Bioscoop: A Domain-Specific Language for FFmpeg Filtergraph Composition

Daniel Szmulewicz[a]

a Independent Researcher

Abstract FFmpeg is the dominant open-source multimedia processing framework. Its filtergraph syntax—a declarative language for composing processing pipelines—is expressive but static: it offers no variables, no abstraction mechanisms, and no iteration. Filtergraphs of any complexity must be written out in full or generated by string manipulation in the host language.

We ask whether filtergraph generation can be treated as a *compilation* problem rather than an API problem, and what architectural properties such an approach entails.

We present Bioscoop, a domain-specific language compiler that targets FFmpeg filtergraph syntax. Bioscoop provides two syntactically distinct front-ends—an external text-based DSL parsed via GLL parsing, and an internal macro-based DSL embedded in Clojure—that converge on an identical abstract syntax tree before evaluation.

We characterize several properties of the system. *AST convergence*—the property that both front-ends produce identical ASTs, enforced by a shared test suite—enables a single definitional interpreter to handle both input paths, guaranteeing semantic equivalence by construction. An *isomorphic intermediate representation*, whose grammar is in bijection with the FFmpeg filtergraph BNF grammar, reduces the back-end to a serializer and enables round-trip correctness, established by structural induction. A *first-class iteration construct* enables data-driven generation of filtergraph topology and filter arguments from computed values. *Two-phase locals injection* via `&env` threads Clojure lexical bindings into the DSL runtime. A *fail-soft error discipline* accumulates errors and returns typed sentinels rather than aborting, appropriate for interactive creative coding. *Dual resolution strategies*—runtime reflection on the JVM, static lookup in GraalVM native image—maximize the affordances of each execution environment.

Bioscoop is implemented in Clojure and available as open-source software. Its formal properties are established analytically and enforced by a test suite that runs identical assertions against both compilation paths.

This work demonstrates that when a target language is already structured and compositional, the intermediate representation can be isomorphic to the target grammar, the lowering problem disappears, and the back-end reduces to a serializer. It situates Bioscoop within a lineage of transpilers targeting domain notation languages—Sass targeting CSS, Hiccup targeting HTML—where the value lies in providing the abstraction mechanisms that the target language deliberately omits.

ACM CCS 2012

- **Software and its engineering** → **Domain specific languages**; *Compilers*;
- **Theory of computation** → Grammars and context-free languages;

Keywords domain-specific languages, FFmpeg, filtergraph, multi-modal DSL, Clojure, bicameral parsing, intermediate representation, two-phase locals injection

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Interpreters, virtual machines and compilers

1 Introduction

FFmpeg is the dominant open-source multimedia processing framework, used for video and audio encoding, transcoding, filtering, and composition. Its filtergraph syntax—a declarative language for composing processing pipelines—is expressive but static: it offers no abstraction mechanisms, no variables, no iteration, and no conditional logic. Complex filtergraphs must be written out in full or generated by string manipulation outside the framework.

This paper describes Bioscoop, a compiler from a Lisp-like domain-specific language to FFmpeg filtergraph syntax. The key architectural decision that distinguishes Bioscoop from prior work is treating filtergraph generation as a *compilation* problem rather than an API problem. Existing tools expose FFmpeg through function calls or builder patterns in the host language; the filtergraph remains implicit, assembled by the library as a string. Bioscoop instead defines a source language—with its own syntax, evaluation semantics, and abstraction mechanisms—that targets FFmpeg's filtergraph notation as its compilation output. To our knowledge, Bioscoop is the first source-to-source compiler whose target language is FFmpeg's filtergraph syntax.

Bioscoop addresses FFmpeg's expressiveness gap by providing:

- Named, reusable filter graph definitions (`defgraph`)
- Lexical binding (`let`)
- A first-class iteration construct (`for`) that generates parallel filtergraph structures
- Parameterized filter arguments computed at compile time
- Spec-driven parameter validation

The system supports two syntactically distinct front-ends that share a single evaluation pipeline. We describe the architectural consequences of this design, with particular attention to the properties that distinguish Bioscoop from string-generation approaches and from prior DSL work in the multimedia domain.

1.1 Programmatic Video Editing

Video editing has progressed from destructive techniques (cutting the physical negative) to non-destructive ones. This became possible once film went digital: an edit decision list stands in for physical cuts, and the underlying material is never altered. In both regimes the creative process remains manual; edits are made by hand, or by mouse, inside a timeline-based interface.

A third approach, largely unexplored in creative-coding practice, is programmatic: the editor specifies the desired transformations as a program, and an underlying system carries them out. FFmpeg is such a system, capable of expressing nearly any editing operation as a filtergraph. It is embedded in most major platforms and applications that handle media, yet its direct use for making video art remains limited—creators tend to avoid its string-based syntax, whose information density works against exploratory, iterative use.

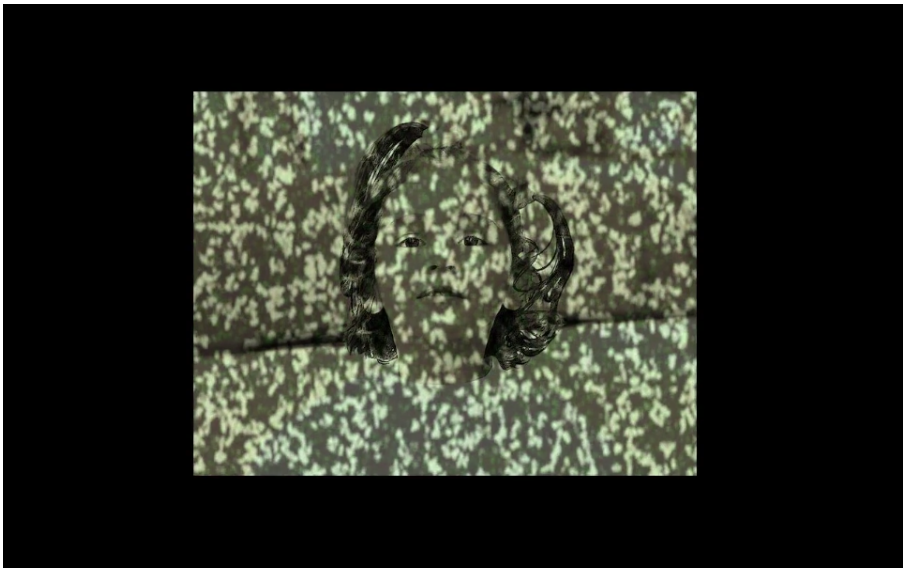

Figure 1 A frame from a music video generated by Bioscoop. Photographs are blended with Game of Life simulations whose cell colours cycle per image.

