**National Textile University**

**Faisalabad**



**Department of Computer Science**

|  |  |
| --- | --- |
| Assignment # | Semester Project |
| Title: | Custom DSL for math(Matrix operations) |
| Course Name: | Compiler Construction |
| Submitted To: | Dr. Hamid Ali |
| Group members: | Muhammad Huzaifa Khalid, Muhammad Shahmeer, Muhammad Bilal |
| Registration No: | 22-NTU-CS-1199, 22-NTU-CS-1209, 22-NTU-CS-1191 |
| Class: | BSCS – 6B |
| Submission Date: | June 24, 2025 |

Matrix DSL Compiler Documentation

### ****Table of Contents****

1. **Overview**
2. **Sample DSL File (sample.dsl)**
3. **How to run streamit for compiler**
4. **Lexer (lexer.py)**
5. **Parser (parser.py)**
6. **Semantic Analyzer (semantic.py)**
7. **Code Generator (codegen.py)**
8. **Compiler Driver (main.py)**
9. **Streamlit Interface (app.py)**
10. **Output Python File (output.py)**
11. **Automatically Generated Files**
12. **Requirements File (requirements.txt)**
13. **README (README.md)**
14. **Streamlit Interface Output Screenshot**
15. **File Structure Screenshot**

# Overview

This project demonstrates the design and implementation of a Domain-Specific Language (DSL) focused on performing matrix operations such as addition, subtraction, multiplication, division, transpose, and determinant calculation. The compiler consists of several components built using Python, PLY (Python Lex-Yacc), NumPy, and Streamlit for GUI-based interaction.

A DSL compiler typically works in several distinct stages:  
**1.** Lexical Analysis: This stage breaks down the raw text into tokens—identifiable pieces such as keywords, operators, identifiers, and literals.  
**2.** Syntax Analysis: Using grammar rules, the parser builds an Abstract Syntax Tree (AST) from the stream of tokens. The AST represents the program structure.  
**3.** Semantic Analysis: Ensures the logic of the code is correct, such as verifying that matrix operations conform to valid dimension rules.  
**4.** Code Generation: Converts the AST into executable code—in this case, Python with NumPy.  
**5**. Execution and Output: Either through a generated script (output.py) or live interface (Streamlit), the output is presented.

This layered structure mirrors the architecture of traditional compilers, offering insight into compiler construction for domain-specific applications.

## HOW TO RUN COMPILER ON STREAMIT..

1. Paste you matrix in dsl on sample.dsl
2. Run **‘python main.py’** on terminal.
3. After an output.py file created automatically Run **‘python output.py’** command on terminal.
4. Then Run **‘streamlit run app.py’** on terminal and paste dsl code same as in sample.dsl file there.

# Sample DSL File (sample.dsl)

This file serves as the entry point where users write their DSL code. Here's a sample of matrix operations:

matrix A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];  
matrix B = [[9, 8, 7], [6, 5, 4], [3, 2, 1]];  
matrix C\_add = A + B;  
matrix T = TRANSPOSE(A);  
scalar D\_det = DETERMINANT(A);  
matrix C\_sub = A - B;  
matrix C\_div = A / B;

These statements define matrices and perform operations using the custom DSL syntax.

# Lexer (lexer.py)

The lexer breaks the input DSL code into tokens using PLY’s lexical module. Each keyword, operator, and identifier in the DSL is transformed into a symbolic token for further processing.

**Key Tokens Defined**:  
- MATRIX, SCALAR: DSL data types  
- ID: Variable names (like A, B)  
- NUMBER: Numeric values  
- TRANSPOSE, DETERMINANT: Unary matrix operations  
- PLUS, MINUS, TIMES, DIVIDE: Binary operators  
- Symbols like =, [, ], (, ), ;

**Purpose:** Tokenization of DSL code into meaningful symbols that the parser can process.

# Parser (parser.py)

The parser uses PLY’s Yacc functionality to define grammar rules for valid DSL expressions. It builds an Abstract Syntax Tree (AST) to represent the hierarchical structure of the code.

**Key Grammar Rules:**  
- Matrix and scalar declarations with expressions  
- Binary operations: A + B, A - B, A \* B, A / B  
- Unary operations: TRANSPOSE(A), DETERMINANT(A)

Each grammar rule returns a structured representation (AST node) containing the operation type and its operands.

**Purpose**: Converts token stream into structured Abstract Syntax Tree (AST) representing DSL instructions.

# Semantic Analyzer (semantic.py)

The semantic analyzer is responsible for ensuring that the code written in the DSL is logically valid. It goes beyond syntax and checks the meaning of each instruction based on the structure built during parsing. This module operates on the Abstract Syntax Tree (AST) and verifies constraints such as shape compatibility between matrices during operations and the validity of scalar operations.

**Key Responsibilities:**  
1. Type and Shape Validation: Ensures that matrix operations follow mathematical rules. For example, matrix addition and subtraction require both operands to have the same shape, while multiplication requires the inner dimensions to match.  
2. Unary Operation Checks:  
 - For TRANSPOSE, the shape is flipped.  
 - For DETERMINANT, the matrix must be square (same number of rows and columns).  
3. Symbol Table Management: Maintains a mapping of matrix/scalar identifiers to their shapes. It acts as a type environment for the DSL during compilation.

