

HalEngine: A Programming Language

Mrs. Smitha P¹, Reshma Hegde², N Harshitha³, Nisarga K⁴

¹Smitha P, Asst. Prof, Dept. of ISE, East West Institute of Technology

²Reshma Hegde, Dept. of ISE, East West Institute of Technology

³N Harshitha, Dept. of ISE, East West Institute of Technology

⁴Nisarga K, Dept. of ISE, East West Institute of Technology

Abstract -This paper represents HalEngine, a programming language which is designed to show how a compact design can still support practical programming tasks. The language is built using Python and LLVM and including features like variables, booleans, strings, basic math functions, conditionals, loops, functions with return-type inference, simple structures, and a minimal standard library. The compiler follows a straightforward architecture with a lexer, parser, abstract syntax tree, environment model, and LLVM IR generator. Despite its simplicity, HalEngine can compile and execute real programs while keeping the implementation easy to understand. Outlining the core design and the minimal runtime required to support the language, this language can be used to demonstrate how a functional compiler can be created with a small and focused feature set.

Key Words: programming languages, syntax, compiler design, EBNF

1. INTRODUCTION

Programming languages form the foundation of modern computing, enabling developers to express logic, define data, and control the behavior of software systems. They continue to evolve as new requirements emerge in areas such as system design, education, research, data analysis and rapid prototyping. While large industrial languages focus on extensive ecosystems and performance, small and minimal languages remain valuable for understanding core concepts, experimenting with new ideas, and demonstrating compiler techniques in a clear and approachable way.

To address the need for simpler and more accessible tools in language design, a minimalist programming language provides a practical solution by offering essential features within a lightweight and easy-to-understand framework. This paper presents HalEngine, a compact programming

language created to demonstrate how a small and focused design can still support meaningful programming tasks.

2. RELATED WORK

Programming languages and educational compilers have been widely studied as vehicles for demonstrating the fundamentals of language design, parsing, and code generation. Prior work on lightweight languages shows how minimal syntactic and semantic rules can be paired with LLVM-based backends to produce executable programs with relatively small implementation effort. Several studies also explore custom frontends that emit LLVM IR directly, emphasizing the value of a stable intermediate representation for portability and optimization. These systems typically prioritize simplicity, often restricting features to the essentials needed for instructional clarity. In contrast, HalEngine maintains a minimal philosophy while offering a slightly broader feature set, including loops, return-type inference, and basic structures thus providing a compact yet practical platform for understanding the full compiler pipeline.

3. LANGUAGE DESIGN

HalEngine is designed to be a minimal, educational programming language that demonstrates the full compiler pipeline- from tokenization to LLVM IR code generation, while still supporting practical constructs.

The primary goal of HalEngine is to provide a compact, minimal programming language that

```

Program ::= [ Function ]
Function ::= "fun" Identifier "(" [ Parameters ] ")" Block "nu"
Parameters ::= Identifier [ "," Identifier ]
Block ::= [ Statement ]
Statement ::= VarDec | ExprStmt | IfStmt | WhileStmt | ReturnStmt
VarDec ::= "var" Identifier "=" Expr ";"
IfStmt ::= "if" Expr Block [ "else" Block ] ("fi" | "esle")
WhileStmt ::= "while" Expr Block "elihs"
ReturnStmt ::= "return" Expr ";"
ExprStmt ::= Expr
Expr ::= Term | ("+" | "-") Term
Term ::= Factor | ("*" | "/" ) Factor
Factor ::= Identifier | Number | String
| "(" Expr ")" | Call
Call ::= Identifier "(" [ Args ] ")"
Args ::= Expr | "," Expr

```

Fig -3.1: Core EBNF Syntax Rules for HalEngine

demonstrates the essential stages of compiler construction without unnecessary complexity. The language is designed to be easy to read, simple to implement, and structurally clean, enabling students and researchers to understand lexical analysis, parsing, AST construction, type inference, and LLVM-based code generation through a small and manageable codebase. The core syntax definitions are shown in Figure 3.1, expressed using an abbreviated EBNF notation. HalEngine focuses on maintaining a balance between minimal syntax and practical capabilities, ensuring that the language remains lightweight while still being expressive enough to model fundamental programming concepts. The structural layout of a HalEngine program is illustrated in Figure 3.2.

