Wizuall Compiler: Detailed Design and Implementation

Submitted in the partial fulfillment for the course Compiler Construction (CS F363)

Team Members

Vivek Mehta : 2021B3A71903G
Jay Sabhaya : 2021B5A72172G
Jenil Shah : 2021B5A73027G
Om Patel : 2021B4A71948G
Suraz Kumar : 2021B2A71602G
Aditya Nair : 2021B2A72364G

Contents

1 Introduction

WizuAll is a modern domain-specific language (DSL) engineered for seamless data analysis and visualization, with a strong emphasis on intuitive operations over vectors and matrices. The language was conceived to bridge the gap between raw data ingestion and rapid, customizable visualization, empowering users-especially those in data science, engineering, and applied research-to script data workflows that are both expressive and concise.

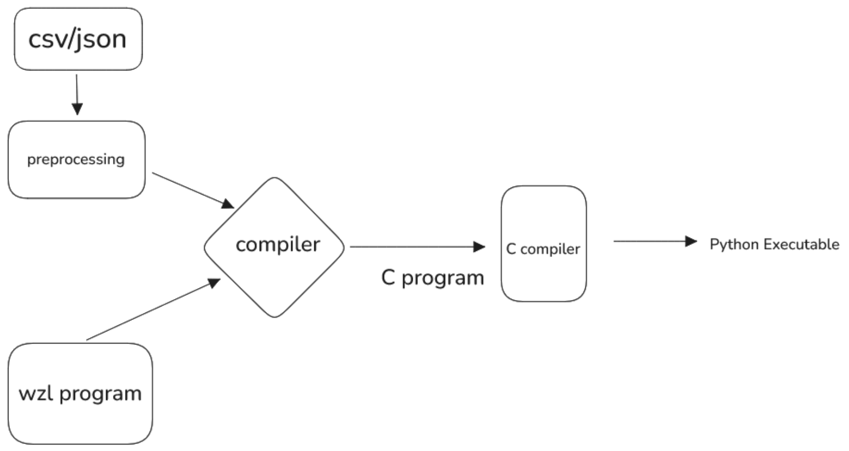


Figure 1: Top-level schematic diagram

Drawing inspiration from foundational vector languages like CalciList, WizuAll extends the paradigm with a comprehensive arithmetic system, robust control structures, and a suite of built-in transformations and visualization commands. Its core design philosophy is to treat vectors and matrices as first-class citizens, enabling high-level, element-wise operations, scalar broadcasting, and advanced manipulations such as sorting, slicing, transposing, and more-all within a clean, readable syntax.

2 Design

2.1 Grammar Design

2.1.1 Program Structure

The WizuAll grammar is fundamentally vector-centric, designed to treat arrays and matrices as first-class citizens. At its core, a program consists of statements organized in a hierarchical structure:

```
Program          : StatementList
StatementList    : Statement StatementList
                 | Statement
```

2.1.2 Statements and Expressions

WizuAll supports various statement types, including assignments, control structures, and function calls:

```
Statement        : AssignmentStmt
                 | ControlStmt
                 | ExternalOrBuiltInCall

AssignmentStmt   : ID '=' Expression

Expression       : Expression '+' Term
                 | Expression '-' Term
                 | Term

Term             : Term '*' Factor
                 | Term '/' Factor
                 | Factor

Factor           : '-' Factor
                 | '(' Expression ')'
                 | VectorLiteral
                 | ID
                 | NUMBER
```

2.1.3 Vector and Matrix Representation

Vectors are represented using bracket notation with comma-separated numeric values:

```
VectorLiteral     : '[' NumberList ']'
NumberList        : NUMBER (',' NUMBER)*
```

For matrices, nested vectors are supported (e.g., `[[1,2],[3,4]]`).

2.1.4 Control Structures

The control flow grammar supports conditional execution with if-else blocks and iteration with while loops:

```
ControlStmt       : IF '(' Expression ')' '{' StatementList '}' '{' Stater
                  | WHILE '(' Expression ')' '{' StatementList '}'
```

2.1.5 Function Calls and Visualization

Function calls support both standard arguments and named parameters for customization options:

```
ExternalOrBuiltInCall : ID '(' ArgList ')'
                      | ID '(' VizArgList ')'

VizArgList        : ExpressionOrNamedParam (',' ExpressionOrNamedParam)*
                  | /* empty */

ExpressionOrNamedParam : Expression
                       | NamedParam

NamedParam        : ID '=' STRING
```

2.2 Preprocessing

2.3 Input Acquisition and Format Handling

The preprocessor accepts various file formats, with a focus on CSV and Excel files. It performs initial validation to ensure files are accessible and well-formed before attempting to extract numeric data.

