Show Us Your Thoughts – But Which Ones?

Demonstrating Abstraction Levels in Domain Specific Languages Through Implementations of Steve Reich's *Clapping Music*

Iván Paz*
TOPLAP Barcelona
ivan@toplap.cat

Niklas Reppel*
TOPLAP Barcelona
nik@parkellipsen.de

Tim Cowlishaw*
Universitat Oberta de Catalunya
tim@timcowlishaw.co.uk

Glen Fraser*
TOPLAP Barcelona
glen@glenfraser.com

Timo Hoogland*
Netherlands Coding Live
hello@tmhglnd.com

Bernat Romagosa
SAP
b.romagosa@sap.com

Luka Prinčič
TOPLAP Slovenia
luka@princic.studio

Lluís Nacenta
TOPLAP Barcelona
lluis@19preguntes.net

Julia Múgica
TOPLAP Barcelona
juliajmg@gmail.com

Roger Pibernat
TOPLAP Barcelona
alo@rogerpibernat.com

Holger Ballweg
Creative Code Club Newcastle
holger.ballweg@gmail.com

ABSTRACT

Since the inception of live coding, but especially in the last 10 years, a central endeavor of the live coding community has been the development of *domain-specific languages* (DSLs), at various degrees of abstraction. In this article, we first discuss the role of abstraction in live coding. Then, we demonstrate and analyze the abstraction levels of different DSLs, based on implementations of Steve Reich's *Clapping Music* (a minimalist algorithmic piece), and discuss the resulting affordances. The hypothesis we are supporting is that abstraction is instrumental in highlighting certain aspects of live coding, by determining which details a performer has to focus on during a live performance, which aspects of the code (and the thought process) are exposed, and which ones are hidden.

1 Introduction

The number of *domain-specific languages* (DSLs) has skyrocketed since the first edition of ICLC in 2015. Back then, there were a few, such as TidalCycles (still just called Tidal), Fluxus, or IxiLang. In 2024, at the time of writing this article, the list of languages and environments on *Awesome Live Coding* (Toplap 20xx) has over 110 entries. A common trend is to create high-level, high-abstraction DSLs, each of which focuses on a specific set of affordances.

Abstraction has been and still is a mildly polarizing topic in the live coding community. While some live coders prefer the detailed level of control (and potentially the riskier environment) that lower abstraction levels offer (Roberts and Wakefield 2018), or go as far as to live code in lower-level, universal languages such as C++ (Villaseñor-Ramírez and Paz 2020), others embrace the limitations of abstract, high-level languages, be it for the reduced cognitive and physical load allowing the performer to focus on the compositional structure (McLean and Wiggins 2011) (Roberts and Wakefield 2018), or the better readability for the audience (where *show us your screen* becomes *show us your thoughts*), giving audiences an opportunity to learn from the code (Burland and McLean 2016). In fact, audience members have expressed disappointment when much code has been pre-written before a performance (Burland and McLean 2016). What is clear is that abstraction impacts the perceived *liveness* of a live coding performance, for both the audience and the performer (see chapter 2).

In September 2024, Bernat Romagosa posted a link to a hardware implementation of Steve Reich's *Clapping Music*[1] on the `Algorithmic-barcelona` Telegram channel, which inspired a wave of implementations of the piece in a variety of

*Principal authors who contributed equally.
[1] https://hannahilea.com/blog/clapping-music-for-flip-disc-displays/

live coding languages[2] by some live coders from that group (shortly after, the repository was also posted on Mastodon, which attracted a few more coders). We discovered that this juxtaposition of implementations of the same musical idea, ranging from 34 lines of code in vanilla SuperCollider to just three lines in Živa or one line in TidalCycles, illustrated well the different aspects of the languages, their focus, and their levels of abstraction.

While a DSL can define the grammar and the abstraction level, and determine what can be manipulated during a performance, we argue that we can learn a lot about the nature and practice of live coding by looking at the abstractions provided by DSLs.

Therefore, in this article, we are analyzing and comparing the various implementations, with special focus on the levels of abstraction, to support our hypothesis that abstraction sets the focus on some specific aspects of live coding, which we will discuss, alongside the differences and commonalities between practices of live coding with different languages and tools.

We'll close with conclusions about how these varied uses of abstractions reflect broader concerns and ideas in the practice of live coding.

2 Abstraction in Live Coding DSLs

The form of close study of source code applied in this article, and in particular of source code used for creative or artistic expression, has been carried out extensively in the fields of *Software Studies*, and *Computational Aesthetics*. The issue of abstraction is fundamental to the idea of an *aesthetics* of computation in both these fields: Beatrice M. Fazi and Matthew Fuller assert that:

> *"To compute involves abstractive operations of quantification and of simulation, as well as the organization of abstract objects and procedures into expressions that can (but also may not) be thought of, perceived, and carried out. Attending to computational aesthetics, then, puts in question the forces of all degrees and kinds that participate in these abstractions, and enquires what level of autonomy one should assign to such forces and abstractions."* (Fazi and Fuller 2016)

This isn't of purely abstract concern, as it also addresses the *affordances* of computational media (in this case, live coding languages), and their expressive power and potential: *"Computation not only abstracts from the world in order to model and represent it; through such abstractions, it also partakes in it."* (Fazi and Fuller 2016). This is of practical concern to the study of live coding, as it speaks to the way in which creative expression and the properties of different languages are linked: the abstractions we employ *"shap[e] and reshap[e] users, their dispositions and habits."* (Fuller and Goffey 2014). They don't just shape the languages we use, but the types of performances we produce, and our own broader live coding practices.

Abstractions also have an interpretative role, as they are necessarily partial, and selective. The process of abstraction involves *" select[ing] properties and behaviors that we think are important to be represented in a program, and ignor[ing] others."* (Cox and Soon 2021) (p147), and therefore necessarily present a particular "reading" of an underlying computational or expressive process that *"relies on the suppression of a lot of other aspects of the world"* (Crutzen and Kotkamp 2008). This suggests that different abstractions of the same underlying process can differ qualitatively, placing us firmly *"on familiar territory for cultural analysis"* (Bek 2008).

In addition, abstractions can happen, simultaneously, at multiple scales - a phenomenon seen in typical live coding performances, which might use the audio generation and DSP-focused abstractions of SuperCollider, controlled via the pattern-expression abstractions of TidalCycles, manipulated with user-defined helper methods and combinators written in Haskell, all controlled through a text editor in which the performer has defined macros for responsive, expressive live performance: *"abstraction exists in many different layers and at many different scales of computing."* (Cox and Soon 2021) (p145). In the case of live coding, many of these different layers of abstraction are developed by live coders as an integral part of their performative practice (Baalman 2015).

