Calimba: a language for computer music

Samuele Giraudo

LIGM, Université Gustave Eiffel

Meetup — Creative Code Paris

January 21, 2021

`https://github.com/SamueleGiraudo/Calimba`

`<</^\|_`

Outline

1. Overview

Main features

Under the hood of Calimba:

- Developed in Caml.
- About 3000 lines of code.
- Requires no particular sound servers or sound libraries.

Paradigm:

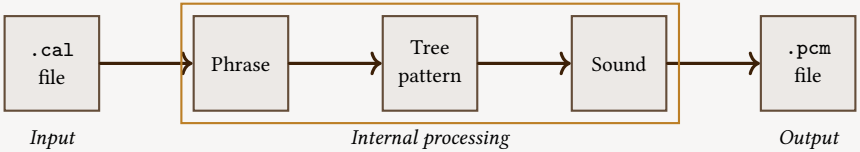
- Implements some functional programming features.
- Represents musical phrases by trees.
- Based on insertion operations on trees.

Features:

- Interprets and plays files specifying music.
- Embeds its own synthesizer, producing sound by additive synthesis.
- Can be used as a live coding environment.

From `.cal` files to sounds

Here is the production chain, constructing a music file from a program:



1. A `.cal` file is a text file written in the CALIMBA language.
2. A phrase is the abstract syntax tree in which the `.cal` file is translated.
3. A tree pattern is a treelike representation of a phrase.
4. A sound is a functional data structure representing a signal.
5. A `.pcm` file (**P**ulse **C**ode **M**odulation) contains a raw signal which can be played (for instance) by `aplay`. It has a sampling rate of 48000 Hz and a depth of 32 bits.

First view of a CALIMBA file

```
put layout = 2 2 2 2 2 2 in
put root = 0 12 -2 in
put time = 2 1 in
put duration = 200 in
put synthesizer = 0.15 0.21 3000 20 100 in

put delay = 100 0.6 in
put scale = 0.9 in

let p1 = 0 * 2 * 1 * 3 * 2 * 4 * 3 * 5 * 4 * 0' in
let p2 = 0 * 2 * 4 * 0' * 2' * 4' * 0'' * 4' * 2' * 0' * 4 * 2 in
let p3 = 0< # (.  * 4) in
let p4 = 0 # 0, in
let p5 = p1 @@ p2 @@ p4 @@ p3 in
let p6 = p5> in
let p = p5 # (p6 * p6) in

p
```

2. Structures

Layouts and degrees

A note in the k tones equal temperament (k-TET) is a triple $\mathfrak{n} := (s, k, o)$ where $0 \leqslant s \leqslant k - 1$ is its step and $o \in \mathbb{Z}$ is its octave.

A layout is a sequence $\boldsymbol{\lambda} := \boldsymbol{\lambda}_1 \dots \boldsymbol{\lambda}_\ell$ of positive integers such that $\boldsymbol{\lambda}_1 + \dots + \boldsymbol{\lambda}_\ell = k$. It specifies the distances in steps between the notes in any octave.

– Example –

$\boldsymbol{\lambda} := 2212221$ is the layout of the major natural scale. $\boldsymbol{\lambda} := 32232$ is the layout of the minor pentatonic scale.

A rooted layout is a pair $(\boldsymbol{\lambda}, \mathfrak{n})$ where $\boldsymbol{\lambda}$ is a layout and $\mathfrak{n}$ is a note, called root.

Given $(\boldsymbol{\lambda}, \mathfrak{n})$, any $d \in \mathbb{Z}$, called degree, specifies a note having d as offset from $\mathfrak{n}$.

– Example –

For the rooted layout $(21414, A)$, where A is the note $A440$, one has the correspondence between degrees and notes:

…	-2	-1	**0**	1	2	3	4	**5**	6	7	8	…
…	$E,$	$F,$	**A**	B	C	E	F	**A′**	B'	C'	E'	…

Time shapes and time degrees

A time shape is a pair $\boldsymbol{t} := (m, d)$ where $m, d \in \mathbb{N} \setminus \{0\}$ and $m > d$. We say that m is its time multiplier and that d is its time divider.

