[Paper] KidLisp: A Minimal Lisp for Generative Art with Social Composition

118 Built-ins, No User-Defined Functions, and the $-Code Abstraction

Jeffrey Alan Scudder
Aesthetic Inc.
New York, NY, USA
mail@aesthetic.computer

ABSTRACT

We present KidLisp, a minimal Lisp dialect for generative art that deliberately omits user-defined functions, recursion, file I/O, networking, and error messages. The language provides 118 built-in functions across 12 categories (graphics, transformations, math, color, system, control flow, timing, text, input, audio, 3D, and composition) and replaces traditional abstraction mechanisms with a social composition system: programs reference other programs by short alphanumeric code using $-syntax, creating an implicit dependency graph across authors. The evaluator re-executes the entire AST every frame at 60 fps, using a deterministic PRNG seeded from the program's content hash to ensure reproducible generative output. Invalid input triggers "chaos mode" — artistic visual output rather than error messages — eliminating the error-wall that discourages beginners. Deployed on Aesthetic Computer (aesthetic.computer) since 2024, KidLisp has been used by 59 authors to write 16,634 programs, with observed embedding depths of up to 7 layers. We describe the language design, the evaluation model, the social composition system, and the adoption patterns, arguing that the constraints imposed by omission — no user-defined functions, no imports, no error messages — produce a creative space that is both more accessible and more socially generative than conventional Lisp dialects or creative coding environments.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**;
• **Human-centered computing** → *Interactive systems and tools*;
• **Applied computing** → *Arts and humanities.*

KEYWORDS

Lisp, generative art, creative coding, domain-specific language, social computing, live coding

1 INTRODUCTION

Creative coding environments face a persistent tension: expressiveness versus accessibility. Processing [12] and p5.js provide general-purpose JavaScript with drawing primitives, offering full expressiveness at the cost of requiring programming knowledge. Scratch [13] provides visual block programming, offering accessibility at the cost of limited creative range. Sonic Pi [1] and Hydra [9] occupy middle positions, but both assume familiarity with programming concepts like functions, variables, and control flow.

KidLisp takes a different approach: it uses the full symbolic power of Lisp syntax [10] but *removes* the features that make Lisp powerful as a general-purpose language. There are no user-defined functions, no recursion, no macros, no file I/O, no networking, and no string manipulation. What remains is a language that can draw, animate, make sound, respond to input, and compose with other programs — and nothing else.

This paper describes the language design (§2), the evaluation model (§3), the social composition system (§4), the chaos mode error handling (§5), and the adoption patterns observed over 18 months of deployment (§6).

2 LANGUAGE DESIGN

2.1 Syntax

KidLisp uses standard S-expression syntax. Every expression is an atom (number, string, or symbol) or a parenthesized list:

```
(wipe "navy")              ; clear canvas
(ink "gold")               ; set drawing color
(circle 64 64 30)          ; draw circle
(repeat 12 i               ; loop 12 times
  (box (* i 10) 50 8 8))   ; draw grid
```

The syntax is chosen for three reasons: (1) it is unambiguous — no operator precedence, no statement terminators; (2) it is uniform — every operation uses the same form; (3) it is the same syntax used since 1958 [10], making KidLisp programs structurally compatible with the entire Lisp literature [2, 7].

2.2 The 118 Built-ins

KidLisp provides 118 built-in functions organized into 12 categories:

The number 118 is not arbitrary but reflects a design principle: every function that a generative artist might need in a single-canvas, single-speaker, immediate-mode environment is provided. No function requires understanding a concept beyond "call this with these arguments."

2.3 Design by Omission

The most important design decisions are what KidLisp *lacks*:

- **No user-defined functions.** The $-code system (§4) replaces them with a social mechanism: reference another program by code.
- **No recursion.** Prevents infinite loops and stack overflows. The only iteration mechanism is `repeat`, which requires an explicit count.

