



Green University of Bangladesh

*Department of Computer Science and Engineering (CSE)
Semester: (Spring, Year: 2024), B.Sc. in CSE (Day)*

Command Line Calculator

*Course Title: Compiler Lab
Course Code: CSE 306
Section: 221-D5*

Students Details

Name	ID
Humayra Afia Hany	221002338
MD Abu Sufian	221002136

*Submission Date: 23-06-2024
Course Teacher's Name: Farhan Mahmud*

[For teachers use only: **Don't write anything inside this box**]

<u>Lab Project Status</u>	
Marks:	Signature:
Comments:	Date:

Contents

1	Introduction	2
1.1	Overview	2
1.2	Motivation	2
1.3	Problem Definition	2
1.3.1	Problem Statement	2
1.3.2	Complex Engineering Problem	3
1.4	Design Goals/Objectives	3
1.5	Application	4
2	Design/Development/Implementation of the Project	5
2.1	Introduction	5
2.2	Project Details	5
2.3	Implementation	5
3	Performance Evaluation	14
3.1	Simulation Environment/ Simulation Procedure	14
3.2	Results Analysis	14
3.3	Results Overall Discussion	15
4	Conclusion	16
4.1	Discussion	16
4.2	Limitations	16
4.3	Scope of Future Work	17

Chapter 1

Introduction

1.1 Overview

This project involves creating a Simple Arithmetic Expression Compiler. It reads arithmetic expressions from the user, breaks them down into tokens, parses these tokens according to the rules of arithmetic, and evaluates the expression to produce a result. The compiler supports basic operations like addition, subtraction, multiplication, division, and parentheses for grouping.

1.2 Motivation

The motivation behind this project is to provide an educational tool that helps students and developers understand the basics of compiler construction. By implementing a simple arithmetic expression compiler, one can grasp fundamental concepts such as tokenization, parsing, and recursive evaluation, which are essential for more complex compiler design.

1.3 Problem Definition

1.3.1 Problem Statement

The goal is to build a compiler that can accurately evaluate arithmetic expressions containing integers and basic arithmetic operators, while correctly respecting operator precedence and parentheses. This involves breaking down the input into manageable tokens, parsing these tokens to understand the expression's structure, and evaluating the parsed expression to obtain the result.

1.3.2 Complex Engineering Problem

The complexity of this problem lies in correctly implementing the parsing and evaluation logic that handles operator precedence and nested parentheses. Ensuring that multiplication and division operations are evaluated before addition and subtraction, and that expressions within parentheses are evaluated first, requires a precise and methodical approach to parsing and evaluating the input expression.

Table 1.1: Summary of the attributes touched by the mentioned projects

Name of the P Attributes	Explain how to address
P1: Depth of knowledge required	A Comprehensive knowledge of language theory, data structures, systems programming, and software engineering principles is essential for effective compiler development.
P2: Range of conflicting requirements	Striking a balance between performance and correctness, flexibility and efficiency, and language features versus compilation speed poses ongoing challenges throughout the compiler development process.
P3: Depth of analysis required	Thorough analysis is crucial for language design, parsing techniques, semantic analysis, and optimization strategies, ensuring robustness, correctness, and efficiency in each phase of the compiler pipeline.
P4: Interdependence	Overcoming new challenges while doing this project

1.4 Design Goals/Objectives

The design goals for this project include

- Create a tokenizer to break down input arithmetic expressions into tokens such as integers, operators, and parentheses
- Ensure the tokenizer handles spaces and correctly identifies each type of token
- Develop a parser to interpret tokens and construct a structure representing the expression
- Respect operator precedence and accurately handle nested parentheses
- Create an evaluation component to compute the result of parsed expressions, ensuring correctness even in edge cases like division by zero
- Prioritize robust error handling to provide meaningful feedback for invalid inputs
- Design a user-friendly compiler to make it easy for users to input and evaluate typical arithmetic expressions

1.5 Application

The Command Line Calculator Compiler project has lots of different uses in many areas:

Math Tools: It can be part of math tools like calculators or software for doing math. People like students, engineers, scientists, and others can use it to quickly get correct answers for tricky math problems.

