# LiveCodeLab 2.0 and its language LiveCodeLang

Davide Della Casa
davidedc@davidedc.com

Guy John
guy@rumblesan.com

## Abstract

We present LiveCodeLab 2.0, a web-based livecoding framework, and its language LiveCodeLang. We describe its operation, its connection with other livecoding frameworks, and its aspects related to functional programming.

***Categories and Subject Descriptors*** D.3.3 [**Programming Languages**]: Language Constructs and Features – Control structures, Frameworks.

***General Terms*** Design, Languages.

***Keywords*** DSL; livecoding; functional programming.

## 1. LiveCodeLab/LiveCodeLang overview

LiveCodeLab [3] is a web-based livecoding [19, 20] environment developed under the TOPLAP manifesto [18] for real-time 3d visuals and sample-based sequencing. LiveCodeLab uses a custom DSL named LiveCodeLang. Although "LiveCodeLab" is the name of the entire project (including the site, the API, the name of the performing duo, the language, the editing environment and the included examples/tutorials), in the context of this paper we'll mean "LiveCodeLab" to be just the livecoding environment, the language and the API. "LiveCodeLang" indicates the language in isolation.

## 2. Influences

LiveCodeLab has been inspired by the TOPLAP manifesto, which articulates in 12 points a set of directives, statements and preferences regarding performances involving (but not limited to) live programming. Technically, LiveCodeLab has been directly influenced by Processing [14], Jsaxus [1], Fluxus [6] and Flaxus [8]. From all these frameworks we've taken the immediate mode rendering style, i.e. scene-graph nodes handles are not explicitly given to and manipulated by users (although Fluxus for example *can also* give users access to primitive data). We've been inspired by Jsaxus' graphic style and Jsaxus' demos have made us aware of the technical feasibility of a livecoding environment implemented in HTML5. We've also been highly inspired by the graphic style of Flaxus, in particular the visual effectiveness of background filling primitives beyond solid fills.


*FARM'14*, September 6, 2014, Gothenburg, Sweden.


## 3. Motive

The motive to develop LiveCodeLab was to bring together all of the elements we loved about other livecoding environments, and to ground such environment on a new language that would be both compact, expressive, and immediately accessible to an audience with low computer literacy. By developing a custom DSL (as opposed to directly using an off-the-shelf language) we can make nimble decisions about how the language should behave and feel like to both the performers and the audience.

## 4. Research/implementation approach

Both the LiveCodeLab language and the API could have evolved in many different ways according to the many combinations of feature ideas and technical options, but often the implementation decisions were made according to a key constraint in our research method: it needed (and to some degree still needs) to be "progressive" in its nature, i.e. partial findings would have to be prototyped and made to work in short iteration cycles.

As an example, version zero of LiveCodeLab used JavaScript as its language - same as per Jsaxus' solution. The next iterations used CoffeeScript, which is more syntactically compact. Subsequent iterations then added simple rewriting rules to avoid the unnecessary parentheses to indicate method invocation with no parameters - as it's often the case that users draw simple graphic primitives that way. Next iterations added more syntactic and API shortcuts by addition (and occasional refactoring) of rewriting rules.

## 5. Core values

> *"It is not necessary for a lay audience to understand the code to appreciate it, much as it is not necessary to know how to play guitar in order to appreciate watching a guitar performance."*
> Aknowledgment point number 1 - TOPLAP Manifesto

Although the TOPLAP manifesto is neutral in regards to the necessity "for a lay audience to understand the code to appreciate it", LiveCodeLab and LiveCodeLang deliberately choose to do what's needed for a lay audience not only to understand but also to pick up and use the language quickly, all while keeping LiveCodeLab expressive. This is a similar approach for example to Thingee and its language ThingeeLanguage by Amy Alexander [19, 20] and, at least it seems to us, David Griffiths' Daisy Chain, Al Jazari and SchemeBricks [7, 11]. The fact that LiveCodeLab is a zero-installation web-based environment also plays well in this respect.

