Chucklib-Livecode Manual

H. James Harkins

May 1, 2021

Contents

1 Introduction

1.1 Overview

Chucklib-livecode (*cll* for short) is a system of extremely compact commands extending the SuperCollider programming language. The commands manipulate musical processes in real time to facilitate live-coding performances. "Processes" in this sense refers to my *chucklib* quark, introduced in *The SuperCollider Book*.[1]

I began implementing *cll* in August 2014, and it reached a stage where I could begin performing with it in March 2015. The public extensions are hosted on github.[2]

cll consists of two main parts:

1. A chucklib *process prototype* (PR) that implements the methods that the musical processes need, in order to receive information from live-coding statements.

[1] Harkins, H. James. (2011). "Composition for Live Performance with dewdrop_lib and chucklib." In Wilson, S., Cottle, D., Collins N. [eds.] *The SuperCollider Book*. Cambridge, Mass.: MIT Press. pp. 589–612.

[2] `http://github.com/jamshark70/chucklib-livecode`

2. A *preprocessor* installed into the SuperCollider interpreter. The preprocessor translates the *cll* command syntax into standard SuperCollider code.

This document begins with a tutorial of usage examples. This is enough to begin making music with it. To add your own sounds, there are two ways. For pitched note-players, create a Voicer factory and use one of the standard pitch processes (see Section 3.6). Otherwise, you can create all-new live-coding processes by cloning `PR(\abstractLiveCode)` (see Section 4).

1.2 Acknowledgments

Thanks are due to:

- James McCartney and all the other developers who contributed to SuperCollider over the years. Without SuperCollider, none of this would exist.
- Alex McLean, for his work on the *Tidal* live-coding language for music.[3] *Tidal* demonstration videos online were the first to capture my imagination about live coding, leading me in this direction.
- Thor Magnusson, whose *ixilang* system[4] provided some of the inspiration for *cll* syntax.
- Canton Becker of `http://sampleswap.org` for granting permission to redistribute some audio files from sampleswap's online collection. These are used in the `ddwLivecodeInstruments` quark, referenced in the Tutorial (Section 3).

2 Installation

cll requires SuperCollider 3.7 or later, and recommends v3.9+. (It is released using the Quarks v2 system, which is not supported prior to SC 3.7.)

2019/01/26: As before, it is recommended to update all of the `ddw*` quarks.

2.1 Installation with *git*

If you have installed the *git* version-control system on your machine, SuperCollider can automatically download and install Quark extensions. Simply evaluate `Quarks.install("ddwChucklib-livecode")`. If there are no error messages, recompile the class library and you should be ready to proceed.

Optionally, also evaluate `Quarks.install("ddwLivecodeInstruments")` to install a pack of ready-to-play instruments. These are used in the Tutorial (3).

[3] McLean, Alex. "Making Programming Languages to Dance to: Live Coding with Tidal." Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design, September 6, 2014, Gothenburg, Sweden, pp. 63–70.

[4] `http://www.ixi-audio.net/ixilang/`, accessed October 4, 2016.

2.2 Installation without *git*

If you haven't installed *git* or don't want to, you can download the required Quark directories manually. In each of these web pages, look for the green "Clone or Download" menu. From here, you can download a ZIP.

- ddwChucklib: `https://github.com/jamshark70/ddwChucklib`
- ddwPrototype: `https://github.com/jamshark70/ddwPrototype`
- ddwCommon: `https://github.com/jamshark70/ddwCommon`
- ddwGUIEnhancements: `https://github.com/jamshark70/ddwGUIEnhancements`
- ddwMixerChannel: `https://github.com/jamshark70/ddwMixerChannel`
- ddwPatterns: `https://github.com/jamshark70/ddwPatterns`
- ddwTemperament: `https://github.com/jamshark70/ddwTemperament`
- ddwVoicer: `https://github.com/jamshark70/ddwVoicer`
- crucial-library: `https://github.com/crucialfelix/crucial-library@tags/4.1.5`
- [optional] ddwLiveCodeInstruments: `https://github.com/jamshark70/ddwLivecodeInstruments`

After downloading these quarks, do the following:

1. In the SC IDE, go to File → *Open user support directory*.
2. In this location, if there is no folder called `downloaded-quarks`, create this empty folder now.
3. Unpack all the ZIP files into the directory. After this, you should have:
    - `downloaded-quarks/ddwChucklib`
    - `downloaded-quarks/ddwPrototype`
    - … and so on. If there is a `-master` suffix on the directory names, please remove it.
4. In SC, run the statement `Quarks.install("ddwChucklib-livecode")`. If it doesn't find the quarks, try recompiling the class library and then run the statement again.
5. If all is successful, recompile the class library and proceed.

2.3 Running *cll* in a session

cll adds convenience functions to load the environment:

- `\loadCl.eval`: Load the *cll* preprocessor and a few helper functions.
- `\loadClExtras.eval`: Load extra user-interface components:mobile control with TouchOSC, and interactive code editor, and ddwLivecodeInstruments if you have installed the quark.
- `\loadAllCl.eval`: Load both of these at once.
- `\makeController.eval`: Open a graphical window with reusable mixing and parameter controls. (See Section 7.)
- `\makeCodeWindow.eval`: Open a code editor window with *cll*-specific features.
- `\cllGui.eval`: Open both the controller and code windows.

These are *not* executed by default at SC startup, because you may not want the preprocessor in every SC session. Once you load the environment, the preprocessor is active until the next time you recompile the class library.

3 Tutorial

First, if you didn't install the optional `ddwLivecodeInstruments` quark, please do so now. Without these, you will have to learn the mechanics of creating a live-coding process before playing any music. See section 2 for installation details.

`ddwLivecodeInstruments` provides a set of standard electronic drums (Section 3.2), and several synthesizers for pitched notes (Section 3.6).

I recommend working step-by-step, starting with the drums (because there are fewer variables and moving parts) before moving on to pitches. You will probably get more out of it by typing the code examples yourself, rather than copying/pasting.[5] I've tried to make it easy to get started, but bear in mind that this improvisational instrument has been in development since mid-2014. You shouldn't expect to understand it all in 15 minutes (just as you wouldn't expect to read a couple of tutorials about SuperCollider itself and "understand" it in depth). Take your time. Experiment. Start with the examples and change them.

If you encounter problems, you could post on the SuperCollider users mailing list[6] or the ddwChucklib-livecode issue tracker.[7] (Also note that this is the

[5] Copying from this PDF is likely to change the code formatting and possibly break the code. If you must copy/paste, use the file `cl-manual-examples.scd`.

[6] `https://www.birmingham.ac.uk/facilities/ea-studios/research/supercollider/mailinglist.aspx`

[7] `https://github.com/jamshark70/ddwChucklib-livecode/issues`

```
\loadAllCl.eval;

// optional
\cllGui.eval;
```

Listing 1: Launching chucklib-livecode.

```
\loadAllCl.eval;
TempoClock.tempo = 124/60;

/hh.(\hardhh);
/hhh = ".-.-.-.-";
/hhh+

/drum.(\tightsnr);
/tsn = " - -";
/tsn+

/drum.(\deepkick);
/dk = "o| o| _o |";
/dk+

// mixing board
/makeEmptyMixer8.();
/hhh => MCG(0);
/tsn => MCG(1);
/dk => MCG(2);

/hhh/tsn/dk-
```

Listing 2: A quick techno-ish drumset.

first version of the tutorial. Many things may be badly explained as yet. Don't hesitate to raise an issue if something is confusing.)

3.1 Starting a session

When you launch SuperCollider, *cll* is not active. (For normal usage, you want to use the standard parser as is.) To use *cll*, you need to load the environment. Optionally, you can open a graphical environment with some additional features (see Section 7).

3.2 Drums

We'll start with drums (Listing 2), because the notation is a little simpler.

Kicks and snares are created by the convenience function `/drum.(name)`; use `/hh.(name)` for hi-hats. Available names are:

- `/drum.(name)`
  - `\deepkick`: `BP(\dk)`
  - `\tightkick`: `BP(\tk)`
  - `\midkick`: `BP(\mk)`
  - `\tightsnr`: `BP(\tsn)`
  - `\fatsnr`: `BP(\fsn)`
  - `\pitchsnr`: `BP(\psn)`
  - `\snr80`: `BP(\s8)`
  - `\clap`: `BP(\clp)`
- `/hh.(name)`
  - `\thickhh`: `BP(\hh)`
  - `\thinhh`: `BP(\thh)`
  - `\hardhh`: `BP(\hhh)`
  - `\synthhh`: `BP(\shh)`

Note the pattern to use an instrument:

- Create it, using the `/drum.(\name)` for drums and `/hh.(\name)` for hi-hats. The result is a `BP` object—a Chucklib "bound process." You can access the process object globally by putting its name in parentheses: `BP(\dk)`, for instance. Many *cll* commands use only the name with a leading slash: `/dk`.
- Give it some music to play (by assigning it a pattern string). More about pattern strings below.
- Start it (`+`). If `+` is start, `-` is stop. You can start and stop several processes at once by listing them on the same line, each name beginning with its slash: Listing 2 stops all three processes using one command, `/hhh/tsn/dk-`. By default, the processes will start or stop on the next bar line (`BP(\name).beatsPerBar`). You can override this by putting a number of beats after the `+` or `-`: `/dk+8` for the next even bar line.

Mixing: `loadAllCl` creates a few `MixerChannel` objects: `~hwOut` (hardware output), `~master` (main output), `~rvbmc` (long-tail reverb), `~longrvbmc` (long-tail reverb), and `~shortrvbmc` (short-tail reverb).[8] If you want to adjust the mix,

[8]The distinction between `~hwOut` and `~master` is for multitrack recording of live sets, where you may wish to record the main, dry mix (`~master`) separately from reverbs. In general, processes and instruments should direct their signal to `~master`; reverbs and other separate channels (e.g., microphones) may connect directly to `~hwOut`.

first run `/makeEmptyMixer8.()`. After that, you can "chuck" mixers, processes or Voicers into mixing board slots: `~master => MCG(7)`, e.g., for the rightmost slot. Later, when you create playing objects, you can chuck them in as well. For instance, where Listing 2 creates a `tightsnr` player, you can do `/tsn => MCG(0)` and its mixer will appear.

3.3 Pattern strings

Cll uses single characters for notes, and spaces for timing placeholders.

- Kick drums: `o` = normal weight, `_` = ghost note
- Snare drums: `-` = normal weight, `.` = ghost note[9]
- Hi-hats: `-` = open hat, `.` = closed hat

By default, the unit of time is one bar (taken from the default `TempoClock`, whose default `beatsPerBar` is 4). The characters and placeholders divide this time span equally: `/hhh` has 8 characters, splitting the bar into 8th-notes, while `/tsn` has 4. You might think the spaces in `/tsn` are rests, but they aren't: they only specify the passage of time, here forcing the two snare drum strokes onto beats 2 and 4.

The kick drum pattern is slightly more complicated. The vertical pipes (`|`) are dividers. Dividers appear *between*, but not surrounding, the time spans; three dividers demarcate four divisions: `"1|2|3|4"`. Each subdivision is divided equally by the characters contained within. In `"o| o| _o |"`, the first beat is a quarter-note, the second divides into 8th-notes and the third into 16ths. With a little practice, you can read the rhythm directly from the ASCII notation.

- Exercise: Edit the given pattern strings to create more interesting rhythms. After every change, reevaluate the line. This is the basic process of improvising with *cll*.

3.4 Generators

Cll can also generate new materials algorithmically. (The tutorial can provide only a brief demonstration, not complete documentation. See Section 3.4 for more detail.)

Generators take a given pattern string as their initial input, enclosed in brackets, and modify it by inserting, deleting or replacing entries. (The initial pattern string can be empty, in which case it may be omitted.) A few basic functions are:

- `\ins("item pool", number, quant)`: Insert *number* new items, randomly chosen from the *item pool*, at rhythmic intervals given by *quant* (e.g. 0.25 = quarter beats = 16th-notes).

[9] The characters for kicks and snares are different, so that a kick and a snare could be combined into one process: `/drum.(#[tightkick, tightsnr]);`.

```
/hhh/tsn/dk+

// A
/tsn = "[ - -]::\ins(".", 2, 0.25)";

// B
/tsn = "[ - -]::\ins(".", 2, 0.5)";

// C
/tsn = "[ - -]::\ins(".", 2, 0.5)::\shift(".", 2, 0.25)";

// D (empty source, so, omitted)
/hhh = "\ins("-", 1, 0.5)::\ins(".", 7, 0.5)";

// E (one closed HH to fill start of bar)
/hhh = "[.]::\ins("-", 1, 0.5)::\ins(".", 6, 0.5)";

// F
/hhh = "[.]::\ins("-", 1, 0.5)::\ins(".", 6, 0.5)::\ins(".", 2,
     0.25)";

// G
/hhh = "\fork("|\ins("-", 1, 0.5)||x")::\ins(".", 7, 0.5)::\ins
    (".", 2, 0.25)";

/hhh/tsn/dk-
```

Listing 3: Generators for drums.

- `\shift("item pool", number, quant)`: Locate *number* of the items in *item pool*, and shift them earlier or later by the rhythmic value given by *quant*.
- I will expand this list later.