The way in which different abstractions engender different affordances and interpretations, and are successively layered within the ecology of live coding environments, has relevance to long-ongoing debates on the *liveness* of live coding. Giovanni Mori's ethnographic analysis of live coding practice (Mori 2015) situates the *liveness* of live coding as engendered by both the *display of creative agency* and *virtuosity* on the part of the live code performer, and the *visibility* of their work, projected during the performance.

Reducing cognitive and physical load through abstraction (McLean and Wiggins 2011) (Roberts and Wakefield 2018) can increase (perceived) creative agency in the sense that it allows the performer to implement noticeable changes more

[2] https://github.com/toplap-cat/clapping-with-code/

Figure 1: *Steve Reich Clapping Music Notated Rhythm*

quickly and dynamically, while the increased readability (or *"closeness of mapping"* (Blackwell and Green 2003)) can make it easier to associate a code gesture with its effect on music or visuals, leading to a reduced overall *"viscosity"* (Blackwell and Green 2003), that is, the resistance to change, in the feedback system between audience, code, and performer.

Abstraction, as *"a tempering of the clutter of the visible"*, according to Cecile Crutzen and Erna Kotkamp (Crutzen and Kotkamp 2008) (quoting Susan Leigh Star (Star 1991)) has an ambiguous relationship with visibility: An example can be seen below comparing the SuperCollider implementation of Clapping Music to the Tidal version: SuperCollider renders the generation of the clapping sound visible while Tidal does not. Either might claim to have higher visibility, either by more comprehensively showing your work in the case of SuperCollider, or by demonstrating a musical idea or motif more clearly in the case of Tidal. Renick Bell, on the other hand, emphasizes the potential of *opacity* that might be introduced by the use of abstractions (Bell 2013), looking at the tradeoff between agency and visibilty from the other end.

The comparative study of how abstraction is deployed in different live coding languages therefore seems to offer promise for better understanding some of these questions, by holding fixed the details of the musical work being expressed, and producing multiple implementations in different languages, the particular abstractions deployed in each language can reveal themselves in a way that is germane to comparison, making this analysis possible. The remainder of this paper describes one such analysis on different implementations of Steve Reich's *Clapping Music*[3], carried out by the Toplap Barcelona community and friends.

3 Clapping Music

The piece *Clapping Music* is as minimal to the extreme, relying on one rhythmic pattern (a variation on the African bell pattern (Wikipedia 2024)), two players, and a single operation for musical development. One player shifts the pattern forward (to the left notation-wise, moving the first note to the end of the measure) by one eighth-note every 8 or 12 bars. The entire musical development of the piece is based on that operation (Haack 1998). This shifted layer technique is commonly found in pattern-based live coding.

The minimalistic and algorithmic nature of the piece serves as an effective litmus test for abstraction. The shifting operation can be implemented in many ways across different languages, highlighting which aspects the live coder must prioritize during the real-time performance – whether it is sound design or the notation of musical/visual structure – depending on the grammar they are using.

4 Case Studies

The following examples were initially shared on the `Algorithmic-barcelona` Telegram channel ("TOPLAP Barcelona Community Report 2023" 2023). Of course, they represent individual takes on the subject. A multitude of longer or shorter alternatives are possible in each language; nonetheless, these takes have been created by experienced live coding practitioners and provide examples of the amount of work needed to perform the task in the various languages. While the examples here were not coded in live performances, most likely similar scenarios have happened in each performers' live practice.

The list is by no means exhaustive, as people followed their curiosity rather than systematic method when adding an implementation in their language of choice, typically motivated by general interest in puzzle-solving. Several were added by the authors of the respective DSL. Timo Hoogland summarized the feeling shared by many of the language authors:

> *I was immediately interested in coding the Clapping Music when the post was made by Bernat Romagosa. Mainly because I wanted to see if Mercury, the language that I have created, was able to code this piece in any way. Besides that it was also a nice exercise to think about the code and find a solution within my own language.*

[3] https://hannahilea.com/blog/clapping-music-for-flip-disc-displays/

Others stated they're interested in implementing "anything that looks or sounds algorithmic in Snap or MicroBlocks", and in contributing as an underrepresented group: women and visualists (in the case of P5LIVE).

Some of the questions we subsequently asked the contributors were: How does your code work? What does the grammar you're using focus on? Is it musical structure, or sound design? Is the readability or visual clarity of the code an important factor, or is there a tradeoff between expressive power or brevity? Which thoughts are abstracted away, that is, not visible on the screen, and which parts ARE shown and coded in real time during a live performance? Abstraction is a practice of interpretation (Cox and Soon 2021) (p147), and therefore what follows are commentaries on each implementation by its authors, explaining how their interpretation of the piece is reflected in their implementation.

4.1 Vanilla SuperCollider

This vanilla SuperCollider version doesn't use samples and subsequently starts by designing a sound. It manually implements the shift operation by extending the sequence. While the sound design code is relatively straightforward, the structural code is more verbose and is concerned with bindings to the instruments, while the rhythmical structure itself is expressed here as numbers.

The synth called `\pols` could of course be replaced by any other percussive synth, or by a clapping sound sample. In this particular example, this should be taken only as an aesthetic preference of the live coder, and should not be considered relevant in the implementation of Steve Reich's piece. The relevant part of the piece is the rhythmic development between the two `Pdefs` that follow, called `\clap1` and `\clap2`.

However, using a synthesized sound instead of a recorded sample offers an interesting opportunity for the live coding performance. Since the rhythmic pattern of the piece is fixed, and the live coder only needs to write it at the beginning and let it develop over time, the piece might be of little interest as a live coding performance. Using a synthesized sound offers the live coder a wide range of possibilities to modify the sound as the gradual phasing process unfolds, so to speak, by itself. It is of course possible to modify the acoustic details of a recorded sound, but once the listener identifies the sound of a clap, successive variations will be more easily perceived as mere details, whereas the less realistic nature of synthetic sound allows for a more varied and sophisticated set of sound modifications.

In terms of abstraction, then, since this implementation of the rhythmic aspect of Clapping Music in SuperCollider is rather abstract, describing the durations with numbers, the interest of the live coding performance requires a lower degree of abstraction in the definition of the sound, so more parameters are available to be changed by the live coder and read by the audience.

