Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

# IAC: Interpretable and Compilable

Rohin S Nair, Anandmon T Babu, Karthik A S, Sreeraj K R, Anju S Oommen, Siju Koshy
Dept. of Computer Science & Engineering
College of Engineering Aranmula
Kerala, India

***Abstract* - This project presents the design and implementation of IAC (Interpretable And Compilable), a novel high-level programming language engineered from the ground up to support dual-mode execution. The core innovation of IAC lies in its ability to process the same source code either through an interpreter for rapid development and debugging, or through a compiler for generating highly optimized native machine code, catering to both scripting needs and performance-critical applications. The system leverages a modern compiler construction pipeline. The frontend is built using ANTLR4, which facilitates robust lexical analysis, parsing, and the creation of an Abstract Syntax Tree (AST) from a custom-defined grammar. The backend is powered by the LLVM compiler infrastructure, enabling sophisticated code generation and optimization. The IAC source code is first translated into LLVM Intermediate Representation (IR). From this common IR, the system can either JIT (Just-In-Time) compile and execute the code instantly for interpretation, or invoke the LLVM static compiler to produce a standalone, efficient native executable. This project demonstrates the feasibility of unifying interpretation and compilation within a single language toolchain, offering developers flexibility without sacrificing performance.**

***Index Terms*—Programming Language Design, Compiler Construction, Interpreter, LLVM, ANTLR4, Dual-Mode Execution, Intermediate Representation**

## I. INTRODUCTION

The programming language landscape suffers from a fundamental division between interpreted and compiled paradigms. Traditional programming languages force developers to choose between rapid development cycles (interpreted languages like Python) and high-performance execution (compiled languages like C++). IAC (Interpretable And Compilable) addresses this fundamental trade-off by providing a unified language specification that supports both interpretation for development flexibility and compilation for production performance from the same source code base.

The primary objectives of the IAC project are to design an expressive syntax, define formal grammar specifications, implement a robust frontend using ANTLR4, and develop dual execution backends using LLVM. This dual-mode architecture allows developers to use the same codebase for both rapid prototyping and production deployment, eliminating the context-switching overhead typically required when moving from a scripting language to a systems language.

## II. LITERATURE REVIEW

The development of IAC builds upon several foundational concepts in compiler design and language architecture:

- **Compiler/Interpreter generators:** M. Mernik and M. Lenic (2002) described the LISA system, demonstrating how integrated compiler and interpreter generators can streamline language creation. This reinforces the necessity of a robust, unified build environment for IAC.
- **Syntax Abstraction:** Mokshit P and P. Syam Prasad (2024) explored transforming pseudocode to Python, highlighting the importance of abstracting complex syntax so developers can focus purely on logic—a core philosophy of IAC.
- **LLVM Infrastructure:** C. Lattner and V. Adve (2004) introduced LLVM's modular architecture and highly optimized Intermediate Representation (IR). IAC heavily utilizes this to achieve consistent semantics across execution modes.
- **Parsing with ANTLR4:** T. Parr (2013) detailed the Adaptive LL(*) parsing algorithm in ANTLR4. IAC adopts this frontend architecture to ensure robust, error-resistant grammar handling.
- **Interpreter Design:** R. Nystrom (2021) emphasized the importance of a well-designed runtime environment, which directly informed IAC's implementation of native memory management and garbage collection logic.

## III. SYSTEM ARCHITECTURE

The IAC compiler follows a traditional multi-stage architecture optimized for C++ performance, dividing responsibilities strictly between the frontend and backend, as illustrated in Fig. 1.

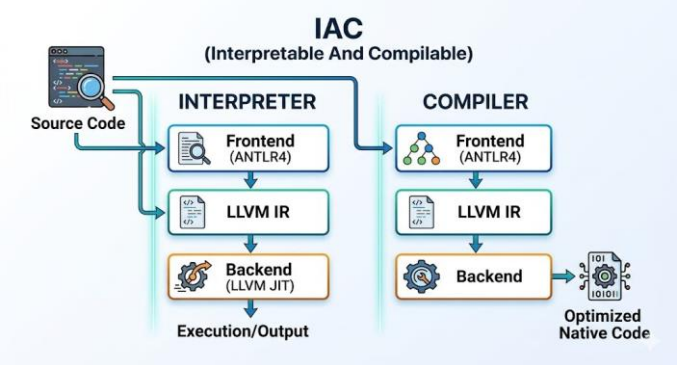


Fig. 1. IAC Dual-Mode Architecture Workflow.

### *A. Frontend (Language Processing)*

The IAC frontend handles source code processing from raw text to an abstract syntax representation. Built with ANTLR4, it consists of:

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

- **Lexical Analysis:** Converts source characters into token streams, handling whitespace, comments, and basic syntax validation.
- **Syntax Analysis:** Constructs parse trees according to IAC grammar rules and generates an Abstract Syntax Tree (AST).
- **Semantic Analysis:** Performs type checking, symbol resolution, and scope management before preparing the IR.