HalEngine introduces a concise set of core features that make the language functional while preserving its minimalist design philosophy. The language supports integer, floats, boolean and undefined values, along simple structures for lightweight data modeling. Control-flow constructs include conditional statements, looping mechanisms. The compiler incorporates automatic type inference and return-type inference, allowing variable declarations and function definitions to remain uncluttered. Additional features such as comments and a minimal math utility header contribute to practical expressiveness without inflating language complexity. Overall, these features offer enough breadth for demonstrating end-to-end compiler design while keeping the system intentionally compact.

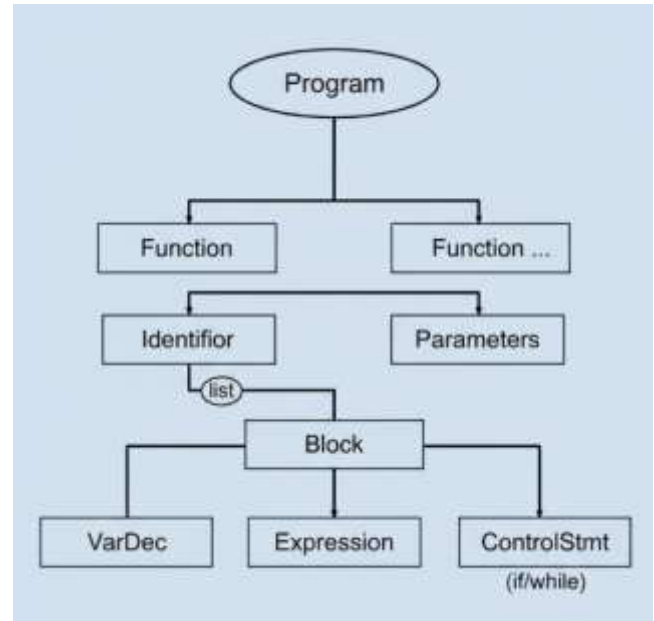


Fig -3.2: Syntax-Level Layout of Functions and Blocks

4. SYSTEM ARCHITECTURE

HalEngine follows a classical yet deliberately lightweight compiler pipeline designed to clearly expose each major stage involved in transforming human-written source code into executable output. The architecture is modular by design, allowing each subsystem such as the lexer, parser, semantic analyzer, IR generator, and execution engine, to be understood, tested, and extended independently. This modular clarity makes HalEngine suitable for students, researchers, and developers who want to observe the end-to-end mechanics of compiler construction without navigating a large or industrial-scale codebase. The overall flow of the system is illustrated in Figure 4.1, which outlines the transformation of source code through each compilation stage.

A. Lexer

The lexer(tokenizer) is responsible for scanning the raw source code character by character and grouping sequences of characters into meaningful tokens. These tokens include identifiers, keywords, numeric literals, operators, punctuation, and syntactic symbols. During this phase, unnecessary whitespace is discarded and comments are ignored. The resulting token stream becomes the structured input that the parser consumes.

B. Parser

The parser applies a recursive-descent parsing strategy guided by the language's EBNF grammar. HalEngine uses a recursive-descent parser for statements and control structures, and a Pratt parsing strategy for expressions to

efficiently handle operator precedence and associativity. It verifies that the token sequence conforms to language's syntax rules and constructs the Abstract Syntax Tree (AST). The parser also provides clear and human-readable error messages pointing to specific lines and tokens, making debugging easier for users. Strict grammar enforcement ensures that only syntactically valid programs advance to later stages.

C. Abstract Syntax Tree (AST)

The AST serves as a hierarchical, structured representation of the program. It abstracts away syntactic details and organizes the program into nodes that represent statements, expressions, declarations, function definitions, and control-flow constructs. By operating on this tree rather than raw tokens, later components of the compiler can analyze and transform the program with accuracy and consistency.

