International Journal of Scientific Research in Computer Science, Engineering and Information Technology
ISSN : 2456-3307
OPEN ACCESS
Available Online at : www.ijsrcseit.com
doi : https://doi.org/10.32628/CSEIT25113379

Exploring the Trade-Off between Language Simplicity and Execution Efficiency: A Study Using Crython and LLVM

Soham Malakar, Sreekanya Pal, Nazlee Rahman, Asoke Nath*

*Post Graduate and Research Department of Computer Science, St. Xavier's College (Autonomous), Kolkata, West Bengal, India

ARTICLEINFO

Article History:

Accepted : 10 June 2025
Published: 15 June 2025

Publication Issue
Volume 11, Issue 3
May-June-2025

Page Number
965-973

ABSTRACT

Contemporary programming languages often pose a challenge due to either complex syntax or performance limitations, especially in dynamically typed environments. The language proposed in this paper, Crython, aims to mitigate these challenges by offering a beginner-friendly, Python-like syntax combined with performance that approaches that of C. It is built with a custom lexer, employs a Pratt parser for expression handling, and uses a Recursive Descent parser for statements. Code generation is powered by LLVM, leveraging its robust toolchain to simplify backend implementation and deliver high execution speed. While the syntax is intentionally similar to Python to facilitate ease of adoption, it also addresses certain ambiguities and inconsistencies found in Python's design. The paper includes performance benchmarks across multiple languages to demonstrate the efficiency and competitiveness of the proposed language.

Keywords: Programming Language Design, Syntax Simplicity, LLVM Code Generation, Pratt Parser, Performance Benchmarking, Python-like Syntax.

Introduction

Modern programming languages are becoming increasingly complex, making them harder to learn, especially for beginners. While languages like *Python* are accessible and easy to learn, they often lag in performance. In contrast, high-performance languages such as C or C++ typically involve a steeper learning curve. In this paper, the authors have combined the best of both worlds thereby making the proposed language fast and easy to learn. As the language's syntax is reminiscent of *Python*'s syntax, users are not required to relearn a whole new language from scratch. Although the syntax is similar, the backend is completely different from *Python* as *Crython* is a *statically-typed* programming language making its performance comparable to *C*. It acts as a bridge between *Python*'s simplicity and *C*'s performance.

In the paper by C. Ye, Z. Shen, Y. Wu, and P. Loskot [10], the authors highlight that *Python* has a clear advantage in terms of its learning curve, largely due to

Copyright © 2025 The Author(s) : This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

its concise and beginner-friendly syntax. This makes it easier for newcomers to quickly get started with programming in *Python*. Given the widespread familiarity with *Python*'s syntax, the authors have designed *Crython* to closely follow *Python*'s structure. The goal is to make the language more accessible and comfortable for users by leveraging the simplicity and popularity of *Python*'s syntax.

Although *Python*'s syntax is beginner-friendly, it has many flaws. Strict indentation rule is one of them. Beginners often overlook proper *whitespace* usage in their code. This often causes significant issues while debugging. Copying a *Python* code might mess up the entire indentation. Sometimes tabs and spaces are both present in the codebase which can make debugging extremely difficult. While indentation enhances readability, it restricts the programmer's freedom in organizing code.

As *Python* does not have a keyword to declare variables it suffers from local and global scoping. Put simply, initializing a variable outside a function definition, creates a global variable. Accessing a variable within a function is generally acceptable, provided that the variable is not modified.

For example,

```
x = 10
def fun():
   print(x)
```

The above program works correctly because the variable *x* is not modified. The *x* simply acts as the global variable.

```
x = 10
def fun():
   x = 5
```

This program also works. In the fun function *x* is a local variable. This does not change the global *x*.

```
x = 10
def fun():
   x += 5
```

The code above produces an error because *x* is treated as a local variable, but it has not been defined locally. Although *Python* provides two keywords *global* and *nonlocal* that solve this issue. [6]. However, this approach can feel cumbersome in practice. In such cases, *C*-style declarations often provide a clearer and more suitable alternative.

```
int x = 10;
void fun() {
   x += 5;
}
```

C-style declaration makes the variable declaration quite simple. In *Crython*, the *var* keyword is used to declare variables. This solves the problem of *Python*'s global and local scope reducing the unintentional bugs. Finally, indentation sometimes causes issues with the multiline statements.

