WizUALL Compiler: Design Report

Pranjay Yelkotwar

April 28, 2025

Abstract

This report presents the design of the WizUALL compiler, a toolchain for a data visualization programming language developed for a Compiler Construction course project. WizUALL processes programs with vector arithmetic, assignments, conditionals, loops, external function calls, and visualization primitives, alongside data streams extracted from PDFs, to generate Python code that produces visualizations. The design emphasizes modularity, flexibility, and leverages Python's visualization libraries (Matplotlib, Seaborn, Scikit-learn). This document covers the input-output relationship, program structure, syntax, semantics, and implementation plan, fulfilling the Design stage requirements.

1 Introduction

WizUALL is a domain-specific language for data visualization, designed to support flexible input and output formats. The toolchain includes a preprocessor, a compiler, and a runtime environment, targeting Python as the output language due to its robust visualization ecosystem. This report details the design choices for the WizUALL compiler, submitted for the Design stage (due March 31, 2025) as per Table 5.1 in the course specifications.

Please note that any earlier report submitted for this project should be disregarded, as it was prepared when I had a limited understanding of the project requirements and scope. This report reflects a comprehensive and accurate design based on the current implementation and test cases.

The toolchain adapts the schematic in Figure 5.1 of the course notes, combining preprocessing and compilation for simplicity while generating Python code that directly produces visualizations. Key features include support for vector arithmetic, six visualization primitives, and a modular architecture. The design prioritizes user accessibility, robust error handling, and organized output storage.

2 Input-Output Relationship and Pipeline

The WizUALL toolchain processes two inputs:

- **WizUALL Program**: A '.wiz' file containing code with vector arithmetic, assignments, conditionals, loops, external function calls, and visualization primitives.
- **Data Stream**: A PDF file with labeled numeric data, extracted into a comma-separated stream (e.g., 'x,1.0,2.0,3.0').

The pipeline comprises: 1. **Preprocessing**: Extracts data from PDFs using PyPDF2, producing a text stream. 2. **Compilation**: The PLY-based compiler tokenizes and parses the ‘.wiz‘ file, performs semantic analysis, and generates Python code. 3. **Execution**: The Python code executes, producing visualizations as ‘.png‘ files in the ‘outputs/‘ directory.

Example:

- Input ‘.wiz‘:

```
x_coords = [1, 2, 3, 4, 5]
y_coords = [2, 3, 5, 7, 6]
histogram(x_coords, 5)
scatter(x_coords, y_coords)
plot(x_coords, y_coords)
```

- Input PDF: Contains ‘x,1.0,2.0,3.0,4.0,5.0‘ and ‘y,2.0,3.0,5.0,7.0,6.0‘.

- Output: ‘outputs/output.py‘ (Python code) and ‘outputs/histogram.png‘, ‘outputs/scatter.png‘, ‘outputs/plot.png‘.

The pipeline supports auxiliary code blocks and direct data stream integration, enhancing flexibility.

3 Program Structure

A WizUALL program includes:

- **Statements**: Assignments, conditionals, while loops, function calls, and visualization primitives.

- **Auxiliary Code**: Enclosed in ‘,* ... ,*‘, copied verbatim to the output Python code.

- **Delimiters**: ‘=‘, ‘‘, ‘()‘, ‘[]‘, ‘,‘, ‘;‘.

- **Entities**: Identifiers (variables), expressions (arithmetic, vectors), and visualization outputs.

Programs are statement sequences with global lexical scope. Auxiliary code enables custom Python functions, bypassing compiler parsing.

4 Syntax

The WizUALL grammar extends the Calcilist grammar (Section 1.3), incorporating subtraction, division, unary minus, and identifier factors.

4.1 Grammar

```
program ::= statement_list
statement_list ::= statement_list statement | statement
statement ::= assignment | conditional | while_loop | function_call |
    auxiliary | expression
assignment ::= ID "=" expression
expression ::= expression "+" term | expression "-" term | term
term ::= term "*" factor | term "/" factor | factor
factor ::= NUM | ID | "(" expression ")" | "-" factor | "[" list_elements "]"
list_elements ::= list_elements "," expression | expression
conditional ::= "if" "(" expression ")" "{" statement_list "}"
              | "if" "(" expression ")" "{" statement_list "}" "else" "{"
                  statement_list "}"
while_loop ::= "while" "(" expression ")" "{" statement_list "}"
function_call ::= ID "(" arg_list_opt ")"
arg_list_opt ::= arg_list | empty
arg_list ::= expression | expression "," arg_list
auxiliary ::= ",*" aux_code ",*"
aux_code ::= /* arbitrary text */
empty ::= /* epsilon */
```

4.2 Descriptions

- **NUM**: Matches numbers (e.g., ‘0‘, ‘1.5‘, ‘-2‘) via regex ‘-?+(
- **ID**: Matches identifiers (e.g., ‘x$_c oords$‘, ‘$histogram$‘)$viaregex$‘$[a-zA-Z_]$$[a-zA-Z_0-9]*$‘.**Operators** : ‘ + ‘, ‘ − ‘, ‘ ∗ ‘, ‘/‘, $unaryminus$(‘ − ‘)$withprecedence$.

- **Auxiliary Code**: Unparsed, flanked by ‘,*‘.
- **Visualization Primitives**: Function calls (e.g., ‘scatter(x, y)‘).

5 Semantics

WizUALL semantics support data visualization with vector operations and modularity.

5.1 Vector Arithmetic

Extends Calcilist with:

- Operations: ‘+‘, ‘-‘, ‘*‘, ‘/‘, unary minus.
- Mapped to NumPy: ‘x + y‘ becomes ‘np.add(x, y)‘ for vectors.
- Example: ‘z = x * 2‘ yields ‘z = np.multiply(x, 2)‘.

5.2 Scope Rules

- Global lexical scope for identifiers.
- Unassigned identifiers default to ‘0‘ (scalar).
- Assignments overwrite existing values.

5.3 Statements

- **Assignments**: Bind expressions to identifiers.
- **Conditionals**: ‘if (E) B [else B]‘ evaluates ‘E‘ (non-zero is true).
- **While Loops**: ‘while (E) B‘ executes while ‘E‘ is true.
- **Function Calls**: Translated to Python calls, with vector arguments using Python list syntax.

5.4 Visualization Primitives

Six primitives, implemented via Python libraries:

- **histogram(data, bins)**: Matplotlib histogram with ‘bins‘ bins.
- **scatter(x, y)**: Matplotlib scatter plot.
- **plot(x, y)**: Matplotlib line plot.
- **reverse(data)**: Reverses vector.
- **slice(data, start, end)**: Extracts sub-vector.
- **cluster(data)**: K-means clustering (2 clusters) via Scikit-learn.

Visualizations are saved as ‘.png‘ files in ‘outputs/‘.

5.5 Auxiliary Code

Copied verbatim, enabling custom functions (e.g., ‘,* def custom$_{f}unc(x) : returnnp.sin(x), *$‘).

6 Design Choices

- **Python Target**: Selected for its visualization libraries, simplifying primitive implementation.
- **PDF Preprocessing**: Used PyPDF2 for flexible data extraction.
- **Visualization Primitives**: Chose six diverse primitives to meet requirements while leveraging existing libraries.
- **Modularity**: Organized into ‘preprocessor/‘, ‘compiler/‘, ‘runtime/‘ for maintainability.

- **Output Organization**: Visualizations saved in ‘outputs/‘ to keep project root clean.
- **Error Handling**: Detailed syntax error messages for user feedback.