COLLEGE OF ENGINEERING
DEPARTMENT OF SOFTWAR E ENGINEERING

SOFTWARE COMPONENT DESIGN

Section B

GROUP 3

NAME	ID
1) HAILELEUL GIRMA	ETS0573/13
2) HAILEYESUS MESAFINT	ETS 0571/13
3) EYOSIYAS SHISEMA	ETS 0478/13
4) EYORAB HAILE	ETS 0473/13
5) HAILEYE ARAGAW	ETS 0574/13

SUBMISSION DATE: 12/18/24

Table of Contents

1. Introduction

1.1 Project Overview

This project implements a simple compiler for a basic programming language. The compiler consists of three main components:

- **Lexer:** Performs lexical analysis, breaking down the input code into a stream of tokens (e.g., keywords, identifiers, operators, constants).
- **Parser:** Performs syntax analysis, verifying that the sequence of tokens conforms to the grammar of the language. It also constructs an internal representation of the code.
- **Runtime Environment:** Manages variables and their values during program execution, allowing for basic arithmetic operations.

1.2 Goals and Objectives

The primary goals of this project were to:

- Gain practical experience in compiler design principles.
- Implement core compiler components: lexical analysis, syntax analysis, and semantic analysis.
- Learn how to handle errors (lexical, syntax, semantic) and provide meaningful error messages.
- Explore techniques for building and testing a compiler.
- Experience collaborative software development using version control systems (specifically, GitHub).

1.3 Project Scope

The scope of this project includes:

- **Supported Language Features:**
    - Variable declarations and assignments.
    - Basic arithmetic operations (addition, subtraction, multiplication, division).
    - Simple expressions.
- **Compiler Functionality:**
    - Lexical analysis to identify and classify tokens.
    - Syntax analysis to check for grammatical correctness.
    - Semantic analysis to evaluate expressions and manage variables.
    - Basic error handling and reporting.
- **Collaboration Features:**
    - Utilizing GitHub for version control, code reviews, and issue tracking.
    - Collaborative development and code merging.

2. System Architecture

2.1 High-Level Design

The compiler follows a traditional three-stage pipeline:

1. **Lexical Analysis:** The input code is read character by character, and tokens are identified and classified. This stage is responsible for recognizing keywords, identifiers, operators, constants, and literals.
2. **Syntax Analysis:** The sequence of tokens is analyzed to determine if it conforms to the grammar of the language. The parser constructs an internal representation of the code, typically an Abstract Syntax Tree (AST) or a similar structure.
3. **Semantic Analysis and Code Generation:**
    1. **Semantic Analysis:** This stage checks for semantic errors, such as type mismatches, undeclared variables, and division by zero. It also performs type checking and resolves variable references.
    2. **Code Generation:** In a real-world compiler, this stage would translate the internal representation into machine code. In this project, the semantic analysis stage directly executes the code and produces the output.

2.2 Components and interactions

- **Components:**
    - **Lexer:** Reads the input code and produces a stream of tokens.
    - **Parser:** Analyzes the token stream, checks for syntax errors, and constructs an internal representation of the code.
    - **Runtime Environment:** Manages variables, their values, and performs arithmetic operations.
- **Interactions:**
    - The Lexer provides tokens to the Parser.
    - The Parser interacts with the Runtime Environment to perform semantic actions and evaluate expressions.

2.3 Classes

- **Lexer:** Contains methods for reading input, identifying tokens, and managing the symbol table.
- **Token:** Represents a single token with its type, value, and line number.
- **Parser:** Contains methods for parsing different language constructs (e.g., statements, expressions).
- **RuntimeEnvironment:** Contains methods for managing variables, performing operations, and checking for errors.

3. Implementation Details

3.1 Lexical Analysis (Lexing)

- **Tokenization:** The Lexer class reads the input code character by character and identifies individual tokens.
    - It recognizes keywords (e.g., "if", "else", "while"), identifiers (variable names), operators (+, -, *, /, =), constants (numbers), literals (strings), and punctuation symbols.
    - Each identified token is classified and stored in a Token object, containing its type, value, and line number.

- **Symbol Table:** The Lexer maintains a symbolTable to store information about encountered identifiers. This can be used for later stages, such as type checking.

3.2 Syntax Analysis (Parsing)

- **Recursive Descent:** The RecursiveDescentParser class implements a top-down parsing algorithm.
    - It starts from the highest-level grammar rule (e.g., parseProgram) and recursively calls functions to parse smaller syntactic units (statements, expressions).
    - The parser checks if the sequence of tokens conforms to the defined grammar rules.
    - If a syntax error is encountered, an appropriate error message is generated and the parser attempts to recover (if possible).