For example,

```
x = "Hello " +
   "World"
```

These kinds of statements sometimes throw syntax errors because of less or more indentation. To address this issue, *Crython* is designed to completely ignore all whitespace. During lexical analysis, whitespace characters are excluded from the token stream, effectively eliminating potential errors caused by incorrect spacing.

But this solution comes with one compromise. Statements must be terminated with a semicolon; otherwise, the compiler cannot determine where the current statement ends. This is a fundamental problem of all programming languages that ignore whitespaces. To address this limitation, an automatic semicolon insertion technique, referred to as the *Sanitizer*, is implemented. It smartly inserts semicolons after each statement. This technique is similar to *Go*'s implementation [5] of auto semicolon insertion. However, this approach is stricter than *Go*'s implementation. This is discussed in detail in the later section of the paper.

Since the language utilizes the *LLVM* backend, the compiler can benefit from extensive optimization opportunities [7]. All the techniques are discussed in detail in the later section.

The implementation of custom lexer and parser is also discussed in that section.

METHODS AND MATERIAL

A. *Literature Review*

A compiler is traditionally divided into two main components: the frontend and the backend. The frontend handles lexical, syntactic, and semantic analysis, while the backend is responsible for code generation and optimization [1].

In this work, the authors have developed a custom frontend from scratch to maintain full control over the compilation process. Although a variety of lexer and parser generators are available such as *Lex* and *YACC*, as well as more modern tools like *ANTLR*; a low-level, handcrafted approach was chosen to maintain greater control over the compilation process. This decision was guided by the intention to maintain focus on the core objectives of the project, rather than diverting attention to the evaluation and integration of additional tools.

The current implementation uses a custom lexer, along with a combination of a *Pratt parser* [3] and a *Recursive Descent parser*. The *Pratt parser* is employed for parsing expressions, as it offers excellent scalability and makes it easy to introduce new operators. A *Recursive Descent parser* is used for statement parsing, as it is straightforward to implement and sufficient for the intended purpose.

On the backend, the authors chose to target *native* code using the *LLVM* infrastructure rather than implementing a *bytecode-based* virtual machine. This decision reflects the authors' aim to ensure optimal performance. Native compilation involves translating source code directly into machine code, which runs directly on hardware and thus offers superior execution speed.

In contrast, *Python* follows a *bytecode* execution model. *Python* source code is first compiled into *bytecode*, which is then executed by the *Python* virtual machine. This indirection introduces overhead and results in slower performance compared to native execution. Bytecode approaches can be categorized into *stack-based* and *register-based* instruction sets. *Python* utilizes a *stack-based* model, which employs *zero-address* instructions, whereas *register-based* models use *three-address* instructions [9]. While *stack-based* virtual machines are simpler to implement, they tend to be less efficient than their *register-based* counterparts.

Despite the relative simplicity of *stack-based* bytecode, implementing a virtual machine from scratch remains a complex and effort-intensive task [8]. Conversely, while native code generation may seem more challenging, *LLVM* significantly lowers the barrier by providing a clean and well-documented interface. In this project, *llvmlite* [4] is leveraged to interact with *LLVM*. This allows the compiler to generate *LLVM intermediate representation (IR)*, which is then compiled by *LLVM* into optimized native code across different architectures. This approach not only simplifies backend development but also results in high-performance executables.

Currently, for the *string* implementation, *heap* allocation is performed making the language un-optimized for the *string*-related operations. A *C* library function *GC_malloc* is called for allocating memory from heap. This is a built-in *garbage collector* also known as *Boehm's Garbage Collector* [12]. It guarantees that no memory leaks while doing string-related operations. The *GC_malloc* function is called using the *Foreign Function Interface* of *Python* through *ctypes* library [11].