Symbol Table Explanation:

The SymbolTable class keeps track of declared variables. It stores each matrix or scalar's name and its associated shape as a tuple. For instance, after processing the following lines:

matrix A = [[1, 2], [3, 4]];  
matrix B = [[5, 6], [7, 8]];  
matrix C = A + B;

The symbol table would contain:  
A -> (2, 2)  
B -> (2, 2)  
C -> (2, 2)

This information allows the semantic checker to ensure that operations like A + B are dimensionally valid.

If an undeclared variable is used or a mismatched shape is encountered (e.g., multiplying a (2,3) matrix by a (2,2)), the analyzer raises a descriptive exception, halting the compilation.

**Purpose**: The semantic analyzer safeguards the logical integrity of the program by enforcing contextual rules based on variable declarations and matrix shapes.

## Code Generator (codegen.py)

Generates equivalent Python code using NumPy from the AST.

- Uses np.array, np.add, np.subtract, np.matmul, np.divide, .T, np.linalg.det

**Purpose:** Converts DSL operations to Python code.

## Compiler Driver (main.py)

This script ties all components together:

- Reads DSL input from sample.dsl  
- Tokenizes and parses it  
- Performs semantic checks  
- Generates output.py with valid NumPy code

**Purpose:** Automates the compilation process from DSL to Python.

## Streamlit Interface (app.py)

A graphical interface for DSL interaction. Users paste DSL code and view the generated Python code and its output.

- Displays code and output in a friendly format  
- Automatically runs Python code and shows variables like matrices and results

**Purpose**: Provides an easy-to-use web interface for compilation and testing.

## Output Python File (output.py)

This file is auto-generated by main.py. It contains:

- Python code for matrix definitions and operations  
- Print statements to show output values

**Purpose**: Acts as the final translated program from DSL.

## Automatically Generated Files

**- parsetab.py**: Created automatically by PLY to store parsing tables  
**- parser.out:** Stores parser debugging and grammar rule information

**Purpose**: Internal files used by the PLY parser engine.

# Requirements File (requirements.txt)

Lists Python packages required:

**- ply  
- numpy  
- streamlit**

**Purpose:** Ensures consistent environment setup.

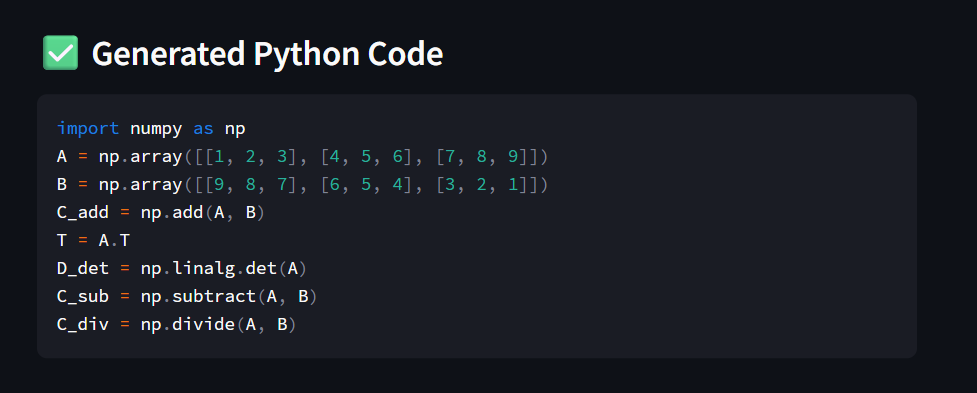
## README (README.md)

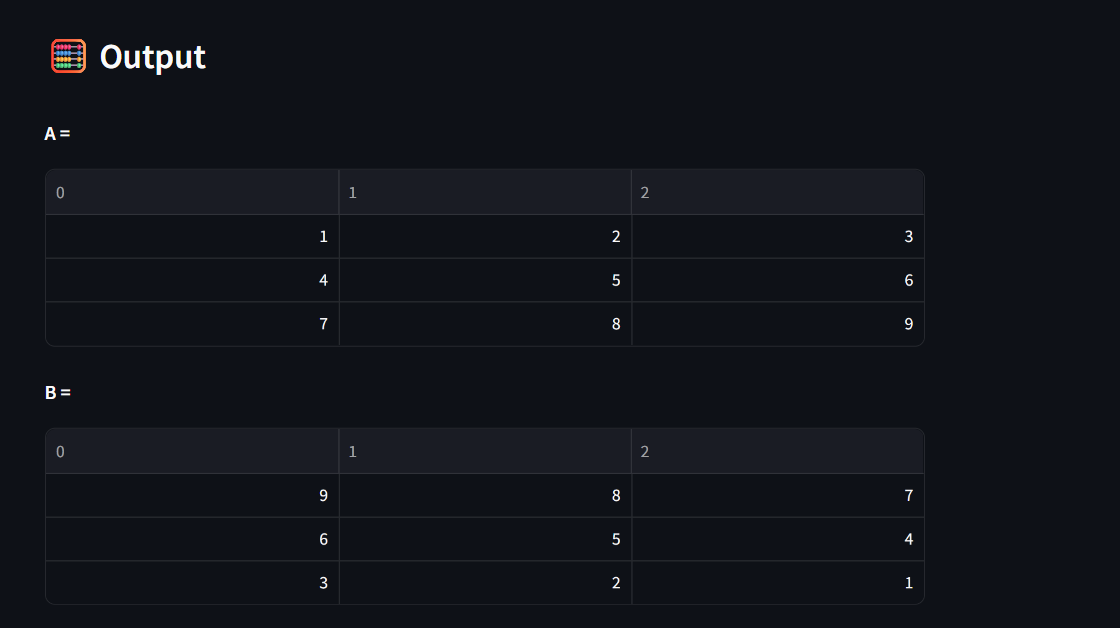
Describes the project, setup instructions, and usage guide.

