The OSCAR PUBLISHING Services
American Journal of Applied Science and Technology

VOLUME Vol.05 Issue 10 2025
PAGE NO. 100-108
DOI 10.37547/ajast/Volume05Issue10-20

Arithmetic Expression Compiler Using Python And LLVM

Abduaziz Ziyodov
Senior Student at Inha University in Tashkent, Uzbekistan

Received: 26 August 2025; **Accepted:** 22 September 2025; **Published:** 24 October 2025

Abstract: This paper presents the implementation of a compiler that translates arithmetic expressions into native machine code using the LLVM compiler infrastructure's Python API. This work demonstrates a complete compilation pipeline. It converts high-level arithmetic expressions into native assembly code using llvmlite, which is the Python binding for LLVM. The compiler performs several tasks. It handles lexical analysis, parsing with operator precedence, constructing the abstract syntax tree, and generating LLVM IR. By connecting native execution and interpreted Python code, the system allows for compilation without the delays of interpretation. This work sets the stage for future experiments in Python-to-native compilation. It also helps us understand how to access modern compiler infrastructures programmatically.

Keywords: Compiler, LLVM, Python, IR.

INTRODUCTION:

Modern computing requires both developer efficiency and performance. Python offers great productivity due to its high-level features and dynamic nature, but this affects its execution speed. In contrast, compiled languages like C and C++ deliver native performance, but they need more detailed code and longer development times. The difference between these approaches has inspired many projects that try to combine Python's user-friendliness with the speed of native code.

The LLVM compiler infrastructure has become a key technology in this area. Projects like Numba use LLVM to translate Python code into native machine instructions, achieving performance close to statically compiled languages. Still, many developers find the ways these tools work unclear. To grasp the process of converting source code to machine instructions, hands-on experience is necessary.

The initial implementation used a straightforward stack-based method to create assembly directly from parsed expressions. While this approach worked, it had limitations in scope and portability. Further research, particularly focusing on Hoyt's work about generating x86-64 assembly from Python, showed that higher-level intermediate representations could offer better abstraction and optimization opportunities.

Exploring LLVM's Python API through llvmlite revealed that accessing modern compiler infrastructure programmatically is relatively easy. This insight changed the project's focus from direct assembly generation to using LLVM's advanced compilation pipeline. The result is a system that converts arithmetic expressions through lexical analysis and parsing into LLVM intermediate representation, which LLVM compiles to native machine code for any supported target architecture.

This paper outlines the complete implementation of an arithmetic expression compiler using LLVM infrastructure. It deals with operator precedence, supports parentheses, checks for compile-time errors, and produces optimized native code. The project reuses lexer and parser components from Gustav, a previously created interpreter, showcasing how compiler and interpreter implementations can share common front-end elements.

METHODS

LLVM Compiler Infrastructure

LLVM is a set of modular and reusable compiler and toolchain technologies. It started as a research project at the University of Illinois and has become

the foundation for several production compilers, including Clang, Swift, Rust, and Julia [3]. The name originally stood for "Low Level Virtual Machine", but the project has expanded well beyond virtual machine technology, making the acronym less relevant.

The main innovation of LLVM is its intermediate representation, known as LLVM IR. This is a low-level programming language similar to assembly but with key differences. LLVM IR is platform-independent, strongly typed, and designed for optimization. Programs written in LLVM IR can be compiled to native code for any architecture that LLVM supports, including x86, ARM, RISC-V, and others [4].

LLVM IR uses a three-address code format, where most instructions have clear source and destination operands. For instance, an addition operation takes two input values and produces one output value, with all three named explicitly. This clarity makes analysis and transformation easier than with stack-based or implicit register models. The language follows Static Single Assignment form, meaning each variable is assigned only once, which simplifies many optimization algorithms [5].