D. Semantic Analysis

Semantic analysis ensures that the program is meaningful, not merely syntactically correct. This stage resolves variable declarations, verifies type compatibility, checks function signatures, enforces scope rules, performs return-type inference, and tracks identifier lifetimes. Errors such as mismatched types, undeclared variables, or invalid operations are detected here. This stage guarantees that only semantically sound ASTs proceed to code generation.

E. LLVM IR Code Generation

After semantic validation, the compiler converts the AST into LLVM Intermediate Representation (IR). This involves generating IR instructions for expressions, branching constructs, loops, variables, function bodies, arithmetic operations, and return statements. Each AST node is lowered into the corresponding LLVM IR construct using `llvmlite`. The resulting IR is architecture-independent, human-readable, and compatible with LLVM's optimization toolchain. This stage bridges high-level language constructs with low-level executable semantics.

F. Execution Engine

The final stage uses LLVM's Execution Engine to run the generated IR immediately, enabling a Just-In-Time (JIT) execution model. This provides fast turnaround for testing and experimentation while still delivering the performance benefits of compiled code. The engine executes the program, handles runtime operations, and returns the final result to the user. Because the IR is

executable directly, HalEngine maintains the feel of a high-level interactive language with the power of low-level compilation.

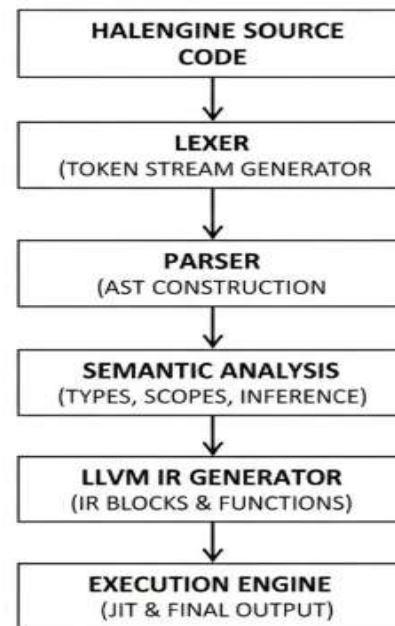


Fig -4.1: HalEngine System Architecture

5. IMPLEMENTATION

HalEngine is implemented in Python and uses LLVM as its backend to generate and execute low-level code. The implementation follows a modular structure, with each compiler stage isolated for clarity, extensibility, and ease of experimentation. The language is intentionally small, allowing the entire pipeline from tokenization to IR execution, to remain understandable to students and researchers studying compiler construction.

A. Lexer Implementation

The lexer is implemented as a hand-written scanner in Python. It maintains explicit pointers for the current character and next read position, enabling efficient handling of multi-character operators and numeric literals. Tokens are represented using a small `Token` class containing type, value, and line number. Whitespace, comments, and newline tracking are handled in the scanner loop without separate states. Numeric literal parsing distinguishes integers and floats directly based on the presence of a decimal point. Identifiers are resolved using a lookup table that maps keywords to token types.

B. Parsing Strategy

HalEngine uses a two-part parsing approach:

1. Recursive-descent functions handle statements, blocks, declarations, and control structures.

2. A Pratt parser handles expressions through binding powers.

Operator precedence is implemented without a precedence table; instead, each operator is assigned a binding power used by the Pratt engine. This keeps expression parsing compact and fully configurable.

Function definitions, parameters, and inferred return types parsed in a dedicated routine that registers the function name before parsing its body, allowing forward references.

C. AST Representation

Each AST node is implemented as a lightweight Python class storing:

1. node type
2. children nodes
3. inferred/static type
4. optional metadata such as source position

The AST structure is minimal, avoiding unnecessary abstractions. Expression nodes unify arithmetic, boolean, and call expressions using a consistent format that simplifies later stages. All statement nodes inherit a simple base representation for uniformity.