This "accessibility" choice to make things easy for new users is not inherently right or wrong, also "easy" and "hard" are subjective aspects, and one could argue for example that something apparently basic for the authors, such as the meaning of the infix operator "=", does not have a single "natural" "easy" meaning and in fact *does* require explanation to a lay audience.

For the scope of this project we just declared this accessibility choice as a requirement we decided to adhere to, and we maintain that we derived a self-consistent system that offers good trade-offs between conciseness, learning curve (both for reading and writing) and expressivity.

Without anticipating a deeper presentation of the language, here are some examples of decisions we made consistently with this choice, many of which are unique in respect to other Livecoding frameworks we are aware of (exceptions noted):

- Graphic state changes commands have meaningful and interesting default behaviour when invoked without arguments (à la Processing).
- Eliminating parentheses for function-invocation notation à la Ruby and Scala to make a few examples, CoffeeScript for many cases, Unix shells, and ThingeeLanguage. Note that, just like in Ruby, Scala and CoffeeScript, precedence is such that arguments belong to the closest function to the left i.e.:

  `a b c,d,e` *is same as* `a(b(c,d,e))`

  If another grouping of arguments is intended, then parentheses are needed for setting priority (but not to indicate the function call):

  `a (b c,d),e` *is same as* `a(b(c,d),e)`

  note again that the parentheses denote priority as in the traditional "arithmetic precedence" sense, and (a little bit like in Lisp) are *not* used to separate the function name from the arguments.
- Making available to users a large set of literals for example all 140 CSS colour literals.
- Making use of indentation as a help to avoid parentheses, braces and semicolons (à la Python, CoffeeScript). We believe in indentation as a strong visual cue that most people associate with "grouping" without need for explanation.
- Making use of indentation to define the scope of graphics state changes (shorter equivalent to the pushMatrix()/popMatrix() command combinations in Processing and Jsaxus, and equivalent to the "with-state" construct of Fluxus).
- Anonymous functions are often used transparently without need of any specific notation ("function" notation, "lambda" notation, arrow notation).
- Providing several shorthands for commands. As an example "`fill red`" can be also be expressed as "`red fill`" or simply as "`red`".
- Multiple steps can generally be inlined. As an example:

```
rotate
red
box
```

  can be inlined into "rotate red box" (with some notes about scoping described later in more detail)

## 6. LiveCodeLab: quick dive-in examples

The simplest program a user can write in LiveCodeLab is:

```
box
```

This will create a cube of unit size in the middle of the screen. Note how LiveCodeLab follows the Fluxus way of default drawing primitives in the middle of the screen (as opposed to Processing, where the world coordinates and camera arrangement cause the equivalent command to draw a cube at the top-left corner). A slightly more interesting scene can be created by using the rotate command.

```
rotate
box
```

Here the cube will rotate freely, still centred on screen. At this point it makes most sense to start looking at most function calls in LiveCodeLab as impure functions that work through their side effects. In this case, the rotate function will affect the global orientation of everything following it. Also the `box` function affects the global graphical state. In fact, all functions that handle matrix transformations and shape creation work in this fashion

There are two things to note about these programs. First, both box and rotate are functions that optionally take parameters, but the "no parameters" default gives an interesting behaviour already. In the example above, "rotate" without parameters animates a continuous rotation (as opposed to rotating the world of a specific amount when parameters are passed). Secondly, these programs will run as soon as they are typed in. LiveCodeLab uses an "aggressive" execution model: whenever there is a change to the program, the environment will immediately attempt to read and interpret it. If the program is valid, then LCL will run it until further edits are made.

LiveCodeLab has borrowed ideas liberally from Processing, many of the keywords and constructs are immediately recognizable:

```
background red
rotate
stroke green
noFill
box
fill white
ball
```

Just like in Processing, the colour and matrix commands immediately affect the graphic state, so the above program will result in a white ball with green strokes, positioned in the centre of the green vertices of a cube, all rotating in front of a red background.