Bioscoop is motivated by this gap. By giving FFmpeg's filtergraph notation the abstraction mechanisms it deliberately omits—naming, iteration, parameterization—Bioscoop makes programmatic video editing a practical mode of creative work, not merely a latent capability of the underlying framework.

1.2 Motivating Example

Bioscoop was used to produce two official music videos for indie bands, published on YouTube.[1] In one of them, the program composites an arbitrary number of photographs with Conway's Game of Life simulations: each iteration renders them as animated cellular automata with per-iteration colour schemes, then layers the photographs through harmonic blending and crossfades. Figure 1 shows a frame from the output.

The core functions are listed in Appendix A. With 14 photographs and 20 s per image, it generates a filtergraph of approximately 3 200 characters; changing the image count, timing, or colour palette requires editing keyword arguments, not rewriting the filtergraph. A simpler slideshow generator, used as a detailed case study in §8.6, exercises the same abstraction mechanisms at a smaller scale.

[1] https://youtu.be/FmbC6oHiD6k, https://youtu.be/GUU4zqeleYY

2 Background and Related Work

2.1 FFmpeg Filtergraph Syntax

An FFmpeg filtergraph is a semicolon-separated sequence of filterchains, each a comma-separated sequence of filters. Filters carry optional bracketed input and output link labels. The BNF grammar is:

Listing 1 FFmpeg filtergraph grammar (simplified)

```
filtergraph = filterchain (";" filterchain)*
filterchain = filter ("," filter)*
filter      = linklabel* filter-spec linklabel*
filter-spec = name ("=" arguments)?
linklabel   = "[" name "]"
```

The grammar is context-free and unambiguous. There are no variables, no abstraction mechanisms, and no iteration. Filtergraphs of any complexity must be written as single expressions.

2.2 Existing Approaches

Existing tools for programmatic FFmpeg filtergraph generation fall into three categories. *API wrappers* such as ffmpeg-python [12] and fluent-ffmpeg [16] expose FFmpeg's options as function calls but treat the filtergraph itself as an opaque string to be concatenated. *Higher-level abstractions* such as MoviePy [18] hide FFmpeg almost entirely, trading expressiveness for simplicity.

A third category is represented by python-ffmpegio [13], which provides a structured Python object model (`Filter`, `Chain`, `Graph`) that serializes to filtergraph strings and can parse native filtergraph strings back into the same representation. This is the closest prior work to Bioscoop's intermediate representation. However, python-ffmpegio's filtergraph builder is an API, not a source language: there is no separate syntax, no named definitions with lexical scope, no iteration construct, and no parameter validation. Users construct filtergraphs by combining Python objects with operator overloading (`+`, `|`). Bioscoop differs in kind: it provides a source language with its own parser and evaluator, named abstraction mechanisms that are compiled away before serialization, and spec-driven validation at compile time.

2.3 DSL Implementation in Lisp

The design of internal DSLs in Lisp is well-studied. Hudak's taxonomy [9] distinguishes shallow embeddings, where host language constructs carry DSL semantics, from deep embeddings, where DSL programs are represented as data and interpreted by a separate evaluator. Bioscoop uses a deep embedding: Clojure forms are rewritten into a tagged vector AST that is then evaluated by a definitional interpreter.

Racket's `#lang` mechanism [6] and Clojure's macro system [8] both support what we call AST convergence naturally, though we are not aware of prior work that names or formalizes this property in the context of multi-modal DSLs.

3 Architecture

3.1 Overview

Bioscoop consists of six layers:

1. *Two front-ends*: a GLL parser for the external text DSL and a macro for the internal Clojure DSL
2. *A shared AST*: tagged Clojure vectors produced identically by both front-ends
3. *A definitional interpreter*: transform-ast, a multimethod dispatching on AST node type
4. *An intermediate representation*: algebraic records (Filter, FilterChain, FilterGraph)
5. *A back-end*: to-ffmpeg, a serializer from IR to filtergraph string
6. *An error channel*: a dynamically-scoped accumulator for compile-time errors

3.2 AST Convergence

A DSL with two front-ends conventionally requires two independent transformation pipelines: an external, text-based front-end parses source into a custom AST that a dedicated transformer walks, while an internal, macro-based front-end typically compiles Clojure forms directly into target code, bypassing any shared intermediate representation. This duplicates the logic that gives AST nodes their meaning across two independently maintained implementations, with no structural mechanism keeping them behaviorally aligned. Bioscoop avoids this duplication by converging both front-ends on the same AST before evaluation.

Both front-ends produce ASTs consisting of tagged Clojure vectors. The node types are: :program, :compose, :for-binding, :let-binding, :binding, :padded-graph, :list, :map, :symbol, :keyword, :string, :number, :boolean, :label, :graph-definition.

Formally, let $P_e : L_e \to AST$ be the external parser and $M_i : L_i \to AST$ be the macro. AST convergence holds when:

$$\forall p \in L_e \cap L_i : P_e(p) = M_i(p) \tag{1}$$

where $L_e \cap L_i$ denotes programs expressible in both surface syntaxes. This enables a single transformer $T : AST \to IR$ to handle both paths, guaranteeing semantic equivalence by construction.

What makes this achievable is what Krishnamurthi [11] calls a *bicameral* parsing pipeline. In Clojure, the reader stage converts source text into standard Clojure data structures—lists, vectors, symbols, maps—and the macro receives these directly at compile time. Crucially, Clojure's reader output *is* standard Clojure data: not an intermediate format, not special syntax objects as in Racket, but the same lists and vectors manipulated anywhere in a Clojure program. The form->ast function can therefore traverse a Clojure form using ordinary Clojure functions (cond, first, mapv) and produce tagged vectors.

