Chucklib-livecode: A live-coding dialect for SuperCollider

H. James Harkins

February 10, 2017

Contents

1 Introduction

Does the world need another live-coding language for music? Perhaps the world didn't, but I did.[1]

I grew interested in live-coding after many years of work in the SuperCollider audio programming language.[2] I depend particularly on a workflow of my design, *chucklib*,[3] to manage resources and signal routing for musical processes. (It was inspired by the eponymous operator of the ChucK programming language, =>, though it does not try to replicate ChucK's most significant innovations.[4]) *Chucklib* facilitates complex behaviors by delegating some of the

[1] DRAFT version, not to be cited or reproduced. For now, I am reserving copyright. Very likely the final version will be Creative Commons.

[2] McCartney, James. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal*, Winter 2002, Vol. 26, No. 4, Pages: 61-68.

[3] Harkins, H. James. "Composition for Live Performance with dewdrop_lib and chucklib." *The SuperCollider Book*. Eds. Scott Wilson, David Cottle and Nick Collins. Cambridge, MA: MIT Press, 2011.

[4] Wang, G.; Cook, P. (2003). "ChucK: A concurrent, on-the-fly audio programming language" (PDF). Proceedings of the International Computer Music Conference.

details to self-contained "process" objects, which wrap SuperCollider patterns with other resources, freeing the user from managing resources directly: the user simply instantiates a *chucklib* process and it takes care of the practical necessities automatically. Once properly tested, *chucklib* processes reduce the chance of onstage failures and save considerable time while designing processes.

Adapting *chucklib* to an existing live-coding language, such as *TidalCycles*,[5] may have been possible, but it would also have changed the nature of a process object from an active agent to a passive recipient of messages. At the same time, I was intrigued by the metrical representation of musical material in Thor Magnusson's *ixi lang*,[6] and felt that its principle would be a viable way to compose material interactively. So, I arrived at a compact notation roughly inspired by *ixi lang*, and a *chucklib* process template that performs the notated material. The entire system is named *chucklib-livecode*, or *cll* for short.

It is beyond the scope of this paper to cover all aspects of *cll*. I will focus here on the crucial user-facing elements: the *chucklib* process to play musical materials, the pattern string language to write musical materials, and generators for algorithmic composition. Interested readers may consult *cll*'s PDF manual for details on components not discussed in this article.[7]

2 Usage example

Chucklib's guiding philosophy is to separate the definitions of musical behaviors from their usage in performance. When the definitions exist in separate files, processes and note-playing instruments may be arbitrarily complex, without cluttering the performance interface with background details. Code used in performance should be compact and simple. Listings 1 and 2 follow this principle.

chucklib process definitions organize into three main object types: "process prototypes" (`PR`), which define behavior; "bound processes" (`BP`), the players, which "bind" the prototype to specific data for performance; and "factories" (`Fact`), which automate the construction of `BP`s from `PR`s. One `PR` can spawn many `BP` players from the same definition, allowing *cll* to implement the common data processing for the user's pattern strings in a master prototype, `PR(\abstractLiveCode)`. All *cll* processes are `BP` instances of this prototype, customized for each musical behavior by passing a `Dictionary` into the `chuck` operation.[8]

[5] McLean, A. (2014). "Making programming languages to dance to: Live coding with Tidal." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*.

[6] Magnusson, Thor. "ixi lang: A SuperCollider Parasite for Live Coding." `http://www.ixi-software.net/thor/ixilang.pdf`, accessed December 10, 2016.

[7] Harkins, H. James. *Chucklib-Livecode Manual*. `https://github.com/jamshark70/ddwChucklib-livecode/raw/master/cl-manual.pdf`. Accessed February 4, 2017.