It is also possible to use all matrix and colour commands with block scoping (note how indentation is significant). Similarly to the `rotate` and `box` functions above, the default usage of the stroke and fill functions will work through their side effects to affect all shapes following them.

```
fill green
rotate // only affects indented parts
    box
    fill red // only affects ball
        ball
box // unaffected by "rotate" and red fill
```

This program will display a green rotating box, a red rotating sphere and a fixed green box. In this case the side effects of the `rotate` and `fill` functions are contained to only the indented blocks. This is achieved by pushing matrix and colour states onto a stack, performing the code in the block and then popping the new states back off. The `ball` command will be performed using this enclosing state, but commands outside of the block will not. In essence these blocks are actually just closures that are passed as arguments to the rotate or fill functions. The user is also free to inline graphic state commands and primitives, and in-lining implies "nested scoping down the line", so the above snippet is exactly the same as:

```
fill green
rotate box fill red ball
box // unaffected by "rotate" and red fill
```

Finally, LiveCodeLab allows the use of loops to iterate over a block of commands:

```
5 times
    rotate
    box
```

or the similar version that supports binding of a variable:

```
5 times with i
    rotate
    box i // nested boxes, i is the scale
```

This will create five cubes, all rotating at different speeds (the rotation is compounded at each loop iteration, just like it would in Processing). Note that in this latter case only the biggest "encasing" box is visible (they are opaque by default), and one of the inner boxes isn't even drawn since its scale is zero ("`times`" index starts at zero). In both of these programs, the rotation transformations are building on the previous matrix state which results from their side effects. If the `box` command were indented so that the push/pop functionality of `rotate` were used, then there would still be five cubes, but their rotation transformations would all be the same amount, so it would appear to the user as being just a single cube.

## 7. LiveCodeLab internals: the four main states

As in all livecoding environments, there is a lot going on in LiveCodeLab (editing, graphics, sounds, housekeeping), apparently all at the same time. And yet LiveCodeLab is a single-threaded environment with four main states, where four groups of activities are independently performed at different intervals.

***Code editor update and translation/parsing.*** LiveCodeLab is in this state every time (or soon afterwards) the program is edited, on keyboard/mouse events. Note that, although LiveCodeLab doesn't pursue this solution at the moment, in theory code editor updates could trigger a separate Web Worker thread to do the translation/parsing rather than doing that immediately in the main thread. The details of the translation/parsing step are discussed in the next chapters.

***User-program running and graphics rendering.*** This is done up to 60 times per second. Among other things, the user program builds the 3d scene and updates the data structures needed to playback the audio samples (more on this below). The running of the user program could, hypothetically speaking, be done at a different interval in a dedicated state separately from the actual 3d rendering. In practice though, it would be of little use to run the user program (which sets up the 3d scene) and then not rendering the scene right away.

***Audio samples playback.*** Audio samples are snippets of pre-recorded (or pre-synthesised) sound. All samples' playbacks are triggered from a dedicated handler running at the interval specified by a "beats per minute" value - the user can change this value via a "bpm" command. Starting the playback is a "fire and forget" asynchronous operation: once the playback is triggered the thread is free to move on and there is no follow-up.

***Autocoder.*** Toggled by the user, it randomly changes the user code (live, in the editor) every second (see next section).

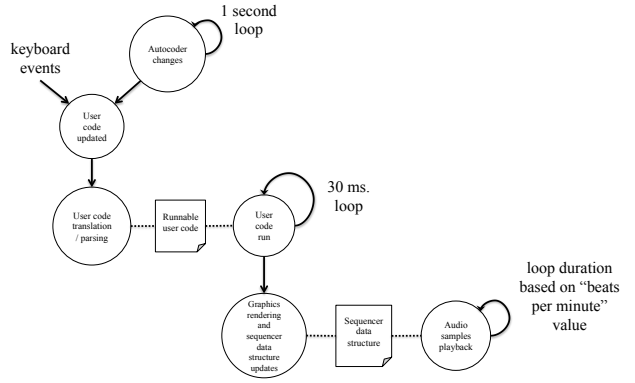


**Figure 1.** LiveCodeLab main states and trigger events

## 8. The Autocoder

The Autocoder feature allows LiveCodeLab to programmatically rewrite/modify user programs while they are running, making live changes within the editor. The Autocoder is meant to be a tool to aid in exploration of the code, providing some degree of randomness, which may give rise to surprising results.