Bicamerality makes AST convergence *possible*; it does not make it *guaranteed*. Any two independent implementations of a source-to-AST transformation will diverge unless actively maintained. The term *homoiconicity*, commonly invoked to explain this kind of macro power, is imprecise: as Krishnamurthi demonstrates, it describes either a trivially universal property (any language with strings can represent programs as data) or a specific and problematic feature (eval), neither of which is the actual mechanism here. The mechanism is the bicameral pipeline. What makes the convergence guarantee hold is a deliberate design decision: form->ast was written to emit the same tagged-vector node types that the GLL parser emits, and the test suite enforces this pairing at every new construct:

Listing 2 Convergence test pattern

```
(deftest macro-string-convergence
  (is (= (to-ffmpeg
          (compile-dsl
            "(chain (scale 1920 1080) (eq))"))
         (to-ffmpeg
          (bioscoop
            (chain (scale 1920 1080) (eq)))))))
```

The external front-end uses Instaparse [4], which implements the GLL (Generalized LL) parsing algorithm [17]. GLL handles the full class of context-free grammars, including ambiguous and left-recursive ones, while retaining the structural clarity of recursive descent. For an unambiguous grammar, as Bioscoop's is, GLL runs in linear time.

The internal front-end is the bioscoop macro:

Listing 3 The bioscoop macro

```
(defmacro bioscoop [& forms]
  (let [ast-nodes (mapv form->ast forms)
        program-ast (vec (concat [:program] ast-nodes))
        locals (keys &env)
        ns-sym (ns-name *ns*)]
    `(binding [config/*dynamic-resolution* true]
       (let [env# (reduce (fn [e# [k# v#]]
                            (env-put e# k# v#))
                          (assoc (make-env) :bioscoop/compile-ns
                                 (the-ns '~ns-sym))
                          ~(mapv (fn [sym]
                                   [(str sym) sym])
                                 locals))]
         (dsl/run-ast ~program-ast env#)))))
```

The ns-sym capture and :bioscoop/compile-ns injection ensure that namespace-aware resolution (for function lookup and var resolution) uses the compilation namespace rather than the runtime *ns*, which is a thread-local binding that can drift under REPL interaction and test runners. This mechanism is structurally parallel to the locals injection described in the following section: in both cases, the macro captures information at expansion time that would otherwise be unavailable at runtime.

form->ast is a structural transformation from Clojure forms to AST nodes. It handles all surface syntax cases: function calls, symbols, keywords, strings, numbers, booleans, padded graphs, maps, and the for, let, and compose special forms.

3.3 Two-Phase Locals Injection via &env

A significant design challenge is threading Clojure lexical bindings into the DSL's runtime environment. Consider:

Listing 4 Parameterized iteration with lexical scope

```
(defn n-transition [n offset]
  (bioscoop
    (for [i (range n)]
      [[(str "v" i)]
       (xfade {:transition "fade"
               :duration   1
               :offset     (+ i offset (* i offset))})
       [(str "t" (inc i))]])))
```

Here n and offset are Clojure locals, not DSL bindings. The bioscoop macro must make them visible to the DSL evaluator at runtime.

The solution exploits the two-phase nature of macro expansion: (keys &env) captures the names of all local bindings at *expansion time*, as a sequence of symbols. The macro then generates code that, at *runtime*, evaluates those symbols to their current values and injects them into the DSL environment:

Listing 5 The locals injection mechanism

```
; At expansion time, for locals [n offset]:
; (keys &env) = (n offset)
; (mapv (fn [sym] [(str sym) sym]) locals)
; produces the literal: [["n" n] ["offset" offset]]
;
; At runtime, n=10 and offset=6, so:
; [["n" 10] ["offset" 6]]
; is injected into the DSL env
```

The key asymmetry is that (str sym) converts the symbol to its string name at expansion time—the only moment the name is available as a symbol—while sym remains unevaluated so that the generated code reads its value from the live call frame at runtime. This is a two-phase computation: names extracted at compile time, values read at runtime.

4 The Intermediate Representation

4.1 Algebraic Structure

The IR is defined by three record types:

Listing 6 IR record definitions

Table 1 IR to FFmpeg Grammar Correspondence

IR	FFmpeg BNF
FilterGraph	filtergraph
FilterChain	filterchain
Filter	filter
:name	filter-name
:args	filter-arguments
:input-labels	input-linklabels
:output-labels	output-linklabels
; between chains	filterchain separator
, between filters	filter separator

```
(defrecord Filter     [name args
                       input-labels output-labels])
(defrecord FilterChain [filters])
(defrecord FilterGraph [chains])
```

These define a context-free grammar over Clojure records:

$$\begin{aligned} \text{FilterGraph} &::= [\text{FilterChain}^*] \\ \text{FilterChain} &::= [\text{Filter}^+] \\ \text{Filter} &::= \text{name} \times \text{args} \times \text{labels}^* \times \text{labels}^* \end{aligned}$$

Well-formedness is enforced by construction: make-filtergraph accepts only vectors of FilterChain records, make-filterchain accepts only vectors of Filter records. Invalid structures cannot be constructed through the public API.

4.2 Isomorphism with the Target Grammar

The IR grammar and the FFmpeg filtergraph BNF grammar are in bijection:

This isomorphism has a significant architectural consequence: the back-end to-ffmpeg is a *serializer* rather than a compiler pass. It traverses a structure already in the shape of the target and emits delimiters:

Listing 7 to-ffmpeg as serializer

```
(extend-protocol Renderable
  Filter
  (to-ffmpeg [f]
    (let [in  (str/join ""
                (map #(str "[" % "]")
                  (:input-labels f)))
          out (str/join ""
                (map #(str "[" % "]")
                  (:output-labels f)))
          args (when (:args f)
                 (str "=" (str/join ":"
```

```
            (map (fn [[k v]]
                   (str (name k) "=" v))
                 (:args f)))))]
    (str in (:name f) args out)))
  FilterChain
  (to-ffmpeg [fc]
    (str/join "," (map to-ffmpeg (:filters fc))))
  FilterGraph
  (to-ffmpeg [fg]
    (str/join ";" (map to-ffmpeg (:chains fg)))))
```

No information is added or removed. The serializer is a homomorphism from the IR algebra to strings under concatenation.

4.3 Round-Trip Correctness

The isomorphism also enables a bidirectional parser: ffmpeg-parser parses native FFmpeg filtergraph strings into the same record structure that the DSL compiler produces. We state the round-trip property formally and give a proof sketch by structural induction.

Let $\mathscr{S}$ denote the set of well-formed FFmpeg filtergraph strings, $\mathscr{I}$ the set of well-formed IR values, $\sigma : \mathscr{I} \to \mathscr{S}$ the serializer to-ffmpeg, and $\pi : \mathscr{S} \to \mathscr{I}$ the parser ffmpeg-parser.

