Report on C- Language Interpreter

1. Introduction

This report details the design and implementation of an interpreter for the C- language, developed as part of a Compiler Design course project. The objective of the project was to create an interpreter capable of executing programs written in C- by processing syntax and semantics according to predefined rules. The interpreter handles variable declarations, arithmetic operations, control flow constructs (like `if` and `while` statements), and error detection.

2. Project Overview

The interpreter is built in C, utilizing Lex (Flex) for lexical analysis and a recursive descent parser for syntax analysis and interpretation. The core components include:

- **Lexer (Scanner)**: Breaks down the input C- source code into tokens.
- **Parser and Interpreter**: Parses the tokens according to the grammar rules and executes the code by performing computations and maintaining an execution environment.
- **Symbol Table**: Stores variables and their attributes, such as name, type, value, and declaration line number.

3. Design and Architecture

3.1. Lexer (Scanner)

The lexer uses regular expressions to recognize keywords, identifiers, operators, literals, and other constructs in the C- language. It generates tokens that are passed to the parser for further analysis. The lexer is implemented using Flex.

Key features of the lexer include:

- **Keywords**: `Program`, `int`, `float`, `if`, `else`, `while`.
- **Operators**: Arithmetic (`+`, `-`, `*`, `/`), relational (`<`, `<=`, `>`, `>=`, `==`, `!=`), assignment (`=`).
- **Punctuation**: Semicolons (`;`), commas (`,`), parentheses (`(`, `)`), braces (`{`, `}`), brackets (`[`, `]`).
- **Identifiers**: Variable names following C-like naming conventions.
- **Literals**: Numeric literals (integers and floats).

3.2. Parser and Interpreter

The parser performs syntax analysis using a recursive descent approach and is integrated with semantic actions to execute the code during parsing. It verifies that the input adheres to the grammar rules of the C- language.

The interpreter:

- **Executes assignments and expressions**: Evaluates arithmetic expressions and assigns values to variables.
- **Handles control structures**: Executes `if`, `else`, and `while` statements.
- **Manages the symbol table**: Inserts and looks up variables, checks for redeclarations, and ensures variables are declared before use.
- **Performs semantic checks**: Detects type mismatches and other semantic errors.

Key grammar components include:

- **Program Structure**: The program must start with the `Program` keyword, followed by an identifier and a block of code enclosed in `{}`.
- **Declarations**: Variables must be declared with a type (`int` or `float`) before use.
- **Statements**: Includes assignments, compound statements, selection statements (`if-else`), and iteration statements (`while`).
- **Expressions**: Supports arithmetic and relational operations.

3.3. Symbol Table

The symbol table is implemented using a simple array structure that stores symbols representing variables. Each symbol contains:

- **Name**: The identifier of the variable.
- **Type**: The data type (`int` or `float`).
- **Value**: The current value of the variable.
- **Declaration Line**: The line number where the variable was declared.

The symbol table supports operations to insert new symbols, look up existing symbols, set variable values, and retrieve variable values.

4. Implementation Details

4.1. Attribute Grammar

Semantic actions are integrated into the parsing functions to execute code and perform semantic checks on the fly. For example, during expression parsing, the interpreter evaluates the expression and checks for type consistency.

Example of expression evaluation:

c

Copy code

```
void additive_expression(VarType *type, void *value) {
    VarType left_type;
    union { int int_val; float float_val; } left_value;
    term(&left_type, &left_value);
    while (lookahead == ADDOP) {
        int op = yytext[0];
        match(ADDOP);
        VarType right_type;
        union { int int_val; float float_val; } right_value;
        term(&right_type, &right_value);
        if (left_type != right_type) {
            semantic_error("Type mismatch in additive expression");
        }
        if (left_type == INT_TYPE) {
            left_value.int_val = (op == '+') ? left_value.int_val +
right_value.int_val : left_value.int_val - right_value.int_val;
        } else {
```

```
            left_value.float_val = (op == '+') ? left_value.float_val
+ right_value.float_val : left_value.float_val -
right_value.float_val;

        }

    }

    *type = left_type;

    if (*type == INT_TYPE) {

        *((int *)value) = left_value.int_val;

    } else {

        *((float *)value) = left_value.float_val;

    }

}
```

4.2. Semantic Error Handling

The interpreter checks for the following semantic errors:

1. **Undeclared Variables**: Using a variable before it is declared results in an error.
2. **Variable Redeclaration**: Redeclaring a variable in the same scope is prohibited.
3. **Type Mismatch**: Mixing types in operations or assignments is not allowed.