Currently the autocoding capability is fairly simple, confined largely to changing number values (separate changes for integers and floating point), colour names, primitive commands and matrix commands. Sample names and sample patterns in "`play`" commands are currently left unchanged. The changes are random - they are not driven by any type of analysis of the program or its behaviour.

We are considering AST-based code analysis and edits that would allow for more interesting program changes, including perhaps code editing assistance and automatic composition.

## 9. Hot-swapping

In LiveCodeLab the user program is updated *while* typed, there is no explicit "register" or "update" mode, no special trigger such as CTRL-enter or shift-enter, no "play button". This behaviour places LiveCodeLab at "level-4 liveness" in the Tanimoto liveness hierarchy [17].

At any moment, while being typed, the user program can either be statically correct or incorrect. In JavaScript and CoffeeScript static correctness roughly equals to correct syntax, as no type system checks are done for example. In some cases the program *is* statically correct but might perform "differently than expected" because of its transient state while being typed: we practically ignore this case as we consider such transient states as part of the "constructive" nature of the performance, and in our experience they don't detract from its quality.

If the program is statically correct, it becomes a candidate for being run at animation speed (up to 60 times per second according to graphics/sound load and host system performance). If the program *is not* statically correct then it's ignored and it's not a candidate for running.

The second check is for runtime errors, for example an array boundary is exceeded or a non-existent function is called at some point - the sort of errors that *in general* can only be detected at runtime (note again that many checks that are normally performed at compile time in other languages can't be done at compile time in JS/CoffeScript due to their dynamic nature).

Programs that fail at runtime need to be swapped-out with hopefully "sane" programs to keep the animation from stopping. To create this "safety net" against misbehaving user programs, these are kept in a "quarantine" state for the first few seconds of their runs. If the program runs *without* run-time errors during this quarantine period, then LiveCodeLab is going to judge it as stable and marks it as the "last stable program". Whenever the *subsequent* program(s) fail (either at compile time or runtime), then this "last stable program" is brought up and run again, with the hope (but not the guarantee) that its good run-time behaviour in the past vouches for it to be a good stand-in.

Note that there is no guarantee that this fallback safety mechanism is going to work *in general*: scenarios can be easily constructed of user programs behaving well for the quarantine period and then failing afterwards. In practice though the hot-swapping mechanism adequately covers practically occurring cases. The most common case is when the user program invokes a function which name *would be valid* if it was typed-in *in its entirety*, but it's actually invalid while being partially typed-in, as the prefix doesn't match the name of any available function.

## 10. State(lessness) across frames and program edits

The only parameters affecting the drawing and sound-playing in LiveCodeLab are a) the time in milliseconds since start of performance and b) the frame count. Note that the sources of randomness available to the user, i.e. the "random" and "noise" (Perlin noise) functions, can optionally be seeded based on a) and/or b) above.

The implication is that the previously described hot-swapping of programs can be done without worrying that there might be data structures to maintain or adapt when the user program changes. The user can obviously code her custom data-structures she might need, but all user-defined data structures are built afresh in each frame, which usually, for small/medium sized data structures, is not a problem. Alternatively the user can calculate only needed parts of data on-demand rather than larger sets of unneeded data.

An example of an **in**expensive data structure that is built anew each frame is a dictionary holding all the "audio samples" patterns. When the user wants a particular rhythm to be played, she types for example:

```
play 'tranceKick', 'zxz'
play 'tranceKick2', 'zzzx'
```

These patterns ('`zxz`' and '`zzzx`' in the example above) indicate when the sample is meant to be played: "z" stands for no-playback and "x" stands for playback, so in the example above, the first trance kick sound will play on beats 2,5,8, 11... while the second trance kick sound will play every 4 beats (4,8,12...). The "`play`" command above just adds the sample name and the pattern to a dictionary, this dictionary is then referenced by the "sample playing" code, running at an independent interval (optionally set by a "`bpm`" command) to actually start the samples playback at the right time according to the specified pattern. Although this storing of (often the same) patterns every frame is redundant, it constitutes no problem since the cost is negligible both in terms of time and memory.

