WizuAll Compiler Project Report

ANKITA VAISHNOBI BISOI (2021B1A72306G)
TANISHQ JAIN (2021B3A72941G)

Overview

This report details the development of the WizuAll compiler, a language processing tool designed for data visualization. The compiler successfully translates WizuAll programs into executable code, primarily targeting Python while maintaining support for C and R languages. The project meets the course requirements by implementing a complete compiler toolchain that separates data and code while providing powerful visualization capabilities.

Scope

The WizuAll compiler aims to simplify data visualization tasks by providing a domain-specific language that abstracts away the complexities of underlying visualization libraries. Its primary purpose is to allow users to focus on what data they want to visualize rather than how to program the visualizations.

The scope encompasses:

- Processing of WizuAll language programs
- Data extraction from external sources
- Translation to target languages
- Generation of visualization code
- Support for basic programming constructs
- Implementation of vector operations and visualization primitives

The compiler is intended for educational and data analysis contexts where users need straightforward visualization capabilities without deep knowledge of graphics programming.

Input-Output Relationship

The WizuAll compiler establishes a clear relationship between inputs and outputs:

Inputs:

1. WizuAll source code (.wzl files) containing program logic and visualization instructions

2. External data sources (CSV files, PDFs) containing the numeric data to be visualized
3. Compiler configuration options (target language, output file, execution preferences)

Outputs:

1. Generated code in the target language (Python, C, or R)
2. Visualization artifacts (plots, charts, graphs) when the generated code is executed
3. Execution results and statistics from the WizuAll program
4. Compilation logs and error messages for debugging

This relationship follows the highest preference category from the assessment rubric, where data and code are completely separated, and visualizations are produced through the execution of generated target code.

Program Structure

The WizuAll compiler follows a modular architecture with distinct components handling specific phases of the compilation process:

1. **Source Code Processing**: This component handles the initial reading of WizuAll source files, preparing them for lexical analysis.

2. **Data Extraction and Processing**: Responsible for extracting numeric data from external files, this component transforms raw data into a format usable by the compiler and generated code.

3. **Lexical Analysis**: This module breaks the source code into tokens, identifying keywords, operators, identifiers, and literals according to the language grammar.

4. **Syntax Analysis**: The parser constructs an abstract syntax tree (AST) representing the hierarchical structure of the program based on the grammar rules.

5. **Semantic Analysis**: This component verifies type correctness, scope rules, and other semantic properties, ensuring the program's meaning is valid.

6. **Code Generation**: Responsible for translating the validated AST into executable code in the target language, incorporating visualization library calls.

7. **Runtime Execution**: The optional execution module compiles and runs the generated code, producing the actual visualizations.

The interconnected components form a pipeline, with each stage processing the output of the previous one to gradually transform the WizuAll source into executable visualizations.

```
wizuall-compiler/
├── preprocessor/
│   ├── __init__.py
│   ├── pdf_extractor.py
│   └── data_formatter.py
├── scanner/
│   ├── __init__.py
│   └── lexer.py
├── parser/
│   ├── __init__.py
│   └── parser.py
├── semantics/
│   ├── __init__.py
│   ├── symbol_table.py
│   └── semantic_analyzer.py
├── visual_primitives/
│   ├── __init__.py
│   └── viz_functions.py
├── runtime/
│   ├── __init__.py
│   └── executor.py
├── tests/
│   ├── test_scanner.py
│   ├── test_parser.py
│   └── test_semantics.py
│   └── test_code_generator.py
├── scripts/
│   ├── build.py
│   └── run_tests.py
├── main.py
└── README.md
```

Figure 1 : WizuAll project structure

The WizuAll compiler codebase follows a clear organizational structure, with modules grouped by functionality rather than implementation details.

- The main.py file serves as the entry point, orchestrating the entire compilation process
- Core modules are organized into distinct directories based on their role in the pipeline
- Supporting utilities and test files are kept separate from the main compilation logic
- Configuration files and documentation are maintained at the top level for easy access

This structure ensures that developers can quickly locate relevant code sections and understand how components interact with one another.

Chronological Development Process

Initial Design and Planning

We began by analyzing the requirements and establishing the WizuAll language specification based on the CalciList foundation. This phase involved:

- Defining the grammar for the language
- Planning the compiler architecture
- Establishing the visualization primitives to implement
- Determining the target language capabilities

Lexical Analysis Implementation

The first technical component developed was the lexical analyzer (**scanner**). This involved:

- Defining token types for the WizuAll language (identifiers, numbers, operators, keywords)
- Implementing tokenization logic to convert source text into a stream of tokens
- Adding support for special elements like comments and string literals
- Building error detection for invalid characters and token patterns

The scanner successfully identifies all language elements and prepares them for parsing.

Syntax Analysis Development

Next, we created the **parser** to analyze the grammatical structure of WizuAll programs:

- Implementing recursive descent parsing techniques for the defined grammar
- Building AST node classes to represent different program constructs
- Handling expressions with proper operator precedence
- Managing statements, control structures, and function calls
- Developing error reporting for syntax violations

The parser transforms the token stream into a structured representation that captures the program's hierarchical organization.

