Design and Implementation of a Mini Compiler with Three Address Code Generation, Optimization, and Interpreter-Based Execution

Vinita Singh[1,] Omkar Kulkarni[2]
[1,2]*Department of Computer Science, faculty of Computer Science, University of Mumbai-400 032*
Hindi Vidya Prachar Samiti's RAMNIRANJAN JHUNJHUNWALA COLLEGE (Empowered Autonomous)
GHATKOPAR (W), MUMBAI – 400 086

***Abstract*—This project presents the design and implementation of a mini compiler for a small statically typed imperative programming language. The compiler supports integer and string data types, variable declarations, assignments, arithmetic expressions, and output statements. It is constructed using a structured multi-phase pipeline consisting of lexical analysis, syntax analysis, abstract syntax tree (AST) construction, semantic analysis, intermediate representation generation using three address code (TAC), an optimization phase, and execution via a custom interpreter backend. The project is developed with a strong educational focus, aiming to provide a transparent view of how high-level source code is transformed into an executable representation. Each compilation stage is implemented explicitly, allowing intermediate artifacts such as tokens, AST structures, and TAC instructions to be inspected and analyzed. The architecture emphasizes modularity, phase separation, and extensibility, enabling future enhancements such as advanced optimization techniques and additional language features.**

***Index Terms*—Compiler, Tokenization, Syntax and semantic Analysis, Three Address Code, Abstract Syntax Tree, Optimization**

I. INTRODUCTION

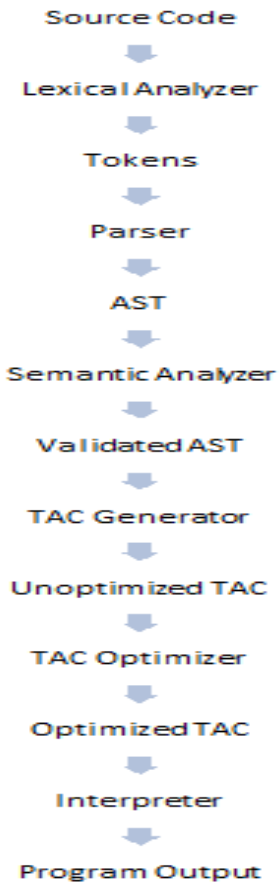
Compilers form the foundational bridge between high-level programming languages and executable programs. They translate human-readable source code into lower-level representations that can be executed efficiently by a machine. Despite their critical role in modern computing systems, the internal workings of compilers are often abstracted behind mature toolchains, making the compilation process appear opaque to learners. This project undertakes the implementation of a compiler for a small statically typed imperative programming language in order to explore and understand the internal structure of a compilation system. Rather than relying on existing compiler frameworks, each stage of the compilation pipeline is implemented explicitly to provide clear visibility into how source code is progressively analyzed, validated, transformed, and executed.

The implemented language is intentionally minimal, supporting integer and string data types, variable declarations, assignments, arithmetic expressions, and print statements. By restricting the language scope, the focus remains on the architecture and mechanics of compilation rather than language complexity. This controlled design enables careful implementation of lexical analysis, parsing, abstract syntax tree construction, semantic validation, intermediate representation generation, optimization, and execution. The compiler follows a classical multi-phase architecture in which each stage produces a well-defined intermediate representation. This structured design enforces separation of concerns, simplifies debugging, and allows incremental extension of the system. The inclusion of an optimization phase further demonstrates how compilers improve program efficiency while preserving semantics. Through this implementation, the project provides both a functional compiler system and a structured study of compiler construction techniques, emphasizing clarity, modularity, and educational transparency.

II. METHODOLOGY

The compiler is designed as a multi-phase system where each phase performs a specific operation on the source program and passes the processed information to the next stage. The overall workflow of the compiler consists of lexical analysis, syntax analysis, semantic analysis, Three Address Code

(TAC) generation, TAC optimization, and interpreter-based execution.
The compiler accepts source code written in a simplified programming language supporting integer and string data types. The language includes variable declarations, assignments, arithmetic expressions, unary operations, and print statements. Each phase of the compiler contributes toward validating, transforming, optimizing, and executing the input program.



III. OBJECTIVE FUNCTION:

1. To design and implement a simplified programming language supporting integer and string data types.
2. To implement the major phases of a compiler, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and execution.
3. To generate Three Address Code (TAC) as an intermediate representation for simplifying expression handling and optimization.
4. To implement optimization techniques such as constant propagation, constant folding, and dead code elimination on the generated TAC.
5. To execute optimized TAC instructions using an interpreter backend instead of machine code generation.
6. To demonstrate the practical application of compiler design concepts through a complete end-to-end implementation.