3.3 Semantic Analysis and Runtime Environment

- **Semantic Actions:** During parsing, semantic actions are performed to:
    - Check for semantic errors (e.g., undeclared variables, type mismatches, division by zero).
    - Manage variables and their values.
    - Evaluate expressions and perform arithmetic operations.
- **Runtime Environment:** The RuntimeEnvironment class maintains a data structure (e.g., a hash table) to store variables and their associated values.
    - It provides methods to:
        - Set the value of a variable.
        - Get the value of a variable.
        - Check if a variable exists.
        - Perform arithmetic operations on variables.

4. Testing and Validation

4.1 Unit Tests

- **Lexer:**
    - Test cases were designed to verify the correct identification and classification of different types of tokens (keywords, identifiers, operators, constants, literals).
    - Edge cases, such as invalid characters and unexpected input, were also tested.
- **Parser:**
    - Unit tests were created to check the correct parsing of various language constructs, including:
        - Simple variable assignments (e.g., x = 5;)
        - Arithmetic expressions (e.g., x = 2 + 3 * 4;)
        - More complex expressions with operator precedence.
    - Tests were designed to ensure the parser correctly handles syntax errors and reports them appropriately.

4.2 Integration Tests

- Integration tests were conducted to verify the interaction between different components:
    - Tested the interaction between the Lexer and the Parser, ensuring that the Parser correctly processes the tokens generated by the Lexer.

- Tested the interaction between the Parser and the Runtime Environment, ensuring that variables are correctly managed and expressions are evaluated correctly.

4.3 Test Coverage

- **Code Coverage:** Efforts were made to achieve high code coverage with unit tests, ensuring that most lines of code in each component were executed during testing.
- **Statement Coverage:** Measured the percentage of statements in the code that were executed at least once during testing.
- **Branch Coverage:** Measured the percentage of branches (e.g., if-else conditions) that were executed during testing.

5. Collaboration using GitHub

5.1 Project Setup and Repository Creation

1. **Project Initialization:** A new GitHub repository was created specifically for this project.
2. **Initial Commit:** The initial project structure, including essential files were committed to the main branch.

5.2 Branching and Merging Strategies

- **Feature Branches:** For each new feature or bug fix, a new branch was created from the main branch. This allowed team members to work independently on different aspects of the project without interfering with each other's progress.
- **Merging:** When a feature was complete or a bug was fixed, a pull request was created to merge the feature branch back into the main branch.

    - This facilitated code reviews and ensured that all changes were properly integrated.
    - Conflicts that arose during merging were carefully resolved through discussions and code adjustments.

5.3 Pull Requests and Code Reviews

- **Pull Requests:** Each pull request included a clear description of the changes made and their purpose.
- **Code Reviews:** Team members reviewed each other's code, providing constructive feedback, identifying potential issues, and suggesting improvements.
- **Discussions:** Discussions within pull requests helped to clarify requirements, resolve design decisions, and improve the overall code quality.

6. Challenges and Solutions

Technical Challenges:

- **Handling Complex Grammar:** Implementing the parser for more complex language features (e.g., nested expressions, control flow statements) presented a significant challenge.
    - **Solution:** We carefully studied and analyzed the grammar rules, breaking them down into smaller, more manageable sub-rules. We also referred to online resources and textbooks on compiler design for guidance.
- **Error Handling and Recovery:** Implementing robust error handling and recovery mechanisms proved to be challenging.
    - **Solution:** We implemented basic error recovery techniques to allow the parser to continue after encountering an error, preventing cascading errors. We also focused on providing informative and user-friendly error messages.

Collaboration Challenges:

- **Merging Conflicts:** Resolving merge conflicts that arose when multiple team members were working on the same files simultaneously required careful attention and communication.
    - **Solution:** We utilized GitHub's conflict resolution tools, discussed conflicts thoroughly, and ensured that all changes were properly integrated.
- **Maintaining Code Consistency:** Ensuring that all team members adhered to the same coding style and conventions was important for code maintainability.
    - **Solution:** We established clear coding guidelines and reviewed each other's code to ensure consistency.

7. Conclusion

Project Outcomes and Achievements:

- Successfully implemented a basic compiler with core functionalities: lexical analysis, syntax analysis, and semantic analysis.
- Gained valuable experience in compiler design principles and techniques.
- Developed practical skills in using GitHub for collaborative software development.
- Produced a working compiler that can successfully parse and execute simple programs.

Lessons Learned and Future Improvements:

- **Enhance Error Handling:** Implement more sophisticated error recovery mechanisms to improve the robustness of the compiler.
- **Extend Language Features:** Add support for more complex language features, such as control flow statements, data structures, and functions.
- **Improve User Interface:** Enhance the user interface with features like code highlighting, debugging tools, and better error reporting.
- **Code Optimization:** Explore techniques for optimizing the generated code to improve performance.