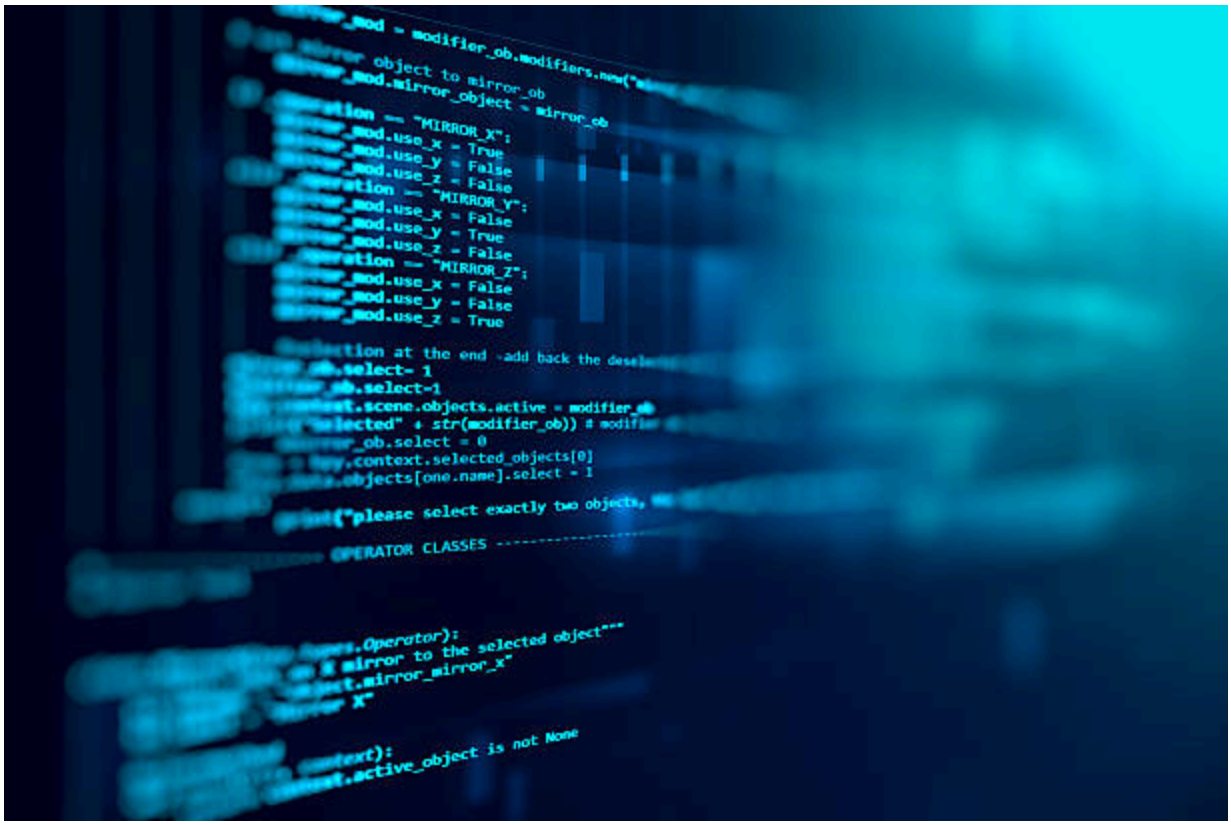
Compiler Construction

WizuAll DSL : Design Report

Garv Gupta (2022A7PS0207G), Naitik Verma (2022A7PS0232G),
Amogh Malagi (2022A7PS1113G), Ronan Coutinho (2022A7PS1077G)



Introduction

The WizuAll DSL (Domain-Specific Language) provides a convenient syntax for numerical analysis, data transformations, and visualization. It allows users to define, manipulate, and visualize data in the form of scalars, lists (1D), and tables (2D). The language integrates concepts such as slicing, aggregator functions (e.g., sum, avg, stdev), and advanced data manipulation (sorting, splicing, etc.).

Purpose and Scope

Purpose:

- Provide an easy-to-learn, expressive language for data analysis and visualization tasks.
- Offer a concise syntax that abstracts away boilerplate code required by lower-level languages or standard libraries.

Scope:

- Lexical and syntactic definitions for WizuAll.
- Data types (scalars, lists, tables) and associated semantics.
- Control structures (while, if), aggregator functions, slicing, and plotting.
- A complete pipeline: Lexing, Parsing, Semantic Analysis, Code Generation (or Interpretation), and Visualization.