Category	Count
Graphics (wipe, ink, line, box, circle, ...)	9
Transformations (scroll, zoom, spin, blur, ...)	11
Math (+, -, *, /, sin, cos, random, ...)	14
Color (named colors, fade, rainbow, zebra)	19+
System/Display (width, height, frame, time, ...)	9
Control Flow (def, if, repeat, once, let, ...)	7
Text/Output (write, type, paste, print)	4
Input (pen, hand, gamepad)	3
Audio (mic, melody, overtone, sound, ...)	6
3D Graphics (cube, form, trans, move, scale, ...)	8
Composition ($code, layer, bake, embed)	4
Data (list, get, set, let)	4
Total	**118**

Table 1: Built-in functions by category.

- **No file I/O, no networking.** Programs cannot access resources beyond their canvas and speaker. This makes every program safe to execute without sandboxing.
- **No error messages.** Invalid input produces visual output via chaos mode (§5), not error text. Beginners never see `SyntaxError`.
- **No imports or require.** The only way to use another program's functionality is `$code`, which fetches and executes it as a layer.
- **No string manipulation.** Strings are used only for color names, text output, and melody notation. They cannot be split, concatenated, or transformed.

Each omission eliminates a class of confusion. No recursion means no stack overflows. No networking means no CORS errors, no fetch failures, no API keys. No error messages means no intimidating red text. The language is safe by construction [6]: not because it validates input, but because it *cannot express* dangerous operations.

2.4 Timing Without Callbacks

KidLisp's timing system uses inline scheduling expressions rather than callback functions:

```
(1s (wipe "red"))       ; after 1 second
(2.5s (ink "gold"))     ; after 2.5 seconds
(0.5s... (spin 45))     ; every 0.5 seconds
(1s! (write "!" 32 32)); fire once at 1s
(30f (ink "blue"))      ; after 30 frames
```

The timing expression wraps a code block and controls when it executes. There are no `setTimeout`, `setInterval`, or `requestAnimationFrame` calls. Time is a property of the expression, not a function applied to a callback. This eliminates the concept of "callback hell" and makes temporal composition as simple as spatial composition.

3 EVALUATION MODEL

3.1 Full AST Re-evaluation

KidLisp re-evaluates its **entire abstract syntax tree every frame** at 60 fps. There is no setup/draw split (as in Processing), no retained scene graph, no persistent state beyond explicitly defined variables. The `frame` counter and `time` value increment automatically.

This model has two consequences:

(1) **Animation requires no explicit loop.** A circle whose position depends on `frame` moves automatically:

```
(wipe "black")
(ink "white")
(circle (+ 64 (* 30 (sin (* frame 0.03))))
        (+ 64 (* 30 (cos (* frame 0.03))))
        8)
```

(2) **Transformations accumulate.** Functions like `zoom`, `scroll`, and `spin` operate on the framebuffer, not the coordinate system. Each frame's transformation is applied to the *result* of the previous frame:

```
(ink "white")
(circle 64 64 2)  ; tiny dot
(zoom 1.01)       ; grows each frame
(spin 1)          ; rotates each frame
; Result: expanding spiral
```

3.2 Deterministic PRNG

The `random` function uses a seeded pseudo-random number generator initialized from the program's SHA-256 content hash. The same program produces identical output on every device, every time. This makes KidLisp programs *reproducible generative art*: the short code is not just an identifier but a complete specification of the artwork's behavior.

4 SOCIAL COMPOSITION

4.1 The $-Code System

KidLisp's primary abstraction mechanism is not functions, modules, or classes. It is the $-code: a reference to another program by its short alphanumeric identifier.

```
($cow)            ; execute program "cow"
($nece 10 20)     ; pass parameters
(wipe "black")
($cow)            ; layer 1
($faim)           ; layer 2
```

When the evaluator encounters `($cow)`, it:

(1) Checks the RAM cache for the program with code "cow"
(2) If not cached, checks IndexedDB (persistent browser storage)
(3) If not stored, fetches from MongoDB via REST API
(4) If not in MongoDB, attempts to retrieve from TEIA (Tezos decentralized archive)
(5) Parses and evaluates the retrieved source as a layer

This multi-level cache (RAM → IndexedDB → network → decentralized archive) ensures that programs remain accessible even if the primary database is unavailable. Content-addressing via SHA-256 ensures integrity.