Semantic Analysis Implementation

With the AST in place, we developed the semantic **analyzer** to validate program meaning:

- Creating a symbol table to track variables and their properties
- Implementing type checking for operations and assignments
- Verifying that variables are used appropriately within their scope
- Adding support for vector operations with appropriate dimension checking
- Building error handling that allows recovery from semantic errors

This component ensures that while the program might be syntactically correct, it also makes logical sense and can be executed without semantic errors.

Code Generation Development

The **code generator** translates the validated AST into executable target code:

- Implementing visitor patterns to traverse the AST
- Creating templates for different target languages (Python, C, R)
- Developing visualization function mappings to appropriate library calls
- Handling variable declarations, operations, and control flow
- Managing appropriate indentation and code structure in the output

This component produces well-formatted, executable code that corresponds to the original WizuAll program.

Visualization Primitives Implementation

A significant development effort went into creating the **visualization primitives**:

- Implementing templates for basic plots (line, scatter, bar)
- Developing statistical visualization functions (histograms, heatmaps)
- Creating vector operation functions (averaging, maximums, minimums)
- Adding advanced analytics capabilities (clustering, classification)
- Ensuring cross-language compatibility for visualization functions

These primitives form the core value of the WizuAll language, allowing users to create sophisticated visualizations with simple commands.

Runtime Execution Implementation

The final core component was the **runtime executor**:

- Building mechanisms to compile generated code in the target language
- Implementing execution capabilities for the compiled code
- Adding support for capturing and displaying execution results
- Creating error handling for runtime issues
- Managing cleanup of temporary files and resources

This component allows for immediate visualization without requiring users to manually execute the generated code.

Testing and Refinement

Throughout development, we implemented comprehensive **testing**:

- Unit tests for individual components (scanner, parser, semantic analyzer, code generator)
- Integration tests for the complete compiler pipeline
- Test cases covering edge cases and error handling
- Sample program testing to verify real-world functionality

Refinements were made based on test results, particularly for handling comparison operators, string literals, and vector operations within function calls.

Implementation Details

The WizuAll compiler uses several implementation approaches to ensure reliable and maintainable code:

- For lexical analysis, we implemented a state machine approach that efficiently transitions between token recognition states. This provides both performance and clarity in the scanning process.
- The parser uses a recursive descent approach with predictive parsing, which aligns naturally with the grammatical structure of WizuAll. This choice simplifies the implementation while providing clear error messages when syntax violations occur.
- For the abstract syntax tree, we employed a composite pattern where each node type inherits from a base AST node class. This creates a uniform interface for tree traversal while allowing specialized behavior for different language constructs.
- The symbol table uses a hashmap-based implementation for O(1) lookups, with scope information encoded in the variable names. This approach balances simplicity with the needs of the WizuAll language's scoping rules.
- Code generation leverages the visitor pattern, where each AST node type implements a visitor interface that generates appropriate code for that construct. This creates a clean separation between the AST structure and the code generation logic.

Project Capabilities

1. **Data Processing**: The preprocessor can extract numeric data from various sources, including CSV files and PDFs, making this data accessible to WizuAll programs without embedding it directly in the code.
2. **Language Features**: The compiler supports core programming constructs including:

    - Vector operations (addition, subtraction, multiplication, division)
    - Variable assignments with appropriate scoping
    - Conditional statements (if-else)
    - Looping constructs (while)
    - Function calls to native visualization libraries
3. **Visualization Primitives**: The implemented visualization functions include:
    - Basic plots (line, scatter, bar)
    - Data distribution visualizations (histograms, heatmaps)
    - Statistical operations (averages, maximums, minimums)
    - Vector manipulations (reversing, products, comparisons)
    - Advanced analytics (clustering, classification)
4. **Multi-language Support**: The compiler can generate code in Python, C, or R, providing flexibility based on the target environment and visualization needs.

Assessment Evaluation

1. **IO Relationship** : The implementation achieves the highest preference category by completely separating data and WizuAll code. It generates target language code that performs visualizations after execution, demonstrating a clean pipeline architecture.

2. **Program Structure** : The compiler demonstrates a well-organized, modular structure with clear separation of concerns between components. The implementation follows established compiler design principles and maintains clean interfaces between stages.

3. **Syntax & Semantics** : The WizuAll language successfully shields users from the complexities of underlying graphics libraries while providing expressive visualization capabilities. The grammar is well-defined and the semantic rules are consistently enforced.

4. **Correctness** : The compiler correctly handles the specified language features and produces functioning target code. The comprehensive test suite helps ensure correctness across different inputs and edge cases.

5. **Completeness** : All required components are present and functional, from preprocessing to execution. The compiler implements all specified language features and exceeds the minimum requirement for visualization primitives.

6. **Visualization** : The project excels in visualization capabilities, providing numerous visualization functions that span basic plotting to advanced analytics. The generated code effectively utilizes target language visualization libraries.

Conclusion

The WizuAll compiler project successfully implements a complete language processing toolchain for data visualization. It meets or exceeds all the requirements specified in the course rubric, demonstrating particular strength in its modular design, visualization capabilities, and separation of data from code.

The compiler effectively abstracts away the complexity of underlying graphics libraries while providing users with powerful visualization tools. The implementation process revealed practical insights into compiler construction techniques and highlighted the importance of systematic testing and error handling.

This project serves as a functional demonstration of compiler design principles applied to the specific domain of data visualization, successfully bridging the gap between user-friendly programming and powerful visualization capabilities.