As this is the first iteration of the compiler, the feature set has been intentionally kept minimal. As a result, support for arrays, file I/O, and a standard library is not included in the current implementation. Due to time constraints, these features could not be implemented, but the authors plan to introduce them in future iterations of the language.

B. *Compiler Pipeline Overview*

The compiler is structured as a modular pipeline, with each stage performing a distinct task and passing its

output to the next stage. The primary components of the pipeline are:

1) *Lexer*
2) *Sanitizer*
3) *Parser*
4) *Code Generator*

Each module operates independently to ensure maintainability and extensibility.

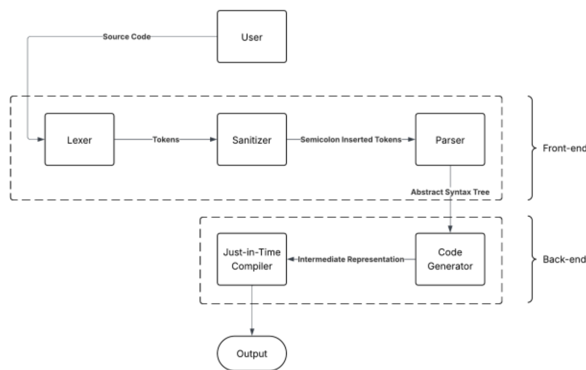

Fig. 1: Compilation Pipeline Diagram

C. *Lexical Analysis*

The compilation process begins with lexical analysis, where the raw source code is read as a *string* and passed to the Lexer. The Lexer processes the input character by character to generate a stream of tokens, each representing meaningful units such as *keywords*, *identifiers*, *literals*, and *operators*.

For example, given the statement:

name = "John";

The Lexer encounters *n* and identifies it as the start of an *identifier* or *keyword*. It then constructs the complete token name and classifies it as an *identifier*. Every token also records its position in the source code, which is essential for precise error reporting. This tokenization continues until the entire input is processed into a structured token list.

D. *Sanitization*

Whitespace tokens, such as newlines, are not retained in the implementation. As a result, statements must be explicitly separated typically with *semicolons*. While retaining *newline* characters could serve as implicit separators, this would complicate parsing multiline expressions.

Example:

```
x = "Hello " +
  "World"
```

Requiring manual *semicolon* placement would be error-prone and tedious, particularly for users familiar with *Python*'s clean syntax. To address this, the authors have implemented automatic semicolon insertion, a strategy inspired by *Go*'s compiler design [5].

The *Sanitizer* analyzes both the last token of the current line and the first token of the next line to determine where semicolons should be inserted unlike *Go*, which considers only the former. This enables support for complex constructs such as method chaining and split-line expressions.

Example:

```
x = 42
  * 69
y = 5
```

This pattern is correctly parsed by the Crython's compiler, whereas *Go*'s strategy would fail.

E. *Syntax Analysis*

The Parser takes the sanitized token stream and constructs an *Abstract Syntax Tree (AST)*. The approach combines two parsing strategies:

1) *Pratt Parsing* for expressions, which enables efficient handling of operator precedence [2] and associativity.
2) *Recursive Descent Parsing* for statements, which simplifies rule-based parsing of control structures and declarations.

Table 1: Simplified Operator Precedence Table

Operator	Precedence
Lowest	0
Sum	1
Product	2
Exponent	3

This table allows the *Pratt parser* to:

1) Determine when to stop parsing a subexpression.
2) Decide which operator binds more tightly.
3) Respect associativity (left-to-right or right-to-left).

F. ***Simplified Pratt Parsing Logic***

```
PARSE_EXPRESSION(min_prec):
  left := PARSE_PREFIX()
  while NEXT_TOKEN and PREC(NEXT_TOKEN) > min_prec:
    operator := NEXT_TOKEN
    ADVANCE()
    left := PARSE_INFIX(left, operator)
  return left
```

Each token type (e.g. *integer literal*, *float*, *parentheses*) has a dedicated handler in the parser. *Right-associative* operators like exponentiation are handled by lowering the binding precedence during infix parsing.