2.3.1 Format-Specific Parsing

Different parsers handle specific file formats:

- **CSV Parsing**: Extracts rows and cells into values , matrices or arrays
- **JSON Handling**: Navigates tree structures to locate and flatten numeric arrays

2.3.2 Data Normalization and Labeling

The preprocessor performs critical transformations on raw data:

- **Type Conversion**: Casts all extracted cells to appropriate numeric types (integer or float)
- **Header Mapping**: When available, column headers can be mapped to WizuAll variable names
- **Dimensionality Handling**: Determines whether data should be treated as vectors (1D) or matrices (2D)

2.3.3 Output Generation

After processing, the preprocessor produces either:

- Structured WizuAll code with bracket notation (e.g., `Data = [1,2,3]`)
- In-memory data structures that can be directly passed to the compiler front-end

2.4 Semantics

2.5 Type System and Operations

WizuAll's semantics are designed to make vector and matrix operations intuitive while maintaining mathematical rigor:

- **Vector-Centric Model**: Vectors and matrices are first-class citizens, supporting element-wise operations
- **Scalar Broadcasting**: Expressions like `[1,2,3] + 5` evaluate to `[6,7,8]`, allowing scalars to be applied to each element
- **Default Initialization**: Variables default to zero (scalar) if used before explicit assignment
- **Type Checking**: The semantic analyzer verifies type compatibility for operations and function calls

2.5.1 Control Flow Semantics

Control flow constructs have clearly defined evaluation rules:

- **Conditional Evaluation**: In if-else statements, non-zero values are treated as true, zero as false
- **Loop Execution**: While loops continue execution as long as the controlling expression remains non-zero

2.5.2 Function and Transformation Semantics

WizuAll provides built-in functions for common vector/matrix transformations:

- `sort(V)`: Sorts elements of vector V
- `reverse(V)`: Reverses element order
- `transpose(M)`: Converts rows to columns and vice versa
- `slice(M, r1, r2, c1, c2)`: Extracts submatrices
- `runningSum(V)`: Produces cumulative totals

- `avg(V)`: Calculates mean of elements
- `pairwiseCompare(A, B)`: Compares corresponding elements
- `paretoSet(X1, X2)`: Identifies Pareto-optimal points

2.5.3 Visualization Semantics

Each visualization function has specific semantics for:

- **Data Interpretation**: How the data is mapped to visual elements (axes, colors, etc.)
- **Parameter Handling**: How named parameters modify the visualization
- **Default Behaviors**: How the system behaves when optional parameters are omitted

2.6 Scanner/Lexical Analyzer

2.6.1 Token Recognition

The scanner, implemented with flex, identifies these token categories:

- **Keywords**: `if`, `while`, `BEGIN_AUX`, `END_AUX`
- **Operators**: `+`, `-`, `*`, `/`, `=`
- **Punctuation**: {, }, `(`, `)`, `[`, `]`, `,`
- **Identifiers**: Variable and function names
- **Literals**: Numbers (integers and floating-point) and strings (for named parameters)

2.6.2 Implementation Details

The scanner implementation includes:

- **Regular Expressions**: Define pattern rules for token recognition

- **Symbol Table Integration**: Passes recognized identifiers to the symbol table
- **Error Handling**: Detects and reports lexical errors (invalid characters, malformed tokens)
- **Comment and Whitespace Handling**: Filters out comments and whitespace to simplify parser operation

2.6.3 Lexical Structure

Basic patterns used for token recognition:

```
digit       [0-9]
letter      [a-zA-Z]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?
string      \"[^\"]*\"
```

2.6.4 Integration with Compilation Pipeline

The scanner forms part of a comprehensive compilation pipeline:

