Group 16, LLC

Command Cruncher
Software Architecture Document

Version 1.2

Revision History

Date	Version	Description	Author
27/Oct/24	1.0	Beginning to develop the software architecture design.	Hannah Prosch, Daniel Van Dalsem, Emma Roy, Nifemi Lawal, Jonathan Kazmaier, Sneha Thomas
04/Nov/24	1.1	Continuing to develop the software architecture plan.	Hannah Prosch, Daniel Van Dalsem, Warren Tan, Johathan Kazmaier, Emma Roy, Sneha Thomas, Nifemi Lawal
08/Nov/24	1.2	Last minute touches, worked on diagram.	Hannah Prosch

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document (SAD) provides an overview of the system's architectural design and implementation, serving as a guide for stakeholders and development teams. It establishes consistency throughout the project by describing key high-level components, interactions, and design principles that highlight the system's structure and behavior.

1.1 Purpose

The role of this document is to provide the various architectural views to showcase every facet of the system. It does so in an organized system that conveys the meaning of every component and its interaction with one another. In addition to this, it keeps each team member in alignment regarding both the goals and constraints of this task while also giving stakeholders a clear breakdown of every concept and execution. Thus, the overall purpose is to give a breakdown of the system to provide a clear understanding to both the team members and any relevant stakeholders.

1.2 Scope

The software architecture document will assist developers in understanding how the code for Command Cruncher should be organized to optimize performance, readability, and the regular maintenance of the code in the future. The way the Command Cruncher program will be organized and written from a coding standpoint will be influenced by this document, which will be used as a blueprint for creating an effective arithmetic parser.

1.3 Definitions, Acronyms, and Abbreviations

N/A

1.4 References

N/A

1.5 Overview

The Software Architecture Document provides an overview of the system’s architecture. After the introduction that outlines the document’s purpose and scope, the Architectural Goals and Constraints section defines key design goals and constraints. The Logical View section describes the main components. The Process View explains runtime behaviors. The Development View describes the physical environment (hardware). The Implementation View describes the coding standards and framework. The Data View highlights data storage and management solutions. Finally, the Risk and Quality Management section defines potential risks.

2. Architectural Representation -

Logical View (model elements: components, relationships and packages)

Development view (model elements: modules, dependencies, and build processes)

Physical View (model elements: nodes, artifacts, and connections)

Process View (model elements: processes, and concurrency)

User Interface View (optional) - includes UI components (sure as interactive elements) as well as user workflow (the sequence of actions a user must complete to finish a task).

3. Architectural Goals and Constraints

It should run be able to be built and run on Linux and Windows environments, and it must be written in C++. It also needs to follow the order of operations. There should be no privacy concerns, and while we should keep security in mind it's not a top priority.

4. Use-Case View

4.1 Use-Case Realizations

5. Logical View

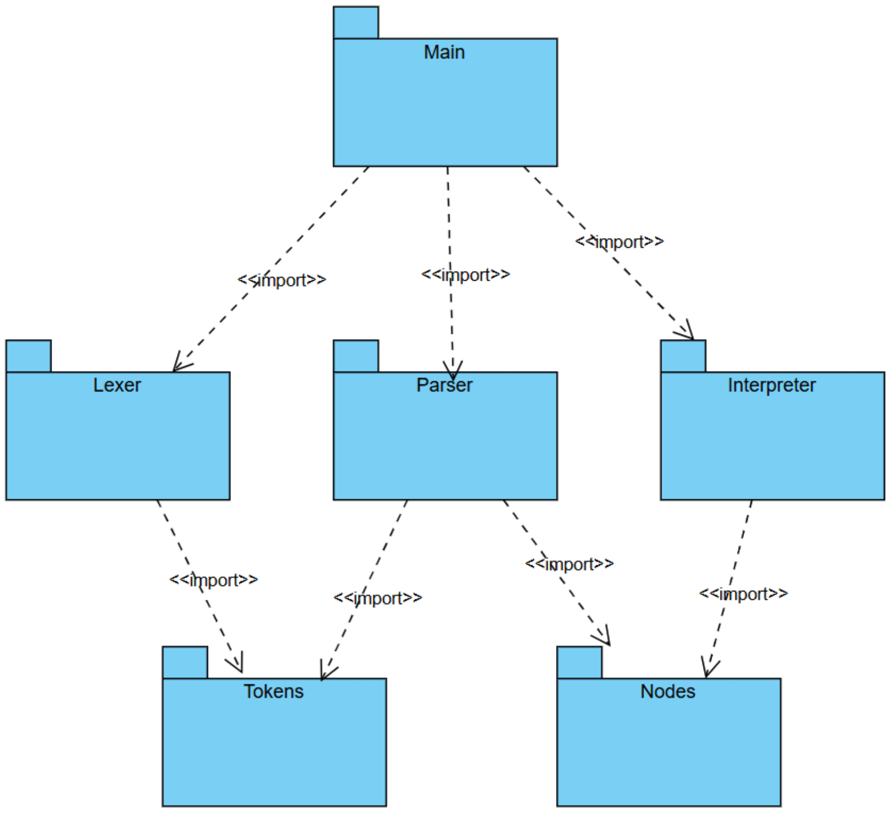
This project will be divided into three pieces, the Lexer, Parser, and Interpreter. The Lexer subsystem processes raw input from the user and transforms it into a series of recognizable tokens, serving as the foundation for syntactic analysis. The Parser subsystem takes these tokens and constructs an Abstract Syntax Tree (AST), which represents the hierarchical structure of the input and captures the relationships between different components of the expression. The Interpreter subsystem then traverses the AST, interpreting and evaluating it to produce a final output. Each subsystem comprises essential classes that handle their specific responsibilities. The main file will use all three systems to output the result.

