**GPP Interpreter Documentation for Lisp**

Overview

The GPP Interpreter is a Lisp-like interpreter designed to process and evaluate custom expressions. This document provides an extensive overview of its functionalities, including defining functions, arithmetic operations, and conditionals.

Key Components

1. Tokenization: Converts input strings into a list of tokens.

2. Parsing: Transforms tokens into an expression tree.

3. Evaluation: Interprets the expression tree and computes the result.

Tokenization (`tokenize` Function)

- Purpose: Splits input into meaningful tokens.

- Implementation: Iterates over each character, grouping them into tokens based on their type (e.g., operators, numbers, parentheses).

Parsing (`parse` Function)

- Purpose: Converts tokens into a nested expression format.

- Implementation: Uses a stack to organize tokens into a tree-like structure, facilitating nested expressions.

Evaluation (`evaluate-expression` Function)

- Purpose: Recursively evaluates the expression tree.

- Implementation: Distinguishes between operators, function calls, and literals to compute the result.

- Supports basic arithmetic operations: `+`, `-`, `\*`, `/`.

- Implements conditional evaluation with `if`.

- Extends functionality to user-defined functions.

Function Definition (`define-function` and `parse-function-definition` Functions)

- Purpose: Enables defining and storing custom functions.

- Usage: `(def functionName param1 param2 (... body ...))`

- Implementation:

- Stores functions in a hash table for later retrieval.

- `define-function`: Stores the function name, parameters, and body.

- `parse-function-definition`: Extracts function details from tokens.

Applying Functions (`apply-function` Function)

- Purpose: Executes user-defined functions with given arguments.

- Implementation: Maps function parameters to provided arguments and evaluates the function body with these bindings.

File Processing (`process-file` Function)

- Purpose: Reads expressions from a file and evaluates them.

- Implementation: Opens a file, reads each line, and processes it using the interpreter.

Test Cases

- Setup: Test cases are written in a file named `test.g++`.

- Execution: `process-file` reads and evaluates each line in the file.

- Cases:

1. Basic arithmetic operations.

2. Function definitions and calls.

3. Nested operations and function calls.

4. Conditional statements.

Error Handling

- Syntax and runtime errors are managed with appropriate messages.

- Includes checks for undefined functions and incorrect operator usage.