4.2 Composition as Social Graph

When author A writes a program that embeds `($xyz)` — a program written by author B — a social edge is created without either author explicitly interacting. The $-code graph is a bipartite network of programs and authors, where edges represent creative dependency.

Observed properties of this graph (as of March 2026):

- Maximum embedding depth: 7 layers
- Hub programs (referenced by many others) tend to be simple visual primitives — solid color fills, basic shapes, gradients
- Cross-author composition is common: authors embed each other's programs without explicit coordination
- The most-referenced programs serve as *de facto standard library entries*, discovered through browsing rather than documentation

This is a fundamentally different abstraction model from user-defined functions. In a conventional language, the programmer creates an abstraction (a function) and gives it a name. In KidLisp, the programmer creates a *complete program* and the system gives it a code. Other programmers discover and embed it. The abstraction is social, not syntactic.

5 CHAOS MODE

When the KidLisp evaluator receives input it cannot parse — gibberish, random characters, unbalanced parentheses — it does not display an error message. Instead, it classifies the input using three heuristics:

(1) **Recognition ratio**: What fraction of tokens are known built-in names? Below 30% suggests non-code input.
(2) **Special character ratio**: Above 50% suggests keyboard mashing or copy-paste noise.
(3) **Parenthesis balance**: A ratio exceeding 3:1 (open to close or vice versa) suggests structural invalidity.

If the input scores high on these heuristics, KidLisp enters "chaos mode": it renders the input as visual material — interpreting character codes as colors, positions, or shapes — producing abstract visual output. The user sees art, not `Unexpected token at line 1`.

This is a pedagogical decision: the single greatest barrier to programming for beginners is the error message [11]. Error messages presuppose that the user intended to write valid code. In a creative environment, the user may be experimenting, playing, or simply curious about what happens when they type their name. Chaos mode rewards that curiosity with visual feedback rather than punishment.

6 ADOPTION

KidLisp has been deployed on Aesthetic Computer since mid-2024. As of March 2026:

Metric	Value
Total programs	16,634
Unique authors	59
Median programs per author	12
Top quartile (programs/author)	180+
Maximum embedding depth	7 layers
Total program hits (views)	>100,000
Programs with Tezos keeps (NFTs)	34
Unique NFT owners	9

Table 2: KidLisp adoption metrics.

6.1 Contribution Distribution

The distribution of programs per author follows a power law [3]: a small core of prolific authors (top 10%) produces the majority of programs, while a long tail of casual creators produces 1–3 programs each. This distribution mirrors patterns observed in open-source software [5] and creative communities like Scratch [13].

Notably, the most prolific authors' early programs are almost exclusively *modifications of existing programs discovered through browsing*. The learning path is: view → fork (save a variation) → compose (embed via $-code) → originate (write from scratch). This "exploration over engineering" pattern [4] is enabled by the $-code system, which makes every program a potential starting point.

6.2 Function Usage Patterns

The `/api/store-kidlisp?stats=functions` endpoint returns weighted usage statistics for every built-in function. Expected patterns based on language design:

- Graphics primitives (`wipe`, `ink`, `circle`, `box`) are the most used — they produce immediate visible output
- `repeat` is the most used control flow construct — it is the only iteration mechanism
- `random` is disproportionately popular — generative art requires randomness
- Transformation functions (`spin`, `zoom`, `scroll`) appear in animation-heavy programs and create the characteristic "feedback" aesthetic
- Audio functions (`mic`, `melody`) are least used — audio requires hardware and is not visible in static previews

7 RELATED WORK

KidLisp occupies a specific position in the landscape of creative coding languages:

Processing/p5.js [12]: General-purpose with drawing primitives. KidLisp is smaller (118 built-ins vs. hundreds of functions), safer (no file/network access), and socially compositional ($-codes).