G. ***Recursive Descent for Statements***

For constructs such as *if*, *while*, and *function* etc., Recursive Descent Parsing is used. For example:

```
PARSE_WHILE_STATEMENT():
  condition := PARSE_EXPRESSION(LOWEST_PRECEDENCE)
  EXPECT(COLON)
  body := []
  while CURRENT_TOKEN != END:
    body.append(PARSE_STATEMENT())
  EXPECT(END)
  return WHILE_NODE(condition, body)
```

Other constructs (*if*, *function*, *break*, etc.) follow a similar structure with dedicated parsing functions.

H. ***Code Generation***

After constructing the *AST*, the next step is *code generation*, involving a recursive traversal of the *AST* to emit *LLVM IR*.

Simplified Code Emission Logic:

```
EMIT(node):
  match node.type:
    case PROGRAM: visit_program(node)
    case IF: visit_if(node)
    case WHILE: visit_while(node)
    case FUNCTION: visit_function(node)
```

Each *visit_** function translates the corresponding *AST* node into *LLVM IR*. The implementation uses *llvmlite*, a *Python* wrapper around *LLVM*, which provides a high-level and Pythonic interface for emitting *IR*. This abstraction enables the generation of efficient, portable, and optimized machine code.

Once *IR* is generated, the one can support two execution strategies:

1) *Just-In-Time (JIT)* Compilation, using *LLVM*'s *MCJIT* engine for immediate execution.
2) *Static Compilation*, producing *native* binaries via standard *LLVM* toolchains.

RESULTS AND DISCUSSION

Currently, *Crython* does not support *arrays* or *matrix* data structures. Thus, the authors are unable to evaluate the programs having array manipulation. Due to time constraints, the authors have planned to add arrays for the future implementation.

Despite this limitation, the authors have benchmarked several programs written in Crython against equivalent implementations in other widely-used languages.

A. ***Test Environment***

1) CPU: AMD Ryzen 5 4600H @ 3.0 GHz (12 logical cores)
2) RAM: 16 GiB
3) Operating System: Arch Linux (Kernel 6.15.2-2-cachyos)

Each program was executed 10 times, and the authors have recorded the *average*, *minimum*, and *maximum* execution times (in seconds) for each language.

B. ***Benchmark Configuration***

Table 2: Benchmark Configuration Table

Language	Compiler/Interpreter Version	Notes
C	gcc (GCC) 15.1.1	Compiled with –O3 flag
Go	go1.24.4	
Python	Python 3.13.5	
JavaScript	v24.2.0	

Language	Compiler/Interpreter Version	Notes
Julia	julia version 1.11.5	Warm up time excluded
Ruby	ruby 3.4.4	
PHP	PHP 8.4.8	
Crython		

C. *Benchmark Programs*

1) Summation of Even Numbers from 0 to 1,000,000,000
2) Recursive Fibonacci (n = 35)
3) Sum of Digits from 1 to 100,000,000
4) Count Numbers Divisible by 3 or 5 up to 1,000,000,000
5) Prime Number Test (n = 999998727)

Source codes for the above programs are given below. These implementations are written in *Crython*.

Program 1:

```
def sum_even_numbers() -> int:
    var sum_val: int = 0
    var i: int = 0
    while i < 1000000001:
        sum_val += i
        i += 2
    end
    print("Sum of even numbers: %d", sum_val)
    return 0
end
sum_even_numbers()
```

Program 2:

```
def fibonacci(n: int) -> int:
    if n <= 1:
        return n
    end
    return fibonacci(n - 1) + fibonacci(n - 2)
end
print("Fibonacci(35): %d", fibonacci(35))
```

Program 3:

```
def sum_digits(n: int) -> int:
    var total: int = 0
    while n > 0:
        total += n % 10
        n /= 10
    end
    return total
end
def sum_all_digits() -> int:
    var total_sum: int = 0
    var i: int = 1
    while i < 100000001:
        total_sum += sum_digits(i)
        i += 1
    end
    print("Sum of all digits from 1 to 100,000,000: %d",
total_sum)
    return 0
end
sum_all_digits()
```

