Chucklib-Livecode Manual

H. James Harkins

October 23, 2015

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.

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

This document will cover the process prototype first. You need to understand its structure in order to understand the commands.

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.

[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`

[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.

- Thor Magnusson, whose *ixilang* system provided some of the inspiration for *cll* syntax.

2 Installation

First, you must install the "ddwChucklib" quark and its dependencies. Consult Quarks documentation for details.

cll's functionality is contained in three files:

`preprocessor.scd` Installs the preprocessor to convert *cll* statements into SuperCollider language syntax.

`preprocessor-generators.scd` Installs *generator* objects (Section 5).

`helper-functions.scd` A handful of useful `Func` definitions.

To use *cll* in a session, simply load the `scd` files in the interpreter. If you place the three files into your user application support directory (run `Platform.userAppSupportDir` to find out where that is), then you can load them easily by:

```
[  "preprocessor.scd",
   "preprocessor-generators.scd",
   "helper-functions.scd"
].do { |name|
   (Platform.userAppSupportDir +/+ name).load;
};
```

Listing 1: Instructions to load cll into a SC language session.

Note: Automatic installation of quarks with dependencies is not supported in SC 3.6.x in Windows. You will have to install the individual quarks by hand, by copying them into the Extensions directory.

- ddwChucklib
- ddwPrototype
- ddwCommon
- ddwGUIEnhancements
- ddwMixerChannel
- ddwPatterns
- ddwTemperament
- cruciallib

3 Process prototype

3.1 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 4.2).
- 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.

3.2 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`.

`defaultParm` The name of the default parameter affected by *Set pattern* statements (Section 4.2). 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.

```
(
PR(\abstractLiveCode).chuck(BP(\beep), 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)
   )
));
)

// Use it, with cll statements:
TempoClock.tempo = 2;

/beep = "^|.. .| .- | .  ";  // "Set pattern"
/beep+;  // start it

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

/beep-;

/beep(free);
```

Listing 2: A simple cll process.

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 2, `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.

Note: Clearly, the code to initialize the process in Listing 2 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 4.5), 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.)

3.3 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 3: 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 4.2.4).

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.

3.3.1 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 4, 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 4: 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—after which, array handling in the *cll* parameter map requires a third level of wrapping.

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 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 5: Arrays for multiple-parameter setting using one cll parameter.

3.4 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.

3.5 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

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 processess	`/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`

Table 1: List of available chucklib-livecode statements.

pattern stored as `phraseSeq`. "Set pattern" has a compact way to express phrase sequences, allowing sequences, random selection (with or without weights) and wildcard matching. See Phrase selection for details (Section 4.2.5).

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.

4 Livecoding statement reference

4.1 Statement types

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 6: Cll statements, one by one or as a batch.

cll supports the statements shown in Table 1, in order of importance.

4.2 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*.

4.2.1 Phrase definition

Most "Set pattern" statements follow this format:

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

Listing 7: 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.

4.2.2 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.

Note: *Set pattern* writes the character identifiers for the values into the pattern: for example, a pattern string `"--"` becomes `Pseq([$-, $-], 1)`. `PR(\abstractLiveCode)` post-processes each parameter, ensuring that the right event keys receive the right values. The conversion from identifier value occurs for each parameter; you should be able to rely on accessing the final values by Pkey. This supports *Generators* (Section 5), which should also return the value identifiers.

1. "--"
2. "----"
3. "- --"
4. "- --- -"
5. "-|-|-|-"
6. "-|--|-|-"
7. "--| - |- --| -"

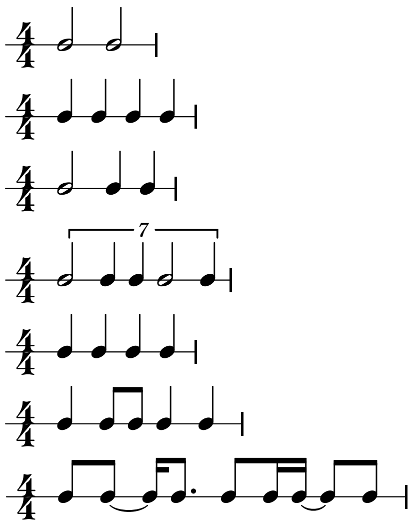

Figure 1: Some examples of cll rhythmic notation, with and without dividers.

4.2.3 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 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 8: Multiple parameters with different timing.

4.2.4 Pitch notation

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

- Scale degrees are given by decimal digits, where 1 is the tonic and 0 is the interval of a tenth above that (following the number row on the keyboard).[4]
- `+` and `-` raise and lower the pitch by a semitone.
- `'` and `,` displace the pitch by an octave up or down, respectively.[5] Multiple apostrophes or commas displace by multiple octaves. (This syntax is borrowed from LilyPond.)
- `.` indicates a staccato note.
- `_` indicates legato (sustain duration slightly shorter than note duration).
- `~` slurs this note into the next note.