A simple example shows LLVM IR syntax:

```
define i32 @main() {
entry:
  %add_tmp = add i64 2, 3
  %result = call i32 (i8*, ...) @printf(i8* %fmt, i64 %add_tmp)
  ret i32 0
}
```

This IR defines a main function that adds two 64-bit integers, prints the result, and returns zero. The syntax clearly names temporary values like %add_tmp and specifies types like i64 for 64-bit integers.

The LLVM compilation pipeline includes several stages:

- First, source code is translated to LLVM IR. This IR can be in a textual format that is easy for humans to read or in binary bitcode format for efficiency.
- Second, optimization passes change the IR to improve performance while keeping the same meaning. LLVM has many optimization passes that use techniques like dead code removal, constant propagation, loop unrolling, and function inlining.
- Third, code generation converts the optimized IR to assembly code for the target architecture. Finally, an assembler turns assembly into machine code in object file format, and a linker combines object files into executable programs.

LLVM's modular design lets developers use only the parts they need. A compiler front-end can create LLVM IR and let LLVM manage optimization and code generation, which saves the trouble of implementing these complex phases on their own. This separation is what makes LLVM a good fit for projects like this one.

Python Interface to LLVM

The llvmlite library offers Python bindings to LLVM's core features [6]. Unlike previous Python LLVM bindings that revealed the entire C++ API, llvmlite targets a lightweight, user-friendly interface for the most commonly used functions. This approach simplifies learning and usage while still delivering enough power for compiler implementation. The llvmlite library has two main components. The IR builder API lets users build LLVM IR programmatically with Python objects and methods. The binding API provides access to LLVM's compilation and execution services, which include parsing IR, running optimization passes, and generating native code.

To create LLVM IR, developers make Python objects that represent modules, functions, basic blocks, and instructions. The following code shows how to construct basic IR:

```
from llvmlite import ir

# Types
int32 = ir.IntType(32)
int64 = ir.IntType(64)

# Create a module
module = ir.Module(name="example")

# Function
func_type = ir.FunctionType(int32, [])
func = ir.Function(module, func_type, name="main")
```

```
# Create a basic block and builder
block = func.append_basic_block(name="entry")
builder = ir.IRBuilder(block)

# Generate LLVM instructions
left = ir.Constant(int64, 2)
right = ir.Constant(int64, 3)
result = builder.add(left, right, name="add_tmp")

# Return
zero = ir.Constant(int32, 0)
builder.ret(zero)
```

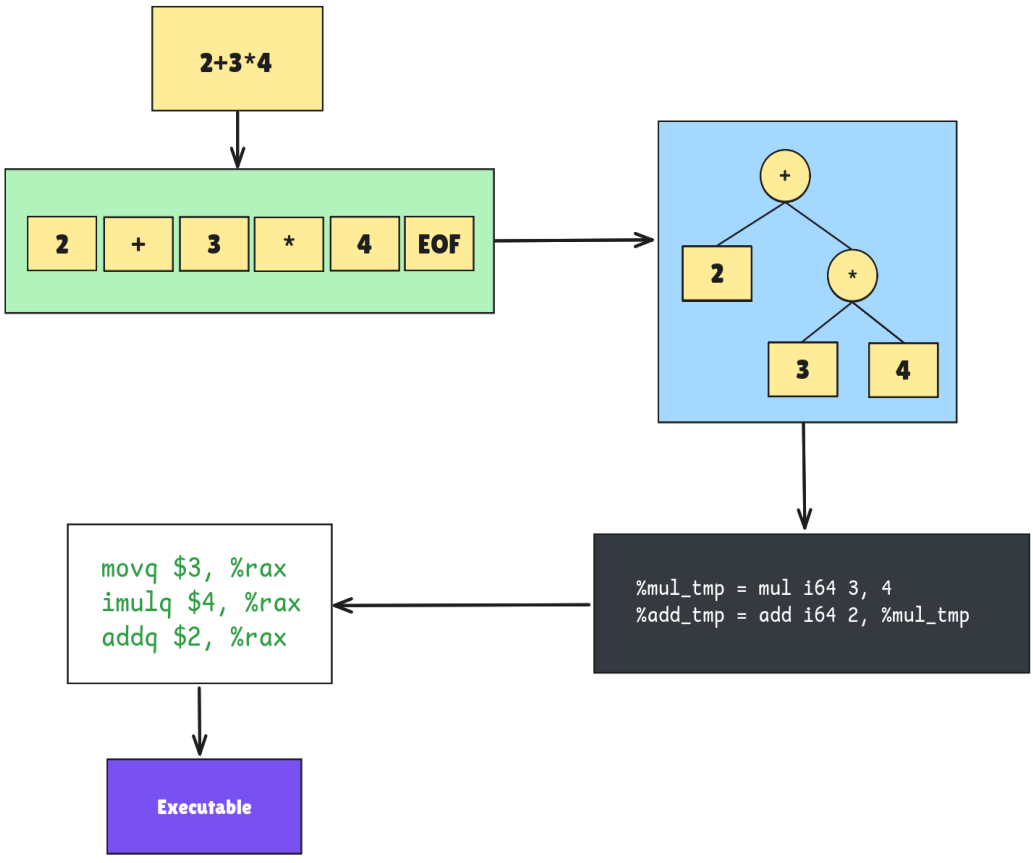
This code creates a module with a main function that includes a single basic block. Inside this block, it adds two constants and returns zero.