For example, the snare drum would benefit from some ghost notes, and it's more fun if they change from bar to bar. We could insert them into any open 16th-note (Listing 3, example A). But if you play this long enough, eventually you will hear some bars with too many 8th-notes. This sounds stilted. It would be better to force the ghost notes onto off-beat 16ths. An easy way to do that is to place the ghost notes onto 8th-notes (B), and then shift them (C). Note the `::` syntax. This creates a *generator chain*, where the result of the a generator (or source) feeds into the the next. (Because the chain provides the source string for `\shift()`, you don't need to write a source—but you still need the comma.)

For hi-hats, a musically sensible way to operate is to place one or more open hats, and then fill the remaining spaces with closed hats (D). Here too, eventually you will run into a musical problem: an open hat on the downbeat sounds awkward. Instead, you can place a closed hat explicitly in the first slot (E), "protecting" that space from insertion. (The example adjusts the number of closed hats to insert at the end, from 7 to 6. While formally correct, it isn't necessary in the performance. `\ins()` will add as many as it can, and not complain if it doesn't reach the requested number.) Finally, for some more spice, you can add a few 16th-notes.

Another way to "protect" part of the bar from a generator is to `\fork()` it. `\fork()` takes a source string, and another string placing generators in specific parts of the bar. In example G, the fork string places an `\ins()` on beat 2, and follows it with an `x` on beat 4. The `\ins()`, then, cannot operate before beat 2, and the `x` ends the `\ins()` generator's range of influence. So the open hat can be inserted in positions 2, 2.5, 3 and 3.5 (not including 4, which belongs to `x`). `x` is not a generator, so it does nothing in the context of `\fork()` except delimit time. After the `fork()`, the remaining generators operate on the whole bar, as before. This is an important technique to control the time over which generators may take effect.

It isn't very useful for drums, but a typical generator usage is to insert wildcards (usually `*` or `@`) to define the rhythm, and then replace them using a number of generators inspired by SuperCollider patterns: `\seq()`, `\rand()`, `\shuf()`, `\wrand()` and such. These are more useful when you have a larger number of possible items to choose from, such as when playing a pitched instrument.

HINT: If you have installed the ddwSnippets Quark, generator objects will automatically add their own templates into the snippets collection. The snippets can help you with the order of arguments: press the snippet hotkey (which you configure using `DDWSnippets.learn`), type a few letters of the generator name, and the template will be inserted into your document.

3.5 Sound effects

```
\loadAllCl.eval;  // If you haven't already done this
TempoClock.tempo = 124/60;

// The beat
BP(#[tk, clp, hhh, tsn]).free;  // Clean up first
/drum.(\tightkick); /drum.(\clap); /hh.(\hardhh);
/tk = "o|| o|";
/clp = "  -   ";
/hhh = "........";

/tk/clp/hhh+

/drum.(\tightsnr);
/tsn = "|||.";

/tsn+

/clp = "|-||  .   ";
/tsn = "|||.   .";

// make the effects
/make(bufBP:mch(set:\machine));
/mch = "|  -||";
/mch+

/mch = "|  -|  ,  ,|";

/tk/clp/hhh/tsn/mch-;
```

Listing 4: Adding sound effects to a simple beat.

The bufBP template provides some unusual percussion sounds, for extra color.

The sounds are organized into "sets," so that each resulting process has a simpler interface. To choose a sound set, use the parameter notation of the /make command: /make(bufBP:name(set:\setName)), substituting your desired process name for name and the name of the set for setName.

- machine: Servomotor clips (3)
- tink: Metallic (2)
- whistle: A sound reminiscent of a train whistle (3)
- wiggle: A, well, "wiggle" (for lack of a better word) (3)

In pattern strings, you can use the following characters (chosen as a visual approximation of pitch level and duration):

Pitch	Long	Short
High	^	!
Middle	-	.
Low	_	,

Note, however, that the tink set has only two sounds, omitting the high pitch level. With this set, ^ and ! are rests (silent).

Figure 4 is a simple example. You should play with the other sets and make your own rhythms.

3.6 Pitched notes

Pitched-note processes require a Voicer instrument and a BP process to play the notes.

Available Voicers:

- anapadVC: Analog-style pad.
- distbsVC: Distorted-sine bass.
- fmMelVC: Maps notes onto FM modulator ratio, playing harmonics. A bit strange.
- fmbassVC: FM bass.
- fmbrassVC: Brassy FM tone.
- fmclavVC: FM clav tone.
- fmpadVC: FM pad.
- klankVC: Detuned bell-like timbre. Not sustaining.

- `pbsVC`: Pulse-wave bass.
- `pulseLeadVC`: Pulse-wave synth lead.
- `staccVC`: Analog-style staccato notes.

Available BP processes:

- `melBP`: Monophonic melody player. Use for basses and leads.
- `chordBP`: Block-chord player.
- `arpegBP`: Chord arpeggiator.

3.6.1 Pitch: Overview

Using pitch in *cll* requires a little preparation.

1. Set the tempo. (You should do this in every performance, as already demonstrated in Listing 2.)
2. Set the *mode* by `/changeKey.(\name)`. *ddwLivecodeInstruments* defines a `Mode` object for all seven classical Western modes, for all 12 chromatic steps. Modes are identified by the root pitch (`c`, `cs` = c-sharp, `db` = d-flat, and so on) plus a modal identifier (Table 1). B-flat lydian is `Mode(\bblyd)`; F-sharp minor is `Mode(\fsmin)`.

Table 1: Modal identifiers.

Mode	ID
Major (Ionian)	`maj`
Dorian	`dor`
Phrygian	`phr`
Lydian	`lyd`
Mixolydian	`mixo`
Minor (Aeolian)	`min`
Locrian	`loc`

3. Create instruments and players. Internally, this is a three-step process: 1. Create the Voicer instrument. 2. Create the playing process. 3. Connect the instrument to the player. *Cll* provides a shortcut: If you write a single `/make` command that creates the instrument first (1), followed by the player process (2), then `/make` will automatically assign the instrument to the player (3, implicitly). Listing 5 demonstrates with `/make(pbsVC:pbs/melBP:bs(octave:3));`:
    - `/make`
        - `pbsVC:pbs`: Make the `pbsVC` instrument under the name `pbs`.

- `melBP:bs`: Make the melody player `melBP` under the name `bs`.
  - `* (octave:3)`: A parameter to apply to the new `bs` process.
- (Here, *cll* does `VC(\pbs) => BP(\bs)` for you, but you don't have to write anything for this!)

The general form is `/make(factoryID:instanceID(parameters...))` with additional factory/instance pairs separated by slashes.

3.6.2 Pattern string pitch specification

In pattern strings, the modal root is `1`.[10] Each note begins with a single digit, going up from there: 8 is an octave higher, 9 is the 9th, and 0 is the 10th (octave above the third).[11] This follows the arrangement of digits on the keyboard: the further right you go, the higher the pitch. You can also attach various modifiers:

- `'` or `,`: Up or down one octave. `''` is up two octaves, and so on.
- `+` or `-`: Up or down a semitone (like sharp or flat). **NOTE:** Flats are not completely working as of this writing.
- `.` or `_` or `~`: Staccato, legato or slurred articulation. A slur will slide into the next note, if the instrument supports it.
- `>`: Accent articulation. Always prevents a slur, and depending on the instrument, it might hit the note a little harder. `>` may be combined with other articulations.

The digit plus its modifiers becomes a single event—so, in Listing 5, beat 3 contains five characters but four events (`7~` is just one event). Timing is based on events, not characters.

Any event that does not begin with a digit—I often use `x`—is a rest, which cuts off the preceding note (in contrast to a space, which affects timing only).

3.6.3 Basses and leads

Basses and leads should use the `melBP` melody player: one written note produces one sounding note (Listing 5).

For basses in particular, get in the habit of assigning the octave. This is the normal `octave` event parameter from the SuperCollider pattern system. Assuming C as the modal root, octave 5 puts scale degree 0 at middle C. Octave 3 in the example pulls the bass two octaves lower. If you forget, you can correct it later by `/bs(setDefault(\octave, 4))` or whichever octave number you need.

Leads follow the same principles, except in a higher octave.

Exercise: Modify the given template to add more notes. Try the different articulation styles.

[10] In SuperCollider pattern terms, `1` translates into degree 0.

[11] Currently a diatonic scale (7 degrees) is assumed.

```
/changeKey.(\dmixo);
/make(pbsVC:pbs/melBP:bs(octave:3));

/bs = "1_| 1.|  7~4|x";

/bs+
/bs-
```

Listing 5: Bassline template.

```
/make(anapadVC:pad/chordBP:ch(chords:\one));
/ch = "87~05";
/ch+

VC(\pad).gui

MBM(0)[\two] => BP(\ch);

MBM(0)[\smallch] => BP(\ch);

/ch-
```

Listing 6: Chord-playing template.

3.6.4 Chords

`chordBP` (Listing 6) uses the same pitch notation, but to control the top note of a harmony. The harmonies come from chord templates stored in a MIDI buffer object (`MIDIRecBuf`). Currently, four are provided (later documentation: how to create your own chord templates).

- `one`: Single notes (so that the chord player can start as a melody, and grow into harmony).
- `two`: Two-note chords, in all intervals from a second to an octave.
- `smallch`: Three-note chords, not all standard triads.
- `bigch`: Six-note chords.

The chord templates will be adapted to the top note and the current chord root (later documentation: how to control the root).

Articulations (including slur!) are valid.

3.6.5 Arpeggiator

NOTE: The arpeggiator is a bit complex to use, and it isn't a high priority for the first round of documentation. I'm providing an example (Listing 7) to give

```
/make(fmclavVC:fmc/arpegBP:arp(chords:\bigch));

// These are indices, from the top down, into the current chord
    .
/arp = "1234";

/arp+

// Add some lower notes as a second layer.
// Accent articulates the start of the bar.
/arp = "[1>234]::\ins("456", 6, 0.25)";

// Extend the second layer higher.
/arp = "[1>234]::\ins("23456", 7, 0.25)";

// Use wildcards to substitute a sequential pattern.
/arp = "[1>234]::\ins("*", 7, 0.25)::\seq("65432")";

// Change the harmony's top note every bar.
/arp..top = "[*]::\seq("5'6'3'2'")";

// Skip: Play dyads instead of single notes.
/arp..skip = "2";

// Skip can also accent specific notes.
/arp..skip = "20  |20  |20  |20  ";

// same, but algorithmic
/arp..skip = "[2222]::\choke(0.25, "0")";

// Add a second process to change the chord root.
// After this, you should hear tonic, dominant
// and subdominant functions.
// No instrument -- this is for data only.
/make(melBP:root(bassID:\bass));
/root = "[*]::\seq("154")";
/root+

/arp/root-
```

Listing 7: Example of arpeggiator usage.

you some hints, and I want to expand the documentation later. For now, try it, and if you run into trouble, file an issue at `https://github.com/jamshark70/ddwChucklib-livecode/issues`.

The arpeggiator is a bit strange. It uses the same harmony-processing logic as `chordBP`, but the pitches in the pattern string are indices of chord notes, not the actual sounding pitches. 1 is the top note, 2 is the next lower, and so on proceeding down the chord to 7. 8, as a normal pitch, is an octave higher than 1; in the arpeggiator, it takes the top note of the chord and raises it by an octave (and all seven indices do octave displacement in the same way).

To make best use of this process, you need to assign alternate parameters: `top` for the top note of the harmony (which behaves like `chordBP`) and `skip` for extra thickness. The default `skip` is 0, meaning to play single notes. Try the other values (1–5). Note that the harmony will not change unless `top` changes, so it's a good idea to supply a slower-moving pattern for this parameter.

The double-dots are a syntax shortcut. Cll processes can store any number of phrases and switch between them. So far, we are using only the default phrase, `main`. The full form of the `skip` and `top` statements in the example is, in fact, `/arp.main.top = "..."`; omitting `main` leaves `/arp..top`. (You can't leave out one of the dots. If you do, `top` will be interpreted as a phrase name, and it won't behave the way you want.)

3.6.6 Pitched instrument parameters

Voicer instruments have two types of parameters: "global controls" and event parameters.

Global controls act like knobs on conventional synthesizers, by affecting all playing notes. These can be displayed automatically in a GUI window by running `VC(\name).gui`. Alternately, you could load Chucklib's performance GUI (`\cllGui.eval`); for details, see Section 7.

Event parameters should normally take care of themselves. If you need to override, use the form `/processName(setDefault(\parameter, value))`. It should be rare to need to do this, but it's useful, for instance, if you forget to set the octave of a bass process and it starts playing in the middle register. Just do `/process(setDefault(\octave, 3))` (or other value) and the next note will be lower.

(This is a tutorial, not reference documentation for all the instruments. Documentation to be expanded later.)

3.7 Phrases

The examples so far repeat a single bar's worth of content. Cll processes allow you to define multiple bars, and choose between them.

