Grenze International Journal of Engineering and Technology, January Issue

Py++ (Python based C++ Approached Compiler with Win32 Editor and Debugger)

Prof. Sheetal Phatangare[1], Ishaan Yadav[2], Yash Maheshwari[3], Shriraj Yamkanmardi[4] and Vipra[5]
[1-5]Department of Computer Engineering, Vishwakarma Institute of Technology, Pune, India
Email: sheetal.phatangare@vit.edu, ishaan.yadav22@vit.edu, yash.maheshwari22@vit.edu, shriraj.yamkanmardi22@vit.edu, undefined.vipra22@vit.edu

***Abstract*— This paper presents the systematic design and development of a modular mini-interpreter framework that replicates core compiler functionalities such as lexical analysis, parsing, symbol table creation, data type handling, expression evaluation, and function processing. Using the powerful combination of Flex and Yacc for frontend parsing and C++ for backend memory and execution management, the system provides support for primitive data types, nested arrays, type conversion, and runtime variable/function handling. The project is structured across multiple modules, each introducing new capabilities in an incremental and testable manner. The resulting system not only serves as a prototype for language development but also as a powerful educational tool for understanding compiler construction concepts.**

***Index Terms*— Compiler Design, Interpreter, Symbol Table, Type Conversion, Expression Evaluation, Flex, Yacc, C++, Mini-Compiler, Nested Arrays, Function Parsing.**

I. INTRODUCTION

Modern software development relies heavily on robust compilers and interpreters to bridge high-level programming languages with machine-understandable instructions. While industrial-scale compilers such as GCC and Clang support large-scale applications and language features, the foundational concepts behind these tools remain complex for beginners. This paper aims to demystify these principles through the implementation of a modular mini-interpreter that supports essential language operations.

Our framework is divided into several stages, beginning with tokenization and grammar parsing using Flex and Yacc. These stages are followed by the implementation of variable handling, type management, expression parsing, and finally function handling. C++ serves as the runtime engine due to its low-level memory access capabilities and support for object-oriented design.

II. METHODOLOGY

A. Tokenization and Grammar Parsing Using Flex & Bison

Flex (lexer) identifies different types of words in the input using regular expressions and converts them into tokens such as identifiers, numbers, characters, strings, and keywords. Bison (parser) then uses grammar rules to decide whether these tokens form a valid statement, applying precedence rules to resolve ambiguities. Together, they validate the structure of a sentence by breaking it into components like subject, verb, and object, and checking whether it follows the defined grammar.

Grenze ID: 01.GIJET.12.1.149

B. C++ as the Execution Layer for Flex–Bison Parsing

Flex and Bison handle the frontend processing—tokenizing input and parsing it according to grammar rules—while C++ acts as the backend where the actual logic, data structures, and storage mechanisms are implemented. After Flex identifies tokens and Bison validates the syntax, C++ code (such as variable classes or symbol table structures) executes the actions associated with those grammar rules. This allows the language parser to not only recognize valid input but also store variables, manage types, and perform operations using C++ data structures and functions.

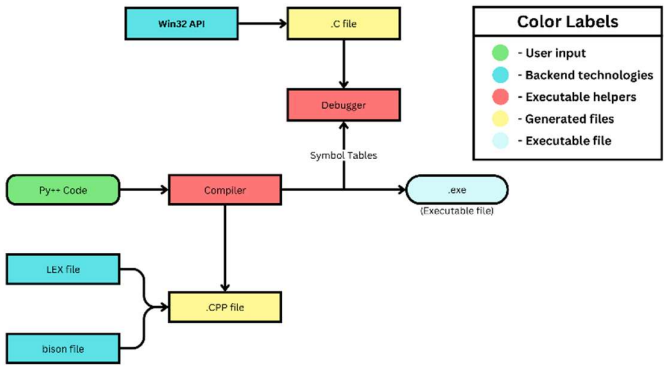

Figure 1 - System architecture and data flow

C. Arithmetic Expression Evaluation Using Flex and Bison –

Flex and Bison were extended to support arithmetic operations by defining grammar rules and helper functions to evaluate expressions. The parser handled operations such as addition, subtraction, multiplication, division, and modulus using a unified evaluation approach. In the grammar, the $ notation was used to access operands within rules (e.g., $1 and $3 for left and right operands). This allowed the parser to compute results for expressions like 9+6 or 9-8 and display the evaluated output.

```
expression: NUMBER '+' NUMBER   {$$=$1+$3;}
   |    NUMBER '-' NUMBER   {$$=$1-$3;}
   |    NUMBER             {$$=$1;}
```

D. Using Void Pointers for Python-Style Symbol Table Implementation

Void pointers enable a Python-like flexibility in C by allowing storage of values of any data type without predefined type declarations. Since a void* can hold the address of any variable—whether it is an integer, character, string, or even a complex object—it becomes an ideal choice for building a dynamic symbol table. By allocating memory for the required data type and storing its address as a void*, the symbol table can uniformly store variables of different types. Through type casting, values can later be retrieved and interpreted correctly. This approach allows C programs using Flex and Bison to emulate Python's dynamic typing within the symbol table.