This "from scratch" approach also applies to the scene graph: each frame discards the previous scene graph and creates a new one from zero. Although this could seem wasteful for no apparent good reason, this is just the result of LiveCodeLab sparing the user from having to build scenes as a hierarchy of nodes. This is intentional and, just like in Processing, it is in the interest of rapid prototyping. On the flip side, since node-less scene descriptions don't carry meaningful information on structures/hierarchy of the scenes, LiveCodeLab is not able to smartly update drawings across frames (for example where changing the transformation matrix of a single node would be sufficient); this is just like in Processing or any immediate-rendering 2D graphics framework such as the HTML5 Canvas 2D Context. Compare this to tree-based scene models, for example the DOM [4], where the user classifies and identifies divs and declares the relative positioning and nesting of such structures. Perhaps unsurprisingly, the more semantically rich the description provided by the user (e.g. grouping, tagging, classification, hierarchies, data dependencies graphs, relative positioning, nesting), the more optimisations and affordance can be provided by the underlying framework: it's common for example for UI/game frameworks to handle reflowing, visibility tests, tweened animation, events management, physics, and some web frameworks for example transparently manage view updates based on underlying data changes [13].

To mitigate some cases where "from scratch" is too expensive of an approach, sometimes "backing" (behind-the-scenes) data structures are transparently handled so that costly operations are cached and only performed when needed. As an example, background painting is potentially an expensive operation (rendering of multiple semi-transparent gradients in a 2d Canvas context or

via CSS is surprisingly expensive) and hence the background is repainted in its own separate layer *only if* its appearance actually needs updating rather than blindly being repainted with the same combination of colours over and over again in each frame. Redundant repainting is avoided by comparing new paint directives with the ones issued in the previous frame, stored in a dedicated data structure behind the scenes.

In summary: all graphics and sounds produced in/during each frame are a pure function of time and frame number, so much that LiveCodeLab could in principle expose a control to display the animation back in time, or backwards, or both.

## 11. LiveCodeLang: spec-sheet

As mentioned, one of the main aims in the design of LiveCodeLang was to keep a good balance between a language that would be powerful and expressive, whilst still being easy to learn for those unfamiliar with programming.

Ideally the code would read similar to natural language, and interactions would be clear to users. An even more important aspect is keeping language features to a minimum and having as little syntax as possible. In particular, we prefer ***compactness*** to "natural language looks", as opposed to, say, Lingo, the scripting language of Macromedia Director [12], where several "no-operation" keywords (such as "`the`", "`to`", "`frame`", "`marker`") could be optionally interspersed in programs to achieve prose-like fluency. We'd rather achieve prose-like fluency by careful optional *removal* of keywords as in the case of "rotate red box" not needing the "fill" command before "red". *Shorter* keywording seems to us the way humans prefer to input directions, we certainly feel that way any time we query search engines or map services. A great deal of inspiration was taken from CoffeeScript, Python and Ruby, both in the way they use indentation to define blocks, and also they aim to have a very orthogonal set of features.

Here we summarise more systematically the key principles behind LiveCodeLang, with short examples. For longer / more articulate examples and step-by-step tutorials please reference the LiveCodeLab embedded examples and tutorials.

- Eliminating parentheses for function-invocation notation (in most cases). Example:

```
box // draws a box
```

- Making available to users a large set of literals, for example all 140 CSS colour literals. Example:

```
fill red
box
```

- Making use of indentation as a help to avoid parentheses, braces and semicolons. Example:

```
if random > 0.5
    box
else
    peg
```

- Making use of indentation as to define the scope of graphics state changes. Examples:

```
rotate // only affects the box
    box
peg
```

and:

```
fill red // only affects the box
    box
peg
```

- Providing several shorthands for state changes in many circumstances. Example:

```
red // instead of fill red
box
```

- Multiple instructions can be generally inlined (which limits the scope). Example:

```
// rotation and color fill only affect box
rotate red box
// not affected by graphics state changes:
peg
```

- Iteration without binding a variable:

```
6 times
    move
box
```

- Iteration binding a variable:

```
6 times with i
    move i/10
box
```

- Higher-order-functions:

```
either = (a,b) ->
    if random > 0.5 then run a else run b
either <box>, <peg>
```

Note the "<>" notation for inlined anonymous functions, "`run`" is then used to actually evaluate the passed functions.

