1. Project Overview

1.1 Purpose

This project involves developing a syntax analyzer for the G++ programming language, aimed at educational use at Gebze Technical University. The objective is to design an interpreter capable of parsing and analyzing G++ code based on its unique syntax rules.

1.2 Language Characteristics

G++ exhibits the following characteristics:

- Lisp-like syntax
- Interpreted language
- Imperative and non-object-oriented structure
- Static scoping and static binding
- Strong typing
- Native support for precise arithmetic operations

YACC PART

1. Lex File:

The Lex file converts input data into tokens. It defines the basic building blocks of the language, such as keywords, operators, and variable names.

- Token Definitions: Regular expressions (regex) are used to identify keywords (true, false, and, or), operators (+, -, /, *), and custom definitions (VALUEF, IDENTIFIER).
- Syntax Errors: If an unrecognized input is encountered, error messages are generated.

2. Yacc File:

The Yacc file specifies how the tokens defined in Lex are processed based on the grammar of the language. It implements the logic for handling the defined rules.

- Function Definitions: User-defined functions are stored in a list and accessed when needed.
- Fraction Operations: Functions like addFraction and subtractFraction handle basic arithmetic operations (addition, subtraction, multiplication, division) for fractions.
- Operation Results: Each operation creates an OperationOutput structure that tracks the operation status (e.g., success, division by zero error).
-

3. Data Structures:

Custom data structures are defined to support the operations:

- Fraction: Represents fractions (numerator and denominator).
- OperationOutput: Contains the results and error messages for operations.
- Numbers: Handles parsing and processing of values in the custom VALUEF format.
- Functions and Variables: Store user-defined functions and variables.
  4. Helper Functions:
- parseValuef: Parses inputs in the VALUEF format and processes their numeric values.
- fractionToString: Converts fraction structures into string format.
- greatest_common_divisor: Calculates the GCD for simplifying fractions.
  Project Functionality
  This project provides a robust foundation for mathematical operations and user-defined functionality. Key features include:

1. **Performing Mathematical Operations with Operators:**
   - Supports arithmetic operations (addition, subtraction, multiplication, division) with fractions.
   - For example, a fraction in the format 3b5 is processed accurately and simplified when needed.
2. **Defining Variables and Functions:**
   - User-defined functions and variables are stored in memory for later use.

LİSP PART

Overview

This document explains the design, structure, and functionality of the GPP Interpreter, which is a Lisp-based implementation capable of processing input files, parsing tokens, and interpreting commands or expressions. The interpreter supports constructs such as arithmetic operations, function definitions, loops, and conditional expressions.

Key Components

Data Structures

defstruct structure_of_function

This structure is used to define functions in the interpreter. It includes the following fields:

- name: The name of the function.
- operator: The operator used in the function (e.g., arithmetic operations).
- operand1 and operand2: Operands involved in the operation.
- argument-number: Number of arguments required by the function.

Constructors:

1. make-structure_of_function: Default constructor.
2. make-structure_of_function (name operator operand1 operand2 argument-number): Allows specifying all fields at initialization.

hashmap_of_function

A hash table used to store function definitions, where keys are function names and values are corresponding structure_of_function objects.

terminate-program

A global flag to indicate whether the program should terminate.

Core Functions

gppinterpreter (&optional file-names)

The main entry point for the interpreter. It processes input either from files or directly from the command line and writes outputs to output.txt.

Workflow:

1. Opens output.txt for writing results.
2. Defines an inner function process-input to handle token processing using parser.
3. If file-names is provided, iterates over the files and processes them. Otherwise, processes direct input from gpp-lexer.

parser (tokens out-stream)

Processes tokens and generates output based on the interpreter's grammar.

Workflow:

1. Defines recursive-parse as a local function for iterative parsing.
2. Calls the START non-terminal function to evaluate the input.
3. Outputs results or errors to both the console and the out-stream.

START (tokens)

Evaluates the main entry point of the grammar by checking for one of the following:

- EXIT-PROGRAM: Handles program termination.
- EXPR: Evaluates expressions like arithmetic operations, identifiers, or function calls.
- FUNCTION_DEFINITION: Parses and stores new function definitions.

Grammar Functions

EXIT-PROGRAM (tokens)

Handles the EXIT command by setting the global flag *terminate-program* and returning a special value ("$_").

EXPR (tokens)

Evaluates expressions, including:

1. Arithmetic expressions (ARITHMETIC).
2. Identifiers (IDENTIFIER).

3. Literal values (VALUEF).
4. Function calls (FUNCTION-CALL).
5. Loop constructs (FOR-EXPRESSION and WHILE-EXPRESSION).

FOR-EXPRESSION (tokens)

Processes FOR loops with the following steps:

1. Parses loop variable, start, and end expressions.
2. Evaluates the loop body for each iteration.
3. Returns the accumulated result of the loop.

WHILE-EXPRESSION (tokens)

Processes WHILE loops:

1. Parses the loop condition and body.
2. Continuously evaluates the body as long as the condition holds true.

Auxiliary Functions

evaluate-res (value tokens)

A utility function to package a result value with the remaining tokens.

next-token (tokens)

Retrieves the next token from the input.

evaluate-valuef (operator operand1 operand2)

Performs arithmetic or logical operations based on the given operator and operands.

File Structure

- output.txt: Stores results generated by the interpreter.
- gpp-lexer: Assumed to tokenize input for processing.

Error Handling

Errors such as syntax mismatches or invalid tokens are handled within the parser function by outputting "Syntax Error!" to both the console and the output file. Additionally, each grammar function ensures graceful failure and returns a nil result upon encountering invalid input.