E. Uniform Handling of Multiple Data Types Using Void Pointers and Variable Objects

The system treats all data types—integers, characters, arrays, and nested arrays—in a uniform way by representing every value as an object of the variable class and storing its actual data through a void* pointer. Since a void* can hold the address of any data type, the symbol table and array structures can store values of different types without separate implementations.

Each variable object stores:

- name → identifier
- type → (1 = int, 2 = char, 3 = vector/array)
- val → void* pointing to dynamically allocated memory for the actual value
- size → used for arrays
- next → symbol table linking

Type casting is used at both storage and retrieval:

- When storing, the actual data (int, char, vector) is cast to void* and assigned to val.
- When retrieving, val is cast back based on the type field to interpret the memory correctly.

Arrays are also stored as variable objects by allocating a vector<variable*> dynamically and storing its pointer in val through static_cast<void*>. Using a stack-based structure (StackArray), the system supports nested arrays by pushing inner arrays onto the stack, merging them at the correct time, and wrapping them again inside variable objects.
In the symbol table printing function, the process is reversed:

- If type == 1, val is cast to int* and printed.
- If type == 2, val is cast to char* and printed.
- If type == 3, val is cast to var** (array of variable pointers), and each element is printed by again checking its type.

Through this design—combining void*, type tags, and controlled type casting—the system achieves Python-like dynamic typing in C++, allowing all data types (including nested structures) to be handled uniformly, flexibly, and efficiently.

F. Handling Nested Arrays in the Symbol Table

Nested arrays require special handling during parsing because the parser must fully recognize an array before determining how to store it. While simple array elements are parsed in reverse order (due to right-recursive grammar rules), nested arrays retain their original order because they are treated as single composite units. The parser completes the full parsing of a nested array first and then pushes it onto the element stack, which prevents it from being reversed like primitive values.
In the symbol table, only the outer array is stored under the identifier. Each nested array is stored internally as a Variable of type ARRAY inside the parent array's structure (e.g., StackArr). No separate symbol table entries are created for inner arrays unless they have names. This allows natural hierarchical access such as a[1][0].

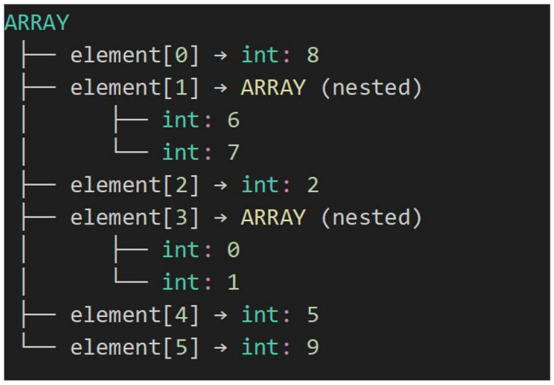

Fig 2

The ordering issue observed in simple arrays is resolved by modifying the grammar from right-recursive:
NUMBER ',' var_list
to left-recursive:
var_list ',' NUMBER
This enforces left precedence and causes array elements—both simple and nested—to be parsed in the correct forward order. Thus, nested array recognition, structure preservation, and integration into the symbol table are handled automatically and consistently.

G. Token Differentiation by the Lexer

In this module, complex expressions were evaluated using a recursive parsing strategy, but at the foundation, Flex (the lexer) is responsible for identifying each token by inspecting characters one by one. The lexer classifies input into categories—identifiers, numbers, operators, parentheses, strings, and other symbols—based purely on character patterns.
For example:
Alphabetic sequences ([a-zA-Z_][a-zA-Z0-9_]*) are recognized as identifiers (possible lvalues).
Digit sequences ([0-9]+) are recognized as numbers (rvalues in expressions).
Operators like +, -, *, /, = are matched as specific operator tokens.
Parentheses (and) direct expression grouping and precedence.
Any matched token is sent to the parser, which then interprets it according to the unified rule

IDENTIFIER = EXPRESSION, rather than separate rules for each type.
Thus, the lexer provides the parser with the correct token stream by distinguishing tokens entirely through character-level pattern matching, enabling generalized expression handling similar to Python.