## 12. LiveCodeLang implementation

There are two implementations of LiveCodeLang:

- Nanopass source-to-source translation [15] to CoffeScript, with subsequent translation to JS, and evaluation via "native" JS runtime.
- Parsing into AST and then sandboxed interpretation.

***Nanopass source-to-source translation solution*** The nanopass source translation is made for quick prototyping, as it allows language enhancements via incremental addition of small rewriting steps following a TDD approach. Surprisingly, the rewriting steps rarely interfere, so analysis and new enhancements can be done in isolation from previous steps.
The matching of the rewriting rules is done via regexes. While regexes are (with a little care) extremely fast matchers, it is well known that they can't match balanced parentheses (and by extension, arithmetic expressions that occur in any program) [16] and are in general inadequate for non-regular language constructs (e.g. nested structures) that can normally occur in LiveCodeLang. Although the source-to-source translator passes a suite of over 270 tests covering common programs (including simple synthetic cases, all the examples of this paper and other demos/examples), complex nested programs can reveal these inherent limitations. Regexes are also in general difficult to read and inspect, so maintainability of this solution somewhat suffers.

(As an aside and for comparison, consider that regex-handled and grammar-less languages/implementations have some wildly successful representatives, see processing.js for the former [9] and bash [2] for the latter).

The obtained CoffeeScript source is then passed to the CoffeeScript compiler for transformation to JS, and then evaluated via the JS runtime.

The evaluation currently runs unchecked and unguarded, which means that a) it uses the only thread made available by the browser to all the JavaScript code within the tab, and b) has the same access to the browser resources as any other part of LiveCodeLab itself. Together with the benefits of native runtime execution and access to the hosting environment itself, there is the possibility of misuse (mistakenly or intentionally) of browser resources, especially should we decide to make user code sharable.

***Parser/AST/Interpretation solution*** A dedicated parser/AST/interpreter has also been created. There are many benefits to this solution.

- With a proper grammar defined, it is much easier to continue extending the language and make some of the more radical changes we want.
- Having access to a AST made it possible to create a "sandboxing" interpreter, allowing greater control over what code is run, and which resources can be accessed.
- The interpreter can be ported to other platforms beyond the browser.
- The AST will also open up the option to emit code, either JS or other languages.

The Jison parser generator was used to create the necessary LALR parser from the EBNF grammar. There is still a small amount of pre-processing that occurs which is primarily due the significance of indentation when defining blocks. This pre-processing inserts braces around blocks, making it possible to define a context-free grammar for the language.

It quickly became clear that to implement some of the language features we wanted, a basic type system would be necessary. This was built into the parser, allowing it to recognise between three classes of functions: primitive commands, matrix commands and style commands.

In this fashion, the parser is able to break down the program

```
rotate 3, 4, 4 scale 0.4 fill red box
```

into the relevant commands and with the relevant scoping.

## 13. LiveCodeLab / LiveCodeLang and functional programming

As mentioned, LiveCodeLab's output is a pure function of time and frame number, and previous chapters discussed how this relates to hot-swapping of programs and management of explicit and behind-the-scenes data structures that need to be maintained across frames.

Other considerations bring us further into "functional" discussions territory.

### 13.1 Functional aspects of JavaScript and the LiveCodeLab loop (specific to the nanopass translation to CoffeeScript)

JS higher-order function support is key in the CoffeeScript-based LiveCodeLab loop: each statically correct edit of the user program is turned into a JavaScript function, and stored waiting to be run at the next frame interval.

