Chucklib-Livecode Manual

H. James Harkins

October 15, 2016

Contents

1 Introduction **2**
1.1 Overview 2
1.2 Acknowledgments 2

2 Installation **3**
2.1 Installation with *git* 3
2.2 Installation without *git* 3
2.3 Running *cll* in a session 4

3 Process prototype **4**
3.1 Data structure 4
3.2 PR(\abstractLiveCode) 5
3.3 Parameter map 7
3.4 Event processing 9
3.5 Phrase sequence 9

4 Livecoding statement reference **9**
4.1 Statement types 9
4.2 Set pattern statement 10
4.3 Start/stop statement 16
4.4 Randomizer statement 16
4.5 Make statement 19
4.6 Passthrough statement 19
4.7 Chuck statement 20
4.8 Func call statement 20
4.9 Copy or transfer statement 20
4.10 Show pattern statement 21
4.11 Helper functions 22

5 Generators **22**
5.1 Generator design 23
5.2 Generator usage 24
5.3 Generators and rhythm 26
5.4 Built-in generators 26
5.5 Writing new generators 28

6 Extending cll **29**
6.1 Statement regular expression 29
6.2 Handler object 31

7 Code examples **31**

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.

[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`

- 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.

2 Installation

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

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.

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`

[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.

Additionally, ddwVoicer (`https://github.com/jamshark70/ddwVoicer`) is not strictly required, but is useful for instrumental sounds.

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.
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 three 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).
- `\loadAllCl.eval`: Load both of these at once.

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 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.

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.

```
(
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 1: A simple cll process.

In Listing 1, `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 1 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 2: 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 3, `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 3: 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 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 4: 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 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.

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.

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

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

Listing 5: 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 6: 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.

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

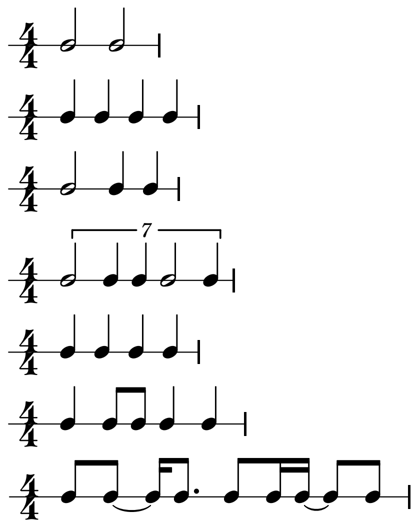

Figure 1: Some examples of cll rhythmic notation, with and without dividers.

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.

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 7: 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).[5]
- `+` and `-` raise and lower the pitch by a semitone.
- `'` and `,` displace the pitch by an octave up or down, respectively.[6] 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.

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

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

```
(
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 8: A retro acid-house bassline, demonstrating pitch notation.

Note: You should use the default event prototype for this process. Include the following in the "chuck" parameter dictionary, as in Listing 8:
`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 8 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.[7] A further refinement would be to add values for filter frequency and `filtMul` into the parameter map.

4.2.5 Phrase selection

Statements to set the phrase sequence follow a different syntax:

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

Listing 9: 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.

[7]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.

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 10: 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 11: 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:

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)`.

```
// 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 12: Examples of randomizer statements.

4.5 Make statement

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

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

Listing 13: 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 14: 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 15: 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 16: 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 17: 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 18: 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.[8] 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
```

[8]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.

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

Listing 19: 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.

`/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 20: 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 design

Generators manipulate lists of events, provided by "set pattern" strings, one bar at a time. (As such, they are not a precise analog to SuperCollider patterns.)

Typically, the generator's first argument is the event source: a "set pattern" string or another generator. The generator requests the event list from the source, processes it and passes the modified list up to its parent. Chaining generators in this way allows complex behaviors from simple units. Generators should be written such that it's possible to use any generator at any point in a chain.

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";

/y+;

/y = "\seq("** *| * *| ** |**", "12456", "*")";

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

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

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

```
/y-;
```

Listing 21: Isorhythmic cycles with generators.

Listing 21 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 (or, write the five possible distinct bars by hand—time moves quickly on stage, so this is painful). 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`.