IV. IMPLEMENTATION

The compiler was implemented in C++ using a modular design in which each compilation phase was developed as a separate component. The lexical analyser is responsible for tokenizing the source code into meaningful lexical units such as keywords, identifiers, literals, and operators. The parser processes the generated tokens and constructs an Abstract Syntax Tree (AST), which serves as the internal representation of the program structure.
Semantic validation is performed using a symbol table that stores information about variables, including their data types and initialization status. This enables the compiler to detect semantic errors such as undeclared variables and invalid assignments.
After successful validation, the compiler generates Three Address Code (TAC) as an intermediate representation. Complex expressions are decomposed into simpler instructions using temporary variables, facilitating subsequent optimization. The optimization module applies constant propagation, constant folding, and dead code elimination to reduce redundant computations and improve execution efficiency.
Instead of generating machine code, the compiler utilizes an interpreter backend. The interpreter executes the optimized TAC instructions directly and produces the final output of the program. This approach simplifies backend implementation while still demonstrating the complete compilation process.

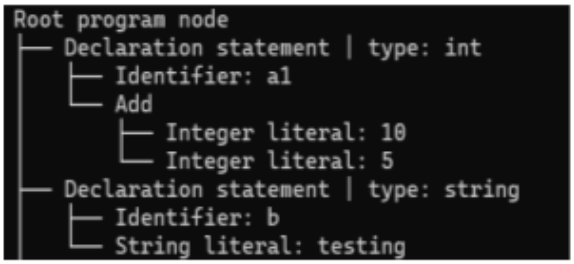
V. RESULTS AND DISCUSSION

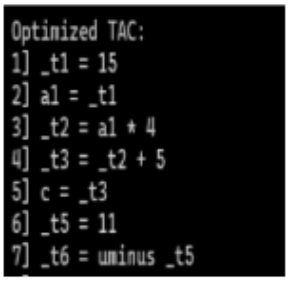
The developed compiler was tested using programs containing variable declarations, arithmetic expressions, string literals, assignments, unary operations, and print statements. The compiler successfully completed all stages of the compilation process, including token generation, syntax tree construction, semantic validation, TAC generation, optimization, and execution.
The generated Three Address Code provided a clear intermediate representation of the source program and enabled the application of optimization techniques. Constant folding reduced arithmetic computations involving compile-time constants,

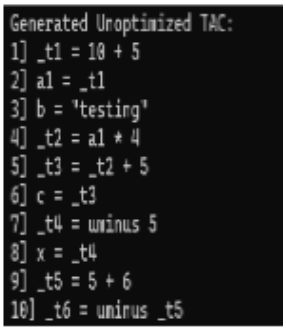
while constant propagation simplified expressions using known constant values. Dead code elimination removed instructions whose results were not utilized, thereby reducing unnecessary execution overhead.

Experimental testing showed that the optimized TAC contained fewer instructions than the unoptimized version while producing identical program output. This demonstrates the effectiveness of intermediate code optimization in reducing runtime work. The interpreter backend successfully executed the optimized TAC and generated the expected results for all test cases considered during development.

The project highlights how intermediate representations and optimization techniques contribute to efficient program execution, even within a simplified compiler environment.







VI. CONCLUSION

This paper presented the design and implementation of a mini compiler for a simplified programming language supporting integer and string data types. The compiler successfully implements the major stages of compilation, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and execution through an interpreter backend.

The use of Three Address Code as an intermediate representation enabled the implementation of optimization techniques such as constant propagation, constant folding, and dead code elimination. These optimizations reduced redundant computations and improved the efficiency of the generated intermediate code while preserving program correctness.

The project demonstrates the practical application of fundamental compiler design concepts and provides a foundation for future enhancements. Possible future work includes the addition of control flow constructs, function support, advanced optimization techniques, and machine code generation to further extend the capabilities of the compiler.

REFERENCES

[1] Brinch Hansen, P. (1985). Brinch Hansen on Pascal Compilers. Prentice Hall. ISBN: 978-0130830982.

[2] Nystrom, R. (2021). Crafting Compilers. Genever Benning. ISBN: 978-0990582939.

[3] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson Education. ISBN: 978-0321486813.

[4] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(4), 451–490.