Scripting: It can be used to write scripts that do math tasks automatically. This is handy for things like doing the same math over and over again or doing math in computer programs.

Libraries for Programs: Developers can use their math abilities in their programs. They can put their math power into their software for things like designing, simulating, or doing financial math.

Small Computers and Devices: Because it's not too heavy, it can work in small computers or gadgets with limited power. It can help do math quickly for things like smart gadgets or devices that control stuff in homes or factories.

Teaching Tools: It can be used in schools to teach math and computer science. Students can use it to learn about math concepts and how computers work by trying things out with it.

Science and Simulation: It's good for doing math in science and simulations. Scientists and engineers can use it to do calculations, solve problems, or make simulations in fields like physics, engineering, chemistry, and biology.

Overall, the Arithmetic Expression Mini Compiler project is a useful tool for doing math in many different places, like schools, businesses, and scientific research. It's easy to use, fast, and can help with lots of different math problems.

Chapter 2

Design/Development/Implementation of the Project

2.1 Introduction

Welcome to the introduction of our Command Line Calculator implemented in the C programming language. Our project aims to efficiently parse and evaluate arithmetic expressions using recursive algorithms. We have developed an intuitive evaluator to handle basic mathematical operations, providing clear error handling and user interaction. Through our Command Line Calculator, we aimed to create a tool that simplifies the process of evaluating arithmetic expressions, whether for simple calculations or more complex tasks. Join us on this journey as we explore the intricacies and effectiveness of our Command Line Calculator, where precision, efficiency, and user-friendliness converge to simplify arithmetic expression evaluation in C programming.

2.2 Project Details

2.3 Implementation

All the implementation details of your project should be included in this section, along with many subsections.

```

8  typedef enum {
9      INTEGER,
0      PLUS,
1      MINUS,
2      MULTIPLY,
3      DIVIDE,
4      LPAREN,
5      RPAREN,
6      POWER,
7      SQRT,
8      SIN,
9      COS,
0      TAN,
1      EOF_TOKEN
2  } TokenType;
3

```

Figure 2.1: Token Types

```

typedef struct {
    TokenType type;
    double value;
} Token;

```

Figure 2.2: Token Structure

```

31 Token getNextToken(char **input) {
32     while (isspace(**input)) {
33         (*input)++;
34     }
35
36     if (**input == '\\0') {
37         return (Token) {EOF_TOKEN, 0};
38     }
39
40     if (**input == '+') {
41         (*input)++;
42         return (Token) {PLUS, 0};
43     }
44
45     if (**input == '-') {
46         (*input)++;
47         return (Token) {MINUS, 0};
48     }
49
50     if (*input == '*') {
51         (*input)++;
52         return (Token) {MULTIPLY, 0};
53     }
54
55     if (**input == '/') {
56         (*input)++;
57         return (Token) {DIVIDE, 0};
58     }
59
60     if (**input == '^') {
61         (*input)++;
62         return (Token) {POWER, 0};
63     }
64
65     if (strncmp(*input, "sqrt", 4) == 0) {
66         *input += 4;
67         return (Token) {SQRT, 0};
68     }