When a semantic error is detected, the interpreter outputs an error message with the line number and description, and terminates execution.

4.3. Execution of Instructions

The interpreter executes instructions during parsing without generating a syntax tree. It evaluates expressions, performs assignments, and handles control flow constructs by maintaining the program state through the symbol table.

4.4. Sample Implementation Snippet

c

Copy code

```c
void assignment_stmt() {
    char name[100];
    VarType var_type;
    var(name, &var_type);
    if (lookahead == ASSIGNOP) {
        match(ASSIGNOP);
        VarType expr_type;
        union { int int_val; float float_val; } value;
        expression(&expr_type, &value);
        if (var_type != expr_type) {
            semantic_error("Type mismatch in assignment");
        } else {
            set_symbol_value(name, expr_type, &value);
        }
    } else {
        syntax_error("'='");
    }
```

```
    match(';');
}
```

5. Test Cases

To validate the interpreter's functionality and error handling, the following seven test cases were designed:

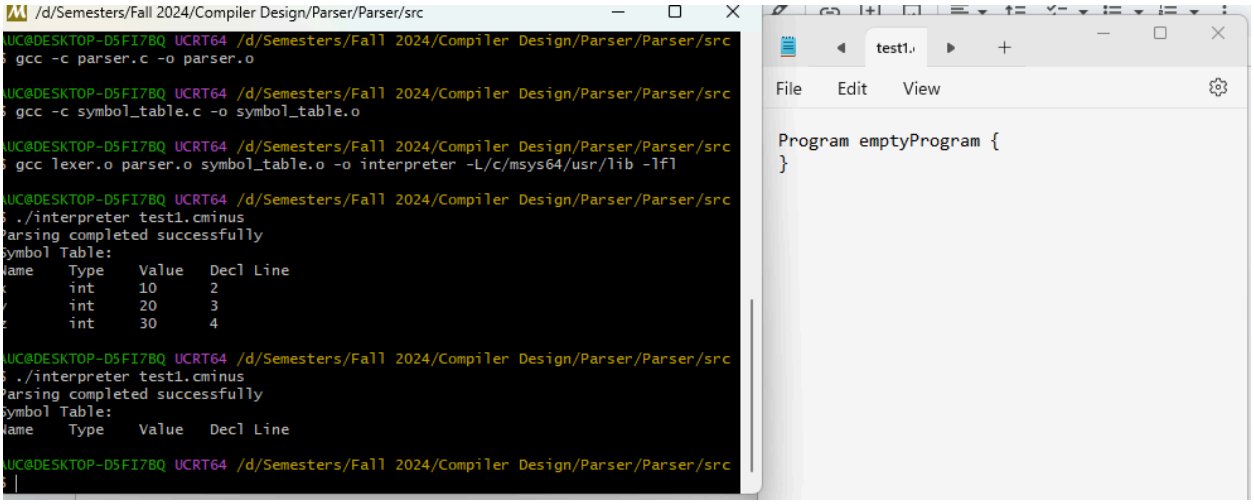
5.1. Test Case 1: Empty Program Block

Input (`test1.cminus`):

c

Copy code

```
Program emptyProgram {

}
```



- The interpreter should parse and execute the program successfully without errors.
- The symbol table should be empty.

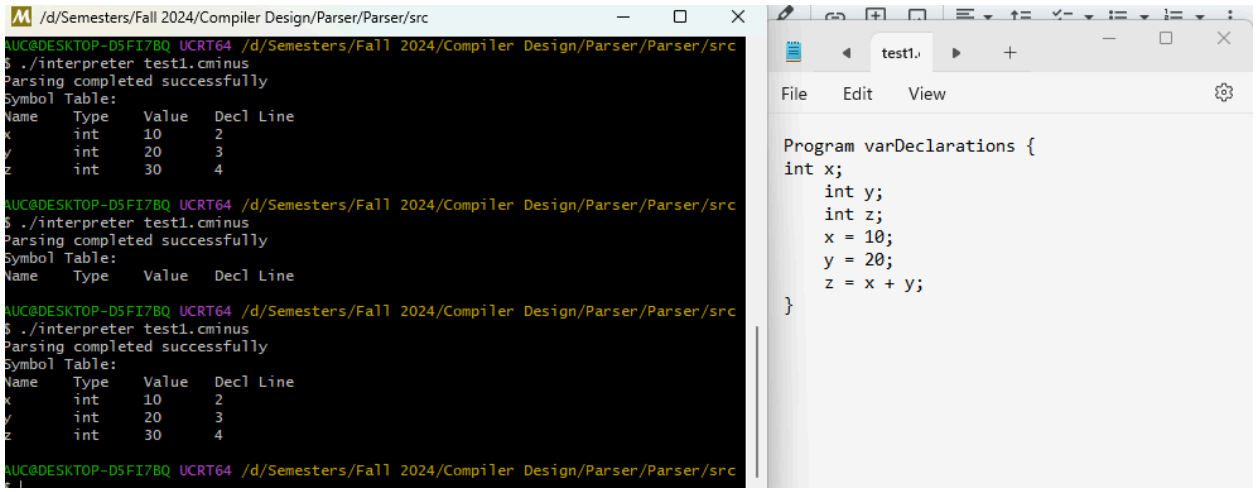
5.2. Test Case 2: Variable Declarations and Assignments