D. Semantic Rules

Type inference is implemented through a single pass over the AST.

1. Variable types are inferred from their initializer expressions.
2. Return types are resolved by inspecting the final return statement in each function.
3. Function signatures are stored in an environment table so that cross-function calls can be validated.

The semantic layer also manages nested scopes using an environment stack, ensuring correct resolution of identifiers inside loops, blocks, and functions.

E. LLVM IR Generation

IR generation is implemented using llvmlite's builder API. Each AST node is translated through dedicated visitor methods that lower constructs into LLVM instructions. Local variables are allocated at the beginning of each function using alloca, and loads and stores are emitted explicitly for every variable access. Conditional statements and loops are lowered into basic blocks connected by branch instructions, and each function is emitted with a signature derived entirely from its inferred argument and return types. A separate symbol table maps high-level identifiers to their corresponding LLVM

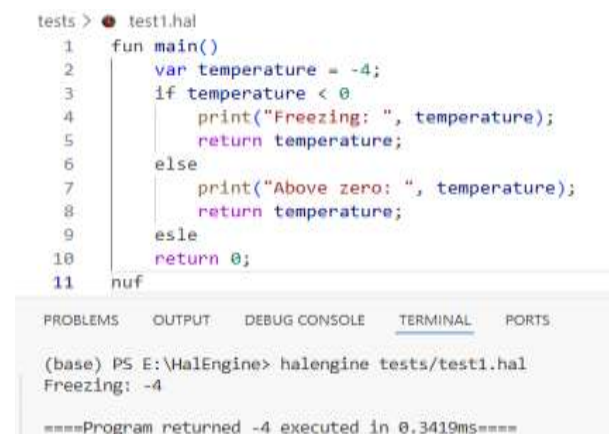
values, ensuring consistent resolution across all scopes during IR generation.

F. Runtime

The runtime layer of HalEngine provides a minimal set of built-in operations. Functions such as print() and length() are implemented in Python and exposed to LLVM through foreign-function bindings, allowing generated IR to call them directly. This small runtime is sufficient for educational use while keeping the language compact and self-contained.

G. JIT Execution

Execution of HalEngine programs is handled through LLVM's MCJIT infrastructure. Once the compiler generates the LLVM IR for a module, the JIT engine finalizes it and immediately executes the compiled code. This provides a fast, interactive execution workflow



```
tests > test1.hal
1 fun main()
2   var temperature = -4;
3   if temperature < 0
4     print("Freezing: ", temperature);
5     return temperature;
6   else
7     print("Above zero: ", temperature);
8     return temperature;
9   esle
10  return 0;
11 nuf

(base) PS E:\HalEngine> halengine tests/test1.hal
Freezing: -4

====Program returned -4 executed in 0.3419ms====
```

Fig -6.1: Conditional Statement Example

where functions return values directly back to the Python host environment. The setup enables near-instant feedback, making it suitable for rapid testing, iterative development, and REPL-style experimentation during the compiler's evolution.

6. RESULTS

To evaluate HalEngine and demonstrate the reliability of its compiler pipeline, a set of representative test programs were designed, implemented, and executed. These programs were selected to cover the major language components, such as conditional logic, arithmetic evaluation, floating-point computation, and loop constructs. Each example was compiled into LLVM IR and executed using the JIT backend to ensure that both parsing and code generation behaved as intended. The resulting outputs were recorded to verify semantic correctness and runtime behavior.

Figures 6.1 through 6.3 present these sample programs alongside their execution results, illustrating how HalEngine handles control flow, numerical operations, and iterative structures while maintaining consistent output across all tested scenarios.