Scratch [13]: Block-based, visual programming. KidLisp is text-based but shares Scratch's commitment to eliminating error messages and its community sharing model.

Sonic Pi [1]: Live-coded music in Ruby-like syntax. KidLisp is visual-first with audio as a secondary capability, and uses Lisp rather than imperative syntax.

Hydra [9]: Live-coded visuals with functional chaining. KidLisp shares the immediate-mode, every-frame-re-evaluation model but uses S-expressions rather than method chaining.

Fluxus [8]: Scheme-based live coding for 3D. KidLisp is 2D-primary (3D is secondary), deliberately simpler, and eliminates user-defined functions.

Racket/HtDP [6]: Pedagogical Lisp with progressive language levels. KidLisp takes the opposite approach: one level with deliberate omissions, rather than multiple levels with progressive additions.

The closest analogue may be **Logo**'s turtle graphics: a constrained environment where simple commands produce visible results and composition builds complexity [11]. KidLisp is Logo for generative art, with social composition replacing procedural abstraction.

8 LIMITATIONS AND FUTURE WORK

KidLisp's constraints, while productive, impose real limitations:

- **No user-defined functions** means complex programs must use deeply nested `repeat` loops or reference external programs via $-codes. Programs exceeding ~100 lines become difficult to manage.
- **No string manipulation** prevents text-generative art (concrete poetry, text permutation, language games).
- **Deterministic PRNG** means that programs cannot produce different output on different viewings unless they use time-dependent expressions.
- **The 118 built-ins are fixed.** Community-contributed functions are not possible; the language evolves only through author updates to the evaluator.
- **59 authors** is a small community. Whether the $-code composition model scales to hundreds or thousands of authors is unknown.

Future work includes: (1) a formal grammar specification for submission to language repositories; (2) a standalone evaluator (currently embedded in the Aesthetic Computer web runtime); (3) empirical analysis of the $-code dependency graph structure; (4) comparative user studies with Processing/p5.js for generative art tasks.

9 CONCLUSION

KidLisp demonstrates that a Lisp dialect with 118 built-ins and no user-defined functions can sustain a community of 59 authors producing 16,634 programs. The $-code composition system — referencing other programs by short code rather than defining functions — replaces syntactic abstraction with social discovery. Chaos mode — rendering invalid input as visual output rather than error messages — eliminates the error-wall that discourages beginners. Deterministic evaluation with seeded PRNG makes every program a reproducible artwork.

The language's power comes not from what it can do but from what it cannot. It cannot crash its host. It cannot access the filesystem. It cannot make network requests. It cannot recurse infinitely. It cannot display an error message. Within these constraints, 16,634 programs exist, some embedding each other up to 7 layers deep, creating a social composition graph that no individual author planned or controls.

KidLisp is not a general-purpose language. It is a Lisp for one purpose: making visible things that move, respond, and compose. That is enough.

REFERENCES

[1] Samuel Aaron. 2016. Sonic Pi – Performance in Education, Technology and Art. In *Proceedings of the International Conference on Live Coding*.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press.

[3] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512.

[4] Kate Compton and Michael Mateas. 2015. Casual Creators. In *Proceedings of the International Conference on Computational Creativity*.

[5] Nadia Eghbal. 2020. *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs* (2nd ed.). MIT Press.

[7] Daniel P. Friedman and Matthias Felleisen. 1996. *The Little Schemer* (4th ed.). MIT Press.

[8] Dave Griffiths. 2007. Fluxus: A Rapid Prototyping Tool for Live Coding Audio Visual Performances. In *Proceedings of the International Computer Music Conference*.

[9] Olivia Jack. 2019. Hydra: Live Coding Networked Visuals. https://hydra.ojack.xyz

[10] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.

[11] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.

[12] Casey Reas and Ben Fry. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press.

[13] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (2009), 60–67.