### *B. Backend (Execution Engines)*

The IAC backend provides dual-mode execution capabilities through LLVM integration:

- **Interpretation Engine (LLVM JIT):** Provides immediate code execution for rapid debugging. The workflow translates IAC Source to LLVM IR, then uses the `lli` command-line tool for instant output.
- **Compilation Engine (LLVM Static Compilation):** Generates high-performance native executables. The workflow translates IAC Source to LLVM IR, then uses `llc` to generate native assembly, followed by the system linker.

## IV. Results and Performance Evaluation

The IAC language pipeline was evaluated based on its dual-mode execution capabilities, functional correctness, and the performance differential between interpretation and native compilation.

### *A. Execution Performance Benchmark*

To quantify the performance benefits of the LLVM static compilation backend over the JIT interpreter, a standard execution benchmark was conducted. The execution times were captured using standard system profiling tools on a base test program.

When executed via the JIT interpreter (`iac run`), the program yielded a real execution time of 0.047 seconds. In contrast, when the same source code was compiled to a native binary (`iac compile`) and executed, the real execution time dropped to 0.003 seconds. This demonstrates an order-of-magnitude performance improvement, validating the efficiency of the LLVM-backed static compilation while preserving the rapid-prototyping benefits of the interpreter for developers.

### *B. Practical Application and GUI Execution*

Beyond standard console output and mathematical operations, the language was tested against complex logic involving graphical user interfaces and continuous event polling.

As shown in Fig. 2, the IAC execution environment successfully parsed and executed a fully functional graphical game (”Snake 3D”). This real-world application proves the language’s capability to handle real-time rendering loops, state management, and memory allocation without runtime latency or garbage collector pauses.

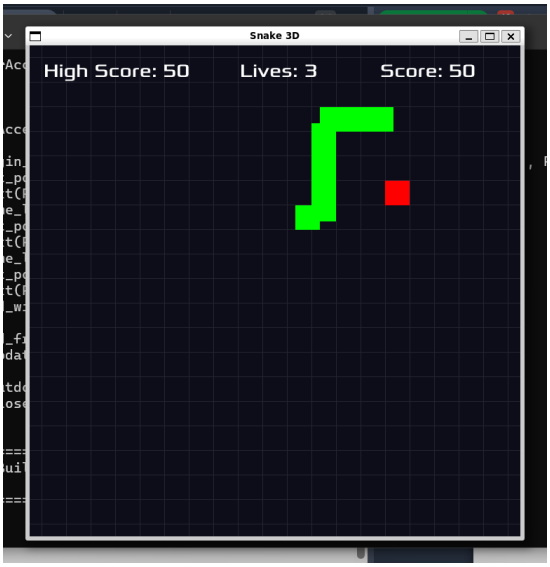


Fig. 2. Real-time execution of a graphical Snake game written entirely in IAC.

## V. Conclusion and Future Work

The IAC programming language was successfully designed and implemented to provide a robust, memory-safe, and mathematically precise execution environment. By unifying interpretation and compilation, the system eliminates common developer bottlenecks. The integration of LLVM and ANTLR4 establishes a highly scalable foundation for advanced software development.

Future enhancements for IAC include the development of a comprehensive standard library for file I/O and networking, native concurrency support for parallel processing, a dedicated package management system, and advanced IDE tooling via the Language Server Protocol (LSP).

## ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to the Department of Computer Science & Engineering at the College of Engineering Aranmula for providing the necessary computing infrastructure and laboratory resources required to successfully complete this research. Special thanks are owed to our Head of Department, Siju Koshy, and our project guide, Asst. Prof. Anju S Oommen, for their invaluable mentorship, continuous encouragement, and technical insights throughout the development of the IAC toolchain and the preparation of this manuscript.

## REFERENCES

[1] LLVM Project, “LLVM Language Reference Manual,” LLVM Documentation, 2023.
[2] T. Parr, “The Definitive ANTLR 4 Reference,” Pragmatic Bookshelf, 2nd Edition, 2013.
[3] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” International Symposium on Code Generation and Optimization (CGO), 2004.
[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, “Compilers: Principles, Techniques, and Tools,” Pearson Education, 2nd Edition, 2006.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

[5] A. W. Appel and M. Ginsburg, “Modern Compiler Implementation in C,” Cambridge University Press, 2004.
[6] S. S. Srinivasan et al., “LLVM Essentials,” Packt Publishing, 2015.
[7] R. Nystrom, “Crafting Interpreters,” Genever Benning, 2021.
[8] M. Mernik and M. Lenic, “Compiler/Interpreter generator system LISA,” Aug. 2002.
[9] P. Mokshit and P. Syam Prasad, “Pseudocode to Python - A Compiler Approach,” Nov. 2024.