This example leads us to postulate that the interest, variety and expressiveness of the live coding performance requires a fine balance in the degree of abstraction, without which the range of possibilities for performance might be too big or too limited.

```
// implementation of Steve Reich's "Clapping Music" in vanilla SuperCollider, by Lluís Nacenta
(
SynthDef(\pols, {
 arg freq, pan=0, vol=0.5, atk=0.01, rel=0.4, width=0.5, des=0.03, curve= -4;
 var sig, env;
 env = EnvGen.kr(Env.perc(atk, rel, curve: curve), doneAction: 2);
 sig = Pulse.ar({ExpRand(freq*(1-des), freq*(1+des))}!19, width);
 sig = sig * env;
 sig = Splay.ar(sig, level: vol, center: pan);
 Out.ar(0, sig);
}).add;
)

(
var durs = [0.2, 0.2, 0.4, 0.2, 0.4, 0.4, 0.2, 0.4];
var amps = [0.2, 0.1, 0.1, 0.2, 0.1, 0.2, 0.1, 0.2];
TempoClock.default.tempo = 1.4;
Pdef(\clap1, Pbind(
  \instrument, \pols,
  \midinote, 51,
  \dur, Pseq(durs, inf),
  \vol, Pseq(amps, 51),
  \pan, Pwhite(-1, 0.0)
 )
).play(quant: 2.4);

Pdef(\clap2, Pbind(
  \instrument, \pols,
  \midinote, 54,
  \dur, Pseq(durs ++ durs ++ durs ++ [0.2, 0.2, 0.4, 0.2, 0.4, 0.4, 0.2, 0.2], inf),
  \vol, Pseq(amps, 52),
  \pan, Pwhite(0, 1.0)
 )
).play(quant: 2.4);
)
```

4.2 Bacalao (SuperCollider)

Bacalao (Fraser 2020) is a toolbox of helpers that add functionality to SuperCollider via new classes and extensions to its native classes. One of its objectives was to reduce the number of keystrokes required for live coding, compared to using SuperCollider's native Pattern and Event classes or JITLib's Node/Pattern/Task Proxies. Bacalao also simplifies access to third-party add-ons, such as VSTPlugin (Ressi 2018). It offers a shortened notation (e.g., `pb` instead of `Pbind`, or `deg` instead of `degree`), which appeals to those seeking a streamlined syntax. While SuperCollider can already be challenging to grasp, Bacalao's condensed syntax may feel cryptic to some users. In addition to its many shortcuts and helpers, Bacalao incorporates a mini-notation, time-based pattern chaining, and functional pattern operations, inspired by TidalCycles but implemented in `sclang`. These abstractions provide more concise ways to express and operate on musical elements such as notes or other properties, together with their rhythms.

Two *Clapping Music* examples are provided to illustrate Bacalao's capabilities. The first demonstrates the use of functional operations (rotate, repeat, collect) to algorithmically describe the complete pattern at once. The second example takes a different approach, using a running task to mimic a live "player", waiting to dynamically adjust the offset of the second pattern every eight bars.

```
// Two implementations of Steve Reich's "Clapping Music" in Bacalao, by Glen Fraser
b = Bacalao().boot.tempo_(1.8);

// Version 1: Pre-baked
(
var clapPattern = [deg: "0 0 0 ~ 0 0 ~ 0 ~ 0 0 ~", amp: 0.7, ins: \clap_electro].pb;
var rotatedPattern = 12.collect{ |i| clapPattern.rotate(i).pn(8) }.pseq;
b.p(\claps, [
  clapPattern.loop <> (pan: -0.7),
  rotatedPattern.loop <> (pan: 0.7)
].par);
)

// Version 2: An "agent" (Task) makes the rotation changes (every 8 bars)
(
var clapPattern = [deg: "0 0 0 ~ 0 0 ~ 0 ~ 0 0 ~", amp: 0.7, ins: \clap_electro].pb;
var rotAmt = 0;
b.p(\claps, [
  clapPattern <> (pan: -0.7),
  Plazy({ clapPattern.rotate(rotAmt) }) <> (pan: 0.7)
].par);

b.tp(\player, { loop {
  12.do{ |i|
    rotAmt = i;
    8.wait;
  };
}});
)
```

4.3 Živa (SuperCollider)

Živa (Pibernat 2023) is a SuperCollider library that makes the most of the sclang's syntax flexibility. It also provides abstractions to quickly set up the environment, load samples and to poll the system for information. It uses SuperCollider's pattern sequencing system with an alternative syntax, allowing for one of the most concise implementations of the piece. Živa allows to explicitly set sound parameters (e.g. the panning) while allowing to work on the structure of the music with the pattern syntax.

In the implementation of *Clapping Music* each part is played by a different agent. The first agent plays the pattern as is. The second agent sets the sample index to a different value and adds it to the original pattern, which is randomly repeated either 8 or 12 times, after which a rest is played for 1/8, offsetting the next iteration.

```
// implementation of Steve Reich's "Clapping Music" in Živa, by Roger Pibernat
Ziva.bpm = rrand(160,184);

c = [0,0,0,r,0,0,r,0,r,0,0,r];
~steve s: \clap n: c.pseq dur: (1/2) pan: (-0.7) >>>.1 0.2;
~reich s: \clap n: [2 + [c!!12, c!!8].prand(1), r].pseq dur: (1/2) pan: (0.7) >>>.2 0.2;
```

4.4 TidalCycles

TidalCycles (McLean and Contributors 2023) allows for a variety of implementations of varying length. Given how TidalCycles works with samples, the actual sound creation is abstracted away completely.

TidalCycles can be seen as a multiple languages nested within each other, each with different affordances and expressive properties. TidalCycles is a domain-specific language, embedded within the general purpose functional programming language *Haskell*. It also contains it a smaller language-within-a-language for expressing musical patterns called *mini-notation*.

All three can be seen in a first example:

```
-- Version 1 - Initial attempt by Tim Cowlishaw
do
  let pattern = "808:3 808:3 808:3 ~ 808:3 808:3 ~ 808:3 ~ 808:3 808:3 ~"
  let struct = "<0%12!8 1%12!8 2%12!8 3%12!8 4%12!8 5%12!8 6%12!8 7%12!8 8%12!8 9%12!8 10%12!8 11%12!8>"
  d1 $ pattern
  d2 $ (struct <~ pattern)
```

The two strings, `pattern` and `struct` are written in TidalCycles' *mini-notation*. The `pattern` binding describes the one-bar-long clapped rhythm, and the `struct` binding describes the *shift*-operation and the overall structure of the piece. Then, the pattern is played unaltered on channel 1, while the "structure" is used to pattern the application of TidalCycles's shift (<~) operator to the same pattern on channel 2.