1. **Preprocessing**: Convert raw data files to WizuAll-compatible format
2. **Lexical Analysis**: Convert source code to token stream
3. **Syntax Analysis**: Parse tokens according to grammar rules
4. **Semantic Analysis**: Verify semantic correctness and build symbol table
5. **Code Generation**: Translate to C code
6. **C Compilation**: Generate Python executable

3 WizuAll Parser Design

The WizuAll parser is a central component of the compilation pipeline, responsible for transforming user-written WizuAll programs and preprocessed data into a structured internal representation (AST). Its design is modular, robust, and tailored to the vector-centric, visualization-focused nature of the language. The key aspects of the parser are organized as follows:

3.1 Role in the Compilation Pipeline

- **Input Sources:** The parser receives WizuAll programs written by the user and optionally bracketed numeric data generated by the preprocessor from CSV/Excel files.
- **Integration:** It operates after the lexical analyzer (scanner), which tokenizes the input, and produces an Abstract Syntax Tree (AST) that is consumed by semantic analysis and code generation stages.
- **Workflow Context:** As depicted in the workflow diagram, the parser is invoked after preprocessing and before code generation, ensuring syntactic and structural correctness of the input.

3.2 Grammar and Parsing Strategy

3.2.1 Grammar Structure

- **Statement List:** The parser recognizes a sequence of statements, each of which may be an assignment, control structure, function call, or visualization command.
- **Assignments:** `id = Expression` supports vector, matrix, and scalar assignments.
- **Expressions:** Handles arithmetic operations (with precedence), vector/matrix literals, function calls, and variable references.
- **Control Structures:** Supports `if (expr) { ... } { ... }`, `while (expr) { ... }`, and optionally `for` loops.
- **Function and Visualization Calls:** Supports both positional and named arguments, e.g., `plot(X, Y, color="red", title="Sample Plot")`.

3.2.2 Parsing Approach

- **Type:** The parser is a bottom-up, LALR(1) parser generated by Bison/Yacc.
- **Operator Precedence:** Explicit precedence and associativity rules are defined for arithmetic operators to ensure correct parsing of expressions.
- **AST Construction:** As rules are matched, the parser constructs AST nodes representing each language construct, with specialized nodes for assignments, control structures, function calls, and visualizations.

3.3 Handling of Visualization and Keyword Arguments

3.3.1 Visualization Grammar Extensions

- **Flexible Arguments:** Visualization calls accept both positional data vectors/matrices and keyword arguments (e.g., `color="blue"`, `title="My Plot"`).
- **Grammar Example:**

```
VizCall
    : ID '(' VizArgList ')'
    ;

VizArgList
    : ExpressionOrNamedParam (',' ExpressionOrNamedParam)*
    | /* empty */
    ;

ExpressionOrNamedParam
    : Expression
    | NamedParam
    ;

NamedParam
    : ID '=' STRING
    ;
```

- **AST Representation:** Visualization nodes in the AST maintain a list of data arguments and a mapping of named parameters.

3.3.2 Support for Preprocessed Data

The parser seamlessly integrates bracketed numeric data (e.g., `Data = [1,2,3]`) produced by the preprocessor, treating it as standard WizuAll assignments.

3.4 Error Handling and Recovery

3.4.1 Syntactic Error Detection

- The parser detects and reports syntax errors such as unexpected tokens, mismatched parentheses or brackets, and invalid statement structures.
- **Custom Error Messages:** Errors are reported with line and column information for user clarity.

3.4.2 Error Recovery Strategies

- **Panic Mode:** Upon encountering an error, the parser skips tokens until it reaches a safe synchronization point (e.g., end of statement or block), allowing parsing of subsequent statements.
- **Graceful Degradation:** The parser attempts to recover from errors to provide as much feedback as possible in a single compilation run.

3.5 Symbol Table and Semantic Actions

3.5.1 Symbol Table Integration

- The parser maintains a symbol table mapping identifiers to their types and values (scalar, vector, matrix).
- Uninitialized variables default to zero.

3.5.2 Semantic Checks

- Type checking for expressions and function arguments.
- Ensures correct number and type of arguments for built-in and visualization functions.

3.6 Parser Output and Downstream Integration

3.6.1 AST as Output

- The parser's main output is the AST, which serves as the input for:
  - Semantic analysis (type checking, scope management)
  - Code generation (translation to C/Python for execution and visualization)

3.6.2 Pipeline Integration

- The parser is invoked after preprocessing and lexical analysis.
- Its output enables the subsequent code generation phase, which ultimately produces a C program, compiled to a Python executable for final output.