Type safety runs throughout the llvmlite API. Every value in LLVM IR has a specific type, and operations can only occur on compatible types. For instance, adding two 64-bit integers requires both operands to be specifically typed as i64. This strong typing helps catch many errors during IR construction instead of at later stages of compilation.

The binding API manages the compilation pipeline. After constructing the IR with the builder API, the code calls binding functions to convert the textual IR representation into LLVM's internal format. Verification functions ensure the IR is well-formed and semantically correct. Optimization passes can be applied by creating pass managers and filling them with the desired transformations. Finally, code generation functions create assembly or machine code for specified target architectures.

System Architecture

The compiler implementation uses a traditional pipeline architecture with four main stages: lexical analysis, syntax analysis, code generation, and assembly emission. This structure keeps concerns separate, allowing each stage to be tested and modified independently. Figure 1 shows the compilation pipeline from source expression to native executable.



Lexical analysis converts input strings into token sequences. The Lexer class keeps track of the current position and uses a simple state machine to recognize patterns.

The number tokenization method demonstrates the lexer's approach:

```
def number_token(self) → Token:
    while self.peek().isdigit():
        self.next_char()

    if self.peek() == "." and self.peek_next().isdigit():
        msg = "Can support integers only"
        raise CompileError(msg)

    part = self.source[self.start : self.current]
    return self.new_token(TT.NUMBER, int(part))
```

This method processes consecutive digits, specifically rejects floating-point numbers, and generates tokens with parsed integer values.

Syntax analysis builds an abstract syntax tree from tokens. The Parser class uses recursive descent [7], where grammar rules correspond to methods. Operator precedence is embedded in the parser's structure through the hierarchy of method calls.

```
def parse_term(self) → Expression:
    expr = self.parse_factor()

    while self.match(TT.MINUS, TT.PLUS):
        operator: Token = self.get_previous()
        right: Expression = self.parse_factor()
        expr = Binary(expr, operator, right)

    return expr

def parse_factor(self) → Expression:
    expr = self.parse_unary()

    while self.match(TT.SLASH, TT.STAR):
        operator: Token = self.get_previous()
        right: Expression = self.parse_unary()
        expr = Binary(expr, operator, right)

    return expr
```

The *parse_term* method manages addition and subtraction by calling *parse_factor* for the operands. Since *parse_factor* takes care of multiplication and division, these operations have a higher priority than addition. This ensures the right order of operations without needing tables.