```

Figure 2.3: Lexical Analysis (Tokenization)


```

70     if (strncmp(*input, "sin", 3) == 0) {
71         *input += 3;
72         return (Token) {SIN, 0};
73     }
74
75     if (strncmp(*input, "cos", 3) == 0) {
76         *input += 3;
77         return (Token) {COS, 0};
78     }
79
80     if (strncmp(*input, "tan", 3) == 0) {
81         *input += 3;
82         return (Token) {TAN, 0};
83     }
84
85     if (**input == '(') {
86         (*input)++;
87         return (Token) {LPAREN, 0};
88     }
89
90     if (**input == ')') {
91         (*input)++;
92         return (Token) {RPAREN, 0};
93     }
94
95     if (isdigit(**input) || **input == '.') {
96         double value = 0;
97         int decimal_place = 0;
98         while (isdigit(**input) || **input == '.') {
99             if (**input == '.') {
100                 decimal_place = 1;
101                 (*input)++;
102                 continue;
103             }
104             value = value * 10 + (**input - '0');
105             if (decimal_place) {
106                 decimal_place *= 10;
107             }
108             (*input)++;
109         }

```

Figure 2.4: Lexical Analysis (Tokenization)

```

94
95     if (isdigit(**input) || **input == '.') {
96         double value = 0;
97         int decimal_place = 0;
98         while (isdigit(**input) || **input == '.') {
99             if (**input == '.') {
100                 decimal_place = 1;
101                 (*input)++;
102                 continue;
103             }
104             value = value * 10 + (**input - '0');
105             if (decimal_place) {
106                 decimal_place *= 10;
107             }
108             (*input)++;
109         }
110         if (decimal_place) {
111             value /= decimal_place;
112         }
113         return (Token) {INTEGER, value};
114     }
115
116     // Return an error token instead of exiting
117     return (Token) {EOF_TOKEN, 0}; // Use EOF_TOKEN to
118 }
119

```

Figure 2.5: Lexical Analysis (Tokenization)

```

122 double evaluateExpression(char **input);
123
124 // Function to evaluate a factor
125 double evaluateFactor(char **input) {
126     Token token = getNextToken(input);
127     double result;
128     double evaluateFactor::result
129     if (token.type == INTEGER) {
130         result = token.value;
131     } else if (token.type == LPAREN) {
132         result = evaluateExpression(input);
133         token = getNextToken(input);
134         if (token.type != RPAREN) {
135             printf("Expected closing parenthesis\n");
136             return NAN; // Return NaN to indicate error
137         }
138     } else if (token.type == MINUS) {
139         result = -evaluateFactor(input);
140     } else if (token.type == Sqrt) {
141         result = sqrt(evaluateFactor(input));
142     } else if (token.type == SIN) {
143         result = sin(evaluateFactor(input) * M_PI / 180);
144     } else if (token.type == COS) {
145         result = cos(evaluateFactor(input) * M_PI / 180);
146     } else if (token.type == TAN) {
147         result = tan(evaluateFactor(input) * M_PI / 180);
148     } else {
149         printf("Invalid factor\n");
150         return NAN; // Return NaN to indicate error
151     }
152
153     return result;
154 }

```

Figure 2.6: Evaluate expression recursively

```

double evaluateTerm(char **input) {
    double result = evaluateFactor(input);
    if (isnan(result)) return result; // Check for error

    while (1) {
        Token token = getNextToken(input);
        if (token.type == MULTIPLY) {
            double factor = evaluateFactor(input);
            if (isnan(factor)) return factor; // Check for error
            result *= factor;
        } else if (token.type == DIVIDE) {
            double divisor = evaluateFactor(input);
            if (isnan(divisor)) return divisor; // Check for error
            if (divisor == 0) {
                printf("Division by zero\n");
                return NAN; // Return NaN to indicate error
            }
            result /= divisor;
        } else if (token.type == POWER) {
            double exponent = evaluateFactor(input);
            if (isnan(exponent)) return exponent; // Check for error
            result = pow(result, exponent);
        } else {
            (*input)--;
            break;
        }
    }

    return result;
}

