Section 1 - Project Description

1.1 Project

Project Troy, developed by team Argonauts:

Bo Thompson, Rakeen Mazumder, Joshua Lozano, Andrew Porter

Developed under the instruction of Professor Robert Bruce

Sonoma State University, 2024.

1.2 Description

Interpreter for custom C-like language as defined in Backus–Naur form.

1.3 Revision History

Date	Comment
11/05/2024	Beginning of documentation
11/13/2024	Function Dictionary added
11/19/2024	Development Decisions added
11/29/2024	Documentation updated
12/5/2024	Documentation finalized

Contents

Section 2 - Overview

2.1 Purpose

To create an interpreter for the C-like programming language, using C++. This interpreter is designed to read, parse, and evaluate C-like code, providing a foundational tool for understanding and simulating program execution within the language. The project involves several components completed in a linear succession.

2.2 Scope

To develop a fully functional interpreter for the C-like programming language, implemented in C++. This interpreter covers several core functionalities of a C compiler, including:

- **Lexical Analysis:** Using Scanner and SourceReader, the interpreter breaks down input source code into tokens, identifying different elements like keywords, operators, and identifiers.

- **Syntax Parsing:** A parser for building an abstract syntax tree (AST) from the tokenized input, which helps in structuring code and managing syntax rules.

- **Symbol Management:** A structure to manage variable and function names, their scopes, types, and other attributes.

- **Expression Evaluation:** An interpreter to evaluate postfix expressions, allowing it to handle calculations or logical operations within the source code.

2.3 Objectives

Gain a comprehensive understanding of the core functions of programming languages, specifically by exploring their syntactic, semantic, and implementation features. Through building a C interpreter, this project allows us to examine key aspects of functional, procedural, and object-oriented paradigms within the context of C, a foundational procedural language. Additionally, we aim to deepen our understanding of:

- **Syntactic Structures:** By implementing a parser, we analyze the rules and structure that define C's syntax, allowing us to see firsthand how programming languages enforce grammar and structure.

- **Semantic Analysis:** Through symbol table management and error handling, we explore the semantic rules that ensure variable declarations, data types, and scope are respected within code, providing insight into how languages maintain logical consistency.

- **Execution and Evaluation:** Building an evaluator for expressions introduces us to language implementation techniques, showing us how procedural languages execute statements and evaluate expressions in real time.

This project also gives us practical experience with lexical analysis, error handling, and symbol management—key components of language processing systems. Overall, our work provides a hands-on survey of programming language concepts across various paradigms, enhancing our ability to understand and implement fundamental language features.

2.4 Requirements

- **Ignore Comments:**

Utilize a procedurally-driven deterministic finite state automoton (DFA) to identify comments. Table-driven DFA solutions will have reduced scores (per the grading rubric) since such solutions are typically automatically generated by compiler tools such as Yacc, Lex, Bison, etc.

- **Tokenization:**

Write a program in C or C++ that will identify and remove comments from an input test file using a deterministic finite state automoton (DFA) then use a DFA to convert the input file into a series of tokens. Display the tokens as output (if no syntax errors occurred) or an error message instead.

- **Recursive Descent Parser:**

Write a program in C or C++ that creates a concrete syntax tree (CST) using a recursive descent parsing technique, using procedurally-driven deterministic finite state automotons (DFA). No table-driven DFAs should be implemented. Furthermore, utilize an LCRS binary tree (Left-Child, Right-Sibling) to store the CST. Display the resulting CST in breadth-first order.

- **Symbol Table:**

Write a program in C or C++ that creates a symbol table (a linked list) of all the defined variables (including their type and scope) and the names of all functions and procedures. Functions and procedures may also have an input parameter list of variables and types. These too should be added (with appropriate scope) to the symbol table. Lastly, functions have a return datatype which must be noted in the symbol table as well.