In this example, the majority of the score of the piece is expressed in *mini-notation*, which describes patterns for the application of the shift operator in the TidalCycles DSL. The result is certainly explicit (the pattern, and the repetition and shifting structure, are written out longhand), but is verbose, and perhaps occludes the underlying logic of the piece: the instruction to *shift the rhythm by one beat every 8 bars* could be represented more explicitly.

We can produce a shorter, and more explicit version, by expressing the structure in the TidalCycles DSL, rather than as mini-notation, using the `iter` combinator that TidalCycles provides to express the *shifting pattern* (it *shifts* the pattern in the way we require, every cycle), and the `jux` combinator to express the layering of the two parts - it layers the clapping pattern, unchanged, with the result of applying the `iter` combinator, and pans each left and right.

```
-- Version 2 - shorter, clearer (but slightly accelerated) version by Niklas Reppel
-- the "phase" of the two pairs of clapping hands changes every cycle instead of every 8 or 12
d1 $ jux (iter 12) $ s "808:3 808:3 808:3 ~ 808:3 808:3 ~ 808:3 ~ 808:3 808:3 ~"
```

The result expresses the underlying logic of the piece more succinctly, but doesn't quite express the piece entirely: it shifts after every cycle, instead of every 8th or 12th. We might therefore try and look f or a way of implementing an equivalent to the `iter` combinator which shifts every `n` cycles, and one approach could be the following:

```
-- Version 2a, by Niklas Reppel
d1 $ jux (every 8 (iter 12)) $ s "808:3 808:3 808:3 ~ 808:3 808:3 ~ 808:3 ~ 808:3 808:3 ~"
```

However, TidalCycles is a grammar for patterns and modifications as a function of (cyclical) time, and inherently stateless. Therefore, this doesn't achieve the intended result, either, as the `jux`-taposed pattern isn't brought into a new state, but shifted every 8th cycle for one cycle. We might therefore attempt to define a new `iterEvery` combinator, using constructs from Haskell itself along with existing TidalCycles combinators:

```
-- Version 3: Deriving an `iterEvery` function, by Tim Cowlishaw
do
  let flap fs x = map ($ x) fs
  let repeatEvery n xs = xs >>= replicate n
  let iterEvery n m p = cat $ flap (map (rotL . (% m)) (repeatEvery n [0..(m-1)])) p
  d1 $ jux (iterEvery 8 12) "808:3!3 ~ 808:3!2 ~ 808:3 ~ 808:3!2 ~"
```

However, this seems torturous, the TidalCycles DSL does not easily afford this type of operation (shifting every `n` cycles), and the length of the boilerplate definition of `iterEvery`, compared to the brevity of the final statement of the *Clapping Music* piece, makes this friction clear. This does not mean, however, that TidalCycles is unsuited to expressing this particular musical idea, simply that it can perhaps be expressed more clearly in another manner.

In the intent above, we concentrate on trying to delay the application of the shift operation by 8 cycles, in order to produce a repetition of 8 cycles of each *position* of the shift operation. However, we could just as easily think of the original 12-cycle pattern (in which the *shift* position changes every cycle) as a blueprint for a pattern with its repetition structure abstracted away, and attempt to *stretch* it after the fact. This way of thinking about the pattern turns out to produce a much more succinct implementation in TidalCycles:

```
-- Version 4: Elongating the pattern with repeatCycles, by Tim Cowlishaw
d1 $ repeatCycles 8 $ jux (iter 12) $ s "808:3!3 ~ 808:3!2 ~ 808:3 ~ 803:3!2 ~"
```

Here, we take the second implementation, then modify it, simply, and declaratively: in order to repeat each cycle 8 times - the (iter 12) operation itself remains unchanged, generating this underlying structure for later repetition using the `repeatCycles` combinator. In this way we can see that certain languages don't just afford the expression of certain musical ideas, but certain *analyses*, or ways of thinking about the same musical idea. In TidalCycles, *Clapping Music* expressed as a phase shift which is applied every 8 bars turns out to be difficult to express, while expressing a general pattern of phase-shifting, then stretching it, by repetition, to the desired length, turns out to be easy, and this difference in expressive affordances applies as much to the component parts of a single language like TidalCycles (which can include mini-notation, code written in the TidalCycles DSL, and code written in Haskell itself).

This also makes clear that the *level of abstraction* of a musical idea is not a simple, one-dimensional linear scale. Musical ideas can be abstracted in multiple different ways, each of which highlight different aspects of that musical idea.

4.5 Strudel

Strudel (Roos and McLean 2023) is a web port of TidalCycles with some slight changes and syntax closer to JavaScript, since that is the underlying language.

At first, the basic rhythm block was *clearly spelled out* using the default clap sample:

```
s("cp cp cp ~ cp cp ~ cp ~ cp cp ~")
```

Of course, the trick is in shifting the second clapping "performer" which shifts this basic block every 8 bars. At first the first mention of polymeter in strudel documentation was employed. Strudel - like TidalCycles - is based on the idea that all cycles always stay aligned and of the same length. With this shifting pattern in the *Clapping Music* piece we are looking for a possibility to create a cycle that is one eighth shorter. One of few options in Strudel is to consider each "note" (event) as a full bar (cycle) and raise the cycle-per-minute value (tempo). A fully working example uses symbols for replication (`!`), elongation (`@`), and alternation "each cycles" (`<>`) within the mini-notation to achieve proper repetitions and lengths:

```
setcpm(220)

$: s(`<  [cp cp cp ~ cp cp ~ cp ~ cp cp ~]@6 >`).pan(0)

$: s(`< [[cp cp cp ~ cp cp ~ cp ~ cp cp ~]!7 ]@42
         [cp cp cp ~ cp cp ~ cp ~ cp cp  ]@5.5 >`).pan(.8)
```

However this somehow seemed redundant and too *manual*, and with a bit of inspiration from the TidalCycles example also shown in this paper (Niklas Reppel and Tim Cowlishaw), a possibility with two methods came up: `.iter()`, and `repeatCycles()`:

```
let p = "cp cp cp ~ cp cp ~ cp ~ cp cp ~"
s(p).pan(1).stack(s(p).iter(12).repeatCycles(8).pan(0))
```

This last version was then made even more elegant by Felix Roos, the author of Strudel. Stacking and panning the two parallel streams (above done with `stack()` and twice `pan()`) is actually achieved with just the `.jux()` method.