Each bar, or *phrase*, has a name. Attach it after the process name, with a dot: `/process.phrase = "content"`. Then, use a phrase selection pattern (Section 5.2.5) to choose the bars in succession. Listing 8 demonstrates with drum

```
TempoClock.tempo = 124/60;

/drum.(\tightkick); /drum.(\tightsnr); /hh.(\thinhh);

/tk = "oooo";
/tsn = " - -";
/thh = "[.]::\ins(\"-\", 1, 0.5)::\ins(\".\", 6, 0.5)";

/tk/tsn/thh+

/tk.fill = "o||   _|o __";

// mid-bar source string:
// in this position, it fills 3 eighth-notes
/tsn.fill = "|-| [ - ]::\ins(\".\", 4, 0.25)|";

/tk = (main*3.fill); /tsn = (main*3.fill);

/tk/tsn/thh-
```

Listing 8: Phrase selection for drum fills.

processes, playing the basic pattern for three bars and a fill pattern for the fourth.

Alternately, you can create multi-bar structures using a few convenience functions (Listing 9):

- `/setupbars.(\proc, n, \prefix)`: Creates *n* empty bars, named `\prefix0`, `\prefix1` and so on.
- `/setm.(\proc, n, \prefix)`: Tell the process to use a multi-bar phrase set for playback.
- `/bars.(\proc, n, \prefix)`: Do both at the same time.

IMPORTANT: Do not omit the `.` between the function name and the arguments. Cll distinguishes between a *function-call* shortcut `/name.(args)` and a *method-passthrough* shortcut `/proc(method(args))`, with the dot to tell the difference.

Note that `/bars.()` will tell the process to start playing a silent phrase structure. So, you should use it only when setting up a new process. If you're already playing material, it's better to `/setupbars.()` first, fill the bars with material, and then switch to the material using `/setm.()`.

An alternate syntax for `/setm.()` is `/proc = (prefix**n)`. This command also sets the process's `quant` to the same number of bars, so that the process will start and stop on the boundaries of the entire phrase set. Be careful when switching from a single-bar structure to multiple bars: you should hit `/setm.()` or the alternate syntax within the bar before the boundary.

```
// If the bass doesn't exist, first do this:
/make(pbsVC:pbs/melBP:bs(octave:3));

/bars.(\bs, 2, \a);

/bs.a0 = "1>|4~5~7 | 4~|3'~";
/bs.a1 = " 5>~|6| 4~| 3";

/setupbars.(\bs, 2, \b);

/bs.b0 = "9>.9.9 | 4'~|   3'|8~7~8~ ";
/bs.b1 = "  33.| 4.5~ | 431.|6.6. 6.";

// short form of /setm.(\bs, 2, \b)
/bs = (b**2);

/bs+
/bs-
```

Listing 9: Multi-bar bassline.

For convenience, `/setupbars.()` will try to insert a code template with the empty bars into the current SC-IDE document.

3.8 Errors

Cll syntax errors are reported in the form "ERROR: …" (with a brief explanation).

A common error is "ERROR: clPatternSet: BP('abc') does not exist," meaning that a cll command referred to a process that hasn't been created. Look for a misspelled name. (Cll is a translator, converting its own syntax into SC language code. Many of the translations depend on information within the object. If the objects don't exist, translation is impossible. So, it fails in the translation stage—but the translation happens in SC language code, so it must be reported as an execution error.)

If an error occurs within a process while it's playing, usually the bottom of the stack trace will refer to `awake` or `prStart`. Please report such errors to `https://github.com/jamshark70/ddwChucklib-livecode/issues`; as much as possible, cll should try to continue playing without stopping processes. If this does happen to you, it should be possible to recover by simply playing the process again. (Internally, the process will try to reset itself. If this fails for some reason, you can manually `stop` and `reset` before trying again to play it.)

4 Process prototype

Cll is not limited to the preset instruments. In theory, *cll* can play any `SynthDef` (or complex of synths!) by modifying a *process prototype*, `PR(\clAbstractLiveCode)`. This prototype is the interface between *cll* patterns and your sounding material. We've seen how pattern strings use single-character identifiers to denote rhythm. `PR(\clAbstractLiveCode)` translates these identifiers into real parameter values, and provides hooks to supply other information necessary for playing.

Before we get to the details, you should understand the data structure.

4.1 Data structure

cll organizes musical behavior, and musical content, hierarchically:

- Chucklib *processes* (BP) contain any number of *phrases*. Every process has its own variable scope (i.e., independent namespace). Activity in one process does not interfere with other processes.
- Each *phrase* contains multiple *parameters*. (The phrase itself is implemented as a `PbindProxy`, so that its contents can be changed at any time.)
- Each *parameter* is defined by a pattern string, parsed and rendered into SuperCollider pattern syntax by the *Set pattern* statement (Section 5.2).
- Parameter values are defined by the *parameter map* (`parmMap`).

cll processes create two phrases by default:

`main` The default phrase, which plays if the user hasn't specified a different phrase sequence. `main` is also the default phrase that *Set pattern* acts on—thus, a user can work with single-bar loops using only `main`, and never specify a phrase ID.

`rest` An empty phrase, which only occupies time.

4.2 PR(\abstractLiveCode)

To create a *cll* process, "chuck" `PR(\abstractLiveCode)` into a BP ("Bound Process"), with a parameter dictionary providing the details. Parameters to include in the dictionary:

`userprep` A function, called when the process is created. Use this function to create any resources that the process will require.

`userfree` A function, called when the process is destroyed. Clean up any resources allocated in `userprep`.

```
(
(
defaultName: \beep,
make: { |name|
   PR(\abstractLiveCode).chuck(BP(name), nil, (
      userprep: {
         ~buf = Buffer.read(
            s, Platform.resourceDir +/+ "sounds/a11wlk01.wav",
            4982, 10320
         );
         ~defaults[\bufnum] = ~buf;
         SynthDef(\buf1, { |out, bufnum, pan, amp, time = 1|
            var sig = PlayBuf.ar(1, bufnum),
            eg = EnvGen.kr(
               Env.linen(0.02,
                  min(time, BufDur.ir(bufnum) - 0.04), 0.02),
               doneAction: 2
            );
            Out.ar(out, Pan2.ar(sig, pan, amp * eg));
         }).add;
      },
      userfree: {
         ~buf.free;
      },
      defaultParm: \amp,
      parmMap: (
         amp: ($.: 0.1, $-: 0.4, $^: 0.8),
         pan: (
            $<: -0.9, $>: 0.9,
            $(: -0.4, $): 0.4,
            $-: 0
         )
      ),
      defaults: (instrument: \buf1),
      postDefaults: Pbind(
         \time, (Pkey(\dur) * 0.6 / Pfunc { ~clock.tempo }).
             clip(0.04, 0.2)
      )
   ));
}, type: \bp) => Fact(\beepBP);
)
```

Listing 10: Defining a simple cll process as a factory.

```
\loadCl.eval;  // or \loadAllCl.eval;
TempoClock.tempo = 2;

/make(beepBP);  // defaultName is 'beep' so you get BP(\beep)
/beep = "^|.. .| .- | .  ";  // "Set pattern"
/beep+;  // start it

/beep..pan = "<><><><>";  // change something

/make(beepBP:beep2);  // ':' overrides defaultName
/beep2 = " ..-| .^ |. ..| .";
/beep2+

/beep..pan = "<";
/beep2..pan = ">";

/beep/beep2-;

/beep(free); /beep2(free);
```

Listing 11: Using the cll process factory in a performance.

`defaultParm` The name of the default parameter affected by *Set pattern* statements (Section 5.2). The default parameter also controls rhythm.

`parmMap` A nested dictionary of parameters, their allowed values, and the characters that will identify these values in pattern strings.

`defaults` An `Event` or event pattern providing default values for the events that the process will play.

`postDefaults` (optional) An event pattern that can do further calculations on the parameter values.

> **Note:** *Chucklib* documentation says to place the initialization function into `prep`, and cleanup into `freeCleanup`. `PR(\abstractLiveCode)` uses these functions for its own initialization and cleanup, and calls `userprep` and `userfree` from there. Do not override `prep` and `freeCleanup`, or your process will not work properly.

This dictionary is not limited to these items. You may add any other data and functions that you need, to define complex behavior in terms of simpler functions and patterns.

In Listing 10, `userprep` loads a buffer and `userfree` releases it. By default, *Set pattern* will operate on `amp`, and `parmMap` defines three values for it (soft, medium and loud). `parmMap` also provides some panning options. The `defaults`

dictionary specifies the SynthDef to use (it may provide other synth defaults as well, not needed in this example), and `postDefaults` calculates the sounding duration of each note based on rhythm.

Note the line `~defaults[\bufnum] = ~buf`: You may add values into `defaults` as part of `userprep`. That's necessary in this case because the buffer number is not known in advance. The only way to supply the buffer number as a default is to read the buffer first, and put it into the defaults dictionary only after that.

After loading the definition, Listing 11 demonstrates how rapidly a performer can instantiate and use multiple copies of the prototype.

> **Note:** Clearly, the code in Listing 10 is too long to be practical to type in the middle of a performance. For practical purposes, you should place all of the process definitions into a separate file, which you would load once at the beginning of a performance. See also the *Make* statement (Section 5.4), which makes it easy to instantiate the processes as needed during the performance, reducing the overhead of initial loading. (In fact, Chucklib was designed from the beginning to "package" complex musical behaviors into objects that are simpler to use, once defined. *cll* is an even more compact layer of control on top of this, following the same design principle: *definition* and *performance usage* are different, and call for different types of code.)
> Listing 10 illustrates how to wrap the `PR(\abstractLiveCode).chuck(...)` statement into a Chucklib *factory* object. I strongly recommend following this model; it "publishes" your process for use with `/make()` and avoids the need to manage large, complex code blocks in performance.

4.3 Parameter map

The parameter map `parmMap` is easiest to write as a set of nested Events:

```
parmMap: (
   parmName: (
      char: value,
      char: value,
      char: value...
   ),
   parmName: (...)
)
```

Listing 12: Template for the parameter map.

`parmName` keys should be Symbols. The keys of the inner dictionaries should be characters (`Char`), because the elements of the pattern strings that represent "notes" are characters.

The inner dictionaries may contain two other items, optionally:

`isPitch` If `true`, enables pitch notation for this parameter (Section 5.2.4).

`alias` An alternate name for this parameter, to use in the pattern. For example, if the parameter should choose from a number of SynthDefs, it would be inconvenient to type `instrument` in the performance every time you need to control it, whereas `def` would be faster. You can do this as follows:

```
parmMap: (
   def: (
      alias: \instrument,
      $s: \sawtooth, $p: \pulse, $f: \fm
   )
)

// Then you can set the "instrument" pattern:
/proc.phrase.def = "s";
```

Written this way, `def` in the *Set pattern* statement will be populate `instrument` in the resulting events.

4.3.1 Array arguments in the parameter map

Array arguments are valid, and will be placed into resulting events as given in the parameter map. In Listing 13, `freqs` will receive the array `[200, 300, 400]` and process that array according to the event prototype's rules.

```
parmMap: (
   freqs: (
      $2: [200, 300, 400],
   ),
   parmName: (...)
)
```

Listing 13: How to write arrays in the parameter map.

Envelopes may be passed to arrayed Synth controls in the same way: `Env.perc(0.01, 0.5).asArray`.

Note: The above is valid for the event prototype used by default in `PR(\abstractLiveCode)`. This is not SuperCollider's default event; it's a custom event prototype defined in *chucklib* that plays single nodes and integrates more easily with `MixerChannel`. Because each such event plays only one node, array arguments are passed as is. The normal default event expands one-dimensional arrays into multiple nodes. The way to avoid this is to wrap the array in another array level.

parmMap array format	**singleSynthPlayer meaning**	**Default event meaning**
`[1, 2, 3]`	Pass the array to one node	Distribute the three values to three nodes
`[[1, 2, 3]]`	Invalid	Pass the array to one node

One other use of the parameter map array is used to set disparate Event keys using one *cll* parameter. `Pbind` allows multiple keys to be set at once by providing an array for a key. *cll* supports this by using an array for the alias!

```
parmMap: (
   filt: (
      alias: [\ffreq, \rq],
      $x: [2000, 0.05]
   )
)
```

Listing 14: Arrays for multiple-parameter setting using one cll parameter.

4.4 Event processing

Every event produced by a *cll* process goes through three stages:

1. Insert all the items from `defaults`.
2. Insert the values from the current phrase (defined by pattern strings).
3. Insert any values from `postDefaults`. This may be a `Pbind`, and it has access to all the values from 1 and 2 by `Pkey`.

Thus, you can use `postDefaults` to derive values from items defined in the parameter map, or to check for invalid values.

4.5 Phrase sequence

cll "Set pattern" statements put musical information into any number of phrases. When you play the process, it chooses the phrases one by one using a pattern stored as `phraseSeq`. "Set pattern" has a compact way to express phrase

Table 2: List of available chucklib-livecode statements.

Type	Function	Syntax outline
Set pattern	Add new musical information into a process	`/proc.phrase.parm = "data"`
Start/stop	Start or stop one or more procesess	`/proc/proc/proc+` or `-`
Randomizer	Create several randomized patterns at once	`/proc.phrase.parm *n +ki "base"`
Make	Instantiate a process or voicer	`/make(factory/factory)`
Passthrough	Pass a method call to a BP	`/proc(method and arguments)`
Chuck	Pass a chuck => operation to a BP	`/proc => target`
Func call	Call a function in chucklib's Func collection	`/funcname.(arguments)`
Copy	Copy a phrase or phrase set into a different name	`/proc.phrase*n -> new`
Transfer	Like "Copy," but also uses the new phrase for play	`/proc.phrase*n ->> new`
Show pattern	Copies a phrase pattern's string into the document, for editing	`/proc.phrase.parm`

sequences, allowing sequences, random selection (with or without weights) and wildcard matching. See Phrase selection for details (Section 5.2.5).