[8] The examples in this paper use a shorter syntax for Dictionaries: `(name: value, ...)`. Technically, this produces an `Event`, which is a subtype of Dictionary. In the context of collections of

```
s.boot;
TempoClock.tempo = 2;

(
SynthDef(\buf1, { |out, bufnum, start, pan, amp = 0.1, time = 1|
   var sig = PlayBuf.ar(1, bufnum, startPos: start),
   startTime = start / BufSampleRate.ir(bufnum),
   eg = EnvGen.kr(
      Env.linen(0.02,
         min(time, BufDur.ir(bufnum) - startTime - 0.04), 0.02),
      doneAction: 2
   );
   Out.ar(out, Pan2.ar(sig, pan, amp * eg));
}).add;
);

(
(make: { |name|
   BP(name).free;
   PR(\abstractLiveCode).chuck(BP(name), nil, (
      // BEGIN customizing the live-coding process prototype
      userprep: {
         ~buf = Buffer.read(
            s, Platform.resourceDir +/+ "sounds/a11wlk01.wav"
         );
         ~defaults[\bufnum] = ~buf;
      },
      userfree: {
         ~buf.free;
      },
      defaultParm: \clip,
      parmMap: (
         clip: (
            $^: [86271, 24604],  // start, numframes
            $-: [140463, 15105],
            $.: [124891, 11990],
            $_: [158215, 13237],
            alias: [\start, \frames]
         ),
         pan: (
            $<: -0.9, $>: 0.9,
            $(: -0.4, $): 0.4,
            $-: 0
         )
      ),
      defaults: (instrument: \buf1, amp: 0.1),
      postDefaults: Plazy {
         Pbind(\time, Pkey(\frames) / ~buf.sampleRate)
      }
      // END customizing
   ));
}, type: \bp) => Fact(\cl);
);
```

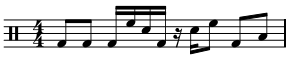
Listing 1: Factory definition of a playable live-coding process.

```
// Load the cll framework, and the process definition
\loadCl.eval;
(thisProcess.nowExecutingPath +/+ "demo-init.scd").load;

// Create the player, provide initial material, and play
/make(cl);
/cl = "....";
/cl+;

/cl = ".|.-|.|.";

/cl = ".|.-|.-  |.";

/cl = ".|.-|.-  |._";

/cl = "\ins(".|.-|.-  |._", "^", 2, 0.25)";

/cl = "\fork(".|.-|.-  |._", "|\ins(, "^", 2, 0.25)||")";

/cl = "\fork(".|.-|.-  |._", "|\ins(, "^", 2, 0.25)::\shift(, ".", 1, 0.25)||")";

/cl = "\fork(".|.-|.-  |._", "\ins(, ".", 1, 0.25)|\ins(, "^", 2, 0.25)::\shift(,
     ".", 1, 0.25)||")";

/cl..pan = "<><><><>";

/cl-;

/cl(free);
```

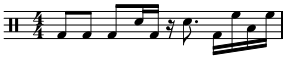
Listing 2: Development of a bar of rhythm in performance.

Figure 1: A few sample results from the longest generator string above.

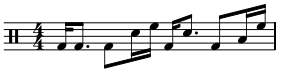
1. ". . |.^-.| -^ |. _ "



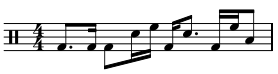
2. ". . |. -.| - |.^_^"



3. ".. |. -^|.- |. _^"



4. ". .|. -^|.- |.^_ "



Listing 1 is a short *chucklib* object definition file, containing a `SynthDef` defining the signal processing for a single note, and a *chucklib* "factory" (`Fact`) that manufactures a *cll* "process" to be played later. In the example process, each event chooses one of four "clips" drawn from the canonical SuperCollider example audio file. The process definition illustrates some key *cll* features:

- Parameter declaration: Performers provide new musical material to *cll* by issuing "set pattern" statements toward synthesis parameters defined in each process's "parameter map." In the example, `parmMap` associates the parameter names `clip` and `pan` to a number of single-character identifiers, representing the data values that will be sent to synthesis nodes. A normal parameter definition is a simple list of characters and values (`pan`). Parameters can also bundle several values together, as in `clip`: the `alias` means that the two array items will translate into entries for `start` and `frames`, respectively, in the resulting events.

- Default parameter: The primary, or "default," parameter receives new pattern strings when no other parameter name is given. It also controls the rhythm that the process will play. Its name is assigned to `defaultParm`.