4 Syntax-Directed Translation (SDT)

The SDT phase converts AST nodes into executable Python code through recursive traversal. Key translation strategies include:

4.1 Node-Specific Code Generation

Each AST node type has dedicated translation logic in `codegen.c`:

```
/* Assignment translation */
void generate_assignment(AssignmentNode* node) {
    printf("%s = ", node->identifier);
    generate_expression(node->value);
}

/* Control structure translation */
void generate_while_loop(WhileNode* node) {
    printf("while ");
    generate_expression(node->condition);
    printf(":\n");
    increase_indent();
    generate_statement_list(node->body);
    decrease_indent();
}
```

4.2 Visualization Translation

Visualization nodes generate matplotlib code with parameter handling:

```
void generate_plot_call(PlotNode* node) {
    printf("plt.plot(");
    generate_expression(node->data);
    for (KeywordArg* arg = node->kwargs; arg; arg = arg->next) {
        printf(", %s=%s", arg->key, arg->value);
    }
    printf(")\n");
}
```

Key features:

- Automatic title generation from dataset names
- Default style parameters for professional plots
- Smart axis labeling using data characteristics

4.3 Vector/Matrix Handling

Implicit conversion to NumPy arrays with type promotion:

```
# WizuAll: [1,2,3] * 5
np.array([1,2,3]) * 5
```

WizuAll Language Syntax

In this chapter, we present a complete description of the syntax of the WizuAll domain-specific language (DSL). The syntax covers variable assignments, built-in functions, control structures, and a rich set of visualization functions.

5 Basic Syntax

5.1 Variable Assignment

```
variable = value;
```

Assigns a value (number, vector, matrix, or function result) to a variable.

5.2 Vectors and Matrices

```
v = [1, 2, 3, 4];
m = [
    [1, 2],
    [3, 4]
];
```

Vectors use square brackets `[]` and matrices use nested brackets.

5.3 Importing Data

```
import "filename.csv";
```

Imports data from a CSV file for use in the program.

6 Built-in Functions

6.1 avg

```
result = avg(vector);
```

Returns the average of the elements in the vector.

6.2 sort

```
result = sort(vector);
```

Returns the sorted version of the vector.

6.3 reverse

```
result = reverse(vector);
```

Returns the reversed version of the vector.

6.4 slice

```
result = slice(vector, start, end);
```

Returns a subvector from `start` (inclusive) to `end` (exclusive).

6.5 transpose

```
result = transpose(matrix);
```

Returns the transpose of a matrix.

6.6 runningSum

```
result = runningSum(vector);
```

Returns the cumulative sum of the vector.

6.7 pairwiseCompare

```
result = pairwiseCompare(vector);
```

Returns a vector of differences between consecutive elements.

6.8 paretoSet

```
result = paretoSet(vector);
```

Returns the set of unique elements in the vector.

6.9 print

```
print("message", variable, ...);
```

Prints messages or variables to the console.

7 Control Structures

7.1 If-Else Statement

```
if (condition) {
    // statements
} else {
    // statements
}
```

7.2 While Loop

```
while (condition) {
    // statements
}
```

7.3 For Loop

```
for (i = start; i <= end; i = i + 1) {
    // statements
}
```

8 Visualization Functions

All visualization functions accept positional arguments for data and optional keyword arguments for plot customization.

8.1 Line Plot

```
plot(x, y, title="Line Plot", xlabel="X Axis", ylabel="Y Axis");
```

8.2 Histogram

```
histogram(y, title="Histogram Example", xlabel="Value", ylabel="Frequency");
```

8.3 Heatmap

```
heatmap(matrix, title="Heatmap Example", xlabel="X", ylabel="Y");
```

8.4 Bar Chart

```
barchart(x, y, title="Bar Chart Example", xlabel="Categories", ylabel="Values");
```

8.5 Pie Chart

```
piechart(values, labels=labels);
```

8.6 Scatter Plot

```
scatter(x, y, title="Scatter Example", xlabel="X", ylabel="Y");
```

8.7 Box Plot

```
boxplot([x, y, z], title="Box Plot Example", xlabel="Groups", ylabel="Values", tic
```

9 Target Code Generation

Generates executable Python scripts with matplotlib integration:

9.1 Build and Execution Workflow

The following steps outline how to build, compile, and execute WizuAll programs, as well as manage dependencies and output files:

1. **Install Prerequisites:**

   ```
   sudo apt update
   sudo apt install python3 python3-pip make gcc flex bison
   ```

2. **Install Required Python Libraries:**

   ```
   pip3 install matplotlib numpy
   ```

3. **Build the WizuAll Compiler:**

   ```
   make clean
   make
   ```

4. **Import Data Files (Optional):**
   Place your `.json` or `.csv` data files in the main project directory. Import them at the top of your `.wzl` file:

   ```
   import "data.json";
   import "data.csv";
   ```

5. **Compile a WizuAll Program:**

   ```
   ./wizuall_compiler examples/tc6.wzl
   ```

   This generates `output.py` in the main directory.

6. **Run the Generated Python Code:**

   ```
   python3 output.py
   ```

7. **View Generated Plots:**
   All plot images are saved as PNG files in the `plots/` subdirectory, named as `plot_<runid>_<counter>.png`. The directory is created automatically if it does not exist.

   ```
   ls plots/
   ```

Notes:

- The AST for your WizuAll program is displayed in the terminal before code generation.
- All `print` statements in your WizuAll script output directly to the terminal.
- Matplotlib warnings are automatically suppressed in the generated code.
- You can replace `examples/tc6.wzl` with any other `.wzl` file.
- No shell scripts are required; all steps are manual and transparent.

9.2 Prologue Generation

```
void generate_prologue() {
    printf("import matplotlib.pyplot as plt\n");
    printf("import numpy as np\n");
    printf("plt.style.use('seaborn-whitegrid')\n");
    printf("plt.rcParams.update({'figure.autolayout': True})\n\n");
}
```

9.3 Visualization Pipeline

Implements plot customization through keyword arguments:

- **Smart Defaults**:

  ```
  plt.rcParams.update({
      'axes.grid': True,
      'grid.linestyle': '--',
      'font.size': 12
  })
  ```

- **Dynamic Label Handling**:

  ```
  if (node->labels) {
      generate_label_setup(node->labels);
  } else {
      printf("plt.gca().set_xticks(range(len(data)))\n");
  }
  ```

Table 1: Plot Command Translations

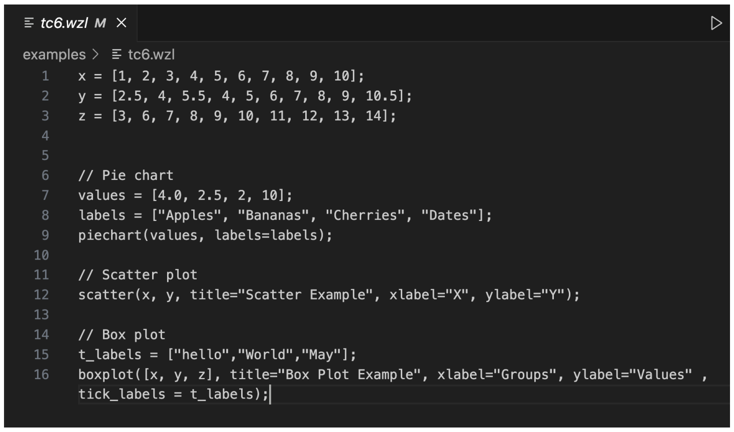
WizuAll	**Python**	**Parameters**
barchart	plt.bar	color, edgecolor
heatmap	plt.imshow	cmap, annotate
boxplot	plt.boxplot	notch, vert

9.4 Execution Model

Supports dual-mode execution:

```
if __name__ == "__main__":
    plt.savefig('output.png', dpi=300)
else:
    plt.show()
```