Since the pattern doesn't need to be repeated in the code twice (in `s(p)`), we can let go of the variable assignment (`let p`) and also further reduce the whole pattern using the repetition (`!`) symbol. The final code becomes an elegant one-liner:

```
s("cp!3 ~ cp!2 ~ cp ~ cp!2 ~").jux(x=>x.iter(12).repeatCycles(8))
```

4.6 Mégra

The Mégra language (Reppel 2024) allows for one of the more concise versions of the piece, relying on prefabricated sounds (samples, in this case, but Mégra barely distinguishes between samples and synthesized sounds). All work regarding sound design and/or sample loading is abstracted away in this case.

```
;; implementation of Steve Reich's "Clapping Music" in Mégra, by Niklas Reppel
(sx 'clapping #t
  (xspread
    (every :n (mul 8 12) (skip 1))
    (loop 'music "hc hc hc ~ hc hc ~ hc ~ hc hc ~")))
```

Nearly all of the keywords here are concerned with the structure, where the main structure is a `loop` (named `'music`), and the `xspread` duplicates the structure with the shift operation applied, thus adding the second "player".

The pattern notation `"hc hc [...]"` (here written as a shorthand string) represents the basic bell pattern in a human-readable format, yet the effective time spacing and duration of the events are more details that are abstracted away. If not specified, the default (200ms) is used.

The overall structure represents the event flow. Events flow from bottom right to top left, where the `sx` function sets the context (named `'clapping`) for event evaluation. This is where the events are sent to the synthesizer to be turned into sound. The syntax relies more on English words, which leads to slightly elevated verbosity, as compared to e.g. the Živa implementation (which relies more heavily on operators). The overall goal is a high *closeness of mapping* using a declarative programming style.

Mégra provides a syntax for a probabilistic finite automata (PFA) grammar that is inherently stateful (Reppel 2020). All basic structures, even deterministic ones such as the `loop`, are represented as PFAs. The current automaton state is stored, the next one calculated as needed. Each state transition yields an event. Thus, the shift operation is easily implemented. Every 96 steps, the next two transitions are calculated, and the first result is dropped. In a simple loop like this one, the that effectively translates to skipping one step of the loop. The procedure is abstracted as `(every :n (mul 8 12) (skip 1))`.

4.7 Mercury

The Mercury language and environment is a domain specific and human-readable language for the live coding of algorithmic electronic music (Hoogland 2019). It is minimal in the sense that it is highly abstracted and the programmer doesn't need much complex code to start playing a looping sound. It is not minimal in the sense of the amount of characters that have to be typed. On the contrary, in order to keep the language human-readable most functions have to be written as full words (although some have aliases). The language was designed with both the performer and the audience in mind. The high level of abstraction allows the performer to quickly focus on the composition and at the same time the audience can clearly follow what the artist is working on. By using musical words for functions and settings the transparency and comprehension are increased.

Mercury finds its roots in the concept of (Total) Serialism, a musical composition technique, where all parameters such as pitch, rhythm and dynamics are expressed as a series of values that adjust the instruments state over time (Magnuson 2009). These series are referred to as `lists`. Lists start with a row (often procedurally generated) and can be further transformed by applying functions such as `rotate`, `join`, `reverse` and `shuffle`. These features build upon the idea described by Spiegel as "a basic library consisting of the most elemental transformations which have consistently been successfully used on musical patterns" (Spiegel 1981). All the lists are generated and transformed upon evaluation of the code, making them static while the music is running.

In order to code the *Clapping Music* with Mercury, we need to create two instruments with `new sample`. This will instantiate two samplers that both have a clap sample loaded that can be played at an eight note (`1/8`) `time()` interval. They can be `pan()`-ned hard left and right to distinguish them. The global transport of the environment will have a `set tempo` of `170` BPM. The initial rhythm can be created as a list of `1`'s and `0`'s. This list will be sequenced by the samplers via the `play()` method.

We now code the right-hand part of the piece, applying the various transformations to the list in order to get the full sequence. These transformations are: Repeat the list 8 times (`copy()`), rotate the list to the left, 0, 1, 2, ..., 10 and 11 steps (`rotate()`, or `rot()`), and concatenate the lists into one long list (`join()`).

```
// implementation of Steve Reich's "Clapping Music" in Mercury, by Timo Hoogland
set tempo 170

list ptn copy([1 1 1 0 1 1 0 1 0 1 1 0] 8)
list ptn2 join(ptn rot(ptn -1) rot(ptn -2) rot(ptn -3) rot(ptn -4) rot(ptn -5) rot(ptn -6)
               rot(ptn -7) rot(ptn -8) rot(ptn -9) rot(ptn -10) rot(ptn -11))

new sample clap_808 time(1/8) play(ptn) pan(-1)
new sample clap_909 time(1/8) play(ptn2) pan(1)
```

Making this exercise with Mercury revealed a limitation of how the language currently is implemented and how the list functions work. You will notice there is quite a lot of repetition in the code, resulting in lots of typing (which would've been even more when using the full `rotate` function name). At the moment there is no function in Mercury that can be used to make an iterative process. A `for`-statement is not part of the language because all loops were abstracted away in list functions. This immediately shows a downside of working at higher levels of abstraction in a domain specific language; it is not always easy to add functions, extend the grammar or change the behaviour on-the-fly.

While reflecting on this limitation in the design of the language, several alternatives were considered to overcome this hurdle in the future:

1. The simplest and fastest solution is to implement a feature in the `rotate()` function that allows a `list` as input for the rotation argument. The list is used to iteratively evaluate the rotation for every item in the list. This would look like: `rotate(ptn [0 -1 -2 ... -11])`
2. A more general solution could be to implement a function that allows to iteratively evaluate some other function, concatenating the results of that function in the output. The iterative function could look like: `iter([0 -1 -2 ... 11] rotate(ptn i))`, where the first list is used to iterate over, placing the number on the `i` in the `rotate()` function.
3. Lastly an extension on the grammar of Mercury could be made, where placing a list as an argument in a function always uses the list to evaluate the function iteratively based on the content of the list. However, this could have breaking consequences. For example `add([1 2 3] [10 20])` currently results in `[11 22 13]`, while with this new grammar it would result in for example: `[11 22 13 21 12 23]`.

```
// possible future implementation
set tempo 170

list ptn copy([1 1 1 0 1 1 0 1 0 1 1 0] 8)
list ptn2 rotate(ptn [0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11])

new sample clap_909 time(1/8) play(ptn)
new sample clap_808 time(1/8) play(ptn2)
```

