operate in a top-down manner, with functions to handle specific constructs (e.g., expressions, statements).
* Recursive descent or another parsing strategy will be employed.

**8. Runtime Result**

**Design Considerations:**

* **Consistency**: The runtime result should consistently represent the outcome of executing a piece of code, whether it’s a value, an error, or a control flow signal (like return or break).
* **Clarity**: The result should clearly distinguish between successful execution and runtime errors.

**Assumptions:**

* Runtime results will be passed up through the call stack, allowing higher levels of the interpreter to handle them appropriately.

**9. Values**

**Design Considerations:**

* **Abstraction**: Values should abstract the various data types in the language, such as integers, strings, functions, and objects.
* **Interoperability**: Values should support common operations (e.g., arithmetic, comparison) and interact seamlessly with the interpreter.

**Assumptions:**

* Each value type will be represented by a distinct class or structure with methods for operations specific to that type.
* Type checking and conversion will be handled at runtime.

**10. Context**

**Design Considerations:**

* **Scope Management**: Context should manage variable scopes, keeping track of variables, functions, and other symbols defined in a particular scope.
* **Isolation**: Different execution contexts (e.g., function calls) should be isolated from each other to prevent unintended side effects.

**Assumptions:**

* Each execution context will have its own symbol table and parent context.
* Contexts will be linked to facilitate access to variables in outer scopes.

**11. Interpreter**

**Design Considerations:**

* **Correctness**: The interpreter must correctly execute the AST, respecting the semantics of the language.
* **Efficiency**: The interpreter should be optimized to minimize execution time and resource usage.

**Assumptions:**

* The interpreter will traverse the AST and execute nodes in a depth-first manner.
* Error handling and control flow (e.g., return, break, continue) will be managed by the interpreter.

**12. Symbol Table**

**Design Considerations:**

* **Efficiency**: The symbol table should allow for quick lookups of variable and function names within the current context.
* **Encapsulation**: The symbol table should encapsulate all the symbols within a particular scope and support operations like adding, updating, and deleting symbols.

**Assumptions:**

* Symbol tables will be hierarchical, with each scope having its own table linked to its parent’s table.
* Global symbols will be stored in the root symbol table.

**13. Run**

**Design Considerations:**

* **Entry Point**: The Run component serves as the entry point for executing a program. It must initialize the necessary components (lexer, parser, interpreter) and manage the overall execution flow.
* **Error Handling**: Run should handle any unhandled errors and provide a summary of issues that occurred during execution.

**Assumptions:**

* Run will be responsible for coordinating the components and ensuring that they interact correctly.
* It will provide a user-friendly interface for executing programs, possibly including options for debugging or tracing execution.

**14. Lexer**

**Design Considerations:**

* **Performance**: The lexer should tokenize the input efficiently, minimizing the time spent on character processing.
* **Simplicity**: The lexer’s design should be simple, translating source code into tokens based on predefined patterns or rules.

**Assumptions:**

* The lexer will operate in a linear pass, converting a string of source code into a sequence of tokens.
* Lexical analysis errors will be handled within the lexer, potentially with mechanisms for error recovery or reporting.