Program 4:

```
def count_divisible() -> int:
    var count: int = 0
    var i: int = 1
    while i < 1000000001:
        if i % 3 == 0:
            count += 1
        elif i % 5 == 0:
            count += 1
        end
        i += 1
    end
    print("Count of numbers divisible by 3 or 5: %d",
count)
    return 0
end
count_divisible()
```

Program 5:

```
def is_prime(n: int) -> bool:
    if n < 2:
        return false
    end
    if n == 2:
        return true
    end
```

```
    if n % 2 == 0:
        return false
    end
    var i: int = 3
    while i < n ** 0.5 + 1:
        if n % i == 0:
            return false
        end
        i += 2
    end
    return true
end
var n: int = 999998727
if is_prime(n):
    print("%d is prime", n)
else:
    print("%d is not prime", n)
end
```

D. ***Benchmark Results***

Table 3: Benchmark Table 1

Language	Program 1			Program 2		
	Avg.	Min.	Max.	Avg.	Min.	Max.
C	0.001	0.001	0.001	0.02	0.02	0.03
Go	0.267	0.257	0.27	0.06	0.05	0.08
Python	16.06	15.91	16.22	1.33	1.32	1.36
JavaScript	0.42	0.41	0.428	0.13	0.12	0.14
Julia	0.148	0.143	0.164	0.2	0.18	0.22
Ruby	19.8	19.63	20.08	0.87	0.87	0.88
PHP	4.706	4.684	4.742	0.48	0.48	0.5
Crython	0.336	0.335	0.337	0.13	0.124	0.127

Table 4: Benchmark Table 2

Language	Program 3			Program 4		
	Avg.	Min.	Max.	Avg.	Min.	Max.
C	0.67	0.661	0.674	0.633	0.615	0.66
Go	0.86	0.851	0.863	1.18	1.171	1.19
Python	49.0	48.8	49.4	67.1	66.8	67.8
JavaScript	2.75	2.74	2.76	1.53	1.53	1.55
Julia	0.83	0.82	0.85	1.8	1.78	1.82
Ruby	28.4	28.3	28.5	72.8	72.5	73.6
PHP	20.2	20.1	20.3	18.1	18.0	18.1
Crython	1.13	1.13	1.13	0.85	0.80	1.27

Table 5: Benchmark Table 3

Language	Program 5		
	Avg.	Min.	Max.
C	0.001	0.001	0.001
Go	0.001	0.001	0.002
Python	0.013	0.011	0.028
JavaScript	0.032	0.024	0.05
Julia	0.155	0.141	0.171
Ruby	0.043	0.041	0.053
PHP	0.015	0.012	0.031
Crython	0.082	0.082	0.083

Table 6: Mean Speedup Table

Language	Mean Execution Time (seconds)	Speedup (vs Crython)
Python	26.6984	52.97x
Ruby	24.3719	48.35x
PHP	8.6927	17.25x
JavaScript	0.9737	1.93x
Julia	0.6266	1.24x
Crython	0.504	1x
Go	0.4723	0.94x
C	0.2646	0.53x

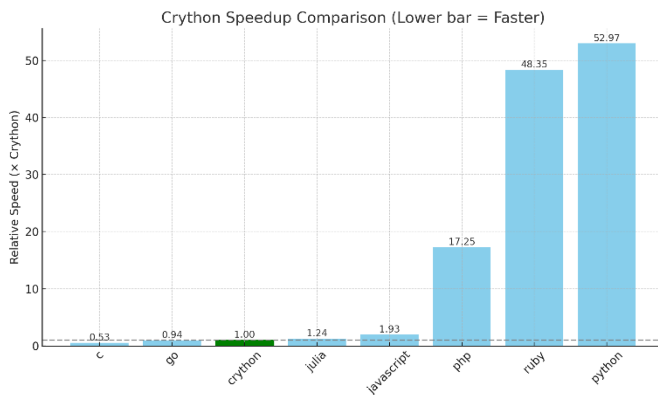

Fig. 2: Speedup Barchart

E. *Discussion*

From Fig. 2, it is clear that *Cryton* outperforms many *interpreted* languages such as *Python*, *Ruby*, *PHP* and *JavaScript* in these specific use cases. As *Crython* uses *LLVM* tools to generate *native* code, it performs considerably well among these programming languages. Although, *Go* and *Julia* perform slightly better in some cases, *Crython* remains close. As expected, *C* is the fastest overall due to its maturity and *low-level* optimization. With maturation of *Crython*, further performance improvements are expected in future iterations.