- **Abstract Syntax Tree:**

Write a program in C or C++ that creates an Abstract Syntax Tree (AST) based on the Concrete Syntax Tree (CST) you created in programming assignment 3. An Abstract Syntax Tree is not a clone of a Concrete Syntax Tree. Utilize an LCRS binary tree (Left-Child, Right-Sibling) to store your AST. Display the resulting AST in breadth-first order. You may output your AST as text to the screen as long as I can identify adjacent children and adjacent siblings in the tree.

- **Interpreter:**

Write a program in C or C++ that interprets programs written in our C-like programming language defined in Backus-Naur Form (BNF)

Section 3 - Development

3.1 Deadlines

Description	Deadline
Ignore Comments	9/12/2024
Tokenization	9/24/2024
Recursive Descent Parser	10/17/2024
Symbol Table	10/29/2024
Abstract Syntax Tree	11/07/2024
Interpreter	12/10/2024

3.2 Development Methodologies

Our team employed a Feature-Driven Development (FDD) methodology. This facilitated a structured approach, enabling us to divide the project into small, manageable features. Each feature was designed, built, and tested within a series of two-week iterations, allowing the team to review progress, incorporate feedback, and adjust priorities based on the instructor's timetable.

Development Phases: Planning and Feature Identification: At the start of each iteration, we identified critical features—such as lexical analysis, syntactic parsing, symbol management, and expression evaluation—defining the deliverables for the upcoming iteration.

Design and Build by Feature: Assignments are structured around these features, allowing team members to focus on distinct, functionally cohesive elements. This approach also helped in modularizing the project, making each component independent yet integrable with the overall system for each deliverable.

Testing and Integration: As features were completed, they were tested individually and then integrated with existing components. This ensured each iteration delivered a stable, working version of the interpreter with incremental improvements.

Review and Adjustment: After each iteration, we conducted reviews to assess progress, gather feedback, and make adjustments as needed. This approach allowed us to refine our implementation based on real-time evaluations and align our work with project goals.

3.3 Version Control

Version control for the project was managed through a centralized GitHub repository. Initially, the main repository served as the base, with individual team members creating separate branches to design and implement specific features. Each feature was developed and tested within its branch, ensuring that experimental or in-progress code did not disrupt the main codebase. Once a feature was complete and verified, it was merged back into the main repository through a pull request.

However, as the project progressed, we encountered merge conflicts due to file discrepancies between branches. These conflicts arose from concurrent updates to shared files, highlighting the need for a more synchronized workflow. To address this, we adopted the following improvements:

- **Regular Synchronization:** Team members pulled changes from the main repository into their feature branches more frequently, reducing discrepancies and minimizing conflicts.

- **Clear Branching Protocols:** We established guidelines on branch naming, structure, and update frequency, which helped maintain consistency across development efforts.

- **Collaborative Conflict Resolution:** When conflicts did arise, team members collaborated during the merge process, reviewing and resolving issues collectively to ensure code integrity.

By refining our version control practices, we improved collaboration efficiency and maintained a stable codebase, enabling smoother integration of features into the main repository.

Section 4 - System Architecture

4.1 Structure

- **Main Controller (`main.cpp`)**:

Serves as the program's entry point, handling initial configurations and file inputs. This file coordinates the flow between other modules, initializing components such as the scanner and parser, and directing them to process the input C code.

- **Source Reading Layer (`SourceReader.cpp`)**:

Manages input file handling, reading source code character-by-character or line-by-line, and includes methods to manage input flow, such as "ungetting" strings. It also handles comment stripping and newline tracking, allowing downstream components to receive a clean source stream.

- **Lexical Analysis Layer (`Scanner.cpp` and `StringParser.cpp`)**:

The `Scanner` component tokenizes the input, converting sequences of characters into tokens that represent language constructs (e.g., keywords, operators, and identifiers). It uses a state machine approach with macros for efficiency. The `StringParser` assists with string-specific parsing, handling escape sequences and multi-character tokens.