This code could be reduced even further with the procedural list functions `hexBeat` (`hex`) and `spreadInclusive` (`spreadInc`). The hexBeat function generates a binary pattern from a hexadecimal value. This way a single letter `e` represents a 4-bit pattern of `1 1 1 0`. This follows that `hex('ed6')` results in `1 1 1 0 1 1 0 1 0 1 1 0`. The spreadInclusive function generates a list of a specified length, starting at one value and ending at another value, creating equal-distant steps in between. So in order to generate the list of all rotations the function `spreadInc(12 0 -11)` does the trick, resulting in `[0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11]`. The final result of the whole pattern, with future implementation idea, will look like:

```
// a shorter notation using the hexadecimalBeat and spreadInclusive functions
// with the lists written in line instead of declared before the instruments are instantiated
set tempo 170

new sample clap_909 time(1/8) play(hex('ed6'))
new sample clap_808 time(1/8) play(rotate(copy(hex('ed6') 8) spreadInc(12 0 -11)))
```

4.8 MicroBlocks

MicroBlocks is an educational, block-based programming language featuring a virtual machine (inspired by the Squeak Smalltalk-80 VM) that runs on a wide variety of 32-bit microcontrollers. As the user codes, the (visual) scripts are compiled into byte code incrementally and sent in real time to the microcontroller. MicroBlocks wasn't designed for live coding music, but it is a concurrent and interactive language nevertheless, which made it possible to write a few libraries –written in MicroBlocks itself– that provide music abstractions for harmony, rhythm, and MIDI communications.

Its most versatile abstraction is arguably the `[arpeggiate]` block, that allows users to define a list of notes over which to arpeggiate –usually a chord or scale– and accepts shorthand list notation for both note position order and duration of each note. In this case, the block arpeggiates on MIDI channel 10 –the drums channel in the General MIDI specification– over a list containing the single drum instrument note *Hand Clap*.

In the `order` input, `1` refers to the first item in the list –that is, the *Hand Clap* MIDI drum note–, and `0` indicates a rest. If the list of notes were longer than a single item, one could use increasingly large numbers to refer to its items. The `duration` input contains a single number, `8` –short for $1/8^{\text{th}}$ note duration–, which propagates across all notes. If further items were added to this shorthand list, the corresponding durations would be wrapped over the list of position indices. Note how the first script always starts the arpeggio at beat number 1, while the second script keeps incrementing the start beat by one every eight iterations.

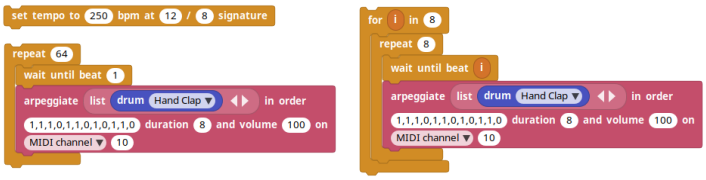

Figure 2: *MicroBlocks Code for Clapping Music*

One of the aims of MicroBlocks is to make code as readable and intuitive as possible while, especially in the case of the libraries for live coding music, being succinct enough to allow for quick music creation. Even though this code could be made slightly shorter, this version tries to maintain the readability of the project, while remaining short enough to be written in a live performance setting.

4.9 Punctual

The piece is expressed here visually, with left/right shaped blips of light, rather than clapping sounds. The base pattern of on/off pulses is expressed as a literal sequence of ones and zeros, and the right side version is shifted every eight cycles of the base pattern by shifting time by increasing amounts.

```
-- Punctual (v0.5) version of Steve Reich's Clapping Music
-- by Glen Fraser
-- (run in https://dktr0.github.io/Punctual/index-0.5.html)
z << 0;
claps << 0.5 * [1,z,1,z,1,z,0,z,1,z,1,z,0,z,1,z,0,z,1,z,1,z,0,z];
base << seq claps;
i << slow 8 $ floor(beat) % 12;
offset << early (i/12) $ seq claps;

slow 4 $
  rect [(i + 0.5 - 6) / 6, -1] 0.1
  + (fit 1 $ (prox [-0.9,0.5] ** 8 * base + prox [0.9,0.5] ** 8 * offset)) >> add;

zoom 0.9 $ move [0,-0.2] $ fb fxy * 0.8 >> add;
```

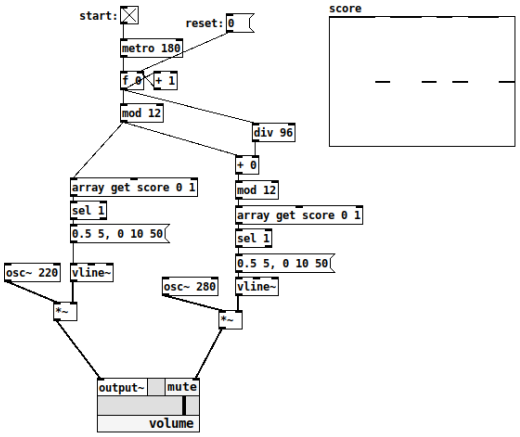

Figure 3: *Pure Data patcher of clapping music*

4.10 Pure Data

Pure Data (Puckette et al. 1997) is a visual programming language first released in 1996. Its permissive licensing, extensibility and good performance contribute to its continued popularity in electronic music. It was not designed with live coding in mind, but can be used as such and there are extensions to facilitate that (Brown 2023).

This implementation of *Clapping Music* uses only functions available in the *vanilla* version of Pure Data. The bell pattern is represented as an array of 0s and 1s seen at the right side, using the standard way arrays are created and displayed in PD. The points in the array can be edited graphically. The `[metro 180]` function provides a metronome that beats every 180ms (roughly 166 bpm). This is fed into the classic Pure Data counter - a `[f]` (float) object holding a number, that is fed its content + 1 every time the metronome "beats". The output of this is a constantly incrementing number. With the use of a modulo operation we produce an index into our array, reading the index at that number, only sending a trigger to a simple envelope generator `[vline~]` if we read a 1 (`[sel 1]`) that gets multiplied with a sine wave `[osc~ 220]` and sent to the left output.

This is the first clapper - the second one works the same except the number produced by the counter is integer divided by 96 (12 * 8) and is used to offset the second clapper's index into the array versus the first by one more every 8 bars and the output is panned hard right.

