F ♭ m as Minimalist Concatenative Design: A Critical Third-Party Review

Executive Summary

F ♭ m (often written "F ♭ m" in project materials) is best understood as an **experimental toy language** and a multi-implementation "learning laboratory," not as a general-purpose systems language. Its design is aggressively minimalist along two axes: (1) a single runtime value type—**arbitrary-precision integers**—and (2) a strictly **postfix / concatenative** surface model where sequencing means function composition-through-execution, with no infix "escape hatches." [1]

What makes F ♭ m more than a mere tiny stack language is the project's insistence on a **tiered toolchain** and a **shared low-level IR** that stays within the language's conceptual universe. In particular, the TypeScript core's compilation model reduces programs to a stream of big-integer instructions that can be encoded as **base64 VLQ** (including signed VLQ coding) and executed by a small interpreter; higher tiers lower to the minimal tier. [2] [3] [4]

Two distinctive semantic idioms structure real F ♭ m programs: (a) the **queue**, which doubles as both an execution stream and a return/auxiliary store (via `q<`/`q>`), and (b) **lazy quotes**, where bracketed code becomes *data*—an integer pointer to a definition—then later becomes *behavior* via `eval` or conditional execution. [4] [5]

The TypeScript/Deno toolchain adds a deliberately non-minimal step: an **optimizer** that runs peephole simplifications, constant folding, and definition inlining based on metadata markers like `.inline` and `.unsafe`. This is a notable design choice: it treats "minimal runtime" as compatible with "nontrivial compile-time structure." [6] [7]

Critically, the same minimalism that makes F ♭ m elegant as a research toy also becomes its primary practical limitation: with only big integers, semantics quickly becomes **encoding-heavy**, and without static structure, programs become **opaque and fragile** as they scale. This review argues that F ♭ m is most valuable as a *didactic specimen* for concatenative compilation/IR design and for studying what kinds of control/data can be recovered from a single-type substrate, rather than as a language one would choose for large maintainable programs. [1]

Abstract

F ♭ m is a minimalist concatenative toy language built around a single runtime value type—arbitrary-precision integers—and a strictly postfix (RPN) execution model. The project organizes its language/tooling into three tiers—**F ♭ m⁰**, **F ♭ m**, and **F ♭ m⁺**—that progressively add surface conveniences while compiling down to a shared low-level representation executed by small interpreters across host languages. [8] [5]

This article critically reviews F ♭ m as an experiment in concatenative minimalism, focusing on (i) its operational model (stack, queue, lazy quotes, pointer encodings), (ii) how its common "bytecode"/IR remains within the language's integer-only worldview and is encoded via base64

VLQ, and (iii) how the TypeScript/Deno implementation adds an explicit optimization pipeline (peephole rules, constant folding, and inlining controlled by `.inline`/`.unsafe`). [4] [3] [6]

F ♭ m is explicitly treated here as an **experimental toy language**—not intended for general-purpose systems programming—and as a vehicle for exploring minimal concatenative semantics across implementations. This is an **AI-generated third-party review** based only on public project artifacts and external context sources. [1]

Introduction

Minimal language design is a recurring methodology in programming-language research and pedagogy: aggressively constrained cores can reveal which abstractions are essential, which are optional, and which can be reconstructed mechanically from lower-level primitives. Concatenative languages are particularly amenable to such experiments because "program = composition" (sequencing is the dominant structure) and environments/variables can often be replaced with stack discipline. [9]

F ♭ m fits squarely into this tradition. It is a tiny, stack-oriented, strictly postfix language where **all runtime values are big integers**, yet the system still supports higher-level surface conveniences by compiling down to a shared low-level tier and reusing a common IR/bytecode strategy among implementations. [8] [4]

Project entry points (as provided in the prompt):

```
Source repository: https://github.com/Hypercubed/f-flat-minor
Web instance:      https://hypercubed.github.io/f-flat-minor/
```

Scope and intent: throughout, F ♭ m is treated as an **experiment and learning vehicle**—a toy language meant to be implemented and re-implemented—rather than a proposal for mainstream adoption. The goal is an accurate critical assessment of what this experiment teaches about *minimal concatenative designs* and *IR-centered tiering*, including where minimalism clarifies and where it obstructs. [1]

Background: Concatenative Minimalism

Concatenative programming languages are typically described as point-free, composition-first systems in which juxtaposition denotes composition and computation proceeds by transforming an implicit data structure (often a stack). Wikipedia's summary emphasizes that concatenation/juxtaposition corresponds to function composition and that most such languages are stack-based, though stackless models exist. [9] This is the conceptual neighborhood in which F ♭ m operates.