- **Syntactic Analysis Layer (`DescentParser.cpp`)**:

Implements a recursive descent parser to analyze the sequence of tokens and construct an abstract syntax tree (AST). The parser uses `CodeNode` structures to represent nodes in the parse tree, facilitating hierarchical organization of parsed elements such as expressions and statements. This component plays a key role in translating tokenized input into a structured representation that can be evaluated or interpreted.

- **Symbol Table Management (`SymbolTable.cpp`)**:

Maintains a symbol table for storing information about identifiers (e.g., variables, functions, and procedures), along with their types, scopes, and array properties. This structure supports semantic analysis, ensuring variables are correctly declared and accessible within their scopes.

- **Expression Evaluation (`PostFixEvaluator.cpp`)**:

Evaluates expressions in postfix notation, managing precedence and associativity for operations. This evaluator is essential for interpreting mathematical and logical expressions, as it can handle complex expressions with nested parentheses and operator hierarchies.

- **Hierarchical Code Representation** (`CodeNode.cpp` and `CodeNode.hpp`):

The `CodeNode` and `SyntaxTree` classes handle the hierarchical representation of the program's code structure. The `CodeNode` represents individual nodes, such as expressions, assignments, and declarations, while the `SyntaxTree` manages the overall structure. These components facilitate relationships between nodes (e.g., siblings and children) and provide methods for traversal, debugging, and evaluation. The `getPrintableNodeType` function ensures node types are easily readable.

- **Symbol Management Layer** (`SymbolTable.cpp` and `SymbolTable.hpp`):

The `SymbolTable` system organizes and manages symbols such as variables and functions within the interpreter. The `CodeScope` class represents individual scopes, supporting nested structures for functions and blocks. The `Symbol` class stores attributes for identifiers, including types, values, and array information. Key features include methods for adding symbols, creating subscopes, validating against reserved keywords, and maintaining type consistency in compliance with the BNF grammar.

- **Utility Layer** (`Tools.cpp` and `Tools.hpp`):

The `Tools` module provides general-purpose utility functions used throughout the interpreter. For example, the `getLowercase` function standardizes string formatting by converting all characters to lowercase. These tools enhance code maintainability by centralizing common operations.

4.2 Function Dictionary

1.0 CodeNode:

Constructor:

1.1 CodeNode(const Token &t)
Description: Initializes a CodeNode with a specified token, assigning it to `token` data member (1.9).
Parameters:
- const Token &t: Token for initializing the node.
Return Type: void
Uses Data: 1.9 (token)

Methods:

1.2 getToken()
Description: Retrieves the token stored in `token` (1.9) for this CodeNode.
Parameters: None
Return Type: Token*
Uses Data: 1.9 (token)

1.3 setToken(Token* t)
Description: Sets the token in `token` (1.9) for this CodeNode using a pointer.
Parameters:
- Token* t: Pointer to the token to set.
Return Type: void
Uses Data: 1.9 (token)

1.4 setToken(const Token& t)
Description: Sets the token in `token` (1.9) for this CodeNode by copying a given token.
Parameters:
- const Token& t: Token to copy and assign.
Return Type: void
Uses Data: 1.9 (token)

1.5 getChild()
Description: Returns the child node stored in `child` (1.10).
Parameters: None
Return Type: CodeNode*
Uses Data: 1.10 (child)

1.6 getSibling()
Description: Returns the sibling node stored in `sibling` (1.11).
Parameters: None
Return Type: CodeNode*
Uses Data: 1.11 (sibling)

1.7 setChild(CodeNode *c)
Description: Assigns a child node to `child` (1.10) in this CodeNode.
Parameters:
- CodeNode *c: Pointer to the child node.
Return Type: void
Uses Data: 1.10 (child)

1.8 setSibling(CodeNode *s)
Description: Assigns a sibling node to `sibling` (1.11) in this CodeNode.
Parameters:
- CodeNode *s: Pointer to the sibling node.
Return Type: void
Uses Data: 1.11 (sibling)

Data:

1.9 name: token
type: Token
purpose: Stores the token associated with this CodeNode.
used by: 1.1, 1.2, 1.3, 1.4