Input (`test2.cminus`):

c

Copy code

```
Program varDeclarations {
      int x;
    int y;
    int z;
    x = 10;
    y = 20;
    z = x + y;
}
```



- Variables `x`, `y`, and `z` are declared and assigned values.
- The final values are:
  - `x = 10`
  - `y = 20`
  - `z = 30`

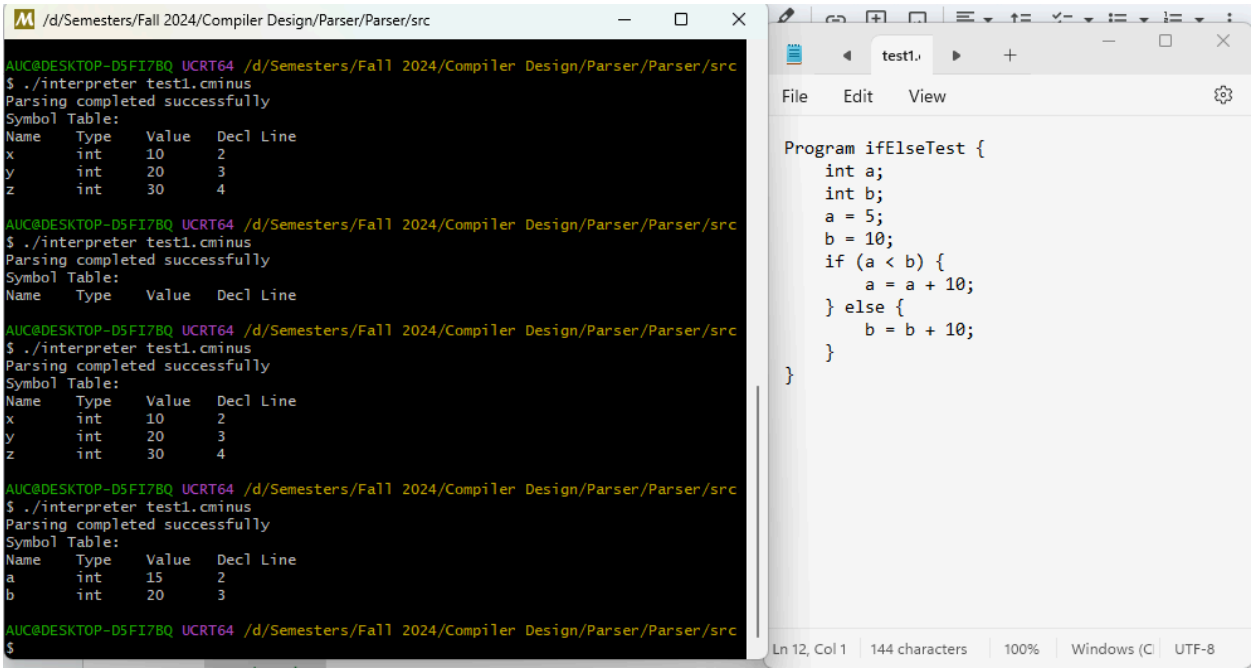
5.3. Test Case 3: If-Else Statement Execution

Input (`test3.cminus`):

c

Copy code

```
   Program ifElseTest {
    int a;
    int b;
    a = 5;
    b = 10;
    if (a < b) {
        a = a + 10;
    } else {
        b = b + 10;
    }
}
```



Expected Result:

- Since `a < b` is true, `a` is incremented by 10.
- Final values:
  - `a = 15`
  - `b = 10`