This design supports musical contrast. The performer can create divergent materials under different phrase identifiers. Then, during the performance, she can change the phrase-selection pattern to switch materials on the fly. Sudden textural changes require changing many phrase-selection pattern at once. ~~For this, Register commands can save sequences of statements to reuse quickly and easily.~~

5 Livecoding statement reference

5.1 Statement types

cll supports the statements shown in Table 2, listed in order of importance.

cll statements begin with a slash: `/`. Statements may be separated by semicolons and submitted as a batch.

```
// run one at a time
/kick.fotf = "----";
/snare.bt24 = " - -";
```

```
// or as a batch
/kick.fotf = "----"; /snare.bt24 = " - -";
```

Listing 15: Cll statements, one by one or as a batch.

5.2 Set pattern statement

Set pattern is the primary interface for composing or improvising musical materials. As such, it's the most complicated of all the commands.

This statement type subdivides into two functions: phrase *definition* and phrase *selection*.

5.2.1 Phrase definition

Most "Set pattern" statements follow this format:

```
/proc.phrase.parm = quant"string";
```

Listing 16: Syntax template for the Set pattern statement.

Syntax elements:

`proc` The BP's name.

`phrase` (optional) The phrase name. If not given, `main` is assumed.

`parm` (optional) The parameter name. The BP must define a default parameter name, to use if this is omitted.

`quant` (optional) Determines the phrase's length, in beats.

- A number, or numeric math expression, specifies the number of beats.
- `+` followed by a number indicates "additive rhythm." The number is taken as a base note value. All items in the string are assumed to occupy this note value, making it easier to create fractional-length phrases. (If only `+` is given, the BP may specify `division`; otherwise 0.25 is the default.)
- If `quant` is omitted entirely, the BP's `beatsPerBar` is used. Usually this is the `beatsPerBar` of the BP's assigned clock.

`string` Specifies parameter values and rhythms.

Note: Both the phrase and parameter names are optional. That allows the following syntactic combinations:

Syntax	Behavior
`/proc = "string"`	Set phrase "main," default parameter
`/proc.x = "string"`	Set phrase "x," default parameter
`/proc.x.y = "string"`	Set phrase "x," parameter "y"
`/proc..y = "string"`	Set phrase "main," parameter "y"

Of these, the last looks somewhat surprising. It makes sense if you think of the double-dot as a delimiter for an empty phrase name.

5.2.2 Pattern string syntax

Pattern strings place values at time points within the bar. The values come from the parameter map. Timing comes from the items' positions within the string, based on the general idea of equal division of the bar.

Two characters are reserved: a space is a timing placeholder, and a vertical bar, |, is a divider.

If the string has no dividers, then the items within it (including placeholders) are equally spaced throughout the bar. This holds true even if it's a non-standard division: #4 (Figure 1) has seven characters in the string, producing a septuplet.

If there are dividers, the measure's duration will be divided first: n dividers produce $n + 1$ units. Then, within each division, items will be equally spaced. The spacing is independent for each division. For example, in #6 below, the first division contains one item, but the second contains two. For all the divisions to have the same duration, then, - in the second division should be half as long as in the first.

Note: It isn't exactly right to think of a space as a "rest." `"- - "` is not really two quarter notes separated by quarter rests; it's actually two half notes! If you need to silence notes explicitly, then you should define an item in the parameter map whose value is a `Rest` object.

5.2.3 Timing of multiple parameters

Each parameter can have its own timing, but a `Pbind` can play with only one rhythm, raising a potential conflict.

The `Pbind` rhythm is determined by the pattern string for the `defaultParm` declared in the process. When you set the `defaultParm`, the rhythm defined in that string is assigned to the `\dur` key, where it drives the process's timing. Other parameters encode timing into a `Pstep`, to preserve the values' positions within the bar. Think of these as "sample-and-hold" values, where the control

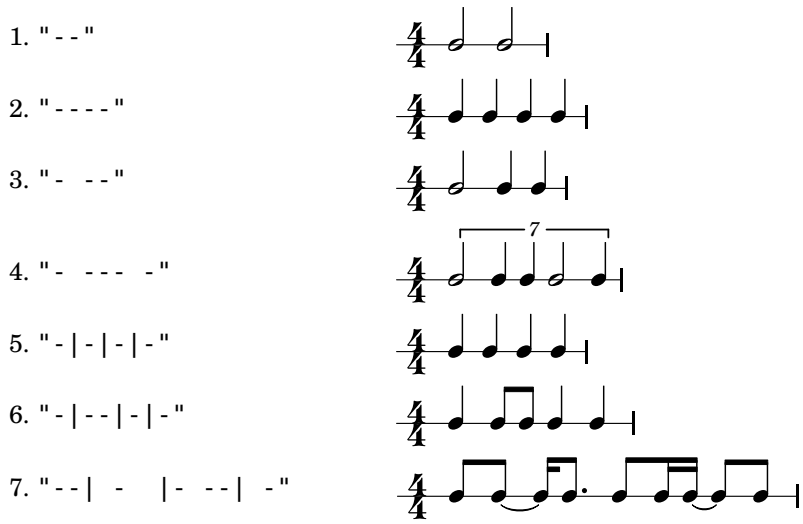

Figure 1: Some examples of cll rhythmic notation, with and without dividers.

value *changes* at times given by its own rhythm, but is *sampled* only at the times given by the `defaultParm` rhythm.

For example, here, the default parameter's rhythm is two half notes. At the same time, a filter parameter changes on beats 1, 2 and 4. The process will play two events, on beats 1 and 3. On beat 1, the filter will use its `a` value; on beat 3, it will use the most recent value, which is `b`. *The filter will not change on beat 2*, because there is no event occurring on that beat!

What about `c`? There is no event coming on or after beat 4, so `c` will be ignored in this case. But, if you add another note late in the bar, then it will pick up `c`, without any other change needed.

```
/x = "--";
/x.filt = "ab c";  // "c" is not heard

/x = "-|-  -";  // now "c" is heard on beat 4.5
```

Listing 17: Multiple parameters with different timing.

5.2.4 Pitch notation

If a parameter's map specifies `isPitch: true`, then it does not need to specify any other values and the rules described in Section 3.6.2 apply.

```
// Initialization code
(
SynthDef(\sqrbass, { |out, freq = 110, gate = 1,
   freqMul = 1.006, amp = 0.1,
   filtMul = 3, filtDecay = 0.12, ffreq = 2000, rq = 0.1,
   lagTime = 0.1|
   var sig = Mix(
      Pulse.ar(Lag.kr(freq, lagTime) * [1, freqMul], 0.5)
   ) * amp,
   filtEg = EnvGen.kr(
      Env([filtMul, filtMul, 1], [0.005, filtDecay], \exp),
      gate
   ),
   ampEg = EnvGen.kr(Env.adsr(0.01, 0.08, 0.5, 0.1),
      gate, doneAction: 2);
   sig = RLPF.ar(sig, (ffreq * filtEg).clip(20, 20000), rq);
   Out.ar(out, (sig * ampEg).dup);
}).add;

(
keys: #[master],  // ~master comes from \loadAllCl.eval
defaultName: \sqrbs,
initLevel: -12.dbamp,
argPairs: #[amp, 0.5],
make: { |name|
   var out;
   ~target = MixerChannel(name, s, 2, 2, ~initLevel, outbus:
       ~master);
   out = Voicer(5, \sqrbass, target: ~target);
   out.mapGlobal(\ffreq, nil, 300, \freq);
   out.mapGlobal(\rq, nil, 0.2, \myrq);
   out.mapGlobal(\filtMul, nil, 8, [1, 12]);
   out
},
free: { ~target.free },
type: \vc) => Fact(\sqrbsVC);
)

// Performance code:
\loadAllCl.eval;
TempoClock.tempo = 132/60;
Mode(\fsloc) => Mode(\default);

/make(sqrbsVC/melBP:bs(octave:3));
/bs = "1_  1.|5~3_9.4.|7.2~4_5'.|5_8~2_4.";
/bs+;

/bs-;
```

Listing 18: A retro acid-house bassline, demonstrating pitch notation.

Note: Items in pitch sequences may include more than one character: `3` is one note, as is `6+,~`. They are converted into `SequenceNote` objects in the pattern, because `SequenceNotes` can encode pitch and articulation information. Post-processing in `PR(\abstractLiveCode)` extracts the articulation value and assigns it to `\legato` (or `\sustain` for staccato notes).

To use pitched notes, I strongly recommend installing the *ddwLivecodeInstruments* quark, and using the `melBP` process (or the other pitched processes described in the tutorial), as demonstrated in Listing 18. The general procedure is:

- In your initialization script, package your instrument as a Voicer factory, following the template shown. Note the following keys in the factory's dictionary:
  - `keys`: Import items from the top environment. In particular, `\loadAllCl.eval` creates mixer channels `~master`, `~rvbmc` and `~shortrvbmc`. You can use these when creating the Voicer's `~target` mixer.
  - `defaultName`: The name of the `VC` object that `/make()` will create.
  - `initLevel`, `initRvb`: Not strictly necessary, but *ddwLivecodeInstruments* uses these as a convention to set the levels of the mixer channel and reverb post-send.
  - `argPairs`: Parameters and defaults specific to this instrument, to inject into the note player when using the instrument. One convention in *ddwLivecodeInstruments* is a default amplitude of 0.5, overriding the default event prototype's 0.1.
- In performance, `/make` the instrument and a `melBP` process together.
- Specify the key: `Mode(\keyname) => Mode(\default)`. The "legacy example" below uses F-sharp Locrian, hence `Mode(\fsloc)`. (This is a global setting; you do not need to repeat this step for every process.)

The particular benefit of a Voicer + `melBP` is that slurred notes will tie across the bar line correctly. (*cll* needs to evaluate processes on every bar line, whether a note sounds at that time or not. If a pitched note ties over the bar line, the last note in the bar does not know when the next note in the bar will occur. This is extremely difficult to handle using the default event prototype. *ddwLivecodeInstruments* pitched processes use a custom event prototype, with features of Voicer, to make it transparent to the user.)

If you must use only the basic features of *cll* (`\loadCl.eval`), you can use `PR(\abstractLiveCode)` and place a dummy `PmonoArtic` into `postDefaults`, as in Listing 19.[12] If you do it this way, you lose support for accents, and notes

[12]Note the trick to get monophonic synthesis. Assigning a `PmonoArtic` into `postDefaults` effectively turns the entire event-producing chain into a `PmonoArtic`—even if it adds no musically useful information into the resulting events. *Caveat*: If you will have any notes slur across the barline, make sure to include `alwaysReset: true` in the BP parameter dictionary.

```
// Use the same SynthDef as in the previous example

BP(\acid).free;
PR(\abstractLiveCode).chuck(BP(\acid), nil, (
   event: (eventKey: \default),
   alwaysReset: true,
   defaultParm: \degree,
   parmMap: (
      degree: (isPitch: true),
   ),
   defaults: (
      ffreq: 300, filtMul: 8, rq: 0.2,
      octave: 3, root: 6, scale: Scale.locrian.semitones
   ),
   postDefaults: PmonoArtic(\sqrbass,
      \dummy, 1
   )
));

TempoClock.tempo = 132/60;
)

/acid = "1_  1.|5~3_9.4.|7.2~4_5'.|5_8~2_4.";

/acid+;
/acid-;
```

Listing 19: Pitch notation in PR(\abstractLiveCode); not generally recommended.

will not automatically tie across the bar line. You can work around this by explicitly repeating the note in the next bar. This is very difficult to do with generators, however.[13]

5.2.5 Phrase selection

Statements to set the phrase sequence follow a different syntax:

```
/proc = (group...);
```

Listing 20: Syntax template for "Set pattern" phrase selection.

`group` can consist of any of the following elements:

Phrase ID The name of any phrase that's already defined, or a regular expression in single quote marks. If more than one existing phrase matches the regular expression, one of the matches will be chosen at random; e.g., to choose randomly among phrases beginning with `x`, write `'^x'`.

Name sequence Two or more of *any* of these items, separated by dots and enclosed in parentheses: `(a0.a1.a2)`. These will be enclosed in `Pseq`.

Random selection Two or more of any of these items, separated by vertical bars (`|`) and enclosed in parentheses: `(a0|a1|a2)`. These will be enclosed in `Prand`. *One* will be chosen before advancing to the next ID.

Phrase group A name, followed by two asterisks and a number of bars in the phrase group. If a four-bar phrase is stored as `a0`, `a1`, `a2`, and `a3`, you can write it simply as `a**4`. The preprocessor will expand this to regular expression matches, as if you had written `('^a0'.'^a1'.'^a2'.'^a3')`. The use of regular expression matching here is to make it easier to have slight variations on the bars within the phrase group, while keeping the same musical shape.

Any of these items may optionally attach a number of repeats `*n`: `(a*3.b)` translates to `Pseq([Pn(\a, 3), \b], inf)`, and `(a*3|b)` to `Prand([Pn(\a, 3), \b], inf)`.