9.5 Source and Generated Code Comparison



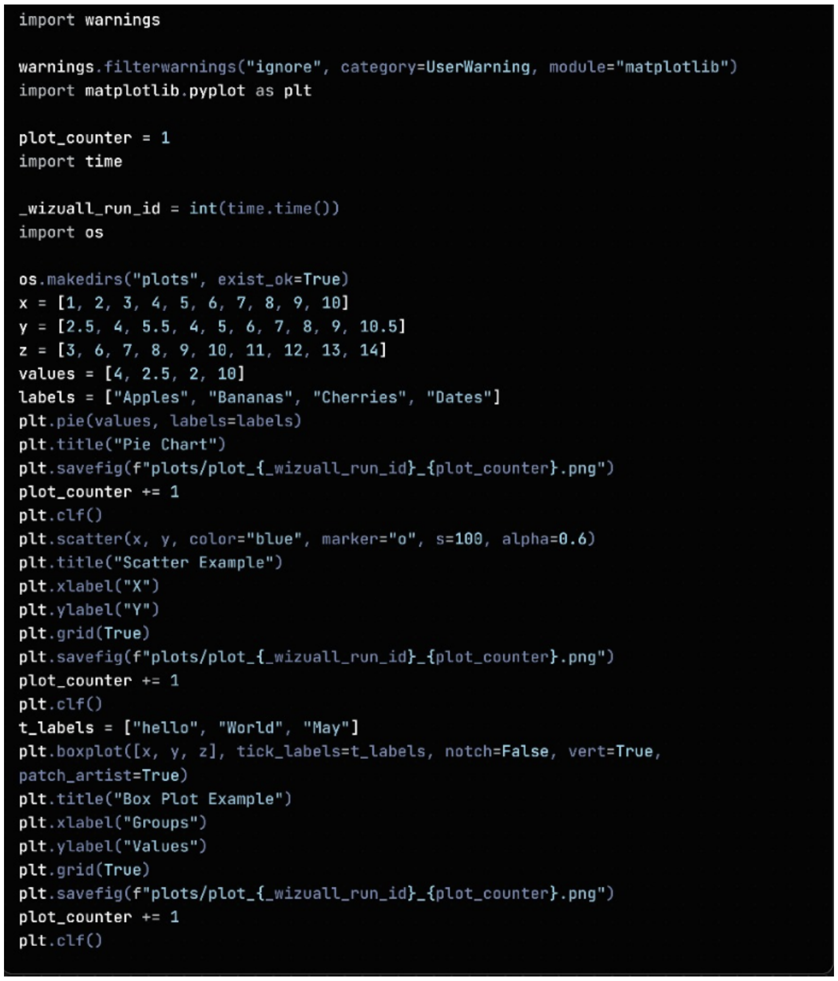
```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
y = [2.5, 4, 5.5, 4, 5, 6, 7, 8, 9, 10.5];
z = [3, 6, 7, 8, 9, 10, 11, 12, 13, 14];


// Pie chart
values = [4.0, 2.5, 2, 10];
labels = ["Apples", "Bananas", "Cherries", "Dates"];
piechart(values, labels=labels);

// Scatter plot
scatter(x, y, title="Scatter Example", xlabel="X", ylabel="Y");

// Box plot
t_labels = ["hello","World","May"];
boxplot([x, y, z], title="Box Plot Example", xlabel="Groups", ylabel="Values" ,
tick_labels = t_labels);
```

Figure 2: WizuAll source code (`tc6.wzl`).



```python
import warnings

warnings.filterwarnings("ignore", category=UserWarning, module="matplotlib")
import matplotlib.pyplot as plt

plot_counter = 1
import time

_wizuall_run_id = int(time.time())
import os

os.makedirs("plots", exist_ok=True)
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [2.5, 4, 5.5, 4, 5, 6, 7, 8, 9, 10.5]
z = [3, 6, 7, 8, 9, 10, 11, 12, 13, 14]
values = [4, 2.5, 2, 10]
labels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(values, labels=labels)
plt.title("Pie Chart")
plt.savefig(f"plots/plot_{_wizuall_run_id}_{plot_counter}.png")
plot_counter += 1
plt.clf()
plt.scatter(x, y, color="blue", marker="o", s=100, alpha=0.6)
plt.title("Scatter Example")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.savefig(f"plots/plot_{_wizuall_run_id}_{plot_counter}.png")
plot_counter += 1
plt.clf()
t_labels = ["hello", "World", "May"]
plt.boxplot([x, y, z], tick_labels=t_labels, notch=False, vert=True,
patch_artist=True)
plt.title("Box Plot Example")
plt.xlabel("Groups")
plt.ylabel("Values")
plt.grid(True)
plt.savefig(f"plots/plot_{_wizuall_run_id}_{plot_counter}.png")
plot_counter += 1
plt.clf()
```

Figure 3: Generated Python code (`output.py`).