5.4. Test Case 4: While Loop Execution

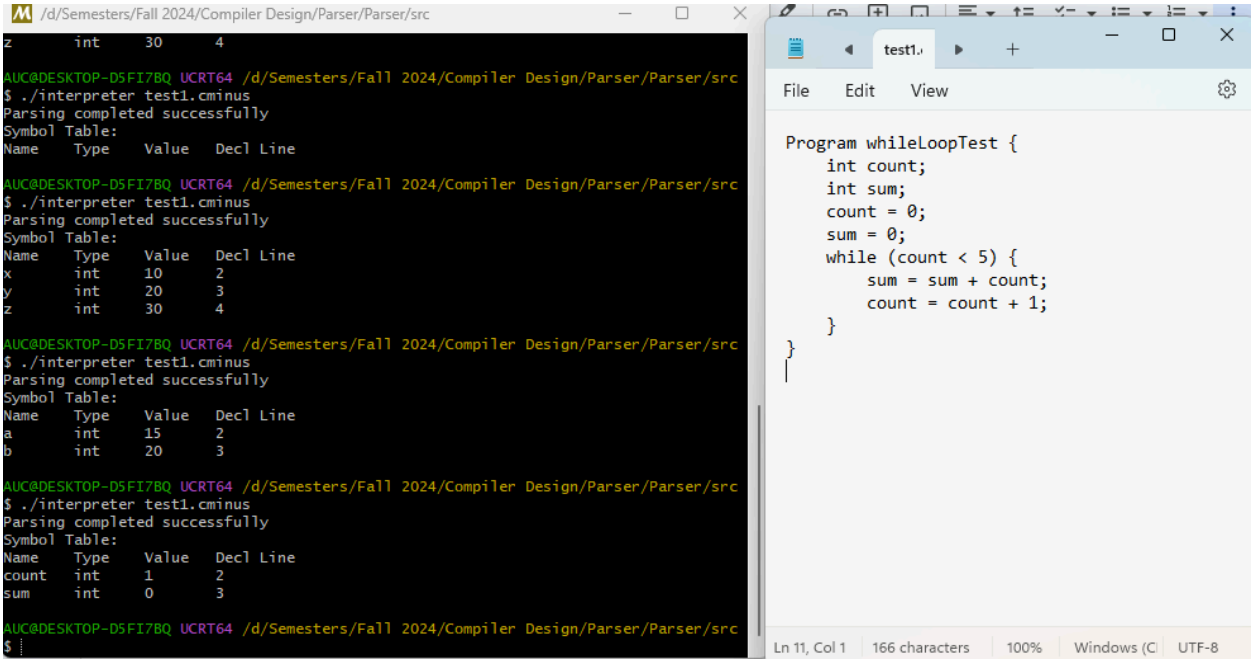
Input (`test4.cminus`):

c

Copy code

```
Program whileLoopTest {

    int count;

    int sum;

    count = 0;

    sum = 0;

    while (count < 5) {
```

```
        sum = sum + count;

        count = count + 1;

    }

}
```



5.5. Test Case 5: Type Mismatch Error

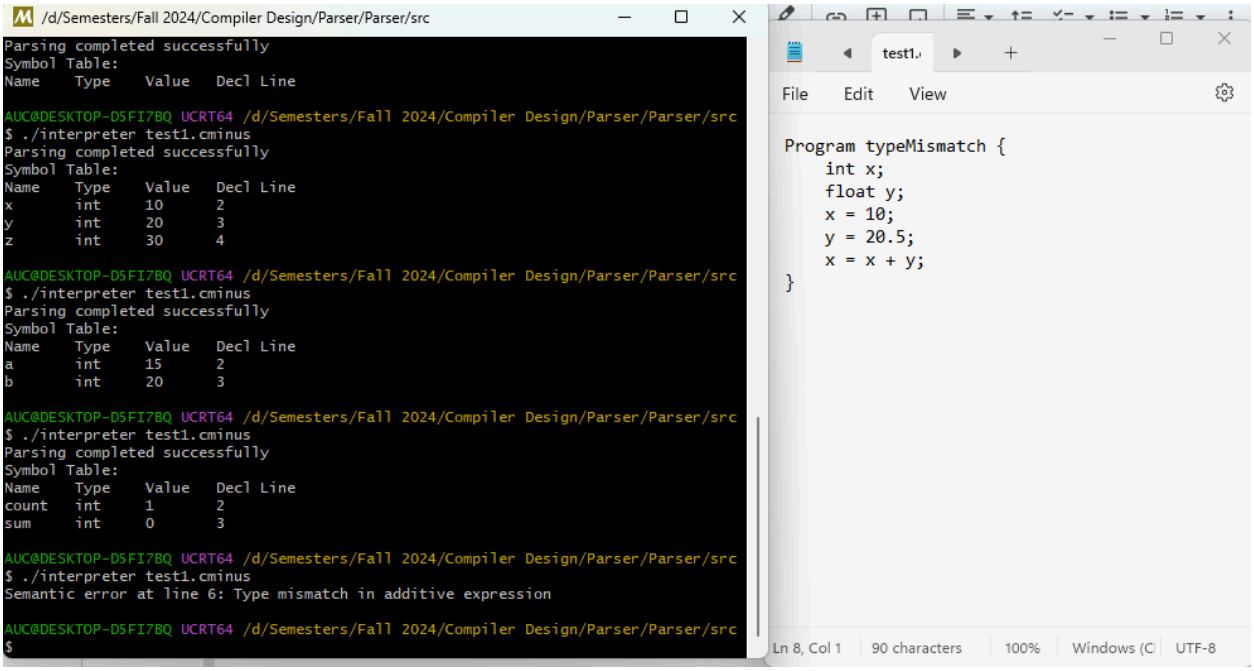
Input (`test5.cminus`):