Theorem 4.1 (Round-trip correctness)**.** *For all $v \in \mathscr{I}$, $\pi(\sigma(v)) = v$.*

Proof sketch. By structural induction on the IR grammar.

Base case: Filter. Let $f = \langle n, a, \ell_i, \ell_o \rangle$ where n is the filter name, a the argument map, ℓ_i the input labels, and ℓ_o the output labels. The serializer produces:

$$\sigma(f) = \underbrace{[\ell_{i_1}]\cdots[\ell_{i_k}]}_{\text{input labels}}\ n\ \underbrace{= a_1 : a_2 : \cdots}_{\text{args}}\ \underbrace{[\ell_{o_1}]\cdots[\ell_{o_m}]}_{\text{output labels}}$$

The FFmpeg grammar rule filter = linklabel* filter-spec linklabel* parses this string unambiguously, recovering n from filter-name, a from filter-arguments, ℓ_i from leading linklabels, and ℓ_o from trailing linklabels. Since labels are stored as first-class fields (not metadata), equality holds: $\pi(\sigma(f)) = f$.

Inductive case: FilterChain. Let $c = [f_1, \ldots, f_n]$ where each f_j is a Filter. The serializer produces $\sigma(f_1), \sigma(f_2), \cdots, \sigma(f_n)$. The FFmpeg grammar rule filterchain = filter ("," filter)* partitions this string at each top-level comma (commas do not appear in filter names, argument keys, or labels), yielding n substrings. By the base case, $\pi(\sigma(f_j)) = f_j$ for each j. Therefore $\pi(\sigma(c)) = [f_1, \ldots, f_n] = c$.

Inductive case: FilterGraph. Let $g = [c_1, \ldots, c_m]$ where each c_k is a FilterChain. The serializer produces $\sigma(c_1)\,;\sigma(c_2)\,;\cdots;\sigma(c_m)$. The FFmpeg grammar rule filtergraph = filterchain (";" filterchain)* partitions this string at each top-level semicolon (semicolons do not appear within filterchains). By the inductive case, $\pi(\sigma(c_k)) = c_k$ for each k. Therefore $\pi(\sigma(g)) = [c_1, \ldots, c_m] = g$. □

The converse—$\sigma(\pi(s)) = s$ for all $s \in \mathscr{S}$—does not hold in general because the FFmpeg serialization format admits multiple syntactic representations of the same

filtergraph (equivalent argument orderings, optional whitespace). Bioscoop normalizes these to a canonical form during parsing. The round-trip that is guaranteed is IR → string → IR, which is the direction that matters for the compiler's correctness.

5 The Evaluator

5.1 Definitional Interpreter in Environment-Passing Style

transform-ast is a definitional interpreter [15] written in environment-passing style. Each defmethod gives the meaning of one syntactic form by structural recursion over the parse tree:

Listing 8 The evaluator entry point

```
(defmulti transform-ast*
  (fn [node env] (first node)))

(defn transform-ast [node env]
  (trace> node env (transform-ast* node env)))
```

The evaluator is:

- *Single-pass*: one recursive descent over the AST
- *Eagerly evaluated*: applicative-order evaluation
- *Lexically scoped*: the environment is a chain of frames linked by parent pointers
- *Total*: every well-formed input produces a FilterGraph, possibly empty

Tracing is delegated to trace>, defined in a separate bioscoop.trace namespace:

Listing 9 The trace> helper

```
(defn trace> [node env result]
  (tap> (cond-> {:node-type (first node)
                 :node      node
                 :env       env
                 :result    result}
          (instance? FilterGraph result)
          (assoc :ffmpeg (to-ffmpeg result))))
  result)
```

Every node's type, environment state, and result are forwarded to Clojure's tap> facility. The :ffmpeg key is added conditionally via cond->: it is only present when result is a FilterGraph, so the tap map never carries a nil serialization. Absence of :ffmpeg is itself the signal that the result is a partial value—no separate boolean flag is needed. Callers register tap subscribers to capture trace data; no subscribers means no overhead.

5.2 Environment Model

The environment is a Clojure map extended with a parent pointer:

Listing 10 Environment operations

```
(defn make-env
  ([] {:errors (atom [])})
  ([parent] (assoc (select-keys parent [:errors :bioscoop/compile-ns])
                   :parent parent)))

(defn env-get [env sym]
  (if (contains? env sym)
    (get env sym)
    (when-let [parent (:parent env)]
      (env-get parent sym))))

(defn env-put [env sym val]
  (assoc env sym val))
```

New scopes are created with (make-env parent), establishing the parent chain for lexical scope traversal; child scopes share the parent's error atom (so errors raised at any depth surface in the single atom inspected after compilation) and inherit the compile-namespace binding for var resolution.

contains? rather than if-let is used to correctly handle bindings to falsy values.

5.3 The for Construct

The for construct is the most significant addition to the filtergraph language. It iterates over any seqable Clojure value and produces a FilterGraph by merging the results of evaluating the body expression for each iteration value:

Listing 11 for-binding evaluation

```
(defmethod transform-ast* :for-binding
  [[_ [_ sym-name] range-node body] env]
  (let [xs (transform-ast range-node env)]
    (vec (for [x xs
               :let [loop-env (env-put env sym-name x)]]
           (transform-ast body loop-env)))))
```

The method returns a plain Clojure vector of per-iteration results. Composition into a FilterGraph is handled upstream by promote-to-filtergraph, which contains a sequential? branch that recursively promotes each element and combines them with compose-filtergraphs:

Listing 12 promote-to-filtergraph* with sequential? branch

```
(defn promote-to-filtergraph* [f]
  (fn promote [x env]
    (cond
      (instance? FilterGraph x) x
      (instance? FilterChain x) (make-filtergraph [x])
      (instance? Filter x)
        (make-filtergraph [(make-filterchain [x])])
      (sequential? x)
        (apply compose-filtergraphs
               (map #(promote % env) x))
      :else (f env x :not-a-filtergraph))))
```

This separation of concerns means the for-binding evaluator need not know whether its callers want a FilterGraph, a label list, or some other interpretation. The range expression is evaluated against the current environment, so it can reference DSL bindings and Clojure locals injected via &env. All of Clojure's sequence-producing functions—range, map, filter, iterate—are available as range expressions.