Items in a random selection may also attach a weight `%w`, which must be given as an integer: `(a%6|b%4)` has a 60% chance of choosing `a` and a 40% chance of `b`. If no weight is given, the default is 1. Weights are ignored for sequences (separated by dots).

Groups may be nested, producing complex structures compactly. For example, to have an 80% chance of `a` for four bars, then an 80% chance of `b` for two bars, you would write:

```
((a%4|b)*4.(a|b%4)*2)
```

Listing 21: Nested phrase-selection groups.

[13] I present the `PmonoArtic` example only to demonstrate that it is possible. This way, however, is not as powerful as `melBP`, so I maintain the recommendation to use `melBP` instead.

You may also include both `.` and `|` in a single set of parentheses. The dot (for sequence) takes precedence: `(a.b|c)` evaluates as `((a.b)|c)`.

5.3 Start/stop statement

The start/stop statement takes the following form:

- Start: `/proc1/proc2/proc3+quant`
- Stop: `/proc1/proc2/proc3-quant`

Any number of process names may be given, each with a leading slash.

`quant`, an integer, tells each process to start or stop on the next multiple number of beats. In 4/4 time, `/proc+4` will start the process on the next bar line; `/proc+8` will start on the next event-numbered bar line (i.e., every other bar). `quant` is optional; if not given, each process will use its own internal `quant` setting. By default, this is one bar; however, the `setm` helper function overrides this for the given number of bars.

5.4 Make statement

The make statement instantiates one or more *chucklib* factories.

```
/make(factory0:targetName0(parameters0)/factory1:targetName1(
    parameters1)/...);

// Or, with autoGui
/make*(factory0:targetName0(parameters0)/factory1:targetName1(
    parameters1)/...);
```

Listing 22: Syntax template for make statements.

`factory` The name of a `Fact` object to create.

`targetName` (optional) The name under which to create the instance. If not given, the make statement looks into the factory for the `defaultName`. If not found, the factory's name will be used.

`parameters` (optional) A dictionary of `key: value` pairs to insert into the Factory prior to making the resulting object, to override defaults. NOTE: It is up to the Factory to pass values into the final object. `Fact()` does not scan for unknown keys and automatically forward them.

Multiple `factory:targetName` pairs may be given, separated by slashes. Both `BP` and `VC` factories are supported.

As noted earlier, the code to define *cll* processes is not performance-friendly. Instead, you can write this code into `Fact` object, and then `/make` them as you need them in performance.

```
(
// THIS PART IN THE INIT FILE
(
defaultName: \demo,
octave: 5,  // a default octave
make: { |name|
   PR(\abstractLiveCode).chuck(BP(name), nil, (
      event: (eventKey: \default),
      defaultParm: \degree,
      parmMap: (degree: (isPitch: true)),
      // Here, the Factory transmits ~octave
      defaults: (octave: ~octave)
   ));
}, type: \bp) => Fact(\demoBP);
)

// DO THIS IN PERFORMANCE
/make(demoBP:dm(octave:6));  // :dm overrides defaultName

/dm = "1353427,5,";
/dm+;
/dm-;

/dm(free);
```

Listing 23: Example of the make statement.

`/make*` instead of `/make` will try to chuck process or voicer mixers into chucklib `MCG` objects, for display in a mixing board, and voicers into empty chucklib `VP` objects, for speed in setting up players during performance. `MCG` and `VP` arrays are created by `BP.loadGui`.

Parameters: *ddwChucklib* has two ways to write a "chuck" operation:

- Simplified: `source =>.adverb target`, or
- Complete: `source.chuck(target, adverb, parameters)` (`parameters` is a dictionary, usually written in Event-style syntax).

Different object types may handle the parameters differently. The *cll* `/make` statement supports `Fact()` as a source, which imports the parameters into the factory's environment before making the result. The parameters are available to the factory, which may choose to insert them into the resulting object in whatever way is needed. Because Event-style syntax already encloses the names and values in parentheses, the `/make` statement simply reads the dictionary directly in the argument list.

Voicers and BPs: If a single `/make` statement produces first a Voicer and then a BP, `/make` will additionally assign the Voicer to play the BP's notes, saving the time in performance of writing an additional `VC(\name) => BP(\name)` command.

5.5 Passthrough statement

The passthrough statement takes arbitrary SuperCollider code, enclosed in parentheses, and applies it to any existing *chucklib* object. If no class is specified, BP is assumed. No syntax checking is done in the preprocessor, apart from counting parentheses to know which one really ends the statement.

```
// This...
/snr(clock = ~myTempoClock);

// ... is the same as running:
BP(\snr).clock = ~myTempoClock;

// Or...
/VC.bass(releaseAll);  // VC(\bass).releaseAll;
```

Listing 24: Syntax template for passthrough statements.

5.6 Chuck statement

The chuck statement is a shortcut for chucking any existing *chucklib* object into some other object. If no class is given, BP is assumed.

```
// This...
/snr => MCG(0);

// ... is the same as running:
BP(\snr) => MCG(0);

// Or...
/VC.keys => MCG(0);  // VC(\keys) => MCG(0);
```

Listing 25: Syntax template for Chuck statements.

5.7 Func call statement

The Func call statement is a shortcut to evaluate a function saved in *chucklib*'s Func collection. This makes it easier to use helper functions. No syntax checking is done in the preprocessor.

```
/func.(arguments);

// e.g.:
/bars.(\proc, 2, \a);
```

Listing 26: Syntax template for func-call statements.

Note: The dot after the function name is critical! Without it, the statement looks exactly like a passthrough, and the preprocessor will treat it as such.

5.8 Copy or transfer statement

Copy/transfer statements create additional copies of phrases, so that you can transform the material while keeping the old copy. Then you can switch between the old and new versions, setting up a musical form.

```
/proc.phrase*n -> newPhrase;  // copy

/proc.phrase*n ->> newPhrase;  // transfer
```

Listing 27: Syntax template for copy/transfer statements.

`proc` The process on which to operate.

`phrase` The phrase name to copy.

`n` (optional) If given, copy a multi-bar phrase group, treating `phrase` as the prefix. `/proc.a*2 -> b` will copy `a0` to `b0` and `a1` to `b1`. (If `n` is omitted, both `phrase` and `newPhrase` will be used literally.)

`newPhrase` The name under which to store a copy. If `n` is given, this is a phrase group prefix.

The difference between "copy" and "transfer" is:

- Copy (`->`) simply duplicates the phrase information, but continues playing the original phrases. If you change the new copies, you won't hear the changes until you change the phrase selection pattern. This is good for preparing new material and switching to it suddenly.
- Transfer (`->>`) duplicates the phrase information *and* modifies the phrase selection pattern, replacing every instance of the old phrase name with the new.[14] Changing the new copies will now be heard immediately. This is good for slowly evolving new material, while keeping the option to switch back to an older (presumably simpler) version later.

5.9 Show pattern statement

Less a "statement" than an interface convenience, this feature looks up the string for a given phrase and parameter, and inserts it into the code document. Invoke this behavior by typing `/proc.phrase.parm` and evaluating the line by itself. As in other contexts, `phrase` and `parm` are optional and default to `main` and the process's `defaultParm` respectively. For a multi-bar phrase group, type `/proc.phrase*n.parm` (where `n` is the number of bars in the group.)

This is useful after a copy/transfer statement.

[14] It does this by producing a `compileString` from the phrase selection pattern, performing string replacement, and then recompiling the pattern. This should work with all *cll* phrase selection strings (Section 5.2.5). It is not guaranteed to work with hand-written patterns that generate phrase names algorithmically.

```
/snr.a = " - -";

/snr.a -> b;

/snr.b   // now hit ctrl-return at the end of this line

// the line magically changes to
/snr.b = " - -";
```

Listing 28: Demonstration of "Show pattern" statements.

> **Note:** You must be using SuperCollider IDE 3.7 or above. Automatic code insertion is not supported for other editors or older IDE versions.

5.10 Helper functions

A few "helper functions" are defined, to simplify common tasks:

- `/changeMeter.(beatsPerBar, clock, barBeat, newBaseBarBeat)`: Set the meter. The last three arguments are optional. If the last three are omitted, it will set `beatsPerBar` on the default `TempoClock`, on the next barline.
- `/changeTempo.(tempo, clock, barBeat)`: Set the tempo (beats per second). The last two arguments are optional (default `TempoClock`, and the next barline). This is mainly to simplify group performances using the Utopia quark's `BeaconClock`.
- `/changeKey.(\modeName)`: Assign the named Mode object to `Mode(\default)`. All processes using the default mode will switch to the new key immediately.
- `/globalSwing.(array)`: Set an array defining swing subdivisions. To swing 16th-notes, the array's values should add up to 0.5 (eighth-note), e.g. `[0.3, 0.2]`. (Individual processes may override the global swing: `/name(swing = array)`.

Also, three `Func` definitions are provided to make it easier to work with multi-bar phrase groups. I will introduce them using *cll* Func call statement syntax (Section 5.7).

`/setupbars.(\proc, n, \prefix)` Create empty phrases for `prefix0`, `prefix1` up to $n - 1$. This also inserts *Set pattern* (Section 5.2) templates into the code document, for you to start filling in musical material.

`/setm.(\proc, n, \prefix)` Set the process's phrase selection pattern to play this phrase group. It also changes `quant` in the process, so that starting and stopping the process will align to the proper number of bars.

`/bars.(\proc, n, \prefix)` Calls both `setupbars` and `setm` at once.

A typical sequence of performance instructions for me is:

```
/make(kick);
/bars.(\kick, 2, \a);

// the following lines are automatically inserted
/kick.a0 = "";
/kick.a1 = "";
```

Listing 29: Common initialization sequence, using helper functions.

After the templates appear, I edit the strings to produce the rhythms I want, and then launch the process with `/kick+`. In this example, the phrase group occupies two bars. `setm` automatically sets the process's `quant` to two bars, so the process will then launch on an even-numbered barline.

5.11 (Deprecated) Randomizer statement

Previous versions included a "randomizer" statement. Generators are far more powerful. The code remains for backward compatibility, but it will be undocumented and not officially supported going forward.

6 Generators

The basic syntax of the *Set pattern* statement (Section 5.2) denotes fixed note sequences, which always play exactly the same events. *Generators* create phrases whose contents can change on each iteration, adding another dimension of musical interest.

6.1 Generator design

Generators manipulate lists of events, provided by a "chain" operator, one bar at a time.

> **Note:** January 2020: Generator syntax has changed from previous versions. Previously, the source event list was specified as the first argument to a generator. Now, the chain operator provides all generator source lists; the first argument is removed. Old scripts will have to be rewritten.

At present, generators divide into these main categories:

- *Rhythm generators* insert new items into the event list, or delete them. New items may be event characters directly, or wildcards to be replaced by the second category.
- *Content generators* replace wildcards with user-specified values.

- *Filter generators* alter the flow of control.

These are not the only possible generator types, and there is no prescribed sequence for using them. However, it's been most successful so far to use a rhythm generator to embellish a base rhythm, and then apply a content generator to "fill in" the new rhythmic elements.

```
(
BP(\y).free;
PR(\abstractLiveCode).chuck(BP(\y), nil, (
   event: (eventKey: \default),
   defaultParm: \degree,
   parmMap: (degree: (isPitch: true))
));
)

TempoClock.tempo = 140/60;

/y = "12 4| 5 6| 12 |45";  // A

/y+;

/y = "[** *| * *| ** |**]::\seq("12456", "*")";  // B

/y = "[** *| * *| ** |**]::\seq("12456", "*")::\ins("*", 7,
    0.25)";  // C

/y = "[** *| * *| ** |**]::\seq("12456", "*")::\ins("*", 7,
    0.25)::\seq("6,214", "*")";  // D

/y-;
```

Listing 30: Isorhythmic cycles with generators.

Listing 30 demonstrates one possibility. The initial idea is a cycle of five pitches laid over nine notes within a bar. Without generators, it's necessary to drop one pitch at the end of every bar (A). But, using the `\seq()` generator, we can specify the rhythm using a `*` wildcard; `\seq()` replaces each wildcard with successive pitches. `\seq` also remembers its state from one bar to the next, so, in this example, the first bar will begin with `1` and the second, with `6` (B).

To add the first generator, wrap the source string in square brackets (indicating that it is a source), and "chain" (`::`) the generator onto it. The `::` operator takes the previous result (here, the source) and feeds it into the following generator.

Generators are "composed" by chaining further generators. `\ins("*", 7, 0.25)` inserts seven wildcards at randomly chosen 1/4-beat positions (C). (There are 16 per bar, and 9 are already occupied, so this will fill all the empty rhythmic positions.) `*` is not a valid pitch specifier, so these are performed as rests. Chaining one more layer, another `\seq()` (D), overlays a new cycle, four

notes this time. The result is a shifting arpeggiation that should repeat every 20 bars—but written as a single bar's pattern string.

6.2 Generator usage

6.2.1 Generators and pattern strings

Generators are invoked using the syntax `\name(arguments)` within a "set pattern" string.

As noted earlier, every item in a pattern string occupies a span of time within the bar, beginning at its metrical position and continuing until the metrical position of the next item (or the end of the bar). (Spaces in the pattern string are placeholders, not items.) In Figure 2, example A, the first `-` begins on beat 2 and extends until the next item, a `.` on beat 3.25, and so on until the end of the bar. The pattern string puts nothing at the beginning of the bar—remember, spaces and dividers are not items. During generator processing, literally nothing is in this space. When it's time to play the bar's contents, a rest event will be inserted, but this does not affect playback.