Joy is a canonical reference point for concatenative functional thinking. Von Thun's "Rationale for Joy" frames Joy as function-composition-centric rather than function-application-centric, with **quotation** and combinators enabling higher-order behavior without variables; it also highlights "programs as data" via quoted programs. [10] Joy's "Overview" similarly emphasizes quotations and combinators, including the well-known `dip` rewrite rule (`a [P] dip -> P a`)—a small equation that encapsulates much of concatenative technique. [11]

Floy (a “flat Joy” subset) is relevant because it argues that powerful behavior can be recovered even when quotation structure is restricted; the Floy note explicitly discusses flatness and relates this to compilation/interpreter composition ideas. [12] XY is relevant because it foregrounds a **stack+queue** operational model and treats execution as manipulation of a `[stack queue]` pair —explicitly describing the queue as “future of the computation.” [13]

This gives a useful lens for F ♭ m: it pushes minimalism by combining (i) a single runtime type (big integers) and (ii) a strict postfix surface discipline, while still offering “structured” features (quotes, definitions, macros) that lower to a small kernel plus an IR/bytecode scheme. [8]

Language Design and Tier Semantics

F ♭ m’s tiering is central to understanding it. The project describes three tiers—**F ♭ m⁰**, **F ♭ m**, and **F ♭ m⁺**—which can be summarized as follows (the exact names and their intended relationship are described in project documentation and tooling). [8] [5]

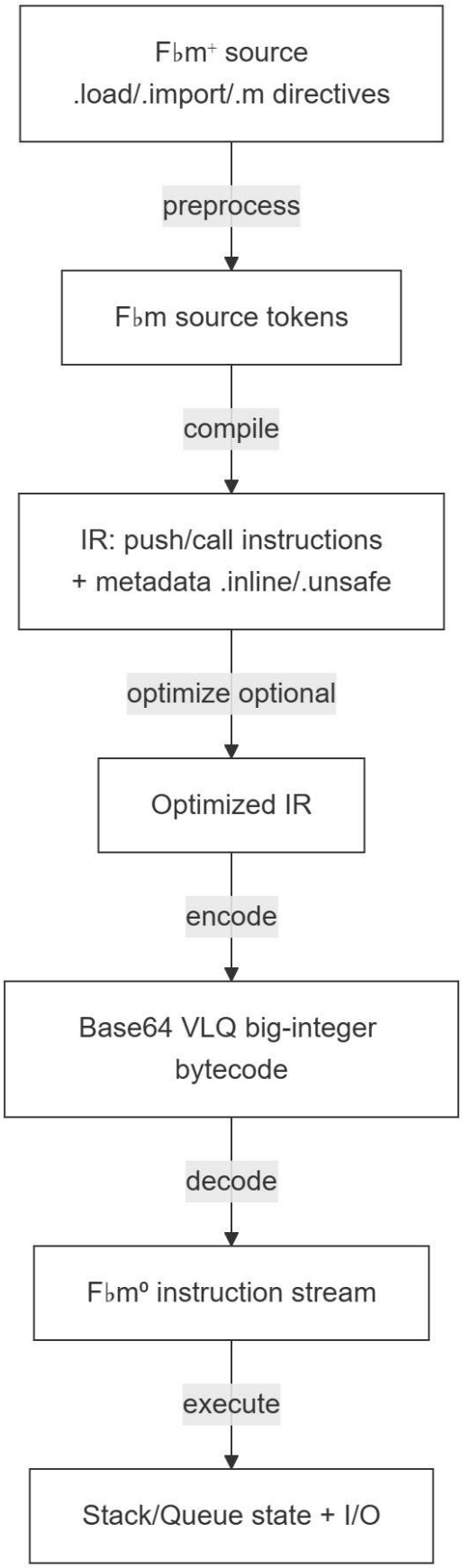
F ♭ m⁰ is the minimal kernel: a tiny instruction vocabulary, numeric literals, and the execution model. It effectively defines the “machine” that everything else targets. In the TypeScript core, this kernel corresponds closely to the `systemWords` mapping (numeric opcodes for `dup`, `swap`, `+`, `?`, `[/]`, `:/;`, `q</q>`, etc.) and the engine that executes encoded instructions. [4]