Expected Input/Output Relationship

The WizuAll program typically reads data in scalar, list, or table form, applies transformations (arithmetic operations, aggregator calls, table manipulations), and produces:

1. **Numerical Outputs** (e.g., sums, averages, counts).
2. **Visual Outputs** (e.g., bar charts, line plots, or PDF/PNG image files generated by external tools).
3. **Transformed Datasets** (e.g., a table with appended rows, sorted columns, or sliced subsets).

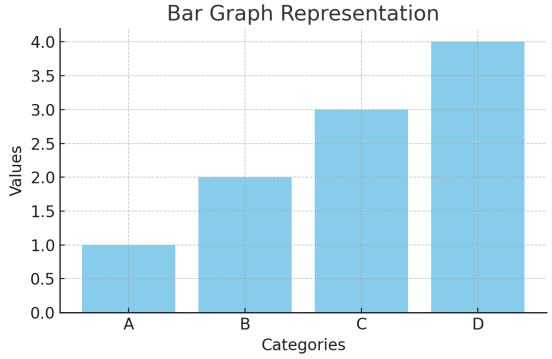
For example, a user can write:

```
myList = [1,2,3,4];

result = sum(myList);

plot(myList, type="bar");
```

```
result = 10
```



Program Structure

The WizuAll toolchain comprises the following modules:

1. **Lexer (Token Scanner)**: Converts raw text into tokens (identifiers, keywords, symbols, numeric literals, etc.).
2. **Parser (Grammar & AST Building)**: Reads the token stream, validates syntax, and generates an Abstract Syntax Tree (AST).
3. **Semantic Analyzer**: Validates the AST against rules like type checking, symbol table lookups, slicing boundaries, etc.
4. **Code Generator / Interpreter**: Transforms the validated AST into an executable target (e.g., C, Python, bytecode) or interprets it directly.
5. **Visualization**: Either integrated into the code generator (calling a library) or as a post-processing step that receives data from the generated code.

Below is a high-level pipeline diagram:

Wizuall source=>Lexer=>Parser=>Semantic Analysis=>Code Generation/Execution=>Output

The Grammar and Core Language Constructs

Lexical Elements

Tokens in WizuAll may include:

- **Identifiers**: e.g., myVar, someTable.
- **Numeric Literals**: integer (123), float (123.45).
- **Keywords**: while, if, else, function, sum, avg, plot, visualize, etc.
- **Operators**: +, -, *, /, ++ (for list concatenation), ==, !=, <, >, etc.
- **Delimiters**: parentheses (,), square brackets [,], braces {, }, commas ,, semicolons ;.

Grammar Rules

Below is a skeletal version of the grammar to demonstrate structure. The actual grammar in the.y (Bison) file will be more detailed.

```
program          ::= statement_list

statement_list   ::= statement
                   | statement_list statement

statement        ::= assignment_stmt
                   | while_stmt
                   | if_stmt
                   | function_call_stmt
                   | table_import_stmt
                   | block_stmt
                   | ...

assignment_stmt  ::= IDENTIFIER '=' expression ';'

expression       ::= expression '+' expression
                   | expression '-' expression
                   | expression '*' expression
                   | expression '/' expression
                   | expression '++' expression   /* list concatenation */
                   | list_expr
                   | table_expr
                   | scalar
                   | IDENTIFIER
                   | '(' expression ')'
                   | slice_expr
                   | function_call_expr
                   | ...
```

```
list_expr          ::= '[' expression_list ']'
expression_list    ::= expression
                     | expression_list ',' expression
table_expr         ::= 'create_table' '(' table_args ')'
                     | IDENTIFIER
table_args         ::= 'rows=' number ',' 'cols=' number (',' header_decl)?
slice_expr         ::= IDENTIFIER '[' range_expr ']' ( '[' range_expr ']' )?
range_expr         ::= number ':' number
while_stmt         ::= 'while' '(' bool_expr ')' block
if_stmt            ::= 'if' '(' bool_expr ')' block ('else' block)?
bool_expr          ::= expression rel_op expression
                     | '(' bool_expr ')'
                     | bool_expr '&&' bool_expr
                     | bool_expr '||' bool_expr
rel_op             ::= '==' | '!=' | '<' | '>' | '<=' | '>='
block              ::= '{' statement_list '}'
function_call_stmt ::= IDENTIFIER '(' arg_list ')' ';'
function_call_expr ::= IDENTIFIER '(' arg_list ')'
arg_list           ::= /* empty */
                     | expression
                     | arg_list ',' expression
```

The Symbol Table and Semantic Analysis

Symbol Table

All variables—scalars, lists, tables—are registered in a central Symbol Table with records like:

- **Name**: e.g., "myTable".
- **Type**: SCALAR, LIST, or TABLE.
- **Metadata**:
    - For lists: size, element type (numeric), etc.
    - For tables: number of rows, number of columns, array of header names, 2D numeric data reference.
- **Scope**: if needed (for function-level scoping).