1.10 name: child
type: CodeNode*
purpose: Holds a pointer to the child node of this CodeNode.
used by: 1.5, 1.7

1.11 name: sibling
type: CodeNode*
purpose: Holds a pointer to the sibling node of this CodeNode.
used by: 1.6, 1.8

2.0 DescentParser:

Constructors:

2.1 DescentParser()
Description: Initializes an empty DescentParser instance.
Parameters: None
Return Type: void

2.2 DescentParser(const string &fname)
Description: Initializes DescentParser with `filename` (2.6).
Parameters:
- const string &fname: Filename for parser initialization.
Return Type: void
Uses Data: 2.6 (filename)

Methods:

2.3 parse(const string&, SyntaxTree*&, CodeScope*&)
Description: Parses input data into a syntax tree using `scope` (2.7).
Parameters:
- const string&: Input data.
- SyntaxTree*&: Reference to syntax tree.
- CodeScope*&: Reference to code scope.
Return Type: void
Uses Data: 2.6 (filename), 2.7 (scope)

2.4 getParameters(list<Token>::iterator&, CodeScope*&)
Description: Retrieves parameters within the current `scope` (2.7).
Parameters:
- list<Token>::iterator&: Iterator over tokens.
- CodeScope*&: Reference to the code scope.
Return Type: void
Uses Data: 2.7 (scope)

2.5 handleArrayDeclaration(list<Token>::iterator& t, Symbol* s)
Description: Handles declarations of arrays in the current `scope` (2.7).
Parameters:
- list<Token>::iterator& t: Iterator to a token.
- Symbol* s: Pointer to the symbol representing the array.
Return Type: void
Uses Data: 2.7 (scope)

Data:

2.6 name: filename
type: string
purpose: Stores the name of the file being parsed.
used by: 2.2, 2.3

2.7 name: scope
type: CodeScope*
purpose: Manages the current scope during parsing.
used by: 2.3, 2.4, 2.5

3.0 PostFixEvaluator:

Constructor:

3.1 PostFixEvaluator()
Description: Initializes an instance of PostFixEvaluator, setting up `PostfixEquation` (3.4) and `Stack` (3.5).
Parameters: None
Return Type: void
Uses Data: 3.4 (PostfixEquation), 3.5 (Stack)

Methods:

3.2 Eval(Token Symbol)
Description: Evaluates a postfix expression using `Stack` (3.5) and `Precedence` (3.6).
Parameters:
- Token Symbol: Token used in the postfix expression.
Return Type: bool
Uses Data: 3.4 (PostfixEquation), 3.5 (Stack), 3.6 (Precedence)

3.3 getEquation()
Description: Returns the equation stored in `PostfixEquation` (3.4).
Parameters: None
Return Type: vector<Token>
Uses Data: 3.4 (PostfixEquation)

Data:

3.4 name: PostfixEquation
type: vector<Token>
purpose: Holds the equation in postfix form.
used by: 3.1, 3.3

3.5 name: Stack
type: stack<Token>
purpose: Temporarily stores tokens during evaluation.
used by: 3.1, 3.2

3.6 name: Precedence
type: map<string, int>
purpose: Defines precedence levels for operators.
used by: 3.2

4.0 Scanner:

Constructors:

4.1 Scanner()
Description: Initializes an instance of Scanner with an empty `tokens` list (4.7).
Parameters: None
Return Type: void
Uses Data: 4.7 (tokens)

4.2 Scanner(const string &fname)
Description: Initializes Scanner with `filename` (4.8) and opens it for reading.
Parameters:
- const string &fname: Filename to open.
Return Type: void
Uses Data: 4.7 (tokens), 4.8 (filename)

Methods:

4.3 open(const string &fname)
Description: Opens a file with `filename` (4.8) in Scanner.
Parameters:
- const string &fname: Filename to open.
Return Type: void
Uses Data: 4.8 (filename)

4.4 error(string msg)
Description: Outputs an error message, referencing `tokens` (4.7).
Parameters:
- string msg: Error message to display.
Return Type: void
Uses Data: 4.7 (tokens)