Note: You should use the default event prototype for this process. Include the following in the "chuck" parameter dictionary, as in Listing 9:
`event: (eventKey: \default)`

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).

Listing 9 illustrates the kind of articulation that is possible with this notation, using a 90s-throwback acid-style bassline. Though the sound is not as cool as a real TB303, careful use of slurs and staccatos mimics the feel of the venerable old machine.[6] A further refinement would be to add values for filter frequency and `filtMul` into the parameter map.

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

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

[6]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.

```
(
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;

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

4.2.5 Phrase selection

Statements to set the phrase sequence follow a different syntax:

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

Listing 10: 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 11: Nested phrase-selection groups.

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)`.

4.3 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 (Section 4.11).

4.4 Randomizer statement

Randomizers create randomized variations on a given string:

```
/proc.prefix.parm *n +ki %q "string"
```

Listing 12: Syntax template for randomizer statement.

`proc` The process into which the new variations will go.

`prefix` A phrase identifier. *Mandatory.*

`parm` (optional) The parameter to control.

`n` The number of variations to create. Each becomes a new phrase: `prefix0`, `prefix1` up to $n - 1$.

`k` The number of sequence items to add.

`i` The sequence item: either a single character (defined in the parmMap) or the name of a Func, with a leading backslash `\`.

`q` (optional) The quantization factor, determining where in the bar the new notes may be placed.

`string` A template, providing items and rhythms that should be constant over all variations. You may use an existing pattern string from any process by omitting the quote marks and substituting `phrase.parm` (if the template comes from the same process) or `/proc.phrase.parm` (if it comes from a different process).

Note: At present, the string must contain vertical-bar dividers (|). I may remove this limitation in a future version. For now, passing a string without dividers will cause an error.

The randomizer's algorithm is:

```
// assuming BP(\snr) defines:
// "-" (normal note)
// "." (softer note)
// Produces strong notes on 2 and 4, and one note elsewhere
/snr.a *10 +1. "|-||-";
/snr = ('^a');  // randomly choose one variation for each bar

// "-" = open, "." = closed
/hh = "..|..|..|..";  // all closed at first

// add an open HH on any empty 16th
/hh.a *10 +1- main;  // "main" refers to the above
/hh = ('^a');

// totally random HH rhythm (probably sounds stupid)
{ "-.".wchoose(#[0.16, 0.84]) } => Func(\randHH);
/hh.b *10 +9\randHH "|||";

// or random notes on 8ths
/hh.b *10 +5\randHH %0.5 "|||";

// or, random notes, but don't allow two "-" in a row
(
{ |prev|
   if(prev == $-) { $. } {
      "-.".wchoose(#[0.16, 0.84])
   }
} => Func(\randHH);
)

/hh.b *10 +9\randHH "|||";
```

Listing 13: Examples of randomizer statements.

1. Use `q` to determine the valid time points at which to place notes. In 4/4 time, with the default `q` = 0.25, there will be 16 time points.
2. Evaluate the string, to find out where notes already exist. Remove these time points from the available list.
3. Randomly choose `k` time points, and add `i` at each of these points.
4. Write the results into a pattern string, and call the *Set pattern* statement (Section 4.2) to add the pattern into the process.
5. Do the above `n` times.

4.4.1 Functions as items

Normally, `i` is simply a character indicating a specific value from the parameter map. If you want the item itself to be randomized, define a function to calculate the random value, save it in a *chucklib* `Func`, and use the `Func`'s name in place of the item.

For each new item, the `Func` will be passed two arguments: the item before the randomly-chosen time point (or nil) and the item after the time point (or nil). You may add other arguments, in parentheses, after the function name; e.g. `+3\myRand(1, 3)` would call `\myRand.eval(prev, next, 1, 3)`.

4.5 Make statement

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

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

Listing 14: 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.

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,
make: { |name|
   PR(\abstractLiveCode).chuck(BP(name), nil, (
      event: (eventKey: \default),
      defaultParm: \degree,
      parmMap: (degree: (isPitch: true))
   ));
}, type: \bp) => Fact(\demoBP);
)

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

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

/dm(free);
```

Listing 15: Example of the make statement.

4.6 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 16: Syntax template for passthrough statements.

4.7 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 17: Syntax template for Chuck statements.

4.8 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 (Section 4.11). No syntax checking is done in the preprocessor.

```
/func.(arguments);

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

Listing 18: 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.

4.9 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 19: 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.[7] 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.

4.10 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.

```
/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 20: 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 in SC 3.6.x (as it uses new features introduced in SC 3.7).

4.11 Helper functions

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 4.8).