5.1 Overview

This subsection describes the overall decomposition of the design model by detailing its package hierarchy and layered structure. The model is organized into three primary layers: **Lexical Analysis**, **Syntactic Analysis**, and **Evaluation**. The **Lexical Analysis** layer, represented by the **Lexer** package, is responsible for processing raw input and converting it into tokens. The **Syntactic Analysis** layer, embodied by the **Parser** package, takes these tokens and constructs an Abstract Syntax Tree (AST) that outlines the input's structure. Finally, the **Evaluation** layer, encapsulated by the **Interpreter** package, traverses the AST and performs computations to get the final result.

5.2 Architecturally Significant Design Modules or Packages

There are 6 files we will need to write, the Main module, Token module, Node module, Lexer package, Parser package, and the Interpreter package.



5.2.1 Main Module

Contains a loop that takes user input, pushes the input through the Lexer, Parser, and Interpreter, and outputs the final result.

5.2.2 Token Module

An enumerated type that represents numbers and all mathematical operations. Generated by the Lexer and used by the Parser.

5.2.3 Node Module

Represents a component of an arithmetic expression, such as a number or operation. Used as the building block of the Abstract Syntax Tree built by the Parser

5.2.4 Lexer Package

Takes the user input and goes character by character, turning it into a list of tokens for the parser to dissect.

- advance(): updates the pointer to be the next character in the input.
- generate_tokens(): converts the input into a list of categorized tokens by iterating through each character and identifying its type or raising an exception for unrecognized characters.
- generate_number(): constructs a number token by taking characters from the input that represent digits or a decimal point, ensures the number string is properly formatted if it starts or ends with a decimal, and returns it as a Token of type NUMBER.

5.2.5 Parser Package

Takes a list of tokens and creates an Abstract Syntax Tree of nodes.

- advance(): updates the pointer to be the next token in the input.
- parse(): verifies that the token is valid before running it through all other functions in the package. The main function of the package
- expression(): builds a node if the token is addition or subtraction
- factor(): builds a node if the token is multiplication, division, or modulus
- term(): builds a node if the toke is exponentiation
- primary(): builds a node is the token is a number or builds a new node tree if the token is a parenthesis

5.2.6 Interpreter Package

Evaluates an Abstract Syntax Tree and outputs the result

- visit(): takes a node and returns its associated type function
- visit_{node_type}(): takes a node and returns its mathematical expression. For example, the visit_AddNode would return the node's left child added to its right child.

6. Interface Description

The system's user interface operates through a command-line setup, where users input commands directly, and results are displayed in the terminal. Internally, the Lexer splits these inputs up into tokens, which are structured by the parser into a syntax tree. The tree is passed to the interpreter which processes it to make the final output displayed in the terminal

7. Size and Performance

The system aims to keep memory usage under 50 MB for standard operations. The execution of tasks should feel responsive and quick. Basic tasks should complete in under 100 milliseconds and more complex tasks within 500 milliseconds. The architecture should be flexible enough to handle future additions without major slowdowns and should be compatible with both Linux and Windows environments. The system will run on C++.

8. Quality

The system architecture should support future features/additions with minimal structural changes. Components will have the ability to be adapted or extended without full redesign. In addition, it should handle memory, inputs, and errors effectively. The system must perform under multiple environments with minimal adjustments.