The for construct also appears in label position, where it generates a sequence of label strings rather than a filtergraph:

Listing 13 for in label position

```
; Generates input labels ["b0c" "b1c" ... "b(n-1)c"]
; for the stacking filter:
[[(for [i (range n)] (str "b" i "c"))] stacking]
```

This is handled naturally by the padded-graph evaluator: when a label expression evaluates to a sequential value, the handler flattens it via mapcat. Since for-binding returns a plain vector, a for expression in label position produces the expected sequence of label strings without any special dispatch on the node type.

5.4 Chain Flattening

The chain construct flattens heterogeneous sequences of Filter, FilterChain, and single-chain FilterGraph values into a single FilterChain:

Listing 14 chain flattening via promote-to-filterchain

```
"chain"
(make-filterchain
  (vec (mapcat #(promote-to-filterchain % env)
               transformed-args)))
```

promote-to-filterchain is a normalization function parameterized by an error handler:

Listing 15 promote-to-filterchain* factory

```
(defn promote-to-filterchain* [f]
  (fn promote [x env]
    (cond
      (instance? Filter x) [x]
      (instance? FilterChain x) (:filters x)
      (instance? FilterGraph x)
        (if (= 1 (count (:chains x)))
          (:filters (first (:chains x)))
          (f env x :chain-parallel-filtergraph))
      (sequential? x) (vec (mapcat #(promote % env) x))
      :else (f env x :not-a-filtergraph))))
```

This allows defgraph-defined named graphs to be used transparently as chain stages, since a defgraph with a linear body produces a single-chain FilterGraph that flattens into its constituent filters. Multi-chain filtergraphs—representing parallel structure—are rejected as a type error.

6 Error Handling

6.1 Fail-Soft with Typed Sentinel

Bioscoop targets interactive creative coding at the REPL. In this context, a hard-failure model—where compilation errors throw exceptions and abort evaluation—is counterproductive. Partial results and immediate feedback matter more than strict failure boundaries.

The error discipline is fail-soft with explicit sentinel: all error paths return (make-filtergraph []) and accumulate a structured error record in a dynamically-scoped error atom. The first error is printed immediately at the point of accumulation. The atom is inspectable after compilation completes:

Listing 16 Error accumulation with deduplication

```
(defn accumulate-error*
  ([env error]
   (accumulate-error* (make-filtergraph []) env error))
  ([return-val env error]
   (let [info (ex-data error)]
     (when-not (some #(= (ex-data %) info) @(:errors env))
       (log/warn (if *warn-verbose* info (:error-type info)))
       (swap! (:errors env) conj error)))
   return-val))
```

The when-not guard prevents the same error from being logged and accumulated more than once. This matters when error-producing nodes appear in paths that are traversed multiple times—for instance, a graph definition that is referenced repeatedly. Without the guard, the error atom would accumulate duplicate entries and the log would be flooded.

The return-val parameter, defaulting to an empty FilterGraph, allows error handlers to return type-appropriate sentinels. Filterchain contexts return [] since their results feed mapcat. The promotion functions are wired directly to the accumulator:

Listing 17 Direct wiring of promotion functions to error handler

```
(def promote-to-filtergraph
  (promote-to-filtergraph* accumulate-error))

(def promote-to-filterchain
  (promote-to-filterchain* (partial accumulate-error [])))
```

promote-to-filtergraph passes accumulate-error directly, since the factory will supply the empty-FilterGraph sentinel via the default arity. promote-to-filterchain pre-applies the [] sentinel inline, since filterchain contexts must return an empty vector for mapcat rather than an empty FilterGraph.

6.2 Invariant: transform-ast Always Returns FilterGraph

The central invariant of the evaluator is that transform-ast, when called in filter-producing position, always returns a FilterGraph. Error paths return an empty FilterGraph rather than a map, an exception, or nil. This eliminates defensive type-checking throughout the evaluator: no method needs to guard against receiving a non-FilterGraph from a recursive call.

The invariant is established at the program boundary by promote-to-filtergraph, which is the only function that transitions from the untyped world of evaluated expressions to the typed world of IR values. Every method that consumes a recursive result calls promote-to-filtergraph before working with it.

7 Spec-Driven Validation

Filter parameters are validated against clojure.spec [7] specifications before construction. Each filter has a corresponding spec defining the valid types and ranges of its parameters:

Listing 18 Example spec for the lagfun filter

```
(s/def ::decay  (s/double-in :min 0 :max 1))
(s/def ::planes (s/int-in 0 16))
(s/def ::lagfun (s/keys :opt-un [::decay ::planes]))
```

Parameter maps use unqualified keys in DSL source. spec-aware-namespace-map resolves these to their correct namespace-qualified forms by consulting the spec registry:

Listing 19 Namespace resolution via spec introspection

```
(defn spec-aware-namespace-map [spec-keyword m]
  (let [spec-form  (s/form spec-keyword)
        key-mapping
        (into {}
          (map (fn [qk]
                 [(keyword (name qk)) qk])
               (all-keys spec-form)))]
    (into {}
      (map (fn [[k v]]
             [(get key-mapping k k)
              (coerce-numeric v)])
           m))))
```

The coerce-numeric function converts Clojure Ratio values (produced by integer division) to Double, ensuring compatibility with specs defined via s/double-in. This is a latent type error that expression arguments expose: (/ 1 3) produces a Ratio in Clojure, not a Double.

8 Discussion

8.1 Applicability of the Isomorphic IR Approach

The absence of a lowering problem in Bioscoop is a consequence of domain analysis, not a limitation of ambition. FFmpeg filtergraphs are already a high-level, compositional, declarative language. The source language and target language are at the same level of abstraction. There is no semantic gap to bridge, only a syntactic and ergonomic one.

This situation is not universal. A compiler from a higher-level language to machine code must lower closures, garbage collection, type erasure, and many other abstractions that have no counterpart in the target. The isomorphic IR approach is appropriate when:

1. The target language is already structured and compositional
2. The source language adds abstraction mechanisms rather than new semantic concepts
3. The primary function of the compiler is elaboration—name resolution, validation, iteration expansion—rather than semantic transformation

FFmpeg filtergraphs satisfy all three conditions. Configuration languages, build system descriptors, query languages, and other declarative domain notations typically do as well.

8.2 AST Convergence as a Design Pattern

AST convergence carries a structural tradeoff. The benefit is strong: *every* construct added to both front-ends is guaranteed semantically equivalent for all inputs on which the two surface syntaxes overlap—no manual synchronization of separate evaluation pipelines, no diverging semantics. The cost is that every new language construct incurs a two-site change: the grammar must gain a production, and `form->ast` must gain a corresponding case, both emitting the same tagged-vector node type. The test pattern shown in §3.2 catches any mismatch, but it cannot write the second implementation—the developer must do that.