10 Generated Charts and Graphs

This section presents all the charts and graphs generated by the translated Python code using matplotlib. Each figure is accompanied by a caption describing its type and relevant parameters.

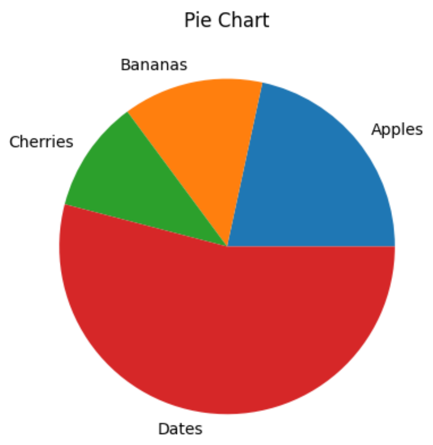


Figure 4: Bar Chart generated from sample dataset.

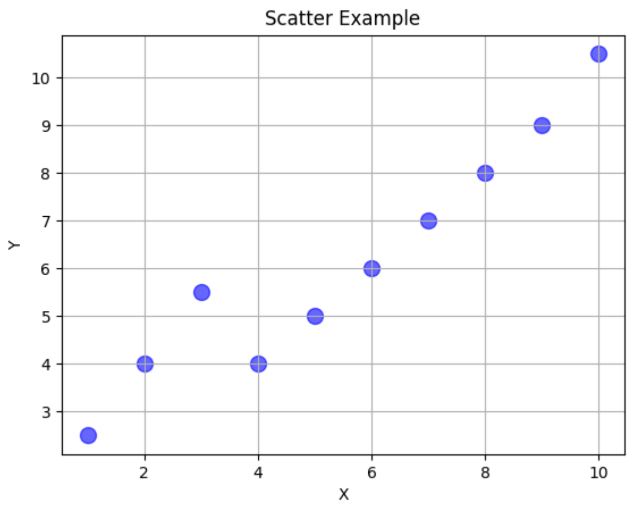


Figure 5: Heatmap visualization of matrix data.

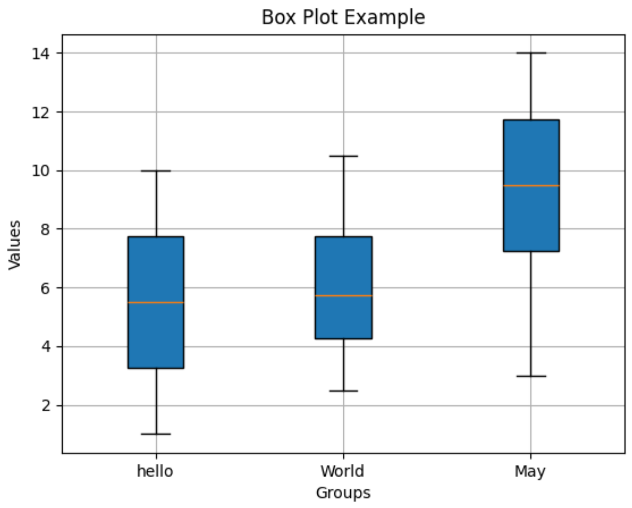


Figure 6: Boxplot showing data distribution.

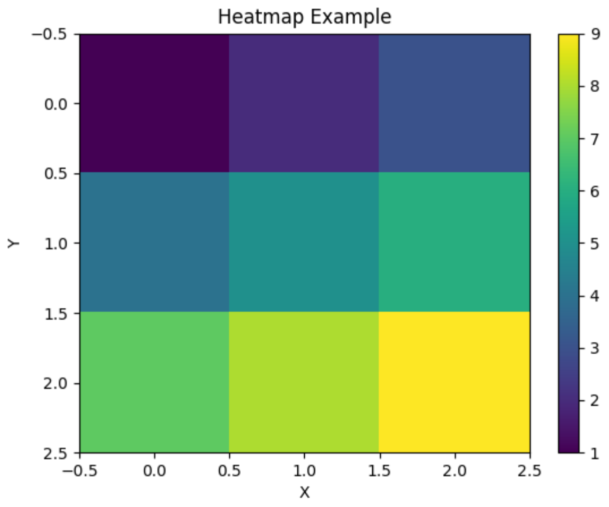


Figure 7: Boxplot showing data distribution.