- Defaults and event data flow: For complex sound designs, it is impractical to write pattern strings for all parameters onstage. In a *cll* process, `defaults` holds initial values for any relevant synthesis parameters. Values from the user's interactively-written pattern strings override these defaults. Subsequently, `postDefaults` can add new, non-interactive data, and massage the pattern-string data into the form needed by the `SynthDef`. The example's `postDefaults` demonstrates the latter scenario: the `clip` parameter provides the clip duration as a number of sample frames, but the `SynthDef` expects a playing time in seconds. `postDefaults` reads the number of frames (`Pkey(\frames)`) and converts to seconds by dividing by the buffer's sample rate.[9]

- Automated resource management: A pair of functions, `userprep` and `userfree`, prepare and release resources that the process needs. The example plays audio from a recorded sound file, so the process needs to load the audio into a `Buffer`.[10] Then, when the process is no longer needed, it is responsible for releasing the `Buffer` memory as well. Because the process uses these hooks, performance code can simply "make" the `cl` player, with no need to manage the buffer directly.

named data values, Events and Dictionaries are functionally interchangeable.

[9]This specific case can be handled without `postDefaults`, by writing the clip duration into the parameter map in seconds, or by writing the `SynthDef` to accept the number of sample frames to play back. Other cases do require post-processing within the process. Also note `Plazy`, which ensures that the correct environment-variable scope is in force to access `~buf`.

[10]The buffer reference should be a constant value in all events, making it suitable for `defaults`. But the buffer is not ready while the factory is processing. So, `userprep` waits until the buffer number is available, and only then puts it into `defaults`.

Listing 2 demonstrates a short performance. First, the user must load the *cll* framework and the process definition(s). Then, she can make a playing instance of the process, create a simple musical phrase to start with (in this case, four quarter notes), and play the process on the next barline. Musical development proceeds by editing the pattern string to insert notes. (Observe the use of vertical-pipe "`|`" dividers to keep the beats clear.) Generator expressions randomize the content. The final variation hints at the kind of interaction among generators that is possible. It represents the algorithm:

1. Start with "`.|.-|.-  |._`".
2. `\fork` divides the bar into two regions for further processing:
   (a) Insert one dot into any empty 16th note slot in the first beat (`\ins`).
   (b) In the remaining three beats, insert two "`^`" into empty 16th-note slots, and shift one dot to be a 16th-note earlier or later.

Some possible results of this algorithm are shown in Figure 1. Note the elements that are common to all the variations (e.g. the positions of the - and _ characters) and the elements that vary.

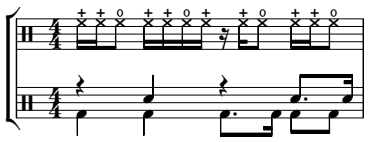
3 Pattern string language

The centerpiece of *cll* is the compact "pattern string" syntax to write "phrases" consisting of musical items and rhythms. (*cll* processes can store as many phrases as the user desires, and stream them out in an order of the user's choosing. Otherwise, *cll* would be limited to single-bar loops.) Like its predecessor, *ixi lang*, it divides durations of time among horizontally-spaced ASCII characters. Spaces are placeholders, positioning visible characters (musical events) at the desired time points. In *ixi lang*, all characters represent the same rhythmic value. *cll* subdivides the phrase duration into equal "beats" using vertical pipes, "`|`"; each "beat" is further subdivided equally by the characters and placeholders within it, for a more efficient representation of sparser beats. (Empty beats may even appear as a pair of consecutive vertical pipes.) As in *ixi lang*, phrases are assumed to be one measure long by default; the user may override this duration.

For comparison, Figure 2 shows a typical house-music drum pattern in Western notation, translated to *ixi lang* and *cll*. The *ixi lang* version resembles a musical score in that moments in time are vertically aligned. *cll* sacrifices this visual element in exchange for less redundancy and clear division of a bar into beats. For example, in the `/kik` pattern, beat 3 divides into four characters representing 16th-notes; the two characters in beat 4 are eighth-notes.[11] This reflects a philosophical difference: *ixi lang* is a display interface as much as a

[11] Note also that this strategy natively supports triplets, quintuplets and so on. Units of time are divided equally; nothing in *cll* enforces duple divisions.