The AST nodes use Python dataclasses, which help with immutability and memory efficiency.

```python
@dataclass(frozen=True, slots=True, eq=False)
class Binary(Expression):
    left: Expression
    operator: Token
    right: Expression

    @t.override
    def accept[T](self, visitor: "ASTVisitor[T]") → T:
        return visitor.visit_binary_expression(self)


@dataclass(frozen=True, slots=True, eq=False)
class Literal(Expression):
    value: int

    @t.override
    def accept[T](self, visitor: "ASTVisitor[T]") → T:
        return visitor.visit_literal_expression(self)
```

The visitor pattern [8] separates tree traversal from operations. Each node implements accept, which sends the call to the right visitor method. Code generation walks the AST and produces LLVM IR. The CodeGenerator class holds an IRBuilder reference for creating instructions:

```python
def visit_binary_expression(self, expression: "Binary") → ir.Instruction:
    lhs, rhs = self.visit(expression.left), self.visit(expression.right)

    match expression.operator.type:
        case TT.PLUS:
            return self.builder.add(lhs, rhs, name="add_tmp")
        case TT.MINUS:
            return self.builder.sub(lhs, rhs, name="sub_tmp")
        case TT.STAR:
            return self.builder.mul(lhs, rhs, name="mul_tmp")
        case TT.SLASH:
            if isinstance(rhs, ir.Constant) and rhs.constant == 0:
                lhs_val = (
                    lhs.constant if isinstance(lhs, ir.Constant) else "computed"
                )
                msg = f"Division by zero: {lhs_val} / 0"
                raise CompileError(msg)

            return self.builder.sdiv(lhs, rhs, name="div_tmp")
        case _:
            raise NotImplementedError(expression.operator)
```

Binary operations visit operands one by one and then create the right instructions. Literals turn into LLVM constants. Unary negation is done by subtracting from zero, using existing arithmetic.

The main function setup shows the LLVM module structure.

This creates a module with a main function returning 32-bit integers, initializes native target support, and sets the target triple for the current platform.

```python
def setup_llvm() → tuple[ir.IRBuilder, ir.Module]:
    binding.initialize_native_target()
    binding.initialize_native_asmprinter()

    module = ir.Module(name="expr_module")
    func_ty = ir.FunctionType(INT_32, ())
    func = ir.Function(module, func_ty, name="main")
    block = func.append_basic_block(name="entry")
    builder = ir.IRBuilder(block)
    module.triple = binding.get_default_triple()

    return builder, module
```

Implementation Details

The complete implementation consists of approximately ~400 lines of Python code. Type constants are defined at module level:

```python
INT_8 = ir.IntType(8)
INT_32 = ir.IntType(32)
INT_64 = ir.IntType(64)
ZERO_32 = ir.Constant(INT_32, 0)
ZERO_64 = ir.Constant(INT_64, 0)
```

Using 64-bit integers for arithmetic prevents overflow on reasonable inputs and remains efficient on modern processors. The system generates both LLVM IR (.ll files) and assembly (.s files) for inspection. You can compile the generated assembly with standard tools:

$ *python3 main.py*

Expr> 2 + 3 * 4

Generates expression.ll and expression.s

$ *gcc expression.s -o expression (or clang)*

$ *./expression*

2 + 3 * 4 = 14

The compilation workflow integrates with existing toolchains, producing standalone executables without runtime dependencies.

RESULTS

The compiler successfully translates arithmetic expressions into native code. Testing covered simple arithmetic, operator precedence, parentheses, and unary negation. All tests produced correct results verified by executing generated executables.

For the expression “(2 + 3) * 4”, generated LLVM IR:

```
; ModuleID = "expr_module"
target triple = "x86_64-pc-linux-gnu"
target datalayout = ""

define i32 @"main"()
{
entry:
  %"add_tmp" = add i64 2, 3
  %"mul_tmp" = mul i64 %"add_tmp", 4
  %".2" = getelementptr inbounds [16 x i8], [16 x i8]* @"format_str", i32 0, i32 0
  %".3" = call i32 (i8*, ...) @"printf"(i8* %".2", i64 %"mul_tmp")
  ret i32 0
}

declare i32 @"printf"(i8* %".1", ...)

@"format_str" = private constant [16 x i8] c"(2+3)*4 = %lld\0a\00"
```