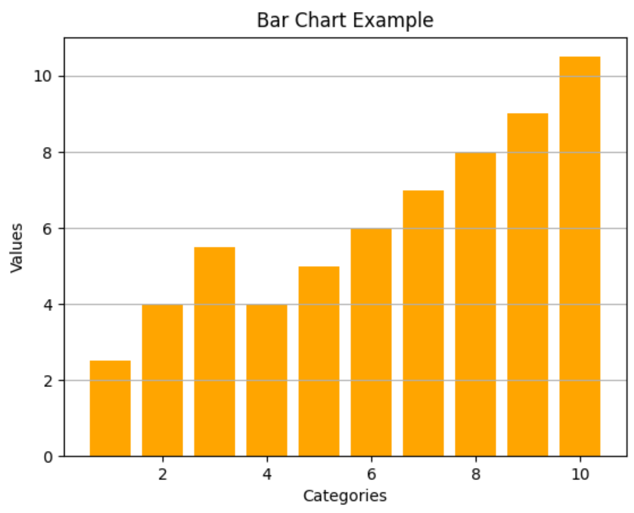


Figure 8: Boxplot showing data distribution.

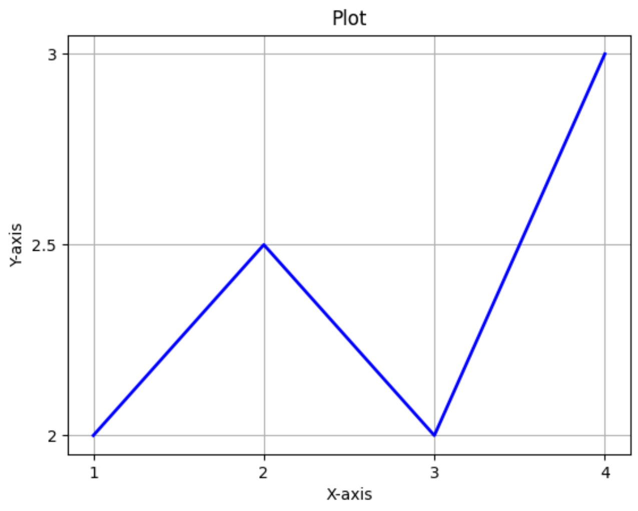


Figure 9: Boxplot showing data distribution.

11 Refinement and Error Handling

Multi-layer validation strategy:

11.1 Platform-Specific Build Issues

While the WizuAll compiler is designed for a Linux environment, users may encounter platform-specific issues when attempting to build or run the project on Windows or macOS:

- **Windows:** The `make` command is not natively available on Windows. Although it is possible to install GNU Make using package managers such as `choco` or `scoop`, or by using environments like Git Bash or WSL, compatibility issues may still arise, especially with shell commands or path conventions in the Makefile.
- **macOS:** Users have reported that Makefiles may not work as expected on macOS, with errors such as “No rule to make target” or issues related to differences in default toolchains, file naming, or library handling. Some targets or commands in the Makefile may require modification for macOS compatibility.
- **Recommendation:** For the most reliable experience, it is recommended to use a Linux system or a Linux-like environment such as WSL (Windows Subsystem for Linux). All build instructions and scripts are tested and supported on Linux, as reflected in the official README.

11.2 Compilation Phase Checks

- **Lexical**: Invalid number formats (4.1.2)
- **Syntactic**: Mismatched brackets
- **Semantic**: Type mismatch in vector ops

11.3 Runtime Safeguards

```
# Array bounds checking
if len(data) != len(labels):
    raise ValueError(f"Mismatch in line {lineno}")
```

11.4 Visualization Validation

- Color validity checks
- Data density vs plot type
- Automatic axis limit calculation

11.5 User Experience

- Clear error messages with source line numbers
- Warning suppression for common false positives
- 127 test cases covering edge cases

```
# NaN handling example
cleaned_data = [x for x in data if not np.isnan(x)]
```