A concrete time shape is a pair $(\boldsymbol{t}, u)$ where u is the unit duration in ms.

Given $(\boldsymbol{t}, u)$, any $t \in \mathbb{Z}$, called time degree, specifies the duration $u\left(\frac{m}{d}\right)^t$ in ms.

– Examples –

- For the concrete time shape $((2, 1), 1000)$, one has the correspondence between time degrees and durations:

…	−3	−2	−1	**0**	1	2	3	…
…	125	250	500	**1000**	2000	4000	8000	…

- For the concrete time shape $((3, 2), 1000)$, one has the correspondence between time degrees and durations:

…	−3	−2	−1	**0**	1	2	3	…
…	296.3	444.4	666.7	**1000**	1500	2250	3375	…

Beats, rests, and atoms

An atom is a pair (d, t) where d is a degree or $d = \cdot$, and t is a time degree.

A beat is an atom (d, t) such that d is a degree. A rest is an atom of the form $(\cdot, t)$.

To gain concision, we express any atom $a := (d, t)$ as follows:

- if $t \geqslant 0$, then a is written as `d` `< ... <` (t times);
- if $t \leqslant -1$, then a is written as `d` `> ... >` ($|t|$ times).

– Examples –

- $(0, 0)$ writes as `0`;
- $(\cdot, 1)$ writes as `.<`;
- $(1, 2)$ writes as `1<<`;
- $(2, -2)$ writes as `2>>`.

For a rooted layout and a concrete time shape, an atom encodes a note or a rest with a duration.

– Examples –

Given the rooted layout $(21414, A)$ and the concrete time shape $((2, 1), 1000)$, the atom

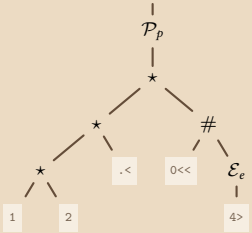
- `0` encodes the note A lasting 1000 ms;
- `1<<` encodes the note B lasting 4000 ms;
- `.<` encodes a rest lasting 2000 ms;
- `2>>` encodes the note C lasting 250 ms.

Tree patterns

A tree pattern is a rooted planar unary binary tree, defined recursively to be either

1. an atom;
2. the concatenation $\star(\mathfrak{t}_1, \mathfrak{t}_2)$ of two tree patterns $\mathfrak{t}_1$ and $\mathfrak{t}_2$;
3. the addition $\#(\mathfrak{t}_1, \mathfrak{t}_2)$ of two tree patterns $\mathfrak{t}_1$ and $\mathfrak{t}_2$;
4. the performance $\mathcal{P}_p(\mathfrak{t})$ of a tree pattern $\mathfrak{t}$, where p is a map sending each atom to a sound;
5. the effect $\mathcal{E}_e(\mathfrak{t})$ of a tree pattern $\mathfrak{t}$, where e is a map from a sound to a sound.

– Example –



is a tree pattern having five atoms (four beats and one rest), three concatenation nodes, one addition node, one performance node, and one effect node.

Tree patterns

This recursive point of view of music is inspired by the notion of temporal medias introduced in [**Hudak**, 2003].

Indeed,

- the concatenation of t_1 and t_2 encodes the sequential playing of t_1 and then of t_2;
- the addition of t_1 and t_2 encodes the simultaneous playing of t_1 and t_2.

Moreover,

- the performance of a tree pattern t is based upon context information (rooted layout, concrete time shape, timbre and envelope of the sound) in order to play the wanted sound specified by each atom;
- the effect of a tree pattern t is used to alter the overall specified sound by introducing some delay, clipping, tremolo, *etc.*

Sounds

A sound σ is a signal which can be encoded as a list of values in $[-1, -1]$, but, in some contexts it is better to encode it as a map

$$\sigma : [\![0, |\sigma| - 1]\!] \to [-1, 1]$$

where $|\sigma| := rt$, t is the duration of the sound in sec, and r is its sampling rate in Hz.