4.5 getTokens()
Description: Returns the list of tokens stored in `tokens` (4.7).
Parameters: None
Return Type: list<Token>
Uses Data: 4.7 (tokens)

4.6 getReadableTokenType(TokenType t)
Description: Converts a TokenType to a readable string.
Parameters:
- TokenType t: Token type to convert.
Return Type: string

Data:

4.7 name: tokens
type: list<Token>
purpose: Stores the list of tokens scanned from the source.
used by: 4.1, 4.2, 4.4, 4.5

4.8 name: current_token
type: Token
purpose: Holds the current token being processed.
used by: 4.5, 4.6

5.0 SourceReader:

Constructors:

5.1 SourceReader()
Description: Initializes an instance of SourceReader, setting up `line_count` (5.9) and `pos` (5.10).
Parameters: None
Return Type: void
Uses Data: 5.9 (line_count), 5.10 (pos)

5.2 SourceReader(const std::string &filename)
Description: Initializes SourceReader with `filename` (5.8).
Parameters:
- const std::string &filename: Filename to open.
Return Type: void
Uses Data: 5.8 (filename)

Methods:

5.3 open(const std::string &filename)
Description: Opens a file with `filename` (5.8).
Parameters:
- const std::string &filename: Filename to open.
Return Type: void
Uses Data: 5.8 (filename)

5.4 unget(const std::string &str)
Description: Pushes a string back into the source for re-processing.
Parameters:
- const std::string &str: The string to push back.
Return Type: void
Uses Data: 5.8 (filename)

5.5 processSource(char &c)
Description: Processes the next character, updating `line_count` (5.9) and `pos` (5.10).
Parameters:
- char &c: Character to process.
Return Type: bool
Uses Data: 5.9 (line_count), 5.10 (pos)

Data:

5.8 name: filename
type: string
purpose: Stores the name of the source file being read.
used by: 5.2, 5.3, 5.4

5.9 name: line_count
type: int
purpose: Tracks the current line number in the source file.
used by: 5.1, 5.5

5.10 name: pos
type: int
purpose: Tracks the current character position within the line.
used by: 5.1, 5.5

6.0 StringParser:

Constructor:

6.1 StringParser()
Description: Initializes an instance of StringParser, setting `parsing` (6.3) to active.
Parameters: None
Return Type: void
Uses Data: 6.3 (parsing)

Methods:

6.2 parse(char &c)
Description: Parses a character if `parsing` (6.3) is true.
Parameters:
- char &c: Character to parse.
Return Type: bool
Uses Data: 6.3 (parsing)

Data:

6.3 name: parsing
type: bool
purpose: Indicates if the parser is currently active.
used by: 6.1, 6.2

7.0 SymbolTable:

Methods:

7.1 print(bool printITypeLine)
Description: Prints symbol information stored in `symbols` (7.4) and `parameters` (7.5).
Parameters:
- bool printITypeLine: Flag to include type line.
Return Type: void
Uses Data: 7.4 (symbols), 7.5 (parameters)

7.2 getReadableSymbolType(const SymbolType&)
Description: Converts a SymbolType to a readable string.
Parameters:
- const SymbolType&: Symbol type to convert.
Return Type: string

Data:

7.4 name: symbols
type: vector<Symbol*>
purpose: Stores symbols within the current symbol table.
used by: 7.1

7.5 name: parameters
type: vector<Symbol*>
purpose: Stores parameter symbols.
used by: 7.1

8.0 Tools:

Methods:

8.1 getLowercase(const string &s)
Description: Converts the provided string to lowercase, returning `lowercase_string` (8.2).
Parameters:
- const string &s: The string to convert to lowercase.
Return Type: string
Uses Data: 8.2 (lowercase_string)

Data:

8.2 name: lowercase_string
type: string
purpose: Holds a lowercase version of a given string.
used by: 8.1

9.0 CodeNode:

Methods:

9.0 CodeNode(const Token &t)
Description: Initializes a `CodeNode` with a specified token, assigning it to the `token` data member (9.9).
Parameters:
- const Token &t: Token for initializing the node.
Return Type: void
Uses Data: 9.9 (token)

9.1 ~CodeNode()
Description: Destructor for the `CodeNode` class, ensuring that dynamically allocated sibling and child nodes are deleted.
Parameters: None
Return Type: void
Uses Data: 9.3 (sibling), 9.4 (child)

9.2 setSibling(CodeNode *c)
Description: Assigns a sibling to the current `CodeNode`. Validates that the sibling is not already assigned to prevent overwrites.
Parameters:
- CodeNode *c: Pointer to the sibling node.
Return Type: void
Uses Data: 9.3 (sibling)

9.3 setChild(CodeNode *c)

Description: Assigns a child to the current `CodeNode`. Validates that the child is not already assigned to prevent overwrites.
Parameters:
- CodeNode *c: Pointer to the child node.
Return Type: void
Uses Data: 9.4 (child)

9.4 getSibling()
Description: Retrieves the sibling of the current `CodeNode`.
Parameters: None
Return Type: CodeNode*
Uses Data: 9.3 (sibling)

9.5 getChild()
Description: Retrieves the child of the current `CodeNode`.
Parameters: None
Return Type: CodeNode*
Uses Data: 9.4 (child)

9.6 setExpression(vector<Token> e)
Description: Assigns a vector of tokens to represent the expression of the current `CodeNode`.
Parameters:
- vector<Token> e: Vector of tokens representing the expression.
Return Type: void
Uses Data: 9.10 (expression)

9.7 getExpression()
Description: Retrieves the vector of tokens representing the expression of the current `CodeNode`.
Parameters: None
Return Type: vector<Token>
Uses Data: 9.10 (expression)

9.8 setType(NodeType t)
Description: Sets the type of the `CodeNode`. Ensures the type is not already set to prevent overwrites.
Parameters:
- NodeType t: Enum value representing the type of the node.
Return Type: void
Uses Data: 9.11 (type)

9.9 getType()
Description: Retrieves the type of the `CodeNode`.
Parameters: None
Return Type: NodeType
Uses Data: 9.11 (type)

Section 5 - Development Decisions

5.1 Version Control Issues

Version Control Issue: 10/29/2024

Issue Overview:

A merge from the symboltable branch into the main branch introduced significant merge conflicts due to discrepancies in file changes. Initially, a force merge was performed under the assumption that only new files were being introduced to the main branch. Unfortunately, this assumption proved incorrect, resulting in unintended changes to other files and causing instability in the codebase.

After the force merge was reverted to restore the stability of the main branch, the symboltable branch became outdated and could no longer be merged directly. This occurred because the main branch had commits ahead of the symboltable branch as a result of the reverted merge.

Resolution Steps:

- Reverting the Main Branch: The changes introduced by the failed force merge were removed, restoring the integrity of the main branch.
- Creating a Fork: To resolve the divergence, a fork of the main branch was created. This allowed the team to safely reconcile the branches without affecting the stable main branch.
- Updating symboltable: The symboltable branch was synchronized with the corrected main branch, ensuring compatibility for future merges.

Lessons Learned:

- Avoid Force Merges: Force merging without thorough review can lead to unexpected and widespread issues.
- Frequent Synchronization: Regularly updating feature branches with changes from the main branch helps to minimize the risk of merge conflicts.
- Improved Branching Protocols: Establishing and adhering to clear protocols for branch management ensures a smoother version control process.

5.2 Instruction Changes

Combining the Concrete and Abstract Syntax Tree

Overview:
The project instructions mandated separate implementations for the Concrete Syntax Tree (CST) and Abstract Syntax Tree (AST). However, during the development process, we identified significant overlap between the two structures. Given the time and resource constraints, maintaining separate implementations for both trees was deemed inefficient.

