Group 16 LLC

Command Cruncher
Software Requirements Specifications

Version 1.3

Revision History

Date	Version	Description	Author
06/10/24	1.0	Began working on the document, completing the introduction and working on the specific requirements.	Hannah Prosch, Jonathan Kazmaier, Emma Roy, Daniel Van Dalsem, Sneha Thomas, Nifemi Lawal
13/10/24	1.1	Finishing up the document.	Warren Tan, Hannah Prosch, Daniel Van Dalsem, Emma Roy, Jonathan Kazmaier, Nifemi Lawal, Sneha Thomas
24/10/24	1.2	Making edits per the grader, for clarity.	Hannah Prosch
27/10/24	1.3	Completing the edits suggested by the grader.	Hannah Prosch, Daniel Van Dalsem, Emma Roy, Nifemi Lawal, Johathan Kazmaier, Sneha Thomas

Table of Contents

1. Introduction 4
    1.1 Purpose 4
    1.2 Scope 4
    1.3 Definitions, Acronyms, and Abbreviations 4
    1.4 References 4
    1.5 Overview 4
2. Overall Description 4
    2.1 Product perspective 4
    2.1.1 User Interfaces 4
    2.1.2 Hardware Interfaces 4
    2.1.3 Software Interfaces 5
    2.1.4 Memory Constraints 5
    2.1.5 Operations 5
    2.2 Product functions 5
    2.3 User characteristics 5
    2.4 Constraints 5
    2.5 Assumptions and dependencies 5
3. Specific Requirements 5
    3.1 Functionality 5
    3.1.1 Programming language 5
    3.1.2 Mathematical Notation 6
    3.1.3 Order of operations 6
    3.1.4 Error handling 6
    3.2 Use-Case Specifications 6
    3.3 Supplementary Requirements 6
4. Classification of Functional Requirements 6
5. Appendices (Optional) 7

Software Requirements Specifications

1. Introduction

The purpose of this document is to outline the requirements specifications for our program, Command Cruncher. Here, we will go into more specifics regarding requirements that the software for Command Cruncher must include, including the overall scope needed. This document also overviews the other relevant definitions and references necessary for the successful execution and management of the project. Command Cruncher's intended audience is anyone seeking a calculator to find the answer for simple, all the way to complex, mathematical statements with ease. Users will enter the expression they want to know the answer of, including the mathematical operators and parentheses, and Command Cruncher will provide a corresponding solution to their request.

1.1 Purpose

The purpose of this document will highlight the behavior of our application, and the pieces that must be included within the software for Command Cruncher to function as intended. The goal for Command Cruncher is that it will take input from users, correctly handle parenthesis and order of operations, to properly calculate mathematical statements provided by the user. We will define the constraints, functional and non-functional requirements to aid in the development of this project.

1.2 Scope

Our Command Cruncher application will be an arithmetic parser that will correctly handle parenthesis and order of operations with mathematical expressions, including addition, subtraction, multiplication, division, and modulus.

This project is associated with the use-case model for an arithmetic expression evaluator. The primary use case models are parsing arithmetic expressions, evaluating the expression, handling parenthesis, error handling, and user interaction.

1.3 Definitions, Acronyms, and Abbreviations

N/A

1.4 References

Software Development Plan Version 0.3 Date 22/09/2024

1.5 Overview

The following pages of this document contain our functional and non-functional requirements, along with the constraints for our project. This document is organized into several categories below, including overall description, specific requirements, functionality and classification of functional requirements.

2. Overall Description

2.1 Product perspective

2.1.1 User Interfaces

The Evaluator will take in inputs from the user and utilize those inputs to parse arithmetic expressions.

2.1.2 Hardware Interfaces

The Evaluator will send requests to the arithmetic logic unit (ALU) and receive said outputs.

2.1.3 Software Interfaces

The Evaluator will receive, perform arithmetic, and send outputs at language L's request.

2.1.4 Memory Constraints

Limitations toward memory can vary in respect to the size of inputs, which expression is being used, and the number of expressions performed.