The benefits of this encoding as maps is that the translation of a tree pattern into a sound is memory efficient and some operations on sounds can be processed quickly.

If σ_1 and σ_2 are two sounds, then the concatenation $\sigma_1 \star \sigma_2$ of σ_1 and σ_2 satisfies

$$(\sigma_1 \star \sigma_2)(i) = \begin{cases} \sigma_1(i) & \text{if } i < |\sigma_1|, \\ \sigma_2(i - |\sigma_1|) & \text{otherwise.} \end{cases}$$

The addition $\sigma_1 \# \sigma_2$ of σ_1 and σ_2 satisfies

$$(\sigma_1 \# \sigma_2)(i) = \begin{cases} -1 & \text{if } \sigma_1(i) + \sigma_2(i) < -1, \\ 1 & \text{if } \sigma_1(i) + \sigma_2(i) > 1, \\ \sigma_1(i) + \sigma_2(i) & \text{otherwise.} \end{cases}$$

By convention, for any sound σ, we set $\sigma(i) := 0$ for all $i \notin [\![0, |\sigma| - 1]\!]$.

3. Language

Phrases

A Calimba program is a phrase, which is an element specified by the grammar

phrase ::=			
	`(`	`)`	`begin` `end`
	`.`	int	int `:` label
	phrase `*` phrase	phrase `#` phrase	
	phrase `+`	phrase `-`	
	phrase `<`	phrase `>`	
	phrase `@` label phrase	phrase `@@` phrase	
	`repeat` int phrase	`reverse` phrase	`complement` phrase
	phrase `'`	phrase `,`	
	`let` name `=` phrase `in` phrase		
	`put` `layout` `=` int$^+$ `in` phrase	`put` `root` `=` int int int `in` phrase	...

Comments are enclosed into `{` and `}` and can be nested.

Each phrase is evaluated into a tree pattern and then into a sound.

Simple phrases

Atom are written as explained before.

- If `p1` and `p2` are two phrases, then `p1 * p2` is a new phrase defined as the concatenation of `p1` and `p2`.
- If `p1` and `p2` are two phrases, then `p1 # p2` is a new phrase defined as the addition of `p1` and `p2`.

– Examples –

- `0 * 1 * 2 * 3`
- `0 # 2 # 4`
- `0 * 2> * 1> * 1 * . * (0 # 2 # 4)`
- `((1<< * 2) # (3< * 4> * 1)) * 0<<`

Without brackets, `*` has a higher priority than `#`.

Operations on phrases

If `p` is a phrase, then `p+` (resp. `p-`) is the phrase obtained by incrementing (resp. decrementing) each degree of `p`.

– Example –

```
(0 # 2 # 4) * (0 # 2 # 4)- * (0 # 2 # 4)++
```

If `p` is a phrase, then `p<` (resp. `p>`) is the phrase obtained by incrementing (resp. decrementing) each time degree of `p`.

– Example –

```
((0 # 4) * 2) * ((0 # 4) * 2)< * ((0 # 4) * 2)>>
```

If p is a phrase, then `p'` (resp. `p,`) is the phrase obtained by incrementing (resp. decrementing) each degree of `p` so that its refers to the octave above (resp. below).

– Example –

```
0 * 0, * (0 # 4 # 0') * (0 # 4 # 0'),,
```

Operations on phrases

If `p` is a phrase, then `repeat k p` is the phrase obtained by concatenating `p` with itself `k` times.

– Example –

```
repeat 3 1>> * 1>> * 5 * .> * 2>
```

If `p` is a phrase, then `reverse p` is the phrase obtained by considering the atoms of `p` from the end to the beginning.

– Example –

```
0 * 2 * (4 # 6) * (reverse 0 * 2 * (4 # 6))
```

If `p` is a phrase, then `complement p` is the phrase obtained by changing its degrees of `p` by their opposite values.