```

Figure 2.7: Evaluating Factors syntax

```

double evaluateExpression(char **input) {
    double result = evaluateTerm(input);
    if (isnan(result)) return result; // Check for error

    while (1) {
        Token token = getNextToken(input);
        if (token.type == PLUS) {
            double term = evaluateTerm(input);
            if (isnan(term)) return term; // Check for error
            result += term;
        } else if (token.type == MINUS) {
            double term = evaluateTerm(input);
            if (isnan(term)) return term; // Check for error
            result -= term;
        } else {
            (*input)--;
            break;
        }
    }

    return result;
}

```

Figure 2.8: Evaluating Terms syntax

```

int main() {
    char expression[100];

    while (1) {
        printf("Enter expression (or 'end' to quit): ");
        fgets(expression, sizeof(expression), stdin);
        expression[strcspn(expression, "\n")] = '\0'; // Remove newline

        if (strcmp(expression, "end") == 0) {
            break;
        }

        char *input = expression;
        double result = evaluateExpression(&input);
        if (isnan(result)) {
            printf("Invalid input. Please try again.\n");
        } else {
            printf("Result: %f\n", result);
        }
    }
}

```

Figure 2.9: Main Function

Chapter 3

Performance Evaluation

3.1 Simulation Environment/ Simulation Procedure

To stimulate the work in this project the software we used is Code::Blocks.

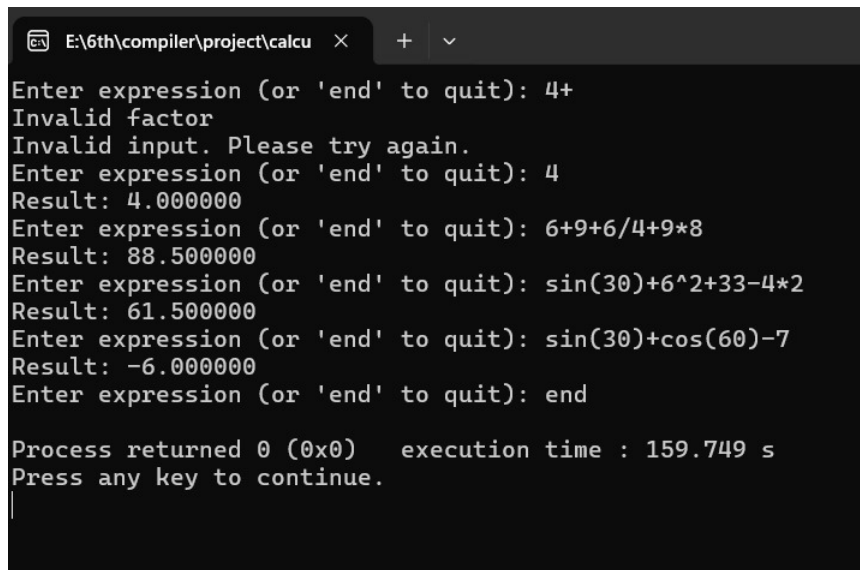


3.2 Results Analysis

```
E:\6th\compiler\project\calcu  X  +  v
Enter expression (or 'end' to quit): 45*2+sin(30)+1.9
Result: 92.400000
Enter expression (or 'end' to quit): 5+
Invalid factor
Invalid input. Please try again.
Enter expression (or 'end' to quit): sin(30)^3+tan(30)-sqrt(9)+6^3
Result: 213.702350
Enter expression (or 'end' to quit): end

Process returned 0 (0x0)   execution time : 79.169 s
Press any key to continue.
```

Figure 3.1: Arithmetic expression



```
E:\6th\compiler\project\calcu x + v
Enter expression (or 'end' to quit): 4+
Invalid factor
Invalid input. Please try again.
Enter expression (or 'end' to quit): 4
Result: 4.000000
Enter expression (or 'end' to quit): 6+9+6/4+9*8
Result: 88.500000
Enter expression (or 'end' to quit): sin(30)+6^2+33-4*2
Result: 61.500000
Enter expression (or 'end' to quit): sin(30)+cos(60)-7
Result: -6.000000
Enter expression (or 'end' to quit): end

Process returned 0 (0x0) execution time : 159.749 s
Press any key to continue.
|
```

Figure 3.2: More Arithmetic expression

3.3 Results Overall Discussion

The Command Line Calculator project has yielded a functional and efficient solution for parsing and evaluating arithmetic expressions in the C programming language. Through the implementation of recursive algorithms, the program successfully handles basic mathematical operations, operator precedence, and parentheses in expressions.