Figure 2: A common house-music beat, with equivalents in ixi lang *and* cll.



```
/* ixi lang:
hh        -> |..- ..-. .- ..- |
snr       -> |    -        -  .|
kik       -> |o   o   o  _o _ |
*/

// cll
/hh = "..- |..-.| .- |..- ";
/snr = "|-||-  .";
/kik = "o|o|o  _|o_";
```

language, while *cll* statements are, first of all, instructions to produce musical phrases.

The general principle of one character per item has two exceptions: pitched parameters and generators. If a parameter is declared as `(isPitch: true)`, the list of characters and values is optional, and Arabic digits represent scale degrees. Additional characters appended to the digit modify the note by octave displacement, accidentals and articulation style. For instance, `5,->` means the fifth scale degree, down one octave, lowered by a half step and accented.

Generators extend pattern strings beyond notated deterministic materials into algorithmic manipulation of note data (Section 4). A generator expression consists of a backslash-escaped name followed by arguments in parentheses: for instance, `\ins("", "*", 3, 0.5)` inserts three `*` items at random eighth-note time points within the time span that the generator covers. Generators may be nested or chained, and can grow to be lengthy.

Internally, all items in a pattern string have two timing properties: `time` for the onset after the phrase's beginning, and `dur` for the length of the item's span. For normal single items, `dur` is usually irrelevant. Generators depend on both properties to know the portion of the phrase's time span over which to act. For the purpose of dividing the phrase duration, or a pipe-delimited subdivision, into onset time points, pitched note strings and entire generator expressions—no matter how long—are considered to be indivisible "items." That is, the pattern string `"1~ 4'~5~"` contains eight characters but represents four items: `1~`, spacer, `4'~` and `5~`. Thus the bar is divided into four units, and the three notes (excluding the spacer) appear on beats 1, 3 and 4 respectively.

4 Generators

Generators are string-rewriting operators that manipulate the contents of pattern strings algorithmically. Internally, *cll* translates a pattern string into an

```
// A. Insert 5 notes randomly in the bar, at 8th-note intervals.
/cl = "\ins("", ".", 5, 0.5)";

// B. Insert 3 quiet notes randomly, around 2 fixed-position louder notes.
/cl = "\ins("^|| -|", ".", 3, 0.5)";

// C. Insert 5 notes randomly, with a 3-note sequence.
// Nesting syntax:
/cl = "\seq(\ins("", "*", 5, 0.5), "^.-")";

// D. Chaining syntax:
/cl = "\ins("", "*", 5, 0.5)::\seq(, "^.-")";

// Evaluation of D:
"  |  |  |  "  // initial state
" *|* |**| *"  // after \ins
" ^|. |-^| ."  // after \seq
```

Listing 3: Basic generator usage.

array of events or generator objects; the events contain an identifier for the note data and an onset time. Every time a phrase begins, any generators within it are resolved by inserting, deleting or modifying events in its source pattern string, eventually leaving an array of events where all generators have been processed and removed.

A key reason for generators to operate by string rewriting is that the source string already places all of its items at a specific time points, favoring a strategy that preserves these pre-existing time points as much as possible. The source string may be empty (Listing 3, example A); if it provides initial items (example B), they become structural time points for generators to ornament. The `\ins` generator illustrated here identifies empty spaces within the source string, at beat intervals given by the quantization parameter (`0.5` here), and inserts the given number of new items, drawing them randomly from a "pool" (`"."`). Other generators can shift items earlier or later within the bar, rotate the rhythm within the bar, or replace "wildcards" with other items by deterministic or randomized sequencing strategies.

Individual generators should implement fairly simple rewriting behavior; as an important early document outlining live-coding aesthetics prescribes, "Insight into algorithms" is preferable to the "obscurantism" of compact notation that hides semantics.[12] If a single generator does too much work by itself, its algorithm is hidden from the audience's sight. Recombining simple processors, by "nesting" or "chaining" them, reveals the connections between them.