A generator expression, from the opening backslash to the closing parenthesis, counts as one item, whose span is determined in the same way. The simplest case is Figure 2, example B, where a single generator occupies the entire bar (there are no spaces or dividers before it, and no items after it). In example C, the generator begins on the second eighth-note, and its span is terminated by an `x` on beat 4.

Metrical position is especially important for the insert `\ins()` generator. `\ins()` chooses metrical positions randomly from a "grid" determined by the generator's starting position and the `quant` argument (fourth in the list). In example B, `\ins()` starts at the beginning of the bar, and the grid is in half-beats (eighth-notes). Example C's `\ins()` grid is in quarter-notes, but beginning off the beat—syncopation—and the generator ends at beat 4. Beat 4.5 matches the "syncopated quarter-note" specification, but it is outside the generator's bounds, so it is unavailable to this `\ins()` instance.

This poses a problem for chaining generators. You can `::` chain a second generator only onto another generator, not normal items. But the generator occupies beats 1.5–4 only; therefore, a chained generator can apply only to this time span, not the entire bar. *Cll*'s solution is a filter generator, `\fork()`, which positions sub-generators within its own time span. In example D, the same `\ins()` generator occurs within a `\fork()`. The `\ins()` still applies only to beats 1.5–4, but the `\fork()` is the only item in the bar and covers the entire span. If you chain additional generators onto the `\fork()`, they will likewise cover the entire bar.

Listing 31 provides some playable examples. In Example 3, beat 2 contains four items: `6,`, `\rand(...)`, space and space. Thus beat 2 is subdivided into 16th-notes, and the generator begins on the second of those.

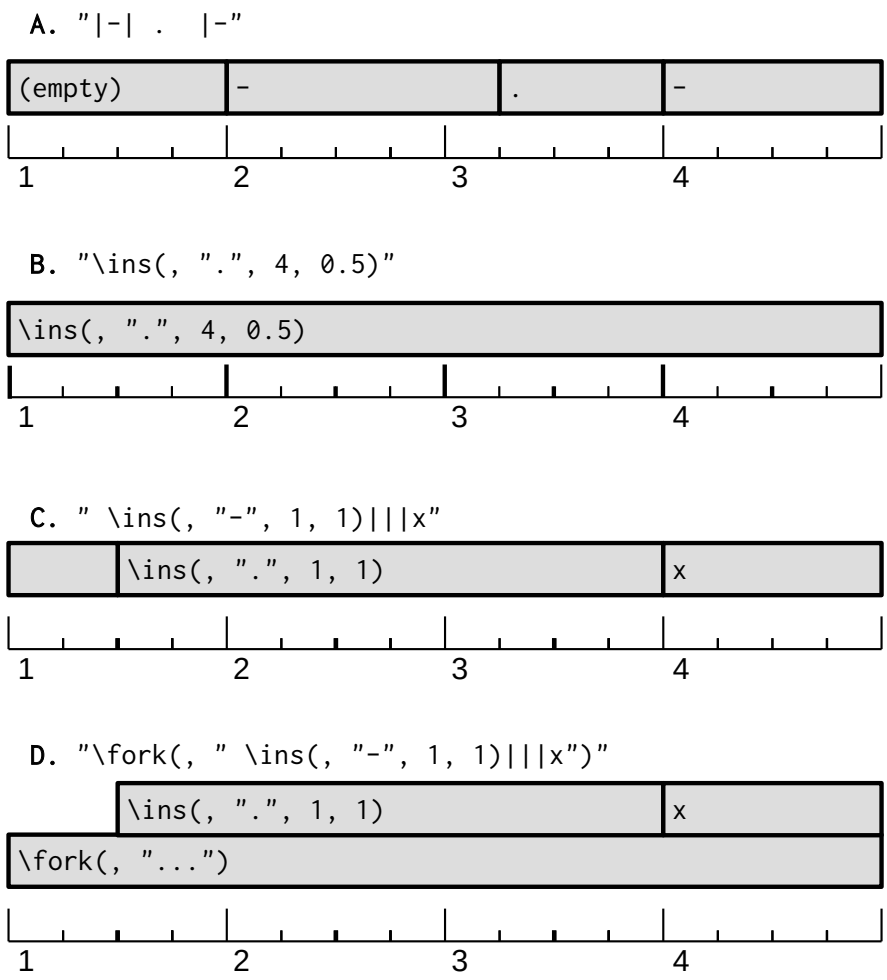

Figure 2: Time spans of items in pattern strings.

```
// 1. Chain starts on the downbeat and occupies the whole bar.
/y = "[1,]::\ins("*", 3, 0.5)::\rand("13467", "*")";

/y+;

// 2. Chain starts on beat 2
// Note that a generator source can appear
// anywhere within the bar!
/y = "1,|[6,]::\ins("*", 3, 0.5)::\rand("13467", "*")||";

// 3. Chain starts on the 2nd 16th-note of beat 2
// Here, '6,' occupies time and is not a generator source.
// So it is not bracketed.
/y = "1,|6,\ins("*", 3, 0.5)::\rand("13467", "*")  ||";

// 4. Chain starts on the 2nd 16th-note of beat 2
// and stop on the 'and' of 4
/y = "1,|6,\ins("*", 3, 0.5)::\rand("13467", "*")  || x";

/y-;
```

Listing 31: Interaction between generator syntax and "set pattern" rhythmic notation.

> **Note:** `\ins("new", num, quant)` inserts *num* new items at possible time points *quant* beats apart. These time points are measured from the beginning of the generator. In Listing 31, examples 3 and 4 offset the generator by one 16th-note—so `\ins()` will syncopate by a 16th.
> Source strings will be compressed to fit into the available duration. In example 2, from the source `[6,]` to the end of the bar is 3 beats, so `[6,||7]` would be correct. `[6,|||7]` would divide the 3 beats into 4 subdivisions.

6.2.2 Generator arguments

Every generator expression currently requires an argument list in parentheses following the generator's name. (If a generator doesn't require arguments, an empty pair of parentheses is currently still required. I may remove this requirement later, but for now, it's not optional.)

Arguments are separated by commas.

Simple arguments are as follows;

- A number, which may be written as follows:
  - An integer (`12`) or floating-point number (`1.5`).
  - An integer range: `1..3`. Especially useful for arguments representing a number of notes to insert or process.

  - A fraction: `1/3`.
  - A tuplet: e.g., `3:4` means "three notes in the space of a quarter note."
  - A pass-through expression in curly braces: `{rrand(1, 4) * 0.5}` for 0.5, 1.0, 1.5, 2.0.

> **Note:** Currently, pass-through expressions in braces may not contain backslashes. If you need to refer to a symbol, use `'singlequote'` symbol literal syntax.

- A Symbol, written in LISP style with an opening backtick: `` `name ``.

Pool arguments provide collections of items. Currently, pools are used in two ways:

- A list of wildcards for matching. For instance, the source string may include wildcards `*` and `@`. A wildcard string `"*"` means that the generator will apply only to asterisks. (`"*@"` would match both.)
- A sequence of items with which to replace a wildcard (or, for rhythm generators, to insert directly). This type of pool may be written as an item string (where each item has equal probability of being chosen, as in `Prand`) or as a subset of certain sequencing generators:
  - `\seq()`
  - `\ser()`
  - `\rand()`
  - `\wrand()`
  - `\xrand()`
  - `\shuf()`
  - See Listing 40 for examples of complex sequences produced by nested generators.
  - When used in a pool argument, a number of repeats may be specified as follows: `\seq*n(args)` where `n` is any of the numeric expressions described above. For example, if you want to embed a sequence of six notes, always starting from the beginning but randomly embedding 3 to 6 steps, write `\ins(\ser*3..6("123456"), ...)` (where `\ins()` or another one of the 6.4.1 determines the rhythm).

Quotes for pool strings should *not* be escaped with backslashes, even though these quoted strings appear within quotes. The set pattern parser reads the pattern string up to a closing quote that appears *outside* generator expressions.

6.3 Wildcard matching

Many generators include a *wildcards* argument. If omitted, the generator will apply to all items with its timespan. Otherwise, it will operate only on items that were produced by that generator. For example, if I want to choke off all open hi-hats after 0.25 beats, but leave closed hi-hats alone, I can specify `"-"` for *wildcards*.

```
TempoClock.tempo = 124/60;
/hh.(\synthhh);
/shh = "\ins("-", 2, 0.5)::\ins(".", 8, 0.5)::\choke(0.25, ".",
    "-")";
/shh+

/shh-
```

6.4 Built-in generators

6.4.1 Rhythm generators

`\ins("new items", numToAdd, quant)` Locates unoccupied metric positions within the bar, every *quant* beats apart beginning with the generator's onset time, chooses *numToAdd* of them randomly, and inserts new items at those positions.

`\shift("shiftable items", numToShift, quant)` Locates *numToShift* occurrences of the *shiftable items* within the source (they must already exist), and moves them forward or back by *quant* beats. A good way to get syncopation is to insert items on a strong beat, and then shift them by a smaller subdivision.

`\rot(quant)` Add *quant* to every item's onset time, and wrap all the times into the generator's boundaries: basically, a strict canon.

```
// Reich, "Piano Phase"-ish

(
BP(\y).free;
PR(\abstractLiveCode).chuck(BP(\y), nil, (
   event: (eventKey: \default, pan: -0.6),
   defaultParm: \degree,
   parmMap: (degree: (isPitch: true))
));

BP(\z).free;
PR(\abstractLiveCode).chuck(BP(\z), nil, (
   event: (eventKey: \default, pan: 0.6),
   defaultParm: \degree,
   parmMap: (degree: (isPitch: true))
));
```

```
)

TempoClock.setMeterAtBeat(3, TempoClock.nextBar);
TempoClock.tempo = 112/60;

/y = "[*^*^*^*^*^*^]::\seq("268", "*")::\seq("37", "^")";

/z = "[*^*^*^*^*^*^]::\seq("268", "*")::\seq("37", "^")";

/y/z+;

/z = "[*^*^*^*^*^*^]::\seq("268", "*")::\seq("37", "^")::
    \rot(-0.25)";

/z = "[*^*^*^*^*^*^]::\seq("268", "*")::\seq("37", "^")::
    \rot(-0.5)";

/z = "[*^*^*^*^*^*^]::\seq("268", "*")::\seq("37", "^")::
    \rot(-0.75)";

/y/z-;
```

Listing 32: Usage of \rot() generator.

`\delta("new items", quant, weights...)` Add items into the source based on random selection of event deltas, as multiples of *quant*. If *quant* is 0.25, then the first *weight* is for 0.25, the second for 0.5 and so on; `\delta(, "*", 0.25, 3, 1, 2)` would choose deltas according to `Pwrand(0.25 * [1, 2, 3], [3, 1, 2].normalizeSum, inf)`. The next delta is added to the current position to find the position for the next item, until reaching the end of the timespan.

`\pdelta("new items", quant, "deltapool")` Like `\delta()`, except where `\delta()` always uses `Pwrand` to produce the durations, `\pdelta()` allows you to sequence multiples of *quant*. A `1` in the deltapool string is 1 times quant; `2` is 2 times quant and so on. A standard hi-hat pattern that could then be written `\pdelta(\seq("..-"), 0.25, \seq("112"))`.

`\rDelta("new items", maxDelta, factor, minNotes, maxNotes, distribution, parameters)` Like `\delta()`, but generates deltas from one of six random functions (whereas `\delta()` corresponds only to `Pwrand()`). It is, then, non-metric.

- `"new items"`: A pool string supplying items to insert.
- `maxDelta`: The largest duration allowed.
- `factor`: Calculates the shortest duration = `maxDelta / factor`.
- `minNotes`: Require a certain minimum number of notes.
- `maxNotes`: Constrain to a certain maximum number of notes.

- `distribution`: A symbol `` `lin ``, `` `exp ``, `` `hp ``, `` `lp ``, `` `beta ``, `` `expb ``. See the example.
- `parameters`: Numbers supplied as additional arguments to the delta pattern. Currently used only by `` `beta `` (supply *alpha* and *beta*) and `` `expb `` (supply *spread*).

```
// equal distribution
// but more total time spent on longer notes
/y = "\rDelta("*", 1, 8, , , `lin)::\wrand("\xrand("12345")
    \xrand("3'4'5'6'")", 2, 1)";

/y+

// equal total time spent on longer notes vs shorter
/y = "\rDelta("*", 1, 8, , , `exp)::\wrand("\xrand("12345")
    \xrand("3'4'5'6'")", 2, 1)";

// Phprand
/y = "\rDelta("*", 1, 8, , , `hp)::\wrand("\xrand("12345")
    \xrand("3'4'5'6'")", 2, 1)";

// Plprand
/y = "\rDelta("*", 1, 8, , , `lp)::\wrand("\xrand("12345")
    \xrand("3'4'5'6'")", 2, 1)";

// beta distribution: similar problem as `lin
/y = "\rDelta("*", 1, 8, , , `beta, 0.2, 0.2)::\wrand("
    \xrand("12345")\xrand("3'4'5'6'")", 2, 1)";

// expb: "exponentialized" beta distribution
/y = "\rDelta("*", 1, 8, , , `expb, 0.2)::\wrand("\xrand("
    12345")\xrand("3'4'5'6'")", 2, 1)";

/y-
```

Listing 33: Usage of \rDelta() generator.