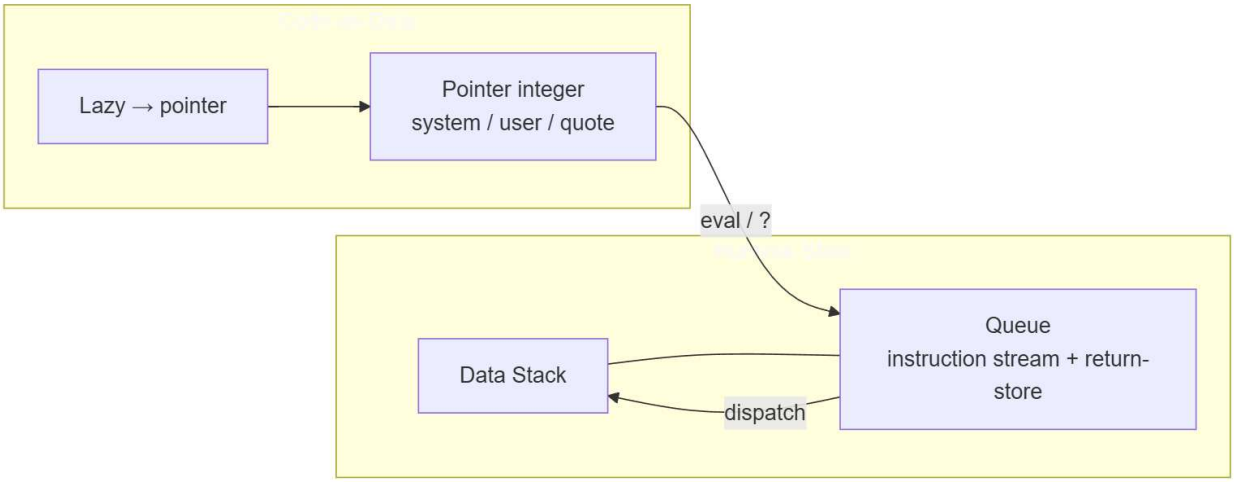
F ♭ m adds surface conveniences while remaining purely language-level (no “compiler commands”): comments, strings expanded to integer character codes, definition shorthand like `name:…;`, and bracketed quoting. In the TypeScript compiler, these emerge as tokenization rules (numbers → pushes; non-numeric tokens → calls; `'...'` strings → multiple pushes; `word:` → definition marker) and pointer forms like `[+]` that push a word-pointer integer rather than calling the word immediately. [2] [5]

F ♭ m⁺ adds preprocessor/compiler directives for scaling programs across files and enabling compile-time computation. The shared toolchain and the web UI help text enumerate directives such as `.load`, `.import`, `.m`, and `.exit`, establishing an explicit “macro/preprocessing” layer above the core language. [14] [15] [16]

A key architectural claim in the project is that higher tiers lower to the minimal tier rather than defining separate semantics. The TypeScript/Deno CLI makes this concrete: it preprocesses (optional), compiles to IR, optionally optimizes, then emits bytecode—with the runtime executing the resulting representation. [7]

Two mermaid diagrams make these relationships explicit.


Fbm+ source
.load/.import/.m directives
preprocess
Fbm source tokens
compile
IR: push/call instructions
+ metadata .inline/.unsafe
optimize optional
Optimized IR
encode
Base64 VLQ big-integer
bytecode
decode
Fbm0 instruction stream
execute
Stack/Queue state + I/O



IR, Runtime Model, and Implementations

Operational model: stack, queue, pointers, lazy quotes

Across implementations, the core operational model is "stack language," but the TypeScript engine is unusually explicit: it maintains a **data stack** and a **queue** and processes a queue of instructions while allowing the same queue to serve as a tail-based store for `q<`/`q>` (a "return stack" role). [4]

The queue matters far beyond an implementation detail: it is part of the programming model. In the tutorial materials, users are warned that `q<` should be matched with `q>`—because values are moved out of the stack and into the queue tail, and later recovered. [5]

F♭m's most distinctive move is to make "code as data" *still just integers*. There are two related constructions:

- **Pointer form**: a token like `[+]` pushes a word-pointer onto the stack (an integer encoding of the `+` word) rather than calling `+` immediately; later, `eval` can execute it. This is implemented directly in the compiler's bracket-token rule. [2] [15]
- **Lazy quotes**: bracketed code `[ ... ]` constructs an unnamed definition and leaves its pointer integer on the stack, enabling higher-order control (e.g., conditionals) without extra runtime types. The engine treats `[` and `]` as immediate structural operators when in "capture" mode, generating anonymous definitions and pushing pointers. [4] [5]

A subtle but important encoding choice appears in the TypeScript toolchain: **user-defined named words are assigned negative integer codes.** The compiler maintains a symbol map with a decrementing counter (starting at −1) for new names, while system words occupy fixed nonnegative opcode values. [2] This means pointers often carry meaning in their sign/range: negative tends to mean "named user word," while large positives can represent anonymous generated definitions (e.g., quotes) distinct from system opcodes. The by-example document shows this in practice by demonstrating pointer values like `-15` for a named word pointer and large positives for anonymous quotes. [5]

IR/bytecode format: base64 VLQ + op-tagging

F ♭ m's bytecode is crucial to its "shared semantics" story. In the TypeScript core, instructions are lowered to a sequence of big integers and then encoded as base64 VLQ. [2] [3]

The VLQ encoding used is the classic "source-map style" signed VLQ: 5-bit groups with a continuation bit, and a sign bit encoded into the least significant bit via a transform that maps negative integers to odd encodings and nonnegative integers to even encodings. [3] This matters because user opcodes may be negative, and the bytecode must represent signed values compactly.

Separately from VLQ sign handling, the compiler also tags each IR instruction as "push" vs "call" by packing an additional bit: it shifts the value left and uses the low bit to represent call-vs-push, then VLQ-encodes that combined integer stream. The engine reverses this by decoding VLQs and splitting out the op-bit and the underlying value (shift right). [2] [4]

A distinctive claim in the project is that "the IR is itself valid F ♭ m code." The TypeScript IR printer supports a "F ♭ m-compatible" textual format in which IR instructions can be emitted as normal tokens (often by pushing an integer pointer and invoking `eval`, or directly printing immediate op tokens), effectively collapsing the boundary between "IR" and "source." [17] This is a meaningful conceptual contribution: rather than treating IR as a foreign metalanguage, it is treated as another (very low-level) F ♭ m surface.

TypeScript/Deno pipeline and the optimizer

In the Deno CLI, the toolchain is explicitly staged: preprocess → compile to IR → validate → (optional) optimize → emit base64 VLQ bytecode (with a header prefix). [7]

The optimizer is a prominent non-minimal choice. It applies peephole rewrites (e.g., eliminating redundant stack shuffles), constant folding (arithmetic simplifications when both operands are compile-time known), and control-flow simplifications (e.g., reducing conditional forms when condition is constant). [6]

Inlining is not merely "always on": the compiler attaches metadata via directives like `.inline` and `.unsafe`, and the optimizer uses those flags plus size heuristics to decide whether to inline a definition at call sites. [2] [6] The standard library (`ff/lib/core.ff`) makes extensive use of `.inline`, indicating that the intended idiom is "expressive library words that compile down aggressively." [18]

Cross-implementation differences: how "the same semantics" flexes

F ♭ m's multi-language ecosystem (TypeScript/Deno/Node, Go, Racket, C++, Rust, plus additional experiments in other directories) provides a real comparative dataset for minimal VM design. [4] [19] [20] [21] [22]

Two implementation axes are especially instructive:

First, **how quotes/definitions are captured** differs. The TypeScript engine supports runtime capture with a depth counter and "immediate words" for `[/]` and `:/;`, building definitions from stack-accumulated instruction fragments. [4] In contrast, the C++ interpreter shown in

`cpp/run.cpp` parses and captures definition bodies by scanning forward in the token queue until a delimiter (`;` for definitions, `]` for quotes), rather than accumulating an explicit IR stream on the data stack. [21] This is not merely an implementation quirk; it changes what kinds of meta-programming are naturally expressible at runtime.

Second, **how "queue" is represented and used** varies. The Go engine uses the queue both as an instruction stream and as a tail-store for `q<`/`q>`; it decides "system vs user" by integer sign/range checks. [19] The Racket engine uses a separate `stash` for `q<`/`q>` and represents deferred code as pairs in a list, emphasizing clarity over raw speed or minimal structure. [20] The Rust `example0.rs` is more limited: it demonstrates big-integer stacks and pointer forms but does not implement full bracket-quote capture in the same way, illustrating the ecosystem's "partial and experimental" reality. [22]

A cautionary note: this diversity is a strength for a toy-language laboratory, but it also means that "F ♭ m semantics" is operationally defined by reference implementations and convention rather than a single formal specification.