`/setupbars.(\proc, n, \prefix)` Create empty phrases for `prefix0`, `prefix1` up to $n - 1$. This also inserts *Set pattern* (Section 4.2) templates into the code document, for you to start filling in musical material.

[7]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 4.2.5). It is not guaranteed to work with hand-written patterns that generate phrase names algorithmically.

`/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 21: 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 Generators

The basic syntax of the *Set pattern* statement (Section 4.2) 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.

5.1 Generator usage

5.1.1 Generators and pattern strings

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

As noted earlier, every character in a pattern string corresponds to a metrical position within the bar.[8] The entire generator string, from the opening backslash to the closing parenthesis, likewise occupies *one and only one* metrical position. The generator remains active until the next event, which may be a literal item or another generator. Spaces in the pattern string are placeholders, and indicate how long the generator should be in force. Listing 22 illustrates. (Argument lists call for further discussion and are not relevant to generators' rhythmic position; so, the examples omit arguments.)

In example 3 of Listing 22, beat 2 contains four items: space, `\rand(...)`, space and space. Thus beat 2 is subdivided into 16th-notes, and the generator begins on the second of those.

[8]The exception is pitch, where a scale degree number may be followed by accidental, octave and articulation designations. In this case, for instance, the four characters `4,+.` make up a single metrical instant.

```
// 1. \rand starts on the downbeat and occupies the whole bar.
/bp = "\rand(...)";

// 2. \rand starts on beat 2
/bp = "|\rand(...)||";

// 3. \rand starts on the 2nd 16th-note of beat 2
/bp = "| \rand(...)  ||";

// 4. \rand starts on the 2nd 16th-note of beat 2
// and stops on the 'and' of 4
/bp = "| \rand(...)  || x";
```

Listing 22: Interaction between generator syntax and "set pattern" rhythmic notation.

5.1.2 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. Each argument should be one of the following:

- A *quoted* string containing items to use for subsequent events. An "item" may be a single character or a generator; if the pattern string is for a pitch parameter, the item may consist of more than one character (including octave, accidental and articulation modifiers). Quotes for these 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.

- A subordinate generator expression (which must begin with a backslash and end with a closing parenthesis).

- An *unquoted* string, consisting of a value that will be interpreted. Especially useful for literal numbers (0.25) or numbers resulting from calculations (1/4).

See Listing 23 for examples.

Each generator object defines its own expected inputs. The built-in generators are documented below; other generators need not conform to these specifications.

5.2 Generators and rhythm

Every process must have a primary rhythm. In chucklib-livecode, the primary rhythm comes from the *default parameter* (see Section 3.2). The process will generate events at the metrical positions of notes given for the default parameter, and not at other times.

A generator given to the default parameter must, therefore, produce rhythm. It can do this in one of two ways:

- The generator can yield two-element arrays `[item, delta]`. The second will become the process's rhythm as long as the generator is active.
- Or, a rhythm generator can be added to a simple-value generator, by using a colon in place of the comma that normally separates arguments. The backslash is omitted after this colon. This generator should return numbers, to be used as event deltas. (The value and rhythm generators will be wrapped in a `Ptuple`, returning the required `[item, delta]` arrays.)

A rhythm generator is optional for non-default parameters. If one is omitted, the value generator will be called according to the default parameter's rhythm. Otherwise, non-default parameters use rhythm to "sample-and-hold" their values. Adding a rhythm generator to a value generator for a non-default parameter wraps both generators within a `Pstep`; the generator returns only the items, polling the sample-and-hold signal at times determined by the default parameter's rhythm.

Listing 23 uses the example process from Listing 2. Note that the last example produces an error message: *ERROR: Generator's output shape must match parm: [amp, dur]*. This tells you that rhythm is required for this parameter, because it's controlling both the default parameter name `amp` *and* timing by `dur`.

```
// non-default parameter, generates values per main rhythm
/beep..pan = "\xrand("<(-)>")";

// non-default parameter, sample-and-hold once per quarter
/beep..pan = "\xrand("<(-)>":randRhy(1, 1))";

// easier to hear with sequence
/beep..pan = "\seq("<>":randRhy(1, 1))";

// default parameter with rhythm generator
// beginning the pattern string with '^' means
// every bar will have a strong downbeat
/beep = "^\rand("-.":randRhy(0.25, 2, 1, 1))  |||";

// switch to triplets, weight the return items
/beep = "^\wrand("-.", 1, 3:randRhy(1/3, 2, 1)) |||";

// default parameter, no rhythm: Error
/beep = "^\rand("-.")  |||";
```

Listing 23: Examples of generators, with rhythm handling.

5.3 Built-in generators

The following generators are defined in *cl-livecode*:

`\seq("items")` Reads the items in sequence, like `Pseq` (with `inf` repeats).