### 13.2 Functional aspects of JavaScript and LiveCodeLang implementation

JS higher-order function support is key in the implementation of LiveCodeLang, as the LiveCodeLang DSL (in both the nanopass and parser/AST implementations) makes use of the "Nested Closure" pattern described in [10].

A topical example of the workings of this pattern is the following:

```
rotate fill red box
```

which is turned into the CoffeScript (or AST) equivalent behind the scenes:

```
rotate (-> fill red, (->box())
```

(the *actual* translation to CoffeeScript is the more compact: "`rotate -> fill red, box`").

When "`rotate`" is invoked, it sets the proper matrix transformation and then evaluates the function passed as its argument (the "`fill`" command). Similarly, the "`fill`" command sets the color-fill to "`red`" and evaluates the chained function (the `box` command, which draws the box). Upon functions returns ("back up the chain"), the changes to the graphics state (color and rotation) are "undone", with the effect of having limited the rotation and color-fill to the box only.

### 13.3 Functional aspects of LiveCodeLang itself

LiveCodeLang itself supports higher-order functions. This is by all means only one aspect of functional programming, yet it is a key one and in this section we indulge in showing some examples straight from the familiar functional programming toolbox.

#### 13.3.1 Higher-order-function support

A simple example of higher-order-function support is:

```
either = (a,b) ->
    if random > 0.5 then run a else run b
either <box>, <peg>
```

which presents a flickering box/cylinder on screen. Note the compact "<>" notation for anonymous functions without bindings.

Also note how simply users can invent their own DSLs:

```
above  = <move 0,-0.5,0>
box above ball above peg
```

and:

```
flashing = <if random < 0.5 then scale 0>
flashing ball
peg // peg doesn't flash
```

#### 13.3.2 More complex examples (specific to the nanopass translation to CoffeeScript)

The nanopass translation to CoffeeScript is (almost) idempotent, hence all the resulting CoffeeScript code is itself accepted as valid code by the environment. In fact the nanopass version of LiveCodeLang accepts most CoffeeScript programs. Although not a purely functional language, CoffeeScript exposes the very same well-known JS functional constructs such as map, filter, reduce, (all three available in ECMAScript 5.1 standard [5]). So all of the followings are valid programs.

Using map:

```
// multiple concentric boxes
noFill
[1..4].map (i) -> box i
```

Another map example:

```
// equivalent to "rotate box line peg"
[<box>, <line>, <peg>].map (i) -> rotate i
```

Example of filter:

```
// draws random combinations of primitives
primitives = [<box>,<line>,<peg>,<ball>]
selected = primitives.filter (x) ->
    random > 0.5
selected.map (i) -> i()
```

Example of reduce:

```
// equivalent to
// "rotate(->scale(->box(->undefined))"
commands = [<box>,<scale>,<rotate>]
drawThis = commands.reduce (acc,x) -> -> x(acc)
drawThis()
```

Also reduceRight is supported:

```
// either a ball moving around a box
// or a box moving around a ball
pieces = [<box>, <move>, <ball>]
if random > 0.5
    drawThis = pieces.reduce (acc,x) -> -> x(acc)
else
    drawThis = pieces.reduceRight (acc,x) -> -> x(acc)
drawThis()
```

Combining filter and reduce:

```
// draw a cube with a random mix of transformations
transforms = [<rotate>, <scale>, <fill blue>]
randomTran = transforms.filter (x) -> random > 0.5
drawThis = [<box>].concat randomTran
drawThisFunction = drawThis.reduce (acc,x) -> -> x(acc)
drawThisFunction()
```

## 14. LiveCodeLab and LiveCodeLang - possible new directions

There is no formal roadmap for LiveCodeLab, and project participants are welcome to work on any idea, but here are a number of possible new directions under discussion:

- Continuous refinement of API and language.
- Static and run-time analysis-led Autocoder.
- Code collaboration.
- Code sharing.
- Interaction with external hw. (beyond currently supported MIDI interface for setting bpm).
- Automatic camelCase-ing of keywords.
- Symbols swapped-in while editing, e.g. on-the-fly replacement of '`->`' and '`PI`' with ➔ and *π.*

## 15. Acknowledgments

The following people also contributed to LiveCodeLab:

- Thomas van den Berg, https://github.com/noio: changes to event system, support for MIDI bpm input, tying of several functions to bpm value.
- Darren Mothersele, https://github.com/darrenmothersele: "shm" and "wave" functions.
- Julien Dorra, https://github.com/juliendorra: French localisation (currently not in codebase).
- Matthew Lawrence, https://github.com/mattus: alternative Autocoder implementation (currently not in codebase).