– Example –

```
0 * (2 # 4) * -1 * (complement 0 * (2 # 4) * -1)
```

Tree pattern insertion

If $b := (d, t)$ is a beat and $a := (d', t')$ is an atom, the product of b by a is the atom $b \times a$ defined by

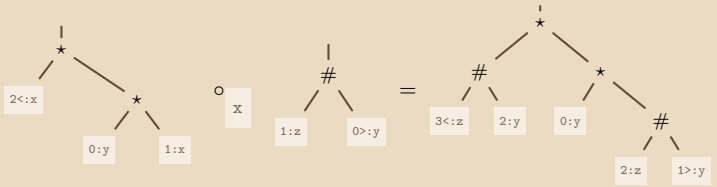
- if a is a beat, then $b \times a := (d + d', t + t')$;
- if a is a rest, then $b \times a := (\cdot, t + t')$.

The action of b on a tree pattern $\mathfrak{t}$ is the tree pattern $b \times \mathfrak{t}$ obtained by replacing any atom a of $\mathfrak{t}$ by $b \times a$.

From now, each beat of a tree pattern can have a label. Several beats can have the same label.

Let $\mathfrak{t}_1$ and $\mathfrak{t}_2$ be two tree patterns and u be a label. The insertion of $\mathfrak{t}_2$ into $\mathfrak{t}_1$ at position u is the tree pattern $\mathfrak{t}_1 \circ_u \mathfrak{t}_2$ obtained by replacing each beat b labeled by u of $\mathfrak{t}_1$ by the tree pattern $b \times \mathfrak{t}_2$.

– Example –



Named and saturated insertions

If `p1` and `p2` are two phrases and `u` is a label, then `p1 @u p2` is a phrase evaluating as the tree pattern $t_1 \circ_u t_2$ where t_1 (resp. t_2) is the evaluation of `p1` (resp. `p2`).

– Example –

The phrases `2:x * 0 * 4:x< @x (repeat 3 3, * 2)` and `(repeat 3 5, * 4) * 0 * (repeat 3 7, * 6)<` are equivalent.

If `p1` and `p2` are two phrases, then `p1 @@ p2` is a phrase evaluating as the tree pattern obtained by inserting t_2 into t_1 onto all its beats, where t_1 (resp. t_2) is the evaluation of `p1` (resp. `p2`).

– Examples –

- The phrases `0 * 1 * 2 @@ 0 * 2 * 4` and `(0 * 2 * 4) * (1 * 3 * 5) * (2 * 4 * 6)` are equivalent.
- The phrases `0 * 1 * 2 @@ (0 # 2 * 4>)` and `(0 # 2 * 4>) * (1 # 3 * 5>) * (2 # 4 * 6>)` are equivalent.

The `let in` construction

Given a phrase `p1`, one can give it a name in order to use it (possibly several times) later, in an other phrase `p2`.

For this, we use `let name = p1 in p2`, where `name` is a name. This phrase is equivalent to the phrase `p2` wherein all free occurrences of `name` are replaced by `p1`.

– Examples –

- The phrases

```
let p = -1 * 0< * 3> in
p+ * 0 * .  * p--
```

and

```
(-1 * 0< * 3>)+ * 0 * .  * (-1 * 0< * 3>)--
```

are equivalent.

- The phrases

```
let a = 0 # 2 in
let b = (7 * 8< * 6 * 7< * 5 * 6<),, in
b @@ a
```

and

```
(7 * 8< * 6 * 7< * 5 * 6<),, @@ (0 # 2)
```

are equivalent.

Contexts

Each phrase is played under a given context. It consists in

1. a rooted layout $(\boldsymbol{\lambda}, \mathfrak{n})$;
2. a concrete time shape $(\boldsymbol{t}, u)$;
3. a synthesizer $\mathfrak{s}$.

The first two components explain how to translate each atom of the phrase into a note or a rest with a duration. The third component explains how to associate a sound with each note.

The default values are

1. $(\boldsymbol{\lambda}, \mathfrak{n}) = (2122122, A)$ where A is the note $A