Idioms, Examples, and Comparative Analysis

Core vocabulary with stack effects

The TypeScript `systemWords` mapping provides a canonical low-level vocabulary for at least one major implementation family (TS core and its Deno/Node/Web frontends). Below is a representative subset with conventional stack effects (top of stack on the right). Effects are inferred from the engine behavior and the opcode naming (and match the typical stack-language interpretation used in the project). [4]

Word	Role	Stack effect (approx.)	Notes
dup	duplicate	(a -- a a)	Fundamental duplication.
swap	exchange	(a b -- b a)	Fundamental permutation.
drop	discard	(a --)	Removes top item.
+ - * / % ^	arithmetic	(a b -- a◦b)	Standard big-int arithmetic; order is stack order.
< = >	comparisons	(a b -- bool)	Booleans encoded as integers (0/1 by convention).
~ & | << >>	bitwise/shift	(a b -- a◦b)	Big-int bit operations.
eval	indirect call	(ptr --)	Executes pointer integer as a word.
?	conditional	(cond ptr --)	Executes ptr iff

Word	Role	Stack effect (approx.)	Notes
			cond is nonzero.
[ ]	quote constructors	(... -- ... ptr)	Constructs anonymous definitions (lazy quotes).
q< q>	queue transfer	(a --) and (-- a)	Moves between stack and queue-tail; critical idiom.
: ;	definitions	(ptr ... --)	Begin/end definition capture; surface sugar (name:) compiles to these.
putc putn getc	I/O	varies	Print character/number; read character code.
.	debug print	(... -- ...)	Prints the stack (non-destructive in many impls).

Example one: factorial via lazy quotes and ?

A compact factorial definition appears in project examples:

```
fact: dup 1 > [ dup 1 - fact * ] ? ;
```

This uses only postfix sequencing plus a quote and a conditional; it is emblematic of how F♭m encodes "structured programming" without structured syntax. [23]

A step-by-step operational sketch (stack shown left→right, top at right):

1) Start: stack = `[n]`.
2) `dup` → `[n n]`.
3) `1` → `[n n 1]`.
4) `>` compares the top two (in stack order) and pushes a 0/1 integer; after `>`: `[n cond]` where `cond = (n > 1)`.
5) `[ dup 1 - fact * ]` creates an anonymous definition and pushes its pointer: `[n cond ptr]`.
6) `?` consumes `cond` and `ptr`. If `cond != 0`, it executes the quoted body; if not, it does nothing. Either way, it leaves `n` available as the base case "answer" candidate.

When `cond != 0`, executing the quote performs: `dup` (copy `n`), `1 -` (compute `n-1`), `fact` (recursive call), then `*` (multiply by original `n`). When `cond == 0`, the recursion stops with `n`

(i.e., `1`) already on the stack. This is the canonical concatenative recursion pattern: the base case is "do nothing," and the recursive case is "schedule more work." [4] [23]

Appreciation: this is conceptually clean—no extra boolean type, no `if` syntax node, no environment.
Critique: the same compactness depends on tacit knowledge (stack order conventions, pointer vs quote subtleties, integer-boolean convention) and is easy to break with small arity mistakes—an ongoing tax of extreme minimalism.

Example two: printing strings with the queue

F ♭ m strings are expanded to character codes (integers) pushed onto the stack; typical code pushes a sentinel `0` before the string so recursion can stop on the sentinel. The `prints` idiom in examples uses the queue as an auxiliary store to preserve a character while recursing:

```
(prints): dup [ q< (prints) q> putc ] ? ;
prints: (prints) drop ;
```

A representative program uses it like:

```
0 'Factorial' 32 '100:' 10 prints
```

These patterns appear in the repository examples and demonstrate a core F ♭ m theme: the queue is not just an execution device—it is user-visible control/data structure. [23] [4]

Operationally, `(prints)` does:

1) `dup` duplicates the current top character code.
2) The quote `[ q< (prints) q> putc ]` will run only if the duplicated code is nonzero (`?`).
3) If nonzero, it `q<` moves the character into the queue-tail (temporarily removing it from the stack), calls `(prints)` recursively on the remaining stack (printing earlier characters first), then `q>` restores the character and `putc` prints it.