Live coding in vanilla Pure Data can get very messy very quickly because of the graphical nature of the language and its low level of abstraction, but this piece really lends itself to Pure Data's straightforward, simple handling of time and data. We can also observe high *visibility*, as the piece is completely self-contained yet compact, with every element from sound generation to structural organization being represented on little screen space. Compared to the other visual language example, *MicroBlocks*, Pure Data focuses on displaying the data flow, rather than the discrete control structure.

4.11 P5LIVE

In the implementation of *Clapping Music* in p5live, two arrays are used to represent the two players: `clap1`, which keeps the pattern fixed, and `clap2`, which rotates the pattern by moving the first element to the last position, while shifting the others to the left. To do this, the `shift()` method was used to remove the first note of the array, and `push()` method to add this note to the last slot of the array. Each time the function `draw()` loops for the length of `clap1`, a counter `rep` increments. After `clap1` has looped 8 times, the notes in `clap2` are shifted. A frame rate of 5 fps is set to approximate a tempo of 92 bpm with 3 beats per bar. To visualize the piece, the canvas is split into two rectangles, where the fill of each changes between white (1) and black (0) based on the notes in the corresponding arrays (`clap1` for the left and `clap2` for the right).

This example follows a clearly imperative paradigm, both in its appearance and its control flow. Yet, no explicit loop is visible, as the main loop that drives the program forward is abstracted away by the convention of the *Processing* type of languages, where the `draw()` method is implicitly called repetitively at the frequency defined by `frameRate(...)`. The iterative nature of the piece is highlighted by classic elements of imperative programming, such as the counter variables `note` and `rep`, and the `if` statements structuring the program. This is in stark contrast to the declarative and functional styles we've seen in the other examples, which are typically focused on expressing the idea "as a whole" and abstract away the details of the process of iteration.

```
/* Implementation of Steve Reich's Clapping Music in P5Live by Julia Múgica */
let clap1 = [1,1,1,0,1,1,0,1,0,1,1,0];
let clap2 = [1,1,1,0,1,1,0,1,0,1,1,0];

let note = 0;
let rep = 0;

function setup() {
    createCanvas(windowWidth, windowHeight);
}

function draw() {
    noStroke();
    background(0);
    frameRate(5);

    clapping(clap1[note], 0, 0, windowWidth/2, windowHeight);
    clapping(clap2[note], windowWidth/2, 0, windowWidth, windowHeight);

    if(note==clap1.length){
        note = 0;
        rep++;
    }
    if(rep == 7){
        rep = 0;
        let first = clap2.shift();
        clap2.push(first);
    }
    note++;
}

function clapping(c, x1, y1, x2, y2){
    if(c == 1){
        fill(255);
    } else {
        fill(0);
    }
    rect(x1, y1, x2, y2);
}
```

5 Discussion

Clapping Music mainly uses rhythmic information (apart form the clap sample). Timbre and dynamics (in the forms of accents) are only a secondary aspect of the piece, arising only due to the player performing. Other musical parameters such as melody and harmony are not part of the score at all. Therefore the languages are barely "tested" in those fields. However, simplifying the dimensions also helps to keep the case study focused, and rhythm is a common denominator in all live coding languages under observation. For a future research it is an interesting idea to look at other algorithmically composed pieces that incorporate any of these other musical parameters and apply this same exercise to all the languages.

On the dimension of compactness (in terms of number of characters) languages already having abstractions for sample playback offer more compact implementations. In those cases the sound design is abstracted from the performance. This does not mean that it cannot be changed (through for example filtering, or other DSP techniques, etc.), but there is a preprogrammed default. The SuperCollider implementation of the "`\pols`", clearly the most verbose, shows all the code of the synth definition on the screen. While this verbosity will lead to more visibility of the underlying algorithms, potentially allowing for adjustments, it comes at the expense of a higher cognitive load in a live coding performance, having to memorize lots of code and requiring more intensive practice, while increasing the chance (or risk?) for errors.

Analyzing the representation of a musical pattern in all languages, it seems most create a duplicated looping structure, with the exception of TidalCycles that already assumes a cyclical structure of time. Mercury assumes a loop structure (the differences between *loop* and *cycle* are nuanced, but crucial), represented by a list, whereas in Mégra, as discussed above, the loop is represented as a PFA (or Markov Chain) where transitions yield events, the loop nature only being a special case of a more general structure.

Regarding time abstractions, SuperCollider implementations are based on the `Pattern` abstract class which schedules events relative to an embedded tempo defined in beats per second. In fact, one of the most common abstractions to be found is a default tempo. In the case of Mercury there is a global transport with a variable named tempo which defines the grid for scheduling (re)-triggering of the instruments. In the Pure Data implementation the read index is incremented every step and passed into the array object without explicitly looping (resetting the global counter). The *loop* is expressed mathematically through modulo operations. MicroBlocks has a `set tempo` command block (every block is a node of an Abstract Syntax Tree) that initializes tempo controlling the loops. P5 implementation uses the `draw` function (controlling frames per second) as time abstraction and in Punctual the *tempo* is embedded in the language although the default tempo can be modified with a global variable.

Discussing with the code authors about the motivations for participating in this exercise, we could see some clear tendencies emerging (see Chapter 3, "Case Studies"). Considering the motivations of the authors is important since live coding operates on socio-cultural, structural, musicological and performative levels. Live coding plays with the technical errors coming from algorithms, sometimes producing unexpected outputs. Live coding embraces the limits of technical systems rather than avoiding errors and can therefore privilege readability over compactness, but as we have seen, the balance is up to the language designers.

Indeed, we could write a language called *Reich* with a single `clappingMusic()` function, arguably to reach the highest possible level of abstraction. Would that be better or worse? Is live coding without the possibility of errors still live coding? Is live coding without seeing the algorithms and thought processes being typed live still live coding? Would the audience find this interesting?

While none of the examples presented have been written in an actual performance, they could have happened inadvertently, given that the piece uses a technique very popular in pattern-based live coding. And indeed, most languages we observed have a pattern-based abstraction, most likely owing to the popularity of algorave-oriented styles.

It is clearly possible to think of alternative implementations in each language. In SuperCollider, for example we could rewrite the synths to make it more compact, whereas in PureData we could use fewer objects, but more connections. However, each implementation is a trade-off involving readability, expressiveness, aesthetic result and cognitive complexity of the execution. The particularities of each grammar and the implementations presented visualize what each language has decided to showcase and to abstract from the live performance. If domain-specific languages can define single-line abstractions, querying it with a particular implementation offers an interesting insight to what it has say about live coding.