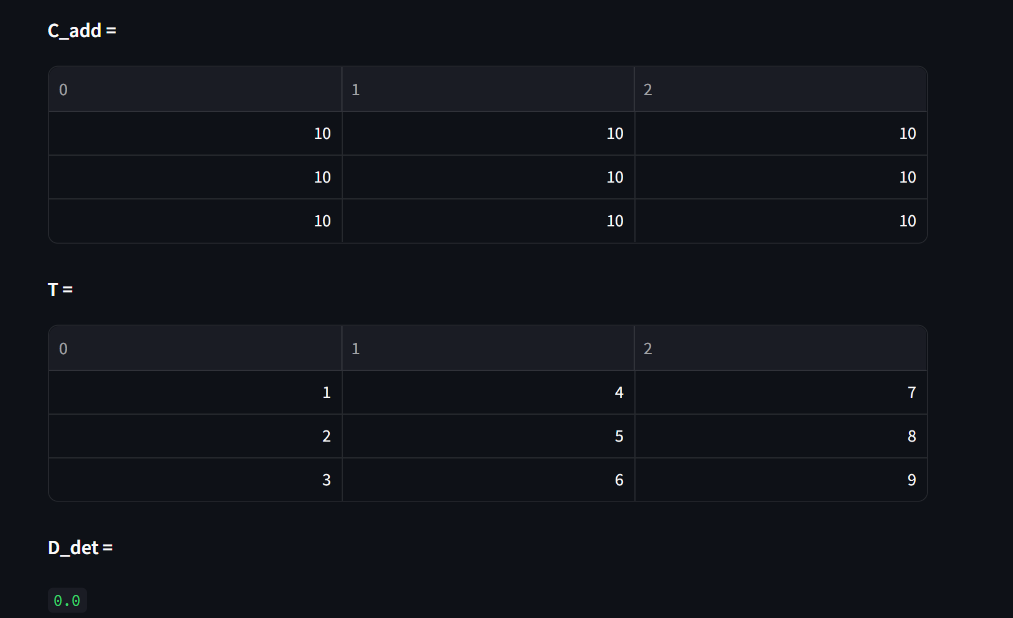
**Purpose:** Helps users understand and use the compiler.

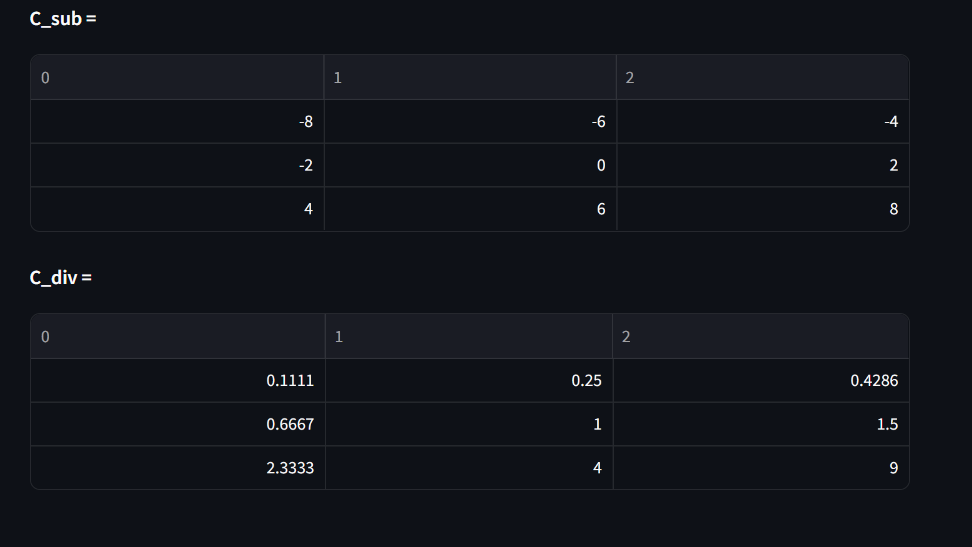
## Streamlit Interface Output Screenshot











## 15.File Structure Screenshot