Execution example showing compilation and running of the program:

```
> python3 main.py
Expr> (10+5)*2-3
2025-10-19 00:54:16,186 INFO: [PARSED AST]

Binary(
    left=Binary(
        left=Grouping(
            expression=Binary(
                left=Literal(value=10),
                operator=Token(type=<TT.PLUS: 'plus'>, lexeme='+', literal=None),
                right=Literal(value=5)
            )
        ),
        operator=Token(type=<TT.STAR: 'star'>, lexeme='*', literal=None),
        right=Literal(value=2)
    ),
    operator=Token(type=<TT.MINUS: 'minus'>, lexeme='-', literal=None),
    right=Literal(value=3)
)

2025-10-19 00:54:16,210 INFO: [GENERATED LLVM IR]

; ModuleID = "expr_module"
target triple = "x86_64-pc-linux-gnu"
target datalayout = ""

define i32 @"main"()
{
entry:
  %"add_tmp" = add i64 10, 5
  %"mul_tmp" = mul i64 %"add_tmp", 2
  %"sub_tmp" = sub i64 %"mul_tmp", 3
  %".2" = getelementptr inbounds [19 x i8], [19 x i8]* @"format_str", i32 0, i32 0
  %".3" = call i32 (i8*, ...) @"printf"(i8* %".2", i64 %"sub_tmp")
  ret i32 0
}

declare i32 @"printf"(i8* %".1", ...)

@"format_str" = private constant [19 x i8] c"(10+5)*2-3 = %lld\0a\00"
```

```
2025-10-19 00:54:16,238 INFO: [GENERATED ASSEMBLY]

    .file   "<string>"
    .section    .ltext,"axl",@progbits
    .globl  main
    .p2align    4
    .type   main,@function
main:
    .cfi_startproc
    pushq   %rax
    .cfi_def_cfa_offset 16
    movabsq $.Lformat_str, %rdi
    movabsq $printf, %rcx
    movl    $27, %esi
    xorl    %eax, %eax
    callq   *%rcx
    xorl    %eax, %eax
    popq    %rcx
    .cfi_def_cfa_offset 8
    retq
.Lfunc_end0:
    .size   main, .Lfunc_end0-main
    .cfi_endproc

    .type   .Lformat_str,@object
    .section    .lrodata,"al",@progbits
    .p2align    4, 0x0
.Lformat_str:
    .asciz  "(10+5)*2-3 = %lld\n"
    .size   .Lformat_str, 19

    .section    ".note.GNU-stack","",@progbits

2025-10-19 00:54:16,247 INFO: Saved into ll_filename='expression.ll' asm_filename='expression.s'
```

```
~/Experiments/llvm-python
> clang -o expression expression.ll

~/Experiments/llvm-python
> ./expression
(10+5)*2-3 = 27

~/Experiments/llvm-python
>
```

The generated assembly integrates with standard toolchains. Executables run independently without Python or LLVM runtime dependencies. Compilation time averages under 100 milliseconds for simple expressions on modern hardware.

DISCUSSION

This project shows that creating a compiler with modern tools is possible for developers who may not have a deep background in compilers. By using LLVM for optimization and code generation, the focus is on front-end tasks: lexing, parsing, and generating intermediate representation (IR).

Reusing the lexer and parser code from the Gustav interpreter highlights an important point. Compilers and interpreters share front-end tools but differ mainly in how they process abstract syntax trees (ASTs). This project produces LLVM IR, which Gustav interprets directly. However, both use the same token definitions and parsing rules. This similarity suggests that programming languages can support both interpretation and compilation with mostly shared code.

Using LLVM IR as the target representation offers clear advantages. LLVM takes care of details specific to different architectures, such as instruction selection, register allocation, and calling conventions. Changing the target configuration in LLVM allows the same code to produce x86, ARM, and RISC-V assembly. Achieving this level of portability with direct assembly generation would be challenging.