This is effectively a **stack reversal via recursion**, using the queue as a return store to preserve the character across the recursive call—similar in spirit to classic `dip`-style patterns in concatenative programming, but encoded through F ♭ m's queue operators rather than distinct combinators. [4] [11]

Appreciation: the idiom is elegant and demonstrates how far recursion + queue can go without extra types.
Critique: it is also a readability cliff: the moment the code is nontrivial, the reader must simulate stack/queue state mentally. For a toy language this is acceptable—even desirable—but it limits maintainability as programs grow.

F ♭ m's "always postfix" feel in practice

A notable aspect of F ♭ m's ergonomics is that constructs often treated as "syntax" elsewhere (definitions, quotes, strings) still behave like token sequences in a postfix world. Even convenience forms (`word:`) are compilation-time sugar over core definition machinery; there is

no structural AST that the programmer can rely on at runtime, only the concatenative discipline and quote/pointer conventions. [2] [5]

Critical Assessment, Meta-Process, and Conclusion

Comparisons to Joy, Floy, XY, and modern concatenative work

F ♭ m is closest in spirit to concatenative functional traditions that treat composition as primary and quotation as central. Joy's rationale explicitly foregrounds quotations as a datatype and combinators that dequote/execute them. [24] Floy is relevant because it demonstrates how "flatness" constraints change what must be encoded and what can be derived—an echo of F ♭ m's own attempt to keep the IR inside the language's simple world. [12] XY is relevant because it makes the **stack+queue** pair explicit as the semantic state, calling the queue the "future of the computation"—a direct conceptual analog to F ♭ m's queue-as-execution-stream design. [13]

Where F ♭ m diverges sharply is the **single-type commitment**. Joy includes heterogeneous datatypes (notably quotations/lists as values), while F ♭ m chooses to reify quotations as integer pointers and represent everything as big integers. This is both its minimalist signature and its main ergonomic hazard: it forces meaning into conventions, not types.

As a contrast in the modern ecosystem, Kitten emphasizes static types and performance, describing itself as a statically typed concatenative language with Hindley–Milner type inference and additional effect/resource machinery. [25] The comparison is instructive: F ♭ m's value is not "what if we made this practical," but "what if we stripped it down until only the concatenative skeleton remains."

Conceptual contributions

F ♭ m demonstrates at least three research-friendly ideas:

It shows how far one can push a concatenative core with a **single universal runtime type** (big ints) while still supporting higher-order-ish behavior via pointers to definitions and quote construction. [4]

It treats the low-level IR not as an alien compiler artifact but as something that can be re-expressed through the language's own notions ("push integer," "call via eval," immediate tokens), reinforcing a "self-consistent" language tower rather than a split between source and IR. [17]

It makes a strong, somewhat underappreciated point: **minimal runtime semantics do not preclude interesting compile-time tooling.** The TypeScript optimizer is nontrivial—peephole rules, constant folding, and inlining controlled by `.inline`/`.unsafe` metadata—and it is used heavily by the standard library to make higher-level idioms "compile away." [6] [18]

Limitations and tradeoffs

F ♭ m's limitations are mostly the expected costs of the chosen experiment rather than "mistakes," but they are real:

Readability and maintainability degrade quickly because programs rely on non-local stack/queue discipline and integer encodings rather than explicit named structure.

The single-type model makes incorrect programs easy to write and hard to debug: integers can be data, booleans, pointers, or opcode-like tags depending on context, and the system can only detect many errors at runtime (or not at all).

Cross-implementation variation is inevitable when the spec is operational and implementation-driven. For a toy-language lab this is arguably a feature; for users expecting stable semantics, it is a hazard. [4] [21] [20]

Even performance comparisons—while interesting—reinforce that the project is an experiment: benchmark numbers differ widely across implementations and are clearly not the primary evaluation criterion.

Lessons for language design

For language designers, F ♭ m highlights a useful boundary: minimalism is enlightening when it exposes compilation structure (how "definitions," "quotes," and "control flow" lower to primitive operations), but it becomes counterproductive when the cognitive overhead of encodings obscures the intended idea.