This cost is acceptable when the language is *small and stable*. Bioscoop's DSL has approximately one dozen AST node types, and its scope is bounded by FFmpeg's filtergraph semantics; new construct additions are rare once the core language is complete. In a DSL with dozens of constructs under active evolution, the two-front-end discipline may become a drag on velocity, and a single-front-end approach—either the external text DSL or the macro-based DSL alone—would be the pragmatic choice. The decision to maintain two convergent front-ends is therefore a judgment about the language's growth trajectory and the value of offering both the REPL-interactive and text-file modalities to the same users.

8.3 Single-Pass Architecture

The evaluator is single-pass in the sense that one recursive descent over the AST produces the final IR. There are no analysis phases, no optimization passes, and no separate code generation step. The :program method processes graph definitions before regular expressions, threading the environment through each definition in turn so that a later definition may reference an earlier one; this establishes an ordering constraint, but it does not constitute a second pass: no new representation is produced, and every graph definition is resolved through the same environment-passing mechanism as the rest of the evaluator.

This characterization aligns with the classical definition: a compiler pass is a complete traversal of the program representation that produces a new representation. Since Bioscoop uses one representation throughout, it has one pass regardless of the traversal order of individual nodes.

8.4 Dual Resolution Strategies

Bioscoop targets two distinct execution environments: the JVM Clojure runtime and GraalVM native image. The resolve-function and resolve-symbol multimethods each select between two resolution strategies via the *dynamic-resolution* dynamic var—one for filter and function calls, the other for named-graph references.

The JVM path uses runtime reflection via ns-resolve. This affords capabilities unique to the JVM's open-world execution model: vars defined after the program started, graphs interned via defgraph in arbitrary namespaces, and fully qualified symbol resolution across dynamically loaded code. User-defined functions are resolved by searching the current namespace and its transitive dependencies at runtime, enabling interactive development patterns where graphs are defined and redefined at the REPL without restarting the system. Named-graph symbols are resolved via resolve-var. Both the local environment and the var namespace are consulted; when a symbol is present in both, the environment binding shadows the var binding—standard lexical scoping—with an optional warning available via *warn-on-shadowing* (which defaults to false).

The GraalVM native image path uses a static lookup table populated at compile time from the public vars of bioscoop.built-in and clojure.core. GraalVM's closed-world compilation model restricts dynamic class loading and requires explicit reflection configuration. The static table is the correct response to these constraints: it is fast, allocation-free, and requires no reflection at runtime. Named-graph symbols follow the same constraint from the other direction: only bindings already present in the local environment resolve, and any other symbol is returned unresolved rather than triggering a var lookup.

These are not two implementations of the same mechanism but two distinct strategies, each maximizing the affordances of its execution environment—for symbol resolution as much as for function resolution. The *dynamic-resolution* dynamic var is the abstraction boundary between them, selected automatically by the macro path and configurable for programmatic use.

8.5 Limitations

The current system has one notable limitation. *Label arity at compose boundaries*: The for construct can generate n parallel filtergraph chains, each with computed output labels. But a subsequent compose stage that collects all n outputs as inputs to a single filter—such as concat or xstack with n inputs—must still list those labels explicitly. A splice operator for label sequences at compose boundaries would close this gap.

8.6 Case Study: Slideshow Generator

We present a case study that exercises Bioscoop's key abstraction mechanisms at a realistic scale. The program generates a cross-fading slideshow from an arbitrary number of input images, using defgraph for reusable filter stages, compose to combine parallel pipelines, and for with conditional logic to generate filtergraph topology driven by the image count.

Listing 20 Slideshow generator (excerpt)

```
(defn screen [duration]
  (bioscoop (color {:color "black" :size "hd1080" :rate 120 :duration duration})))

(defgraph padding
  (chain (pad {:width "'iw*1.25'" :height "'ih*1.25'" :x "'(ow-iw)/2'" :y "'(oh-ih)/2'"})
         (scale {:size "hd1080" :force_original_aspect_ratio "decrease"})
         (setsar {:sar "1"})))

(defn transition [& {:keys [duration offset style] :or {style "fade" duration 1}}]
  (bioscoop (xfade {:transition style :duration duration :offset offset})))

(defn slideshow-base [n duration]
  (bioscoop
   (compose
    [(chain (screen duration) (split {:outputs n}))
     [(for [i (range 0 n)] (str "screen" i))]]
    (for [i (range 0 n)]
      [[(str i ":v")] padding [(str "padded" i)]])
    (for [i (range 0 n)]
      [[(str "screen" i)] [(str "padded" i)]
       (overlay {:x "'(W-w)/2'" :y "'(H-h)/2'"})
       [(str "overlay" i)]]))))

(defn slideshow-concat [{:keys [n duration]}]
  (bioscoop
   (compose (slideshow-base n duration)
            [[(for [i (range 0 n)] (str "overlay" i))]
             (chain (concat {:n n :v 1 :a 0})
                    (format {:pix_fmts "yuv420p"}))
             ["out"]])))

(defn slideshow-xfade [{:keys [n duration]}]
  (bioscoop
   (compose (slideshow-base n (inc duration))
            (for [i (range 0 (dec n))]
              (if (zero? i)
                [[(str "overlay" i) (str "overlay" (inc i))]
```

```
            (transition :offset duration) [(str "out" i)]]
           [[(str "out" (dec i))]
            [(str "overlay" (inc i))]
            (transition :offset (+ (* i duration) duration))
            [(str "out" i)]]))
        [[(str "out" (- n 2))] (format {:pix_fmts "yuv420p"})
         ["out"]]))

(def filtergraph {:xfade  slideshow-xfade :concat slideshow-concat})

(defn make-slideshow [& {:keys [images duration style] :or   {duration 4 style :xfade}}]
  (with-inputs {:filtergraph (to-ffmpeg ((style filtergraph) {:n (count images) :duration duration}))
                :maps ["-map" "[out]"]
                :verbose true
                :blocking true
                :out-filename "slideshow.mp4"}
               images))
```

padding is a reusable filter chain, defined once via defgraph and referenced as a filter-stage expression in slideshow-base. transition is a parameterized helper that wraps xfade with keyword arguments and defaults, reducing repetition in the crossfade variants that follow. slideshow-base uses three for blocks inside a single compose: the first splits a colour source into n labelled outputs; the second pads each input image and assigns computed output labels padded0, padded1, etc.; the third overlays corresponding image and screen pairs, producing overlay0 through overlay($n-1$). The two for blocks that share an iteration variable are sequenced within the same compose, so their label chains connect: the second block's computed output labels supply the third block's computed input labels.

slideshow-concat collects all n overlay outputs at a compose boundary via for in label position—the expression (for [i (range 0 n)] (str "overlay" i)) generates the sequence "overlay0" through "overlay($n-1$)"—and routes them into a single concat filter. This illustrates the limitation noted in §8.5: a for in label position can generate the label list, but the compose boundary must still list it explicitly; a splice operator would remove that last bit of ceremony.

slideshow-xfade layers cross-fades on top. A single for loop with an if branch generates a cascade: iteration 0 consumes overlay0 and overlay1 and produces out0; every subsequent iteration consumes the previous out($i-1$) together with overlay($i+1$) and produces out$i$. A final format stage normalises pixel formats. The offset expression (+ (* i duration) duration) produces the staggered schedule $4, 8, 12, \ldots$ so that each new image appears exactly as the previous transition finishes.

filtergraph is a plain Clojure map associating keywords :xfade and :concat with their implementations. make-slideshow accepts a :style keyword argument, uses it to look up the corresponding function from filtergraph, and passes the image count and duration to the selected constructor. The transition mode is therefore pluggable: changing :style from :xfade to :concat replaces the entire filtergraph structure with a concat-based variant, without touching either implementation. A user-defined transition style would require only a new function conforming to the same calling convention and an entry in the dispatch map.

Table 2 Bioscoop source size versus generated filtergraph size for slideshow-xfade

n	Bioscoop (lines)	Raw FFmpeg (chars)
3	22	553
10	22	2 176
20	22	4 446
50	22	11 256

When invoked with ten images, slideshow-xfade generates the following filtergraph string (typeset to show structure; semicolons separate chains, commas separate filters within a chain):

Listing 21 Generated filtergraph for $n = 10$ (excerpt)

```
color=color=black:size=hd1080:rate=25:duration=5,
  split=outputs=10[screen0][screen1]...[screen9];
[0:v]pad=width='iw*1.25':height='ih*1.25':x='(ow-iw)/2'
  :y='(oh-ih)/2',scale=size=hd1080:force_original_aspect_ratio
  =decrease,setsar=sar=1[padded0];
...
[9:v]pad=...,scale=...,setsar=sar=1[padded9];
[screen0][padded0]overlay=x='(W-w)/2':y='(H-h)/2'[overlay0];
...
[screen9][padded9]overlay=...[overlay9];
[overlay0][overlay1]xfade=transition=fade:duration=1
  :offset=4[out0];
[out0][overlay2]xfade=transition=fade:duration=1
  :offset=8[out1];
...
[out7][overlay9]xfade=transition=fade:duration=1
  :offset=36[out8];
[out8]format=pix_fmts=yuv420p[out]
```

Changing the number of images requires changing a single argument; changing the transition type or the per-image duration requires editing one expression each. All label generation, chain replication, and offset computation are handled by the for construct.

Table 2 quantifies the scaling behavior. Raw filtergraph length grows linearly with n, while the Bioscoop source remains constant.

This case study demonstrates that Bioscoop's abstraction mechanisms—defgraph for reuse, compose for pipeline assembly, and for for data-driven generation—scale to non-trivial filtergraph programs while keeping the source compact and parameterized.

9 Related Work

9.1 Language Workbenches

JetBrains MPS [3] and Xtext [5] provide tooling for multi-modal DSL development, typically maintaining separate grammars for different surface syntaxes. Neither targets the problem of shared intermediate representation between modalities.

9.2 Nanopass Framework

The Nanopass Framework [10] for Scheme decomposes compilation into many small, explicitly named passes, each operating on a distinct intermediate language. This approach maximizes comprehensibility of individual passes at the cost of managing many intermediate representations. Bioscoop's single-representation approach is at the opposite end of this design spectrum, appropriate because the source and target languages are at the same level of abstraction.

9.3 Typed DSLs

Typed embeddings of DSLs in functional languages [1] use the host type system to enforce DSL well-formedness statically. Bioscoop uses runtime validation via clojure.spec rather than static types, trading compile-time guarantees for flexibility and REPL interactivity.

9.4 Source-to-Source Compilation for Domain Notation Languages

The pattern of targeting an existing domain notation language as a compilation output appears in several successful systems. Sass and Less [2] target CSS, adding variables, nesting, mixins, and functions to a language that is deliberately simple and declarative. These systems demonstrate that the source language / notation language split is an architecturally stable arrangement: CSS has gained custom properties and other features, yet Sass remains in wide use because a compilation model can offer abstraction capabilities that a notation language should deliberately omit.

In the Clojure ecosystem, Hiccup [14] provides a DSL—represented as Clojure vectors—that targets HTML. The structural parallel with Bioscoop is real: both generate a well-formed textual notation from a tree-structured source. The analogy is partial, however, because HTML is markup rather than a processing language and its grammar is permissive rather than compositionally constrained. FFmpeg's filtergraph syntax is closer to CSS in this respect: it is a structured, compositional notation for a processing pipeline, with a well-defined grammar and precise semantics for filter composition.

Bioscoop fits this pattern. Its target language is a public, stable API—libavfilter's filtergraph syntax—that is intentionally devoid of abstraction mechanisms. The compilation model is stable for the same reason Sass is stable targeting CSS: the target language's simplicity is a deliberate design decision, not a temporary deficiency to be corrected upstream.

10 Conclusion

We have described Bioscoop, a domain-specific language compiler for FFmpeg filtergraph composition. Its primary contributions are:

- *Novel compilation target*: to our knowledge, the first source-to-source compiler whose target language is FFmpeg's filtergraph syntax, as distinct from API wrappers and builder libraries that construct filtergraphs programmatically without a source language
- *AST convergence*: a deliberately maintained architectural property ensuring that two front-ends produce identical ASTs before evaluation, guaranteeing semantic equivalence by construction
- *Isomorphic IR*: an intermediate representation whose grammar is in bijection with the target language's grammar, reducing the back-end to a serializer
- *First-class iteration*: a `for` construct that enables data-driven generation of filtergraph structure—the first such abstraction mechanism in FFmpeg tooling
- *Two-phase locals injection*: a mechanism using `&env` to thread Clojure lexical bindings into the DSL runtime evaluator
- *Dual resolution strategies*: a design that maximizes the affordances of two distinct execution environments—runtime reflection on the JVM and static lookup in GraalVM native image—rather than reducing both to a common denominator
- *Fail-soft error discipline*: appropriate for interactive creative coding, where partial results are more useful than hard failures

These contributions situate Bioscoop within a lineage of transpilers that target domain notation languages—Sass targeting CSS, Hiccup targeting HTML—where the value is not in replacing the target language but in providing the abstraction mechanisms it deliberately omits. Its most practically significant individual property is the `for` construct, which closes the expressiveness gap between FFmpeg's static filtergraph syntax and the data-driven, parameterized pipeline descriptions that users of complex filtergraphs actually need: graph topology and filter arguments derived from computed values, not written out by hand. More broadly, Bioscoop demonstrates that programmatic video editing—specifying transformations as a program rather than performing them by hand in a timeline—is a practical mode of creative work once the target notation is given the abstraction mechanisms it lacks.

The system is available as open-source software under the MIT license.

Acknowledgements The author thanks the Clojure community for their contributions to the ecosystem that made this work possible, and the FFmpeg project for building a framework expressive enough to warrant a compiler.

A Music Video Program

The listing below shows the three functions that carry the core structure of the music video generator described in §1.2.

Listing 22 Music video generator in Bioscoop (excerpt)

```
(defgraph padded-image
  (chain (scale {:width "'if(gt(iw,1920),1920,iw)'"
                 :height "'if(gt(ih,1080),1080,ih)'"
                 :force_original_aspect_ratio "decrease"})
         ;; sepia tone via color-channel mixing
         (colorchannelmixer {...})
         (pad {:width "1920" :height "1080"
               :x "(ow-iw)/2" :y "(oh-ih)/2"
               :color "black"})))

(defn n-life [n life-colors death-colors]
  (bioscoop
    (for [i (range n)]
      [(let [single (life {:filename "/tmp/names.txt"
                           :mold 10 :rate 25
                           :death_color (nth death-colors i)
                           :life_color (nth life-colors i)})
             blur (boxblur {:luma_radius "3"
                            :luma_power 1})]
         (chain single blur
                (scale {:width 1920 :height 1080})))
       [(str "life" i)]])))
```

padded-image is a reusable filter stage defined via defgraph: it normalizes each photograph to 1920×1080, applies a sepia tone through colour-channel mixing, and pads with black to fill the frame. n-life uses for to generate *n* parallel Game of Life simulations, each with a distinct colour pair drawn from cycling life-colors and death-colors sequences; the cell patterns are initialized from a file whose contents the program writes from a CSV of names before rendering, so each iteration builds the cellular automaton from a data-driven seed.

The top-level filtergraph function (not shown) assembles the complete pipeline with compose: five for-generated sub-pipelines—image padding, split, harmonic blending, time-gated final blending, and crossfading—together with the life generation layer, producing a single filtergraph string that FFmpeg executes.

References

[1] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. “Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages”. In: *Journal of Functional Programming* 19.5 (2009), pages 509–543. DOI: 10.1017/S0956796809007205.

[2] Hampton Catlin and Natalie Weizenbaum. *Sass: Syntactically Awesome Style Sheets*. https://sass-lang.com/. Accessed: 2024. 2023.

[3] Sergey Dmitriev. "Language Oriented Programming: The Next Programming Paradigm". In: *JetBrains onBoard* 1.1 (2004).

[4] Mark Engelberg. *Instaparse: What if context-free grammars were as easy to use as regular expressions?* https://github.com/Engelberg/instaparse. Accessed: 2024. 2023.

[5] Sven Eysholdt and Heiko Behrens. "Xtext: Implement Your Language Faster Than the Quick and Dirty Way". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. 2010, pages 307–309. DOI: 10.1145/1869542.1869625.

[6] Matthew Flatt and PLT. *Reference: Racket*. Technical report PLT-TR-2010-1. PLT Inc., 2010. URL: https://racket-lang.org/tr1/.

[7] Rich Hickey. *clojure.spec*. https://clojure.org/guides/spec. Accessed: 2024. 2016.

[8] Rich Hickey. "The Clojure Programming Language". In: *Proceedings of the 2008 Symposium on Dynamic Languages*. 2008, page 1. DOI: 10.1145/1408681.1408682.

[9] Paul Hudak. "Building Domain-Specific Embedded Languages". In: *ACM Computing Surveys* 28.4es (1996), page 196. DOI: 10.1145/242224.242477.

[10] Andrew W. Keep and R. Kent Dybvig. "A Nanopass Framework for Commercial Compiler Development". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. 2013, pages 343–350. DOI: 10.1145/2500365.2500537.

[11] Shriram Krishnamurthi. *Bicameral, Not Homoiconic*. Parenthetically Speaking. Accessed: 2024. 2023. URL: https://parentheticallyspeaking.org/articles/bicameral-not-homoiconic/.

[12] Karl Kroening. *ffmpeg-python: Python bindings for FFmpeg*. https://github.com/kkroening/ffmpeg-python. Accessed: 2024. 2023.

[13] *python-ffmpegio: Python package to read/write/transcode media files with FFmpeg*. https://github.com/python-ffmpegio/python-ffmpegio. Accessed: 2024. 2023.

[14] James Reeves. *Hiccup: A Clojure Library for Representing HTML in Clojure Data Structures*. https://github.com/weavejester/hiccup. Accessed: 2024. 2023.

[15] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages". In: *Higher-Order and Symbolic Computation* 11.4 (1998), pages 363–397. DOI: 10.1023/A:1010027404223.

[16] Damien Schaffe. *fluent-ffmpeg: A fluent API for FFmpeg*. https://github.com/fluent-ffmpeg/node-fluent-ffmpeg. Accessed: 2024. 2023.

[17] Elizabeth Scott and Adrian Johnstone. "GLL Parsing". In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010), pages 177–189. DOI: 10.1016/j.entcs.2010.08.041.

[18] Zulko. *MoviePy: Video editing with Python*. https://zulko.github.io/moviepy/. Accessed: 2024. 2023.

About the author

Daniel Szmulewicz is an independent researcher working on programming languages, multimedia tooling, and creative coding. Contact him at daniel.szmulewicz@gmail.com.