Introduction

The first version of C4 compiler functioned only with integer calculations. The bonus feature adds complete f64 floating-point capabilities that let users parse floating literals while performing mixed decimal arithmetic operations and implementing comparisons and float-print functionality. Our project involved full description of implementation flow alongside the facing challenges and resolution methods.

Lexer and Parser Enhancements

Lexical Recognition:

The TokenKind type now includes a floating-point number representation as Float(f64). Within the Lexer::next_token() method one decimal point character allows the lexer to detect floating-point literals. The state-machine tracks decimal point presence before f64 parsing of the obtained string.

AST/Bytecode Emission

Instruction included two fresh bytecode entries known as ImmF(f64) for pushing float literals and PushF(f64) in combination with PrintF for printing operations. The “nud” prefix processor of the parser matches TokenKind::Float(f) then produces an instruction ImmF(f). During print(expr), we verify that the most recently produced instruction included ImmF or float involvement to use PrintF instead of Print.

Virtual Machine Changes

Value Tagging

The solution introduced a Vec<Value> stack instead of the former Vec<i64> stack while defining enum Value {Int(i64), Flt(f64)}. The stack combines integer and floating-point values into one structure while maintaining both call frame and local variable protocols without path duplication.

Instruction Dispatch

Arithmetic and comparison operations now match on (Value, Value) pairs, handling four cases:

- Int op Int → Int
- Flt op Flt → Flt
- Int op Flt → Flt
- Flt op Int → Flt

We added explicit panics for unsupported operations (e.g., % on floats).

Print Semantics
The PrintF function removes a floating-point f64 value then utilizes println!("{}", f) for display. The type-tag inspection method checks the last bytecode during print(expr) operation in mixed-mode printing.

Key Challenges and Solutions

Type Mixing and Tagging Overhead
The requirement for distinct integer and float paths within the VM system increases VM complexity by two times. The paths for integer and float statements merged into a single enum which Rust ensured proper safety through exhaustive matching.

Performance Impact
Since pattern matching applies to Value it causes a minor increase in computational expense. The solution for this problem involved maintaining pure integer loops inside optimized arm patterns and tolerating minimal expenses from float branches.

Stack and Frame Integrity
The microcode needed careful implementation within Enter, LoadLocal and Leave to maintain calling convention propriety during the stack extension for holding Value. The process of cloning float values at dereference sites helped the borrow-checker approve code while maintaining stack frames intact.

Lexical Ambiguity with .
C4 simplified the recognition between float literals and dots in other contexts by forbidding structs in its syntax allowing dots to indicate floating-point numbers when near digits.

Print Dispatch Logic
Evaluating the outcome of an expression to determine if it returned a float value instead of an integer becomes complex when working with linear bytecode streams. The solution involved checking the currently active Instruction to determine which print instruction needed appropriate tagging.

Conclusion
The(Float) transformation of C4 independent(Floating-point) from integer-only C4 resulted in a new VM with mixed fundamental modes. The combination of Rust types and enums and pattern matching mechanisms enabled developers to achieve both security and maintenance ease in their development. Our compiled program passes all existing integer tests and newly added floating-point behavior tests which enables a self-hosting C4 with modern f64 features.