Challenges:

- **Structural Redundancy**: The CST and AST shared numerous functionalities and data, leading to unnecessary duplication of logic and increased complexity in managing both structures.
- **Design Philosophy**: The instructions implied that keeping the CST and AST separate was essential for clarity and modularity. This presented a challenge when considering a unified approach.
- **Refactoring Effort**: Modifying the existing CST to incorporate AST functionality required extensive refactoring, which was initially thought to be a prohibitive obstacle.

Decision and Resolution:
After careful analysis, we opted to merge the CST and AST into a unified syntax tree, leveraging the existing CST structure as a foundation. By refactoring the CST code, we repurposed it to fulfill the requirements of both trees. This involved:

- Updating the parser to generate an enriched CST with semantic information required for evaluation.
- Modifying tree traversal methods to support abstract-level operations without sacrificing the CST's original capabilities.

Outcome:
This decision resulted in a more efficient and streamlined implementation, reducing code redundancy and improving maintainability. Testing confirmed that the unified syntax tree met all project requirements, validating our approach. The experience also highlighted the importance of critically evaluating design assumptions and adapting to practical constraints.

Combining Tokenization Rules for Quotes in Strings

Overview:
The original project instructions allowed for separate handling of strings bounded by single (`'`) and double (`"`) quotes. During implementation, we discovered discrepancies in how these strings were parsed, particularly when unescaped quotes appeared inside the string.

Challenges:

- **Ambiguity in Tokenization**: The grammar suggested that strings could contain both single and double quotes without escaping, leading to ambiguity during token parsing.
- **Behavioral Differences**: Tests revealed that unescaped quotes caused parsing failures, conflicting with the expected behavior defined in the instructions.

Decision and Resolution: We adjusted the tokenization logic to enforce stricter rules for quote handling, requiring all inner quotes to be escaped regardless of their type. This decision aligned our implementation with standard practices in similar programming languages, reducing ambiguity and ensuring predictable behavior during parsing.

Outcome:
While this adjustment deviated from the instructions, it simplified the parsing process and ensured compatibility with all test cases.

Redefining Syntax Tree Construction for Arrays

Overview:
In the original design, array expressions were to be parsed into the Abstract Syntax Tree (AST) with specialized handling for indexing operations. However, issues arose during implementation regarding the representation of brackets (`[]`) in postfix notation.

Challenges:

- **Unclear Indexing Rules**: It was unclear how array indexing should be represented in the syntax tree, leading to inconsistencies during evaluation.
- **Postfix Notation Limitations**: Brackets used for array access did not integrate smoothly into the postfix representation, causing errors during tree traversal.

Decision and Resolution: We modified the evaluator to include brackets as explicit tokens in postfix notation. This ensured that expressions like `a[10 + i]` were correctly parsed and represented in the tree. Additionally, the parser was updated to handle these tokens as part of the syntax tree construction.

Outcome:
This change enhanced the evaluator's ability to process array expressions, maintaining clarity and consistency in tree traversal logic.

Refactoring for Local Scope Handling

Overview:
The original instructions emphasized strict separation between local and global variable scopes. However, during testing, we encountered an issue where the parser failed to differentiate variables with identical names in separate scopes.

Challenges:

- **Scope Management Conflicts**: Tests revealed that the parser treated variables in different local scopes as duplicates, violating the instructions' intent for independent scopes.
- **Rubric Inconsistencies**: Certain rubric requirements conflicted with provided test cases, further complicating scope handling.

Decision and Resolution: We updated the scope management logic to allow identical variable names in different local scopes. This required significant changes to the symbol table and parser, but the result was a system that correctly respected variable scope boundaries.

Outcome:
The updated implementation adhered to the rubric's expectations while resolving ambiguities in the instructions. This refactoring effort also improved the modularity of the symbol table.

Section 6 - Final Review

6.1 Outstanding Issues

Incomplete Test Coverage:
While significant testing was performed, particularly by the QA lead, some edge cases in the interpreter's functionality remain untested. This includes advanced scenarios in nested scoping and error handling for invalid expressions. Expanding test coverage with automated regression testing would enhance reliability.