The queue is a particularly interesting design lever. XY's explicit stack/queue semantics provide a precedent for treating "future" computation as a first-class manipulable structure. [13] F ♭ m's use of `q</q>` in everyday library code suggests that "one more structure than a stack" can dramatically increase expressive power in a minimalist setting—at the expense of mental simulation cost.

Meta-section: AI-generated review process

This review was generated by an LLM acting as a third-party reviewer. The human host provided the repository and a web-instance URL, and required a citation-first analysis. The research relied primarily on the GitHub connector artifacts (source files for the TypeScript core, Deno CLI, Go/Racket/C++/Rust implementations, and example F ♭ m programs). [8] [4] [7] [19] [20] [21]

External context (Joy, Floy, XY, Kitten, and a general definition of concatenative languages) was drawn from primary/official sources where possible: von Thun's Joy documents hosted at hypercubed.github.io, the Floy note, Apter's XY page, and the official Kitten web site. [26]

A practical limitation encountered in this session was that direct "open" access to the live web instance and GitHub pages via the web tool was unreliable; therefore, the review emphasizes repository source code and documentation rather than live web UI behavior. This is appropriate for JAIGP-style transparency: where I could not directly confirm live behavior, I did not assert it as fact.

Conclusion

F ♭ m is a compelling example of minimalist concatenative design used as a **toy-language laboratory**. Its defining constraints—**one big-integer runtime type**, **strict postfix**

concatenation, and a **tiered toolchain** ($F \flat m^0 \rightarrow F \flat m \rightarrow F \flat m^+$) that compiles downward to a shared kernel—make it both pedagogically sharp and ergonomically demanding. [8] [2]

The project's most interesting technical ideas are (i) the use of a **queue** as both execution stream and user-visible auxiliary store, (ii) **lazy quotes** as anonymous definitions represented purely by integer pointers, and (iii) an IR/bytecode strategy based on **base64 VLQ of signed big integers** that supports portability across implementations. [4] [3] [19]

For future experiments (still in toy-language spirit), the most promising directions would be: adding optional lightweight static structure (even if only as tooling annotations), experimenting with alternative queue/quote semantics that reduce the "mental simulation tax," and enriching the optimizer's explanations (e.g., emitting rewrite traces) to make the compile-time story as teachable as the runtime story. [6]

---

[8] README.md

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/README.md

[2] typescript/core/src/compiler.ts

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/typescript/core/src/compiler.ts

[3] typescript/core/src/vlq.ts

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/typescript/core/src/vlq.ts

[4] typescript/core/src/engine.ts

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/typescript/core/src/engine.ts

[5] docs/fbm-by-example.md

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/docs/fbm-by-example.md

[6] typescript/core/src/optimizer.ts

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/typescript/core/src/optimizer.ts

[7] deno/bin/ff-compile.ts

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/deno/bin/ff-compile.ts

[9] https://en.wikipedia.org/wiki/Concatenative_programming_language

https://en.wikipedia.org/wiki/Concatenative_programming_language

[10] [24] [26] https://hypercubed.github.io/joy/html/j00rat.html

https://hypercubed.github.io/joy/html/j00rat.html

[11] https://hypercubed.github.io/joy/html/j00ovr.html

https://hypercubed.github.io/joy/html/j00ovr.html

[12] https://hypercubed.github.io/joy/html/jp-flatjoy.html

https://hypercubed.github.io/joy/html/jp-flatjoy.html

[13] https://www.nsl.com/k/xy/xy.htm

https://www.nsl.com/k/xy/xy.htm

[14] typescript/core/src/preprocess.ts

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/typescript/core/src/preprocess.ts

[15] web/src/templates/help.html

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/web/src/templates/help.html

[16] go/src/compiler/compiler.go

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/go/src/compiler/compiler.go

[17] typescript/core/src/ir.ts

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/typescript/core/src/ir.ts

[18] ff/lib/core.ff

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/ff/lib/core.ff

[19] go/src/engine/engine.go

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/go/src/engine/engine.go

[20] racket/private/engine.rkt

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/racket/private/engine.rkt

[21] cpp/run.cpp

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/cpp/run.cpp

[22] rust/example0.rs

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/rust/example0.rs

[23] ff/example.ff

https://github.com/Hypercubed/f-flat-minor/blob/5f30b15c4fed384f146e92ab546f48dfbeb08e97/ff/example.ff

[25] https://kittenlang.org/

https://kittenlang.org/