## References


[1] Brodsky Jonathan. (Mar 23, 2008). Jsaxus, js opengl livecoding. Available: https://github.com/jonbro/jsaxus. Last accessed 14th May 2014.

[2] Brown, Amy and Wilson, Greg. (02 June 2011). The Architecture Of Open Source Applications, Chapter 3.4

[3] Della Casa, Davide and John, Guy. (2014). LiveCodeLab release as of ACM SIGPLAN 2014 paper submission. Available: https://github.com/davidedc/livecodelab/releases/tag/ACM-SIGPLAN-2014-paper. Last accessed 15th May 2014.

[4] "Document Object Model (DOM)." *W3C Document Object Model*. N.p., n.d. Web. 23 June 2014.

[5] Ecma-international. (2011). Standard ECMA-262 5.1 Edition. Available: http://www.ecma-international.org/ecma-262/5.1/. Last accessed 15th May 2014.

[6] Griffiths, D. (2010). Fluxus. Available: http://www.pawfal.org/fluxus/. Last accessed 15th May 2014.

[7] Griffiths, D. 2007. “Game Pad Live Coding Performance.” In J. Birringer, T. Dumke and K. Nicolai, eds. Die Welt als virtuelles Environment. Dresden: TMA Helleraue.

[8] Ivanoff and Jimenez. (2006). Flaxus toplap flash based aplication. Available: http://www.i2off.org/flaxus/. Last accessed 15th May 2014.

[9] Kamermans, Rathbone, Macrae. (2014). Processing.js, parser source. Available: https://github.com/processing-js/processing-js/blob/4e2a5e28daa91128747b910b9c99441be2d315b3/src/Parser/Parser.js#L270. Last accessed 15th May 2014.

[10] Martin Fowler. 2010. Domain Specific Languages (1st ed.). Addison-Wesley Professional. Chapter 38.

[11] McLean, A., Griffiths, D., Collins, N & Wiggins, G. "Visualisation of Live Code", Proceedings of Electronic Visualisation and the Arts Conference, London. 2010.

[12] Phillips, Ian. "Online Lingo Resources." *LingoOnlineMaster.pdf* (1998): n. pag. *Advisory Group on Computer Graphics*. Web.

[13] "React | Why React?" *React | Why React?* N.p., n.d. Web. 23 June 2014.

[14] Reas, C. and Fry, F. 2004. Processing.org: programming for artists and designers. In ACM SIGGRAPH 2004 Web graphics (SIGGRAPH '04)

[15] Sarkar, Dipanwita, Waddell, Oscar, & Dybvig, Kent R. (2005). EDUCATIONAL PEARL: A Nanopass framework for compiler education. Journal of Functional Programming, 15(5), 653–667.

[16] Schwarz (2012). Regular Expressions and The Limits of Regular Languages. Available: http://www.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/15/Small15.pdf. Last accessed 15th May 2014.

[17] Tanimoto. A Perspective on the Evolution of Live Programming. Workshop on Live Programming (LIVE), 2013.

[18] TOPLAP collective. (31 July 2005). TOPLAP Manifesto. Available: http://toplap.org/wiki/ManifestoDraft. Last accessed 14th May 2014.

[19] Ward, Rohrhuber, Olofsson, et al. (2004). Live Algorithm Programming and a Temporary Organisation for its Promotion

[20] Zmölnig, J., Eckel, G., "Live Coding - An Overview", Proceedings of the International Computer Music Conference, Copenhagen, DK, August 2007.