`\ramp("new items", initDelta, midDelta, midpoint, curve)` Generates accelerating and decelerating rhythms. To use this effectively, think of the `\ramp()` generator's time span as being normalized: 0 is the beginning of the span and 1 is the end. Within that span, the generator will ramp from an initial duration to a target, and back to the start duration.

- `"new items"`: A pool string supplying items to insert.
- `initDelta`: The duration at the beginning of the time span.
- `midDelta`: The target duration.
- `midpoint`: Where, between 0 and 1, to arrive at `midDelta`. If 1, `midDelta` is placed at the end, and the ramp will go from `initDelta`

to `midDelta` (the most common use case). If 0, `midDelta` is at the beginning, and the ramp goes in the opposite direction (so, you can choose randomly between acceleration and deceleration by writing `{2.rand}` here). `0 < midpoint < 1` creates a two-segment curve.

- `curve`: Curvature types from `Env` are supported: numbers, or `` `lin ``, `` `exp `` etc. A curvature number applies to the first segment, and will be negated for the second.

```
/y = "\ramp("*", 1, 0.2, 1, `exp)::\pitch("*", "2", 0, 0)";
/y+

// randomly choose accel or decel
/y = "\ramp("*", 1, 0.2, {2.rand}, `exp)::\pitch("*", "2",
    0, 0)";

// alternate between accel and decel
/y = "\ramp("*", 1, 0.2, {Pseq([0, 1], inf)}, `exp)::\pitch
    ("*", "2", 0, 0)";

/y = "\ramp("*", 1, 0.2, 0.5, `exp)::\pitch("*", "2", 0, 0)
    ";

// pulls 1 -> 0.2 curve to the right, biasing long
    durations
/y = "\ramp("*", 1, 0.2, 0.5, 2)::\pitch("*", "2", 0, 0)";

// pulls 1 -> 0.2 curve to the left, biasing short
    durations
/y = "\ramp("*", 1, 0.2, 0.5, -2)::\pitch("*", "2", 0, 0)";

/y = "\ramp("*", 1, 0.2, 0.5, -4)::\pitch("*", "2", 0, 0)";

/y-
```

Listing 34: Usage of \ramp() generator.

`\choke(maxDur, "new items", "wildcards")` For every item within the time span, insert an item *maxDur* beats later (unless the next item comes at that time point or earlier). `\choke("-||-.|", 1, ".")` will insert a `.` at beat 2, because it is one beat later than the first item's position, and there is no other item already "choking off" that note. Beat 4 does not receive an extra item, because the `-` on beat 3 is already choked off a half beat later.

`\stutt("new items", numToAdd, quant, prob, "insItem", "wildcards")` Choose up to *numToAdd* matching items and stutter them according to *quant*. *New items* determines the items to match (if omitted, it matches any items, also taking *wildcards* into account). At *quant* intervals, a new item is inserted

if `prob.coin` is true. *insItem* specifies the item to insert; if omitted, it will insert a copy of the existing item.

`\stuttDur("new items", numToAdd, quant, prob, addDur, "insItem", "wildcards")` Like `\stutt()`, except that *addDur* is a maximum duration over which to repeat/subdivide each item. This may produce jittery patterns with empty space in between.

`\stuttN(("new items", numToAdd, quant, prob, numInclusive, "insItem", "wildcards")` Like `\stutt()`, except that *addDur* is a maximum number of new items to add.

`\div("new items", "replaceItems", numToAdd, quant, divisor, divisible)` Choose up to *numToAdd* matching items (taken from *replaceItems*, which must be at least *quant* beats long, and divide their duration by *divisor*, inserting items from *new items*. *Divisible* can further prevent short notes from being divided; if *quant* is 0.5 and *divisible* is 2.5, notes of duration 2.0 will not be divided.

`\euclid("new items", quant, increment, initial)` Euclidean rhythm generator. Operating on a grid defined by *quant*, *increment* is the step size and *initial* is the shift amount. Both of these are numeric proxies, so you can randomize them using range numbers or curly-brace-enclosed passthrough expressions.

`\golomb("new items", quant, minimum)` A Golomb ruler divides a number into quantized steps, where each partition is unequal. The total time to be partitioned comes from the generator's duration and cannot be overridden. *quant* is the rhythmic value to divide into, and *minimum* is the smallest integer number of *quant* steps that will be permitted. If *quant* = 0.25 (16th-note), then *minimum* = 2 means that nothing shorter than an eighth note will be produced. NOTE: If *minimum* is too high, there may not be a valid partition; in that case, it will fall back to the previous successful minimum, or to 1.

`\unis(`srcParm, "new items", srcBP)` Copy items, or rhythm, from another process ("unison"). The other process should have `leadTime > 0` to guarantee that it evaluates first (you can set `leadTime` while a process is playing). *SrcParm* is the parameter name from the other process, as a symbol `` `name ``; *srcBP* is the name of the other process. If you specify items in *new items*, they will replace the values from the other process. For a fairly complex example, let's play block chords in a pad, and then have an arpeggiator follow the top note's rhythm. The first use of `\unis()` here makes sure the arpeggiator plays the highest note (`1`) at the same time the pad moves to another chord. In `/arp..top`, the `\unis()` follows the top note's rhythm and pitch, and in `/arp..acc`, it adds an accent to the top notes only.

```
TempoClock.tempo = 124/60;
/make(anapadVC:pad/chordBP:ch(chords:\bigch));
/ch(leadTime = 0.01);
/ch = "\delta("*", 0.5, 0, 1, 2, 2)::\shuf("0976")";
/ch+

/make(pulseLeadVC:pl/arpegBP:arp(chords:\bigch));
/arp = "\unis(`note, "1", `ch)::\ins("*", 16, 0.25)::\seq("
    23456")::\artic(".")";
/arp..top = "\unis(`note, , `ch)";
/arp..acc = "\unis(`top, ">", `arp)::\choke(0.25, "-")";
/arp+

VC(\pad).v.gui; VC(\pl).v.gui;  // adjust filters
```

Listing 35: Using the unis() generator to coordinate two harmony processes.

6.4.2 Wildcard-replacement generators

Note that `\seq()`, `\ser()`, `\rand()`, `\wrand()`, `\xrand()`, and `\shuf()` are based on the corresponding SuperCollider patterns `Pseq`, `Pser`, `Prand` etc., particularly for the number of repeats (generally a number of single values to return, except for `\seq()`).

`\seq("items", "wildcards", reset)` Replaces *wildcards* in the source with *items*, one by one, preserving order. *Reset* is optional; if it's a number greater than 0, the item sequence will reset on every bar.

`\ser("items", "wildcards", reset)` Like `\seq()`, but the number of repeats determines how many notes will be output, not how many repetitions of the entire phrase. When used for wildcard-replacement, it is indistinguishable from `\seq()`. Used in a pool argument, however, compare the following (using `\y` defined in Listing 30):

```
/y = "\ins("*", 8, 0.5)::\seq("\seq*2..4("1234")\seq("875")
    ")";
/y+

/y = "\ins("*", 8, 0.5)::\seq("\ser*2..4("1234")\seq("875")
    ")";

/y-
```

Listing 36: Usage of \ser() generator.

`\rand("items", "wildcards")` Like `\seq()`, but chooses from *items* randomly. (*Reset* is not relevant, as there is no order to preserve.)

`\wrand("items", "wildcards", weight0, weight1, weight2...)` Weighted random selection, like `Pwrand`. *Weight0* is associated with the first element

of *items*; *weight1* with the second, and so on. The generator automatically does `normalizeSum` on the weights, so you don't have to worry about making them add up to 1.0. Do not enclose the weights in array brackets. (As in `\rand()`, *reset* is irrelevant.)

`\xrand("items", "wildcards", reset)` Reads the items in random order without repeating the same item twice in a row, like `Pxrand`.

`\shuf("items", "wildcards", reset)` Shuffles the items into random order, and returns each one before choosing a new order, like `Pn(Pshuf(items, 1), inf)`.

`\pdefn(`pdefnKey, "wildcards")` Like `\seq()`, but obtaining replacement items from a `Pdefn`. For non-pitched parameters, the `Pdefn` should yield characters corresponding to `parmMap` items. For pitched parameters, it should yield e.g. `SequenceNote(degree, nil, length)` where `length` is 0.4 for staccato, 0.9 for legato (but rearticulating the next note) and 1.01 for slurred.

- `Pdefn` streams are shared globally across all instances of this generator *for this process*. With care, you can create sequential patterns spanning barlines.
- The behavior of `reset > 0` is undefined.

```
Pdefn(\y, Pn(Pseries(0, 1, 8), inf).collect { |d|
    SequenceNote(d, nil, 0.9) });

/y.a0 = "[*]::\ins("*", 2, 0.5)::\pdefn(`y, "*")";
/y.a1 = "\ins("*", 3, 0.5)::\pdefn(`y, "*")";
/y = (a**2);
```

Listing 37: Usage of \pdefn() generator.

`\gdefn(`pdefnKey, "wildcards")` Like `\pdefn()`, with a difference: `\pdefn()` keeps an independent repository of streams for every BP, while `\gdefn()` keeps a global repository shared among all BPs. *However*… be careful; this does not work the way you think. *Cll* generates *all* of the events for one phrase at one time. In the following example, you might expect values to alternate between `/x` and `/y`. Instead, four values are assigned to `/x` first, and then four to `/y`, and these are streamed out over the course of one bar.

```
(
BP(\x).free;
PR(\abstractLiveCode).chuck(BP(\x), nil, (
   event: (play: { ~x.debug(~collIndex) }),
   defaultParm: \x,
   parmMap: (
      x: ($0: 0, $1: 1, $2: 2, $3: 3, $4: 4)
```

```
    ),
));

BP(\y).free;
PR(\abstractLiveCode).chuck(BP(\y), nil, (
    event: (play: { ~x.debug(~collIndex) }),
    defaultParm: \x,
    parmMap: (
        x: ($0: 0, $1: 1, $2: 2, $3: 3, $4: 4)
    ),
));

BP(\x).leadTime = 0.01;
Pdefn(\x, Pseq("01234", inf).trace(prefix: "pdefn: "));
)

/x = "[****]::\gdefn(`x, "*", 1)";
/y = "[****]::\gdefn(`x, "*", 1)";
/x/y+
/x/y-
```

Listing 38: Usage of \gdefn() generator.

Expected x	Expected y	Actual x	Actual y
0	1	0	4
2	3	1	0
4	0	2	1
1	2	3	2

`\prev("items", "wildcards", staccDur)` Replaces any *wildcards* with either: one of the *items* (which is equivalent to `\rand()`; generally, you should leave *items* empty), or the previous item found in the bar. Useful for repeating notes. For pitched parameters, if the repeated note's eventual duration `<= staccDur`, its articulation will be made staccato. NOTE: If a wildcard is the first item in the bar, then there is no previous item. In this case, `\prev()` will try to determine the last item from the previous bar. If this fails, a warning will be posted and the wildcard will become a rest (silent).

`\repeat1("items", "wildcards", reset, repeats)`: Repeats every value from the `"items"` stream. `\repeat1(\seq("1234"), "*", , 2)` would output 1, 1, 2, 2, 3, 3, 4, 4 etc. the number of repeats may be randomized.

`\pitch(src, "wildcards", "intervalPool", fallbackMin, fallbackMax, "articPool")` Valid for pitched parameters only. Replaces *wildcards* with "Brownian motion" melody. Each wildcard will search backward for the previous note. Then, one of the intervals will be chosen from *intervalPool* and added to the previous note. Intervals are written relative to the scale root: `1` is

unison (no movement), 2 is up a second, 8 is up an octave, and 6, is a third down (e.g., in C major, 6 is up a sixth = A, and 6, lowers this by an octave to be the A a third below the root). If there is no previous note, a pitch will be chosen randomly between *fallbackMin* and *fallbackMax* (given as integers, where 0 = root). For each note, and articulation will be chosen randomly from *articPool*; if omitted, new notes will use the default "unslurred, legato" articulation.

`\pitch2(src, "wildcards", "intervalPool", fallbackMin, fallbackMax, "articPool", connect)`

Like `\pitch()`, but it remembers the previous pitch *that it generated*, instead of looking back for every wildcard. This may be used to produce multiple, interlocking lines. NOTE: This generator resets every bar; unconstrained Brownian motion is risky.

```
TempoClock.tempo = 140/60;
Mode(\cphr) => Mode(\default);

/make(pulseLeadVC:pl/melBP:pl);

// Every "6,7," descending pattern starts at the previous
    given note
/pl = "[5'>| 3'>|4'>  2'>|]::\ins("*", 5..10, 0.25)::\pitch
    ("*", "6,7,", 8, 10, ".")";
/pl+

// The first descent starts from "5'" and keeps going down
    through the bar
/pl = "[5'>| 3'>|4'>  2'>|]::\ins("*", 5..10, 0.25)::
    \pitch2("*", "6,7,", 8, 10, ".")";

// One \pitch() stream;
// another interlocking \pitch2() stream in a higher
    register
/pl = "[5'>| 3'>|4'>  2'>|]::\ins("*", 3..7, 0.25)::\ins("@
    ", 3..8, 0.25)::\pitch("*", "6,7,", 8, 10, ".")::
    \pitch2("@", "6,7,23", 14, 18, ".", 0)";

/pl-
```

Listing 39: Usage of \pitch() and \pitch2() generators.