H. How Functions and Loops Work Similarly at a Low Level

In this module, functions were implemented with support for parameters, local scope creation, and execution of a sequence of statements. At a low or assembly-like level, both functions and loops share the same fundamental mechanism:
they store a set of instructions and execute those instructions multiple times depending on the context.
A function definition stores:

- function name
- parameter list (identifiers with temporary/null values)
- body (a list of statements)
- a local symbol table that becomes active only during a function call

When a function is called, a new local scope is created, parameters are inserted, and then the saved body statements are executed line by line, just like a block of instructions.
This is essentially how loops operate at the assembly level as well:
a loop simply contains a block of statements, and the processor repeatedly executes this block until the loop condition fails.
Functions do the same — only instead of repeating based on a condition, they execute the saved statements once per call, using a fresh local symbol table each time.
Thus, both functions and loops rely on the idea of re-executing a stored sequence of instructions, making their core mechanism similar when viewed from an assembly-level or low-level execution standpoint.

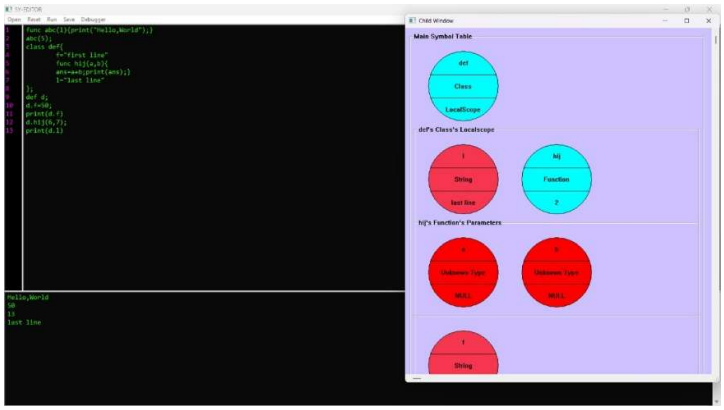

Figure 2 - Py++ editor and debugger

The figure 2 illustrates the developed programming environment consisting of the following:

- Lightweight Win32-Based Source Code Editor - The editor is a lightweight code-editing environment built using the Win32 interface. It provides basic IDE features such as syntax-coloured text input, line-by-line program writing, and command options for running and resetting the program. The editor displays user code in real time and acts as the primary interface for writing programs in the custom language. Its design focuses on simplicity, fast responsiveness, and seamless integration with the interpreter.
- Visual Symbol Table Debugger for Runtime Inspection - The debugger window visualizes the internal state of the program during execution. It displays symbol tables, scopes, variables, and function parameters in a graphical format, helping users understand how data is organized at runtime. Each scope (main, class, function) is shown as a separate section with color-coded nodes for easier interpretation. This tool enhances program transparency by showing how variables change and how nested scopes behave during function and class execution.

III. CONCLUSIONS

The project successfully implemented a functional mini-language featuring dynamic typing, generalized expression parsing, array handling, and user-defined functions. Through the integration of Flex, Bison, and

C/C++, the interpreter achieved Python-like behavior while operating on low-level memory structures. A unified symbol table built using void pointers enabled seamless storage of heterogeneous data types, and recursive expression evaluation ensured correct computation of arithmetic and nested expressions. Scope management using a stack-based model allowed functions and classes to execute with proper isolation and lifetime control.
Overall, the system demonstrated reliable tokenization, parsing, execution flow, and memory handling, validating the feasibility of constructing a high-level language framework atop C/C++ internals.

REFERENCES

[1] https://cfm.ehu.es/ricardo/docs/python/Learning_Python.pdf
[2] https://towardsdatascience.com/how-lists-in-python-are-optimised-internally-for-better-performance-847c8123b7fa/
[3] https://dpvipracollege.ac.in/wp-content/uploads/2023/01/Alfred-V.-Aho-Monica-S.-Lam-Ravi-Sethi-Jeffrey-D.-Ullman-Compilers-Principles-Techniques-and-Tools-Pearson_Addison-Wesley-2007.pdf
[4] https://www3.nd.edu/~dthain/compilerbook/compilerbook.pdf
[5] Win32 API Documentation - https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list
[6] Python Language Reference - https://docs.python.org/3/reference/datamodel.html
[7] Robert W. Sebesta, Concepts of Programming Languages, 12th Edition, Pearson, 2019.
[8] LLVM Language Reference Manual - https://llvm.org/docs/LangRef.html