c

Copy code

```
Program typeMismatch {

    int x;

    float y;

    x = 10;

    y = 20.5;
```

```
    x = x + y;
}
```



Expected Result:

- A semantic error should occur due to mixing `int` and `float` types in the expression `x + y`.
- Error message indicating a type mismatch in the additive expression.

5.6. Test Case 6: Undeclared Variable Error

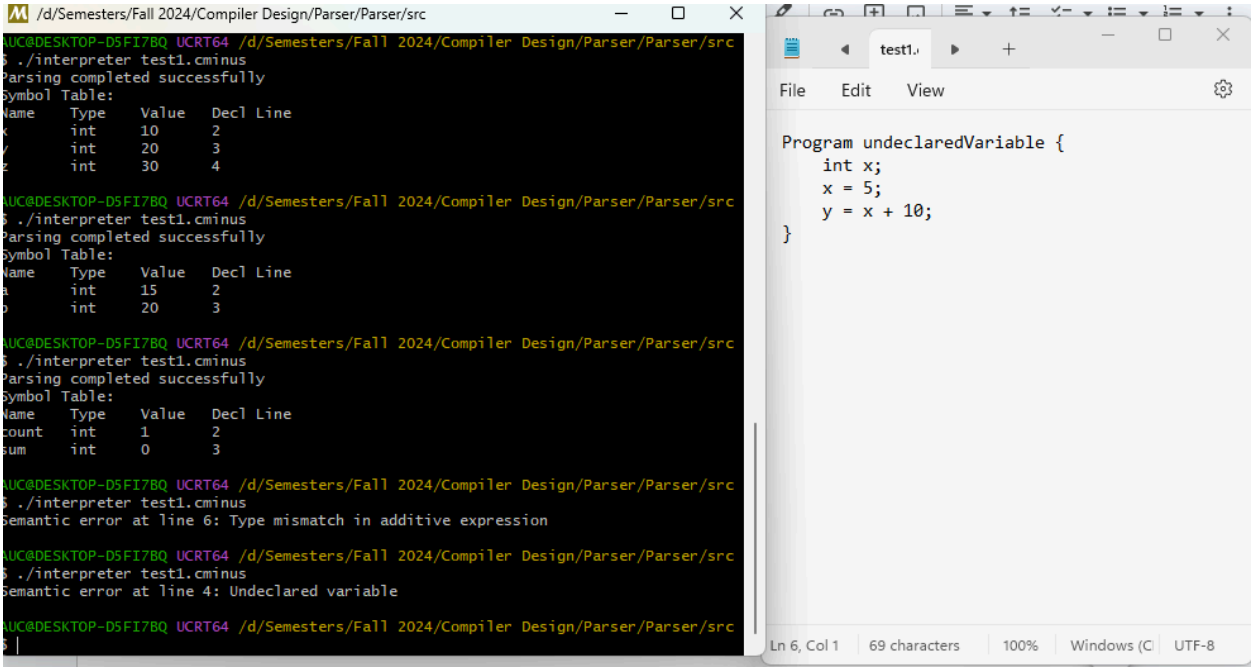
Input (`test6.cminus`):

c

Copy code

```
Program undeclaredVariable {
    int x;
    x = 5;
    y = x + 10;
```

```
}
```



- A semantic error should occur because variable `y` is used without being declared.
- Error message indicating the use of an undeclared variable.