2.1.5 Operations

The Evaluator will handle and manage scenarios like division by zero or invalid expressions.

2.2 Product functions

The Evaluator will parse and evaluate the following arithmetic operators: +, -, *, /, %, and **. It will properly evaluate parentheses as well as numeric constants.

2.3 User characteristics

Command Cruncher is for developers, students, and educators of all experience levels who need a reliable, quick tool for evaluating mathematical expressions.

2.4 Constraints

Optionally, optimizing the arithmetic will take time, possibly until the end of the semester. It should run be able to run on Linux and Windows environments, and it must be written C++.

2.5 Assumptions and dependencies

Development assumes the user has proper knowledge of how each of the basic operator's function. We also assume that we will only need to handle real numbers.

3. Specific Requirements

- For this project, the entirety of its backend must be written in C++. This includes the development of both arithmetic operations and numeric values parsing component as well as handling/prioritizing the correct precedence of these operators and therefore correctly interpreting the expression.
- In this scenario, the precedence of order of operations would be as follows:
  - Parenthesis
  - Exponents
  - Unary Operators
  - Multiplication – same level of precedence as Modularity and Division
  - Modularity – same level of precedence as Multiplication and Division
  - Division - same level of precedence as Modularity and Multiplication
  - Addition - same level of precedence as Subtraction
  - Subtraction – same level of precedence as Addition

3.1 Functionality

The program should be able to take an arithmetic expression and output a mathematically accurate answer. The output should follow the order of operations (PEMDAS) and should be able to handle both integers and floats with addition, subtraction, multiplication, division, modulus, exponentiation, unary operators, and parenthesis. All errors in the input must be handled appropriately.

3.1.1 Programming language

Backend must be written in C++.

3.1.2 Mathematical Notation

Should be able to handle addition, subtraction, multiplication, division, modulus, exponentiation, unary operators, and parathesis

3.1.3 Order of operations

The output should follow the order of operations (PEMDAS). It should be able to handle floats and integers.

3.1.4 Error handling

It should handle error appropriately and avoid crashing.

3.2 Use-Case Specifications

- The goal: Evaluate and parse the user's input to output the correct arithmetic result.
- Actors: the user
- Functional requirements: user provides a valid arithmetic expression. Expression must be correctly parsed into tokens (operators, operands, and parenthesis). Parenthesis must be properly handled and the system must follow the correct order of operations (PEMDAS). Result is computed and displayed. The program must handle invalid inputs such as division by zero or invalid expressions.
- Non-functional requirements: The CLI must have clear prompts and error messages. Error detection must occur within 100ms. The system should handle expressions with at least 100 characters in length. Nested parenthesis should be handled efficiently.

3.3 Supplementary Requirements

Non-functional requirements:

- Adding shifting bits operations (if time permits), and optimizing the speed at which our code can run calculations.
- Should be able to handle expressions up to 100 characters in length.
- All mathematically invalid inputs should have a clear error message describing the mistake

Development constraints:

- Try to complete the project ahead of schedule so we can have time to create a visually pleasing and easily accessible user interface.
- Must be written in C++.

4. Classification of Functional Requirements

Functionality	Type
Expression parsing: Must be able to parse arithmetic expressions correctly	Essential
Operator support: Must have implementations for the basic operators (addition, subtraction, division, etc.)	Essential

Parenthesis handling: Must be able to handle expressions within parentheses correctly	Essential
Identifying numeric constants: Must be able to recognize and calculate numeric constants in the input	Essential
Operator precedence: Must be able to implement operator precedence (PEMDAS)	Essential
Error handling: Must have robust error handling (div by 0, invalid expression, etc.)	Essential
User interface: Must have a command-line interface that allows users to input expressions and see results.	Essential
Correct Evaluation: Must properly evaluate expressions	Essential
Support floating point numbers: Accommodate floating point inputs	Desirable
Frontend Interface: Incorporate a visually pleasing, non-command-line, user interface	Optional

5. Appendices (Optional)