When one generator appears as the first input to another, then they are "nested": the first generator's result becomes the second one's source sequence. Listing 3, example C, nests an "insert" generator (`\ins`) inside a "value sequence" generator (`\seq`). `\ins` evaluates first, adding 5 $*$ wildcards into empty 8th-note spaces. This new event list is passed into `\seq`, which replaces the wild-

[12]The (Temporary|Transnational|Terrestrial|Transdimensional) Organisation for the (Promotion|Proliferation|Permanence|Purity) of Live (Algorithm|Audio|Art|Artistic) Programming. "ManifestoDraft." `https://toplap.org/wiki/ManifestoDraft`. Accessed January 31, 2017.

```
// A. Harmony changes during the bar (assuming a pitched parameter).
/hrm = "\ins("", "*", 12, 0.25)::\fork(, "\rand(, "13")|| \rand(, "24")|")";

// B1. Forking to "protect" part of the bar from an inner generator.
/cl = "\fork("^", "|\ins(, ".", 4, 0.5)||")";

// B2. Written without fork, as this is trivially simple.
/cl = "^|\ins("", ".", 4, 0.5)||";

// C. Non-trivial example:
// Protect first beat, but chain a generator to the whole bar.
/cl = "\fork("*", "|\ins(, "*", 4, 0.5)||")::\seq(, "^.-")";
```

Listing 4: The \fork generator.

cards, one by one, by the given three items in order.[13] Any rhythm generator may combine with any content generator, representing a general paradigm: first, generate rhythm using content-neutral wildcards, and then fill in meaningful musical content.

Nested generators may become unwieldy, however, as the performer adds additional layers; the arguments for the outer generators are separated from the generator name by all of the code for the inner generators. Note, in example C, that `"^.-"` belongs to `\seq`, but fully 25 characters intervene between the argument string and the generator name that gives it purpose. To address this problem, *cll* introduces a chaining operator, `::`, which appears between two generators. Examples C and D represent the same behavior, but in D, `\seq` and its argument are practically adjacent. When chaining, the code structure is linear rather than hierarchical, thus simpler to read and edit.

Of particular interest is a "filter generator," `\fork`, which applies different generators to different time spans within the same source string (which itself may come from other generators). `\fork`'s second input is itself a pattern string containing generators, each of which applies only to its own time span. For example (Listing 4), one could fill `\fork`'s source string with wildcards, and apply different harmonic content to different parts of the bar (example A). The `"13"` generator applies to the first 2.5 beats, and `"24"` to the remaining 1.5, resulting in a "chord change" on the downbeat and on beat 3.5.

`\fork` also solves another common problem. Looking back to Listing 3 B, the loud item `^` represents a strong accent on the downbeat. `\ins`, however, is free to place a note within the first beat, weakening the accent. A user can "protect" the first beat from insertion by forking (Listing 4, example B1). This trivial example could be written without `\fork` (example B2); however, `\fork` is necessary in example C. It is not possible to chain the `\seq` onto the top-level string; but, using `\ins` at the top level fails to protect the first beat from insertion. Example C's use of `\fork` handles all the requirements.

Nested double-quotes are a novel syntax feature. Left-to-right, they scan

[13] The `\seq` generator will place five values in each bar, from a three-item sequence, leaving one item left over. Generators maintain their state from one phrase to the next. The first bar will use items `^.-^.`; the second, `-^.-^` and so on. `\seq` supports a "reset" flag, but this is disabled by default because the quasi-isorhythmic behavior is more interesting.

```
// Code-document "part" organization
/hh.a0 = ".-|.-|.  -|.-";
/hh.a1 = ".- .|.-|. .-| .- ";

/snr.a0 = " - -";
/snr.a1 = "|-| .  |-";

/kik.a0 = "oooo";
/kik.a1 = "o|o|o  _|o";

// Editor window "score" organization
/** a0 **/
/hh.a0 = ".-|.-|.  -|.-";
/snr.a0 = " - -";
/kik.a0 = "oooo";

/** a1 **/
/hh.a1 = ".- .|.-|. .-| .- ";
/snr.a1 = "|-| .  |-";
/kik.a1 = "o|o|o  _|o";
```

Listing 5: cll *"part"-style and "score"-style organization.*

unambiguously: the first double-quote inside parentheses is always an open-quote.