CONCLUSION

In this work, the authors introduce a compiler that combines the syntactic elegance of *Python* with the execution efficiency of *C*. The proposed language *Crython* is designed to support rapid prototyping while delivering significant performance improvements over interpreted languages like *Python*. As evidenced by benchmark results, the language performs competitively and, in many cases, surpasses several popular programming languages in terms of execution speed.

The present compiler follows a conventional multi-phase architecture, comprising lexical analysis, parsing, and code generation. A key innovation lies in the introduction of a *Sanitizer* module, which automatically inserts semicolons during the token stream preprocessing stage. This feature allows developers to omit semicolons without syntactic ambiguity, enabling them to write cleaner and more readable code while avoiding the pitfalls of *Python*'s indentation-sensitive syntax. The result is a language that feels intuitive yet avoids ambiguities during parsing.

For backend *code generation*, the authors utilize the *LLVM* toolchain via the *llvmlite* library. The compiler translates the *AST* into *LLVM IR*, which is subsequently compiled into native machine code. Unlike *Python* which compiles to *stack-based bytecode* executed within a virtual machine; *Crython* produces *machine-level* instructions that run directly on hardware. This *native* code execution offers considerable performance benefits, especially for compute-intensive workloads.

Currently, the language is kept intentionally minimal and does not include features such as arrays, file I/O, or a standard library. Future development will focus on expanding the language with support for both static and dynamic type systems, allowing developers to select either performance-critical or flexible programming paradigms.

The complete source code, along with sample programs and documentation, is publicly available on GitHub: https://github.com/SohamMalakar/purrgram-llvm

References

[1]. A. V. Aho and J. D. Ullman, Principles of compiler design. New Delhi, India: Narosa Publ. House, 1999.

[2]. "6. Expressions — Python 3.10.7 documentation," docs.python.org. https://docs.python.org/3/reference/expressions.html#operator-precedence

[3]. V. R. Pratt, "Top down operator precedence," Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73, 1973, doi: https://doi.org/10.1145/512927.512931.

[4]. "User guide — llvmlite 0.45.0dev0+224.g2fbac84.dirty documentation," Readthedocs.io, 2015.

https://llvmlite.readthedocs.io/en/latest/user-guide/index.html

[5]. "The Go Programming Language Specification - The Go Programming Language," Go.dev, 2024. https://go.dev/ref/spec#Semicolons

[6]. "7. Simple statements," Python documentation. https://docs.python.org/3/reference/simple_stmts.html#the-global-statement

[7]. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." Accessed: May 22, 2025. [Online]. Available: https://www.llvm.org/pubs/2004-01-30-CGO-LLVM.pdf

[8]. "Chunks of Bytecode Crafting Interpreters," Craftinginterpreters.com, 2015. https://craftinginterpreters.com/chunks-of-bytecode.html#bytecode (accessed May 22, 2025).

[9]. Craftinginterpreters.com, 2025. https://craftinginterpreters.com/a-virtual-machine.html#design-note (accessed May 23, 2025).

[10]. C. Ye, Z. Shen, Y. Wu, and P. Loskot, "Reconsidering Python Syntax to Enhance Programming Productivity," International Journal for Research in Applied Science and Engineering Technology, vol. 12, no. 3, pp. 776–785, Mar. 2024, doi: https://doi.org/10.22214/ijraset.2024.58903.

[11]. "ctypes — A foreign function library for Python — Python 3.9.5 documentation," docs.python.org. https://docs.python.org/3/library/ctypes.html

[12]. H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," Software: Practice and Experience, vol. 18, no. 9, pp. 807–820, Sep. 1988, doi: https://doi.org/10.1002/spe.4380180902.