Note also that `\seq()`, `\rand()`, `\wrand()`, `\xrand()`, `\shuf()` and `\repeat1()` may be used in (or as!) pool strings (Listing 40).

```
(
BP(\y).free;
PR(\abstractLiveCode).chuck(BP(\y), nil, (
   event: (eventKey: \default),
   defaultParm: \degree,
   parmMap: (degree: (isPitch: true))
```

```
));
)

TempoClock.tempo = 140/60;

/y = "\ins("*", 8, 0.5)::\seq("123")";

/y+

// can repeat (like Prand)
/y = "\ins("*", 8, 0.5)::\rand("\seq("123")\seq("7854")\seq("
    26,")")";

// no repeats (like Pxrand)
/y = "\ins("*", 8, 0.5)::\xrand("\seq("123")\seq("7854")\seq("
    26,")")";

// 2-5random notes after 6,
// note here that articulation/transposition applies to sub-
    choices
/y = "\ins("*", 8, 0.5)::\xrand("\seq("123")\seq("7854")\seq("
    26,\xrand*2..5("34567")::\xpose("1'")::\artic(".")")")";

// also, articulations and transposition can be streamed this
    way
/y = "\ins("*", 8, 0.5)::\seq("123")::\artic(\seq("\seq*3("_")
    \seq*7(".")"))";

/y-
```

Listing 40: Complex sequencing with sub-generators.

6.4.3 Modifier generators

Modifiers may be part of the main generator chain, or included in subsequences (Listing 40).

`\artic("articPool", "wildcards")` Replaces matching notes' articulation with one of the items randomly chosen from *articPool*. If no *wildcards* are given, all notes match. Otherwise, only those notes with an association to one of the given *wildcards* will match.

```
TempoClock.tempo = 140/60;
Mode(\cphr) => Mode(\default);
/make(pulseLeadVC:pl/melBP:pl);

// No wildcards, all notes match: All are staccato
/pl = "[1574]::\artic(".")";
/pl+
```

```
// Wildcard, but none of the notes came from a
// wildcard operation, so none of them match.
/pl = "[1574]::\artic(".", "*")";

// Notes were inserted by \seq operating on "*";
// all notes match.
/pl = "[****]::\seq("1574")::\artic(".", "*")";

// Two layers of notes with different articulations
/pl = "[****]::\seq("1574")::\ins("@", 4..8, 0.25)::\shuf("
    1'2'3'4'5'6'", "@")::\artic(">.", "*")::\artic("~", "@"
    )";

/pl-
```

Listing 41: Usage of \artic() generator.

`\xpose("intervalPool", "wildcards")` Transposition. For each *wildcard*-matching note, choose an interval from *intervalPool* and transpose the note accordingly.

```
TempoClock.tempo = 140/60;
Mode(\cphr) => Mode(\default);
/make(pulseLeadVC:pl/melBP:pl);

/pl = "\ins("*", 16, 0.25)::\seq("12345432")";
/pl+

// Easy octave displacement.
/pl = "\ins("*", 16, 0.25)::\seq("12345432")::\xpose("
    1111,1'1''")";
/pl-
```

Listing 42: Usage of \xpose() generator.

6.4.4 Filter generators

`\fork("timed generators")` Applies different generators to different segments of the bar. For instance, the *source* could insert *n* wildcards throughout the bar, while *timed generators* could replace wildcards in the first half of the bar with one value, and a different value in the second half. Here, *timed generators* includes two items, and `\fork()` occupies the entire bar. So both `\seq()` instances get half a bar. Source items in any portions of the bar not covered by one of the *timed generators* will pass through unchanged.

```
/y = "\ins("*", 10, 0.25)::\fork("\seq("13", "*")\seq("14",
     "*")")";
```

```
/y = "\ins("1,", 10, 0.25)::\fork(" \seq("13", "1,")x\seq("
    14", "1,")")";
```

Listing 43: Usage of \fork() generator.

`\chain(generator, generator...)` For internal use only.

`\delete("wildcards", probability)` Find existing items produced by a given wildcard, and keep those items based on probability. 0.75 probability means a 3/4 chance of keeping the item; I felt that a higher probability should result in a more dense texture. If the coin toss determines that the note should not be kept, it is removed and its timepoint will be empty.

`\wildcards("wildcards")` If an item had been produced by wildcard replacement, "undo" this: replace the real value with the original wildcard, so that a future wildcard replacement will affect this item. This is especially useful when using `\unis()` to copy data from another process: you can overwrite the other process's data, but keep the wildcards, and then substitute new information in its place.

6.5 Writing new generators

Generators inherit from `PR(\clGen)`.[15] They should implement:

`~prep` Validate the entries in the `~args` array, and return the `Proto` object by finishing with `currentEnvironment`. In general, start with `~baseItems = ~args[0]`.

`~process` Generally begins with `~items = ~getUpstreamItems.();`. Following this, manipulate the `~items` array and return it at the end. Be careful to `copy` or `collect` the array (to avoid corrupting `~baseItems`) and—important!—if you modify any of the items, be sure to copy it first.

Generators should take care to respect their time span, given by `~time` (the generator's onset within the bar) and `~dur` (the number of beats occupied by this generator). Do not modify any items outside this time span. See the definition of `PR(\clGenRot)` for an example.

`~baseItems` and `~items` are arrays of Events, containing:

`item` The entry to be played. For non-pitched parameters, these will generally be characters. Otherwise, pitch strings are parsed into `SequenceNote` objects.

`time` The event's onset time within the bar. This is relative to the bar line, not the generator's onset time.

[15] In `Proto`, inheritance is handled by "cloning" the Proto: `PR(\clGen).clone { ... overrides... }`.

`dur` The number of beats until the next event. This may not be reliable during processing. The top-level generator will correct the `dur` values before streaming out the events.

This documentation may be expanded at a later date.

7 Graphical interface windows

7.1 Code window features

The GUI code window has two advantages for *cll* over the SuperCollider IDE:

- The IDE's syntax highlighter doesn't understand *cll*. You may get incorrect highlighting or indentation with *cll* statements.
- The default font size is larger, for projection in performance.

One limitation: Because of a SuperCollider bug,[16] this window can evaluate single lines only. If you select a block of code to evaluate at once, the selection is ignored and only the line containing the cursor will be executed.

The "autosave" button at the top will save the contents to disk automatically when the window is closed. The file location is `Platform.userAppSupportDir +/+ "cll-sketches"` and files are automatically named by date and time. (I preferred autosave over a standard save dialog specifically because it's a GUI window rather than a code document.) The "load" button allows you to choose a file from `cll-sketches`.

If the cursor is on a 5.2, pressing alt-shift enters an intelligent navigation mode, where arrow keys move through syntactic elements rather than characters:

- Left and right arrows move to the previous or next sibling.
- The up arrow expands the selection to the syntactic unit containing the current element.
- The down arrow contracts the selection to one of the units contained within the current element.

Press alt-shift again when the selection is what you want, to exit navigation mode and retain the selection.

It takes a little time to learn to use this fluently, but it's very effective, especially for selecting entire generators at once. (Note also that, at present, the set pattern statement must be complete and free of syntax errors.)

[16] `https://github.com/supercollider/supercollider/issues/3279`

7.2 Controller window features

The controller window divides into two or three parts: controls, a list of processes and instruments, and an optional cheatsheet.

7.2.1 Controls

The default layout is based on a Korg Nanoktl. (If a Nanoktl is plugged in and configured with the default controller numbers, the hardware knobs and sliders are connected automatically.) You may also call `/cllGui.(\mix16Touch)` for a layout based on TouchOSC's Mix16 preset.

Buttons behave as toggles. Sliders and knobs should be self-explanatory.

Note the slight shading at the top of each control. This is to overlay a text label. Unfortunately, I cannot find a way to add a label and make the widget respond over its entire area. So, be careful to click in the unshaded area at the bottom.

7.2.2 Process list

The process list collects all playable BPs ("bound processes") and VCs ("voicers").

Any item in the list can be controlled-dragged into buttons:

- BPs: the toggle button is for play/stop and the associated fader/knob is for mixing.
- VCs: the toggle button is for mute and the fader is for mixing.

If a BP defines global controls, it will have a + to the left of its name. Right-arrow expands to show the controls that belong to it; left-arrow hides them. Individual controls can be dragged into a button, where they will be attached to the associated fader.

You can use the keyboard to "throw" a list entry up into the GUI. Both `\nanoTouch` and `\mix16Touch` have 16 button/fader pairs, identified as 0-9 and A-F. To "throw," type `^` and the index character (e.g. `^0` for the first one).

Backspace in the list will delete an entry if possible. A BP can be deleted only if it is not playing; a VC, only if it is not attached to an existing BP player.

7.2.3 Cheatsheet

You can write notes to yourself about available instruments and players and save them under `Platform.userAppSupportDir +/+ "cll-cheatsheet.txt"`. If this file exists, its contents will be displayed at the bottom of the controller window. (You can edit the text here and save as well.) This is free text, only for your convenience.

8 Extending cll

cll is designed to be extensible: adding new statements is relatively straightforward.

Processing a *cll* statement goes through two main steps:

1. `PR(\chucklibLiveCode)` tests the statement against a number of regular expressions, to determine what type of statement it is.
2. Then, a `PR` object to handle the statement is instantiated, and the statement is passed to that object's `process` method.

So, to implement a new statement type, you need to do two things, matching the above stages.

8.1 Statement regular expression

First, add a statement ID and regular expression into `PR(\chucklibLiveCode)`. Within this object, `~statements` is an array of Associations: `\statementID -> "regexp"`.

```
~statements = [
   \clMake -> "^ *make\\(.*\\)",
   \clFuncCall -> "^ *`id\\.\\(.*\\)",
   \clPassThru -> "^ *([A-Z][A-Za-z0-9_]*\\.)?`id\\(.*\\)",
   \clChuck -> "^ *([A-Z][A-Za-z0-9_]*\\.)?`id *=>.*",
   \clPatternSet -> "^ *`id(\\.|`id|`id\\*[0-9]+)* = .*",
   \clGenerator -> "^ *`id(\\.|`id)* \\*.*",
   // harder match should come first
   \clXferPattern -> "^ *`id(\\.`id)?(\\*`int)? ->>",
   \clCopyPattern -> "^ *`id(\\.`id)?(\\*`int)? ->",
   \clStartStop -> "^([/`spc]*`id)+[`spc]*[+-]",
   \clPatternToDoc -> "^ *`id(\\.|`id)*[`spc]*$"
];
```

Listing 44: Cll statement regular expression templates.

More restrictive matches should come first. For instance, `\clXferPattern` comes before `\clCopyPattern`. If they were reversed, `->` in the "copy" regular expression would match the "xfer" statement as well as the "copy" statement. Checking `->>` first ensures that the more permissive test takes place only after the stricter test fails.

Within these strings, a backtick (`` ` ``) introduces a macro that will be expanded into part of a regular expression. Available macros are:

```
~tokens = (
   al: "A-Za-z",
   dig: "0-9",
   id: "[A-Za-z][A-Za-z0-9_]*",
   int: "(-[0-9]+|[0-9]+)",
```

```
        // http://www.regular-expressions.info/floatingpoint.html
        float: "[\\-+]?[0-9]*\\.?[0-9]+([eE][\\-+]?[0-9]+)?",
        spc: "    "  // space, tab, return
    );
```

Listing 45: Regular expression macros for SC language tokens.

You should match only as much of the syntax as you need to determine the statement type. This is not the place for syntax validation. For example, the `\clGenerator` statement has a fairly complex syntax, but the matching regular expression is looking only for one or more IDs separated by dots, followed by a space and then an asterisk. This will dispatch to `PR(\clGenerator)`; it is this object's responsibility to report syntax errors (generally by throwing descriptive `Error` objects).

> **Note:** The leading slash is stripped from the statement before regular expression matching. Don't include the slash in your regular expression.

8.2 Handler object

Usually, a statement handler is a `PR` object, containing a `Proto` object prototype. The `PR`'s name must match the statement ID created in the last step.

The `Proto` must implement `process`, which takes `code` (the statement, as a String) as its argument. It should return a string containing the SuperCollider language syntax to perform the right action.

```
Proto {
    ~process = { |code|
        // parse 'code' and build the SC language statement(s)...
        translatedStatement  // return value
    };
} => PR(\clMyNewStatement);
```

Listing 46: Template for cll statement handlers.

Very simple statements may be implemented as functions added into `PR(\chucklibLiveCode)`.

```
PR(\chucklibLiveCode).clMyNewStatement = { |code|
    // parse 'code' and build the SC language statement(s)...
    translatedStatement  // return value
};
```

Listing 47: Adding a function into PR(\chucklibLiveCode) for simple statement types.

9 Code examples

10 Typesetting

Typeset by TeX Live 2017; Edited in Emacs 24.3.1 (Org mode 8.3beta).