5 Additional features

The preceding sections cover the main features of *cll*: `PR(\abstractLiveCode)`, pattern strings and generators. *cll* implements additional commands, and two user interfaces, to support them.

5.1 More commands

cll implements statement types beyond the "set pattern" statement discussed previously. There are commands to create, start and stop processes, copy phrase data to a different phrase name (to produce musical variations while keeping older versions for later reuse), call predefined `Func` object, pass arbitrary code to *chucklib* objects, and retrieve pattern strings from a process and insert them into the current code document for further editing. These are simple but valuable conveniences. For example, normal SuperCollider code to use a `Fact` factory object is `Fact(\factoryName) => BP(\bpName)`. The "make" statement reduces this to `/make(factoryName:bpName)`.

5.2 Code editor interface

Normal SuperCollider code documents can be unwieldy in performance, for two reasons: navigation features in the SuperCollider Integrated Development Environment (IDE) are too basic for the time pressures of live performance, and

the IDE's auto-completion features conflict with *cll* syntax (particularly when using generators, because of the nested double-quote delimiters).

Optionally, users may work within a graphical-interface (GUI) window, organizing processes' phrase data into multiple panels. Code documents tend to read more like printed parts for ensemble music: in Listing 5, all of the hi-hat material clusters together in one place, as does the snare and kick drum material. The GUI instead, displays two panels: `a0` and `a1`; just as a printed score aligns all the material for measure 1 together, the GUI shows all parts' `a0` material together, and so on. The `Esc` key allows the user to jump to a panel with just a few keystrokes.

Traditional navigation jumps from word to word by Ctrl-left or Ctrl-right arrow keys. The high density of punctuation marks in pattern strings makes it clumsy to move through them in this way. In the editor GUI, users can navigate more rapidly through pattern strings using a syntax-aware navigation mode, which parses a pattern string into a syntax tree and uses arrow keys to move through the tree elements (where normal arrow keys move through characters, words or lines). For instance, if syntax navigation is currently highlighting a generator argument, the left and right arrow keys will move to the previous or next argument (no matter how long each argument is).

5.3 Mobile control interface

Continuous parameter control is inconvenient in code. *cll* also provides a GUI window mirroring the "Mix 16" layout from the TouchOSC mobile app.[14] Users can "chuck" processes or individual parameters into the fader slots; toggle buttons start and stop processes. This makes it easier to control the mix or produce crescendo/decrescendo effects interactively.

6 Conclusions

cll is more of a shift in orientation—toward notation embellished by algorithmic manipulation rather than pure code structure—than a radical departure from existing approaches. It is still a text interface, and as such, improvising in it shares with other live-coding interfaces shares the quality of being more cognitive than neuromuscular, more analytical than reactive.

cll is conceptually oriented toward metrical position. This is a benefit—the text notation itself clarifies the rhythm—but also a challenge, in that the performer must mentally identify the metrical position of new notes before being able to write them. In the heat of performance, with time moving steadily forward, it can be difficult to hold onto a note's time point long enough to understand where to put it in the code. In practice, it flows best as an experimental

[14] Hexler Limited. *TouchOSC*. `http://hexler.net/software/touchosc`. Accessed February 8, 2017.

approach of "discovering" materials: place a note *somewhere*, and refine its position on subsequent iterations. Recalling Thelonious Monk, then, successful improvisation is a matter of making the right mistakes.

A large part of *cll*'s value to me as a composer and performer is its flexibility in sound design. I depend heavily on *chucklib*'s PR and BP design to work at two levels of abstraction: a detailed implementation level for sound design, and a higher "summary" level for composition. *cll* integrates into this system, and this is important to me.

Flexibility, however, is in itself a potential trap. If every process has its own intricate and radically different parameters and character identifiers, it is almost impossible to remember every process's coding interface. Each becomes, effectively, a unique musical instrument. In that case, it is not sufficient to learn to improvise in *cll*, but rather to learn to improvise on twenty different *cll* instruments. It takes discipline to standardize the identifiers across multiple processes and keep the parameter interfaces as streamlined as possible, reducing sonic complexity to the minimum number of options. It is this tension between open-ended design and restrained interfaces that characterizes the most fertile territory for live-coding with *cll*.