Chapter 4

Conclusion

4.1 Discussion

The Command Line Calculator project has yielded a functional and efficient solution for parsing and evaluating arithmetic expressions in the C programming language.

1. Efficient Parsing and Evaluation: The compiler effectively parses and evaluates arithmetic expressions using recursive algorithms. It handles operator precedence and parentheses accurately, delivering precise results.

2. Robust Error Handling: The compiler includes robust error handling mechanisms. Swiftly identifies and reports various errors, enhancing user experience and ensuring a seamless compilation process.

3. Educational Tool: Serves as a valuable educational tool, offering practical insights into fundamental concepts in compiler design and computational problem-solving.

4. Foundation for Future Enhancements: Provides a solid foundation for potential extensions and optimizations, paving the way for further exploration into advanced features and enhanced performance.

By implementing these features, the project effectively meets its goals and offers significant value both as a practical tool and an educational resource.

4.2 Limitations

The Command Line Calculator project also has certain limitations that should be acknowledged:

1. Expression Complexity: The compiler efficiently handles basic arithmetic operations and parentheses, but its capabilities may be limited with highly complex expressions, potentially leading to parsing errors or slower compilation times.

2. Lack of Optimization: The compiler parses and evaluates arithmetic expressions without implementing optimization techniques, which may impact performance in scenarios with resource constraints or time-critical operations.

3. Limited Error Reporting: The compiler has error handling but may not provide detailed error messages, which can make debugging complex expressions difficult.

4. Platform Dependency: The compiler's behavior and performance may vary based on platform-specific dependencies, optimizations, and architecture differences, which can limit its portability across different systems or environments.

The Arithmetic Expression Mini Compiler has limitations when used for diverse cases or advanced language features. Future enhancements could improve its utility in broader contexts.

4.3 Scope of Future Work

The future works for the Arithmetic Expression Mini Compiler project are multifaceted, offering numerous opportunities for further development and enhancement.

1.Optimization Techniques: Implementing advanced optimization techniques to enhance compiler efficiency and performance. This could involve using optimization algorithms to minimize code size, reduce redundant computations, and improve compilation speed.

2.Extension to Support Additional Operations: Expanding the compiler's capabilities to support a wider range of arithmetic operations and mathematical functions. This could involve adding support for trigonometric functions, logarithms, exponentiation, or other advanced mathematical operations commonly used in scientific computing or engineering applications.

3.Error Recovery Mechanisms: Enhancing the compiler's error handling mechanisms to provide more informative error messages and robust error recovery strategies, including techniques such as error correction, reporting with line numbers, and interactive debugging features to assist users.

4.Language Extensions: Extending the compiler to support additional language features beyond basic arithmetic expressions, such as variables, constants, control structures, user-defined functions, and data types for more complex computations.

5.Integration with External Libraries: Integrating the compiler with external libraries or frameworks can enhance its capabilities and broaden its utility. This may involve leveraging existing libraries for numerical computation, symbolic mathematics, or mathematical optimization.

6.Cross-Platform Compatibility: Ensuring cross-platform compatibility by adapting the compiler codebase to work seamlessly on different operating systems and platforms, including Windows, macOS, Linux, iOS, and Android.

7.Language Specification and Documentation: In order to improve the Arithmetic Expression Mini Compiler, we need to develop a comprehensive language specification and documentation. This will make it easier for developers to understand and use the compiler. Additionally, expanding the compiler's capabilities will make it more versatile and useful for a wider range of applications and domains.