Redundancy in Syntax Tree Operations:
The merging of the Concrete and Abstract Syntax Trees into a unified structure simplified the architecture, but the current implementation includes redundant operations, particularly in traversals. Optimizing these operations could improve efficiency and maintainability.

Limitations in Proto-C Compliance:
The interpreter adheres closely to the Proto-C language defined by the professor's BNF grammar. However, minor inconsistencies were observed in handling complex function calls and array operations. Addressing these issues would ensure stricter compliance with the language specifications.

Scalability of Symbol Table:
The current symbol table implementation effectively manages small to medium-sized programs but may degrade in performance with significantly larger programs or deeper nesting. Refactoring the symbol table with a more efficient lookup mechanism, such as a hash map, could address this potential bottleneck.

Error Reporting and Debugging Feedback:
The interpreter currently generates basic error messages, which are sufficient for identifying issues but lack detail. Improving error reporting to include precise locations and suggestions for resolution would enhance the debugging experience for users.

Dependency on Manual Testing:
The lack of a fully automated testing pipeline limited the team's ability to quickly identify regressions during development. Implementing a continuous integration (CI) system with automated test cases would streamline the development process and improve reliability.

6.2 Project Conclusion

The successful completion of this project marks a significant achievement for our team. We met the primary goals outlined at the beginning of the project, including the development of a functional interpreter for the Proto-C language. Key milestones, such as implementing lexical analysis, syntax parsing, symbol management, and expression evaluation, were successfully accomplished.

The final product meets the functional requirements defined in the professor's specifications. It processes and executes Proto-C code accurately, adhering to the defined BNF grammar and supporting features like nested scoping, array handling, and postfix expression evaluation. Despite challenges encountered during development, such as structural changes to the syntax tree and ensuring Proto-C compliance, our team effectively collaborated to deliver a robust and reliable interpreter.

Overall, this project provided valuable insights into interpreter design, collaborative software development, and debugging complex systems. The functional product not only satisfies the project objectives but also serves as a strong foundation for future extensions and improvements.

Section 7.0 - Glossary

Glossary of Relevant Terms:

- AST (Abstract Syntax Tree): A tree representation of the abstract syntactic structure of source code, used to represent its logical flow and operations.

- BNF (Backus-Naur Form): A notation used to define the grammar of a language, specifying its syntax rules.

- CST (Concrete Syntax Tree): A detailed tree representation of source code that includes all syntactic elements, often used as an intermediary step in parsing.

- CodeNode: A class used to represent individual nodes in the syntax tree, such as variables, functions, or expressions.

- CodeScope: A class representing a single scope within the symbol table, used to manage variable declarations and function definitions in nested structures.

- CI (Continuous Integration): A development practice where code changes are automatically tested and integrated into the main branch to ensure functionality and compatibility.

- Lexical Analysis: The process of breaking source code into tokens, which are meaningful sequences of characters, such as keywords, operators, and identifiers.

- NodeType: An enumeration used to represent the type of a `CodeNode` in the syntax tree (e.g., assignment, declaration, expression).

- Parser: A component that analyzes the syntactic structure of tokenized code, building a syntax tree that represents the program's structure.

- Postfix Notation (Reverse Polish Notation): A mathematical notation where operators follow their operands, often used for expression evaluation in interpreters.

- Proto-C: A simplified subset of the C programming language used in this project, defined by the professor's specifications and BNF grammar.

- Scope: The context in which variables or symbols are defined, determining their accessibility within the program.

- Symbol: An entity in the symbol table that represents a variable, function, or parameter, along with its associated attributes.

- Symbol Table: A data structure used to store and manage information about symbols (e.g., variables, functions) and their scopes within the program.

- Syntax Tree: A hierarchical representation of the structure of a program, encompassing nodes for statements, expressions, and other syntactic elements.

- Token: A single meaningful unit of source code, such as a keyword, operator, or identifier, generated during lexical analysis.

- Tools Module: A utility module containing helper functions used throughout the interpreter, such as string manipulation and formatting functions.