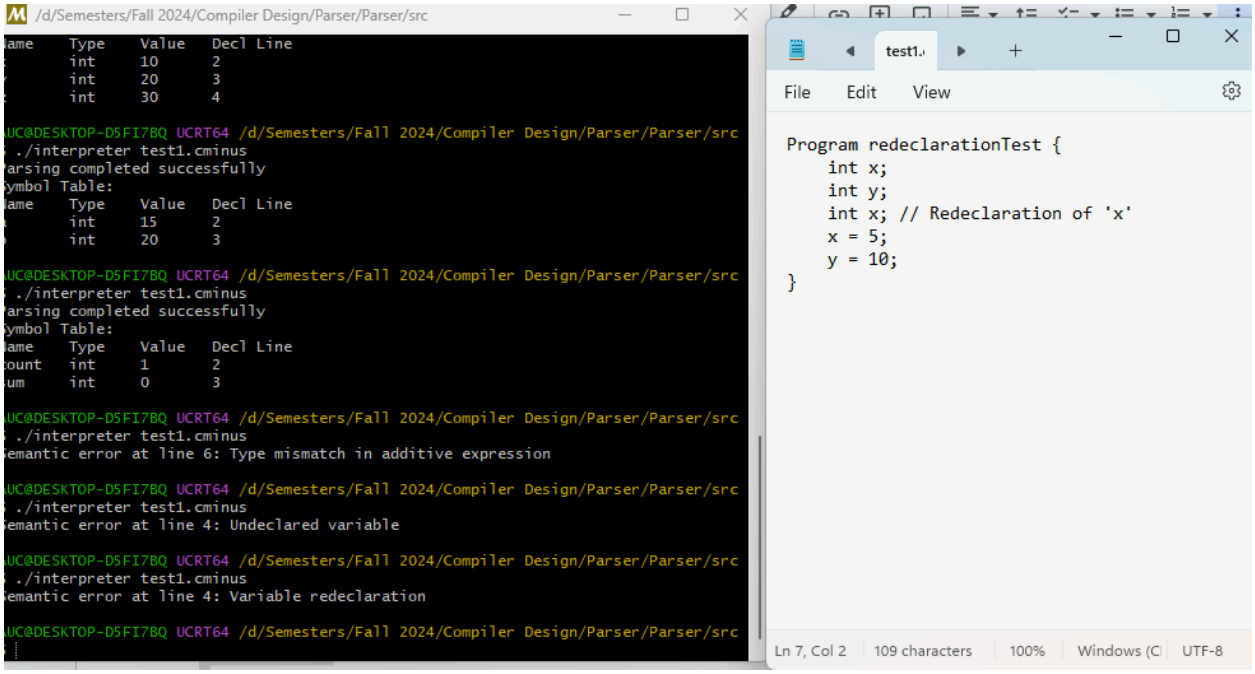
5.7. Test Case 7: Variable Redeclaration Error

Input (`test7.cminus`):

c

Copy code

```
Program redeclarationTest {

    int x;

    int y;

    int x; // Redeclaration of 'x'

    x = 5;

    y = 10;

}
```



- A semantic error should occur due to the redeclaration of variable x.
- Error message indicating that the variable has been redeclared.

6. Results

The interpreter was tested with the above test cases, and the following outcomes were observed:

- **Test Cases 1-4**: The interpreter successfully executed the programs, correctly managing variable declarations, assignments, control flow, and arithmetic computations.
- **Test Cases 5-7**: The interpreter appropriately detected semantic errors, providing clear error messages with line numbers and descriptions.

7. Conclusion

This project demonstrates the ability to construct a simple interpreter for a C-like language. By utilizing Lex (Flex) for lexical analysis and a recursive descent parser for syntax analysis and interpretation, the project successfully tokenizes, parses, and executes C- programs. The interpreter effectively handles variable declarations, arithmetic operations, control structures, and error detection.

Through the implementation of test cases, the system's correctness and robustness were verified. The interpreter can be extended in the future to support more complex features, improve error handling, and optimize performance.

8. References

- **Flex (Fast Lexical Analyzer Generator)**: Tool for generating scanners (lexical analyzers).
- **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie, 2nd Edition, Prentice Hall.