This program illustrates the use of a conditional structure (if... else... esle) in HalEngine. It evaluates a temperature value and determines whether it falls below zero, printing the corresponding message. This test confirms that the compiler correctly parses the conditional syntax, performs the comparison, and executes the appropriate branch. The output verifies that the condition is evaluated accurately and that the runtime behavior matches the expected logical flow.

```
tests > floats.hal
1 fun main()
2   var radius = 5.0;
3   var area = 3.1415 * radius * radius;
4   print("Area:", area);
5   return area;
6 nuf
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(base) PS E:\HalEngine> halengine tests/floats.hal
Area:78.537506

====Program returned 78.53750610351562 executed in 0.4172ms====
```

Fig -6.2: Area Calculation with Floating-Point Arithmetic

This example highlights HalEngine's support for floating-point variables and arithmetic operations. The output shows the computed area of a circle using a float literal, demonstrating that type inference, expression evaluation, and printing are working correctly.

```
tests > while.hal
1 fun main()
2   var i = 1;
3   var prod = 1;
4   while i <= 5
5     prod = prod * i;
6     print("i = ", i, ", prod = ", prod);
7     i = i + 1;
8   elihw
9   return prod;
10 nuf
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(base) PS E:\HalEngine> halengine tests/while.hal
i = 1, prod = 1
i = 2, prod = 2
i = 3, prod = 6
i = 4, prod = 24
i = 5, prod = 120

====Program returned 120 executed in 2.8787ms====
```

Fig -6.3: While-Loop Accumulator

This program illustrates the language's looping mechanism. A while loop iterates over integer values to

compute a running sum, printing intermediate results at each iteration. The output image confirms correct loop evaluation, variable updating, and final return value, validating HalEngine's control-flow handling.

These results confirm that HalEngine's compiler pipeline, from lexical analysis to LLVM IR generation and execution is functioning as intended, providing both accurate computation and efficient runtime performance.

7. CONCLUSION

HalEngine is a minimal, educational programming language designed to demonstrate the complete compiler pipeline from lexical analysis to LLVM IR generation and execution. Despite its compact design, it supports essential features such as variables, control-flow constructs, type and return-type inference, functions, simple structures, floating-point arithmetic and basic operations. The presented results confirm that HalEngine correctly parses, analyzes, and executes programs while maintaining efficient runtime performance.

Future work could extend the language with a richer standard library, more modern features, enhanced error reporting, concurrency support, and additional data types to further explore compiler design concepts. Overall, HalEngine provides a lightweight yet practical platform for students and researchers to study language implementation and LLVM-based compilation techniques.

REFERENCES

- [1] GNU Compiler Collection (GCC), [Online]. Available: <https://gcc.gnu.org/>
- [2] LLVM Compiler Infrastructure, [Online]. Available: <https://llvm.org/>
- [3] Clang: a C language family frontend for LLVM, [Online]. Available: <https://clang.llvm.org/>
- [4] C. Lattner, *CGO: Using LLVM as a Backend for a Compiler*, 2004.
- [5] Journal of Science and Technology, "Design and Implementation of a Minimalist Programming Language Stab using Flex, Bison and LLVM," vol. 4, no. 2, pp. 65–70, 2024.
- [6] ACM, "Design and Implementation of a Minimalist Programming Language Stab using Flex, Bison and LLVM," 2015.
- [7] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT Compiler," in Proc. 2nd

Workshop on the LLVM Compiler Infrastructure in HPC (LLVM'15), 2015, pp. 1–6.

- [8] S. S. Muthukumar, B. Gandham, T. Vijayekkumaran, N. S. Krishna, and N. Panda, "Implementing a Multilingual Lexical Analyzer for Java, C and C++," in 2024 5th IEEE Global Conference for Advancement in Technology (GCAT), 2024.
- [9] C. D. Priya, C. Vikram Raju, A. I. Daniel, G. R. Patel, and M. Belwal, "Parser for TINY Language," in 2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS), 2024.
- [10] M. Kuznetsov and G. Firsov, "Syntax Error Search Using Parser Combinators," in 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), 2021.
- [11] M. di Angelo and G. Salzer, "Tokens, Types, and Standards: Identification and Utilization in Ethereum," in 2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), 2020.
- [12] C. Lattner, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, Ph.D. dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2009.