LLVM's strong typing helped catch several development errors. Errors like mismatched types between instruction operands, incorrect function signatures, and poorly formed constants prompted clear messages during verification. This quick feedback sped up development compared to troubleshooting mistakes in incorrect assembly output.

The project's simplicity shows that Python works well(in such a smaller scale of course) for building compilers, and llvmlite is effective for this purpose. The entire compiler fits in a single ~400-line file, and the code remains readable and easy to maintain without losing functionality.

Currently, the project only supports integer arithmetic with four basic operations. For real-world applications, variables, control flow, functions, and many other features are necessary. The error handling identifies clear mistakes but offers limited guidance. Division by zero detection works only for constants known at compile time, while complete detection would require complex static analysis or checks during runtime.

Future developments could expand the language by adding variables that would need symbol tables and memory operations. Managing control flow structures would require basic block management and branch instructions. Introducing functions would involve calling conventions and managing stack frames. Integrating with the Gustav interpreter could allow hybrid interpretation during development while enabling compilation for production, combining Python's fast development speed with the performance of native code.

The broader context of compiling Python to native code makes this work significant. Projects like Numba and Triton use similar methods on a larger scale. Learning how simple arithmetic compiles to LLVM IR lays the groundwork for understanding how these more complex systems function.

CONCLUSION

This work presents a functional compiler that translates arithmetic expressions to native machine code using LLVM infrastructure. The implementation shows that modern compiler tools make compilation projects easier for developers with moderate experience. By using llvmlite to access LLVM's compilation pipeline, the project achieves native code generation without complicated optimization and code generation phases.

The compiler correctly handles operator precedence, supports parenthetical grouping, performs compile-time error checking, and generates efficient assembly code. The architecture follows standard compiler design, with lexical analysis, parsing, and code generation as separate stages. The visitor pattern allows for extending AST processing without changing node classes. Strong typing throughout catches errors early in development.

The project started from observations during a

lecture and evolved through multiple iterations. Initial stack-based assembly generation shifted to LLVM-based compilation after researching available tools. Code reuse from the Gustav interpreter project shows how front-end components can be shared between interpreters and compilers.

The results indicate that the generated code is correct and efficient. LLVM's optimizations produce assembly that is comparable to hand-written code. The system compiles expressions quickly and works well with standard build tools. Error messages provide enough information for users to identify and fix problems.

Future work will add variables, control flow, and functions to the language. Integrating with the Gustav interpreter would allow for hybrid interpretation and compilation. Experiments with optimization passes could show LLVM's transformation capabilities. The foundation established here supports these extensions while also serving as a valuable standalone demonstration of compilation techniques.

The importance of this work goes beyond the specific implementation. Understanding how arithmetic expressions compile to native code offers insight into how larger systems like Numba and Triton operate. This project demonstrates that compiler construction, often seen as specialized and challenging, can be tackled step by step using modern tools. This accessibility benefits both education and the development of new language implementations.

REFERENCES

1. B. Hoyt, "Writing a simple x86-64 JIT compiler from scratch in stock Python," 2021. [Online]. Available: https://benhoyt.com/writings/pyast64/
2. A. Author, "Gustav: A Tree-Walk Interpreter Implementation in Python," Web of Journals, vol. 1, no. 12, pp. 1-10, 2024. [Online]. Available: https://webofjournals.com/index.php/12/article/view/5047/5084
3. C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in Proceedings of the International Symposium on Code Generation and Optimization, 2004, pp. 75-86.
4. LLVM Project, "LLVM Language Reference Manual," 2024. [Online]. Available: https://llvm.org/docs/LangRef.html
5. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," ACM Transactions on Programming Languages and Systems, vol. 13, no. 4, pp. 451-490, 1991.
6. Continuum Analytics, "llvmlite documentation," 2024. [Online]. Available: https://llvmlite.readthedocs.io/
7. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed. Boston: Addison-Wesley, 2006.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1994.
9. S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 2015, pp. 1-6.
10. P. Tillet, H.-T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2019, pp. 10-19.