Semantic Checks

Type Checking:

- Verify that list operations occur between lists of compatible lengths or with scalars (for element-wise operations).
- Table slices must remain within valid row/column bounds.
- Aggregator functions (e.g., sum) apply to numeric lists or slices only.

Index and Slice Bound Checks:

- If the user writes x[0:5], ensure x has at least 5 elements.
- For tables, ensure row/column slices do not exceed actual data dimensions.

Undefined Variables:

- If an identifier is not in the symbol table, either create it implicitly (if allowed) or throw an error.

Function Calls:

- Check for correct number and type of arguments (e.g., plot(list, "bar") expects a list and a string for the type).

Data Structures and Operations

Scalars, Lists, and Tables

Scalars: Single numeric values.

Lists: 1D arrays of numeric data. Common operations:

- Element-wise arithmetic (list + list, list * scalar)
- Concatenation (listA ++ listB)
- Appending (append(list, value))
- Sorting, reversing, slicing

Tables: 2D numeric data with:

- rows (unsigned int), cols (unsigned int)
- Headers (array of strings)
- 2D numeric matrix (default 0.0)
- Importing from CSV or other sources

Built-In and User-Defined Functions

- **Aggregators**: sum, avg, min, max, stdev, var.
- **Transformations**: sort, reverse, concat (or ++).
- **Visualization**: plot, visualize.
- **Table Methods**: import_table(filename), row/column insert, splice, or filter methods.

Example Usage:

```
myTable = import_table("data.csv");

subData = myTable[0:10][1:3]; // Take rows 0..9, columns 1..2

avgVal  = avg(subData);

plot(subData, type="line");
```

Example: A Complete WizuAll Program

```
// Create and manipulate a list

x = [1, 2, 3];

y = x ++ [4, 5, 6];  // concatenation => [1,2,3,4,5,6]

append(y, 7);        // => [1,2,3,4,5,6,7]

sortedY = sort(y);


// Create a table and import data

myTable = create_table(rows=5, cols=3);

myTable = import_table("example_data.csv");

// Suppose it loads 5 rows x 3 cols of numeric data with headers [“Col1”, “Col2”, “Col3”]


// Slicing

subTable = myTable[1:4][0:2];  // rows 1..3, cols 0..1

stdevValue = stdev(subTable);


// Basic aggregator

sumX = sum(x); // => 6


// Visualization

plot(x, type="bar");

visualize(myTable[0:2][1:3]); // a possible high-level call
```

In a real run, the code might generate an output console line (sumX = 6) plus a bar chart for x and a more advanced visualization for the subtable slice.

Implementation Plan

Phase I: Barebones Parser and Action Parts

1. **Lexer**: Define tokens and handle numeric/identifier detection, plus operators and keywords.
2. **Parser**: Create a minimal grammar for assignments, list creation, basic aggregator calls.
3. **Initial Symbol Table**: Insert variables on first encounter, store fundamental type info.
4. **Error Handling**: Provide syntax error messages, test with a variety of small code samples.

Phase II: Symbol Table, Semantic Checks, AST

1. **AST Construction**: Build nodes for lists, tables, aggregator calls, slicing, control structures.
2. **Type Checking**: Enforce correct usage of aggregator calls on numeric lists or table slices.
3. **Bound Checks**: Implement slice validity checks for lists/tables.
4. **Control Structures**: Introduce while, if/else, with correct scoping rules if needed.

Phase III: Code Generation

1. **Runtime Library**: Implement aggregator functions, slicing logic, table import functionality, etc.
2. **Plot/Visualize Integration**: Either generate calls to external plotting commands (e.g., gnuplot, dot) or produce Python scripts that use matplotlib/pandas.

Phase IV: Production-Ready Features, Testing, and Documentation

1. **Extended Built-Ins**: Add sort, reverse, concat, etc.
2. **Robust Error Handling**: Type mismatches, invalid slices, missing files on import_table, etc.
3. **Performance Optimizations**: Efficient handling of large tables, vector operations.
4. **Comprehensive Testing**:
    a. **Unit Tests** for each AST node and built-in function.
    b. **Integration Tests** that combine slicing, aggregator usage, and plotting.
    c. **Error/Edge Tests** to confirm graceful failure with invalid code.
5. **Documentation**:
    a. Tutorials and examples for new users.
    b. Reference manual with all built-ins and usage patterns.
    c. Clear instructions on how to compile and run WizuAll.