Generators are "composed" by wrapping another generator around the outside: `\ins(..., "*", 7, 0.25)` inserts seven wildcards at randomly chosen 1/4-beat positions. (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. Wrapping in one more layer, another `\seq()`, 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.

The nested notation has the drawback that the parameters of outer-layer generators may be far away from the generator name. A double-colon "chaining" or "composition" operator, `::`, makes it possible to write each generator as an isolated unit. The final variant sounds the same as the nested version, but is easier to read. The `::` operator takes the result of the first `\seq()` and replaces the first parameter of the subsequent `\ins()` with it, and on down the chain. The initial comma inside `\ins()` is required as a placeholder, but nothing need be supplied; empty commas become `nil`.

5.2 Generator usage

5.2.1 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.[9] 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.)

[9]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.

In example 3 of Listing 22, 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.

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

/y+;

// 2. \rand starts on beat 2
/y = "1,|\rand(\ins("6,", "*", 3, 0.5), "13467", "*")||";

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

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

/y-;
```

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

Note: `\ins("source", "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 22, examples 3 and 4 offset the generator by one 16th-note—so `\ins()` will syncopate by a 16th. Further, source strings will be compressed to fit into the generator's duration. If `"|||"` at the outermost layer produces four divisions of one beat each, the same inside example 3's `\ins()` generator would divide 2.75 beats by 4, whereupon each division would consist of 11 64th-notes. These examples avoid the problem by supplying empty source strings. Otherwise, be careful (or, structure your music to take advantage of the resulting Nancarrow-esque polyrhythms).

5.2.2 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. 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).
- A number.
- A Symbol, written in LISP-influenced style with an opening backtick: `` `name``. Currently this is used only in the `\pdefn()` generator.

By convention, the first argument to any generator should be its source: a pattern string or generator. Generators may be freely composed if they follow this rule. Breaking the rule will result in combinations of generators that cannot be made to work. Other arguments are free for each generator to define.

5.3 Generators and rhythm

Previous *cll* versions used a "rhythm generator" to supply timing, when a generator was used for the default parameter. (As discussed in Section 4.2.3, the default parameter controls the rhythm of the entire process.)

Beginning with v0.3, *all* generators are timed according to the rhythm in the source string and any subsequent manipulation. There is no syntactic difference when using a generator in default or non-default parameters.

5.4 Built-in generators

5.4.1 Rhythm generators

`\ins("source", "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("source", "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("source", 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-;
```

5.4.2 Content generators

`\seq("source", "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.

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

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

`\shuf("source", "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("source", `pdefnKey, "wildcards", reset)` 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. This means 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);
```

5.4.3 Filter generators

`\fork("source", "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,")")";
```

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

5.5 Writing new generators

Generators inherit from `PR(\clGen)`.[10] They should implement:

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

`~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.

`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.

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 23: 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 24: 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 25: 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 26: Adding a function into PR(\chucklibLiveCode) for simple statement types.

7 Code examples

1 A simple cll process. 6
2 Template for the parameter map. 7
3 How to write arrays in the parameter map. 8
4 Arrays for multiple-parameter setting using one cll parameter. . 9
5 Cll statements, one by one or as a batch. 10
6 Syntax template for the Set pattern statement. 11
7 Multiple parameters with different timing. 13
8 A retro acid-house bassline, demonstrating pitch notation. 14
9 Syntax template for “Set pattern” phrase selection. 15
10 Nested phrase-selection groups. 16
11 Syntax template for randomizer statement. 16
12 Examples of randomizer statements. 18
13 Syntax template for make statements. 19
14 Example of the make statement. 19
15 Syntax template for passthrough statements. 20
16 Syntax template for Chuck statements. 20
17 Syntax template for func-call statements. 20

18 Syntax template for copy/transfer statements. 21
19 Demonstration of "Show pattern" statements. 21
20 Common initialization sequence, using helper functions. 22
21 Isorhythmic cycles with generators. 23
22 Interaction between generator syntax and "set pattern" rhythmic notation. 25
23 Cll statement regular expression templates. 30
24 Regular expression macros for SC language tokens. 30
25 Template for cll statement handlers. 31
26 Adding a function into PR(\chucklibLiveCode) for simple statement types. 31