6 Conclusion

We are aware that this exercise does not give insight into all the abstractions, characteristic affordances and syntax constraints of every DSL. However, we do think that it offers great insights for further analysis. By querying the code of each language individually, and juxtaposing the different paradigms, under the aspect of the implementation of *Clapping Music*, asking the aforementioned questions, we found it straightforward to highlight the affordances and challenges of each language's grammar, in this case the time-related musical abstractions and the high-level representation of musical patterns.

Further work is needed to define the analysis toolbox more rigorously and extend it to more dimensions beyond the rhythmical patterns, such as timbre, harmony or melody in the music domain, texture, color and shape in the visual domain, or movements in the world of choreography. We also acknowledge that, due to the fact that this paper came out of a post in a Telegram group (on a very short notice) not everyone might have had a chance to contribute with their implementation. We invite everyone reading this paper to contribute more implementations in other (or the same) languages, and also suggest other interesting ideas and pieces to highlight these dimensions. There is a github repository available on the Toplap Barcelona account where you can send a pull request [4].

References

Baalman, Marije. 2015. "Embodiment of Code." In *Proceedings of the First International Conference on Live Coding*, 35–40. Leeds, UK: ICSRiM, University of Leeds. https://doi.org/10.5281/zenodo.18748.

Bek, Wilfried Hou Je. 2008. "Loop." In *Software Studies: A Lexicon*, edited by Matthew Fuller. MIT Press.

Bell, Renick. 2013. "PRAGMATIC AESTHETIC EVALUATION OF ABSTRACTIONS FOR LIVE CODING." *Journal of the Japanese Society for Sonic Art* Vol. 5 No. 2: 1–8.

Blackwell, Alan F., and Thomas R. G. Green. 2003. "Notational Systems – the Cognitive Dimensions of Notations Framework." In *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, edited by John M. Carrol, 103–34. Morgan Kaufmann.

Brown, Andrew R. 2023. "Live Coding Patterns and a Toolkit for Pure Data." *Organised Sound* 28 (2): 264–75.

Burland, Karen, and Alex McLean. 2016. "Understanding Live Coding Events." In *International Journal of Performance Arts and Digital Media*, 12 (2):139–51.

Cox, Geoff, and Winnie Soon. 2021. *Aesthetic Programming: A Handbook of Software Studies*. Open Humanites Press.

Crutzen, Cecile, and Erna Kotkamp. 2008. "Object Orientation." In *Software Studies: A Lexicon*, edited by Matthew Fuller. MIT Press.

Fazi, M. Beatrice, and Matthew Fuller. 2016. "Computational Aesthetics." In *A Companion to Digital Art*, edited by Christiane Paul, 281–96. John Wiley & Sons.

Fraser, Glen. 2020. https://github.com/totalgee/bacalao.

Fuller, Matthew, and Andrew Goffey. 2014. "The Unknown Objects of Object-Orientation." In *Objects and Materials*, 218–27. Routledge.

[4] https://github.com/toplap-cat/clapping-with-code

Haack, Joel K. 1998. “The Mathematics of Steve Reich’s” Clapping Music”.” In *Bridges: Mathematical Connections in Art, Music, and Science*, 87–92.

Hoogland, Timo. 2019. “Mercury: A Live Coding Environment Focussed on Quick Expression for Composing, Performing and Communicating.” In *Proceedings of the Fourth International Conference on Live Coding 2019, Madrid*. Medialab Prado / Madrid Destino. https://doi.org/10.5281/zenodo.3946338.

Magnuson, Phillip. 2009. “Serialism.” In *Sound Patterns, a Structural Examination of Tonality, Vocabulary, Texture, Sonorities, and Time Organization in Western Art Music*. Phillip Magnuson. https://tobyrush.com/soundpatterns/microcosms/serialism.html.

McLean, Alex, and the Contributors. 2023. https://github.com/tidalcycles/Tidal.

McLean, Alex, and Geraint Wiggins. 2011. “Texture: Visual Notation for Live Coding of Pattern.” In *Proceedings of the International Computer Music Conference*, 621–28.

Mori, Giovanni. 2015. “Analysing Live Coding with Ethnographic Approach - A New Perspective.” In *Proceedings of the First International Conference on Live Coding*, 117–24. Leeds, UK: ICSRiM, University of Leeds. https://doi.org/10.5281/zenodo.19343.

Pibernat, Roger. 2023. “Živa - Easy Live Coding with SuperCollider.” In *Proceedings of the 7th International Conference on Live Coding 2023*.

Puckette, Miller S et al. 1997. “Pure Data.” In *ICMC*.

Reppel, Niklas. 2020. “The Mégra System - Small Data Music Composition and Live Coding Performance.” In *Proceedings of the International Conference on Live Coding 2020*, 95–104.

———. 2024. https://github.com/the-drunk-coder/megra.rs.

Ressi, Christof. 2018. https://git.iem.at/pd/vstplugin.

Roberts, Charlie, and Graham Wakefield. 2018. “Tensions & Techniques in Live Coding Performance.” In *Oxford Handbook of Algorithmic Music*, edited by Alex McLean, 293–317. Oxford University Press.

Roos, Felix, and Alex McLean. 2023. “Strudel: Live Coding Patterns on the Web.” In *Proceedings of the 7th International Conference on Live Coding 2023*.

Spiegel, Laurie. 1981. “Manipulations of Musical Patterns.” In *Proceedings of the Symposium on Small Computers and the Arts*, 19–22. IEEE Computer Society Catalog No. 393.

Star, Susan Leigh. 1991. “Invisible Work and Silenced Dialogues in Representing Knowledge.” *Women, Work and Computerization: Understanding and Overcoming Bias in Work and Education. Amsterdam: North Holland*, 81–92.

“TOPLAP Barcelona Community Report 2023.” 2023. In *Proceedings of the 7th International Conference on Live Coding 2023, Utrecht, Netherlands*. Zenodo. https://doi.org/10.5281/zenodo.7843519.

Toplap, Earth. 20xx. https://github.com/toplap/awesome-livecoding.

Villaseñor-Ramírez, Hernani, and Iván Paz. 2020. “Live Coding from Scratch: The Cases of Practice in Mexico City and Barcelona.” In *Proceedings of the International Conference on Live Coding 2020*.

Wikipedia. 2024. “Wikipedia – Clapping Music.” https://en.wikipedia.org/wiki/Clapping_Music.