`\rand("items")` Reads the items in random order, like `Prand`.

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

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

`\wrand("items", weight0, weight1, weight2...)` Weighted random selection, like `Pwrand`. 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.

`\randRhy(base, weight1, weight2, weight3...)` Random rhythm generation. `weight1` is the probability of returning `base * 1`; `weight2` is the probability of `base * 2` and so on. Typically used after a colon (and *without* the leading backslash).

Two additional generators are defined: `\clStepGen` and `\clRhythmGen`. These are for internal use. You should not override their behavior.

5.4 Writing new generators

Generators inherit from `PR(\clGen)`,[9] and may override the following:

`~asPattern` A function returning a pattern to evaluate. Generator arguments are available in the `~args` array. The pattern may implement any logic that is required.

- In `~args`, item strings are converted to arrays. (They cannot remain Strings, because they may contain values other than characters.) For example:
  - `"abc"` becomes `[ $a, $b, $c ]`.
  - `"abc\rand("def")"` becomes `[ $a, $b, $c, a Proto ]` (where the `Proto` represents the `\rand()` generator).
- Item strings for pitch parameters become arrays of `SequenceNote` objects: `SequenceNote(pitch, nil, legato)`. A pitch consists of a pitch number (usually scale degree) and articulation (staccato, portato or slurred). `SequenceNote` can maintain both values and allow basic math operations on the pitch number.

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

- If the generator is returning items *and* rhythm, it should ensure that the sum of the outgoing rhythm values is exactly ~dur. The easiest way is to generate an infinite-length pattern and wrap the time pattern: Pconst(~dur, timePattern). (Item-only generators do not need to worry about this; attaching a rhythm generator by a colon automatically applies Pconst.)

~protoID The name of the PR for this generator. This is required to render the generator's compile string. If this is not populated, the wrong generator object will be produced and it will not work as you expect. The name must be a symbol, \clGenName; substitute the name of your object for Name, and be sure it is capitalized. (Generator usage in *Set pattern* strings will take the lowercase form.)

~yieldsKeys Override this for generators that produce rhythm as well as the items to play. Simple value generators do *not* need to override this. Combination item-plus-rhythm generators should specify: ~yieldsKeys = { [~parm, \dur] }.

~patternStringKeys A list of Proto variable names to render in the compile string. By default, these are #[bpKey, args, dur, isPitch, parm]. In general, you should not need to override this.

The following Proto variables are populated by the *Set pattern* compiler:

~bpKey The name of the BP object to which the generator is assigned. The generator can access BP variables through BP(~bpKey).

~isPitch true if the parameter is for pitch, false otherwise.

~parm The parameter name.

Listing 24 defines a simple "pyramid" generator, which yields the first item, then the first two items, then the first three and so on. In a simple case like this, it is sufficient only to provide ~asPattern and ~protoID.

```
(
PR(\clGen).clone {
   ~asPattern = {
      Pn(Pser(~args[0], Pseries(1, 1, inf).asStream), inf)
   };
   ~protoID = \clGenPyr;  // note: *capital* Pyr
} => PR(\clGenPyr);
)

/beep..pan = "\pyr("-()<>")";  // lowercase in Set pattern
```

Listing 24: Defining a new "pyramid" generator object.

A final example, Listing 25, demonstrates a simple use of pitch in a generator by playing an ascending scale starting with a given note. The starting note is given as a quoted-string argument, which parses pitches as `SequenceNotes`. The first argument is `~args[0]`, and we are interested only in the first pitch in this argument, hence `~args[0][0]`. `SequenceNotes` respond to basic math operations, meaning they may be used in `Pseries`. Each subsequent value coming from this `Pseries` will add 1 to the previous `SequenceNote`, raising the pitch by one scale degree. `:randRhy()` specifies 16th-notes.

Because each generator runs until the next event in the enclosing pattern string, true rests—the `x` characters—indicate the generators' durations. The first rest occurs on the second 16th of beat 2; this allows the first generator to play through the first 16th of that beat, for five 16ths in total.

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

PR(\clGen).clone {
   // ~args[0] comes from e.g. "1", which becomes
   // [SequenceNote(0, nil, 0.9)]. So ~args[0][0]
   // is the starting note.
   ~asPattern = { Pseries(~args[0][0], 1, inf) };
   ~protoID = \clGenUpscale;
} => PR(\clGenUpscale);
)

/pgen = "\upscale("1":randRhy(0.25, 1))| x  |\upscale("6,":
    randRhy(0.25, 1))|   x";

/pgen+;
/pgen-;
```

Listing 25: A pitch generator definition, with rests for duration.

6 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.

6.1 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 26: 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 27: 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.

6.2 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 28: 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 29: Adding a function into PR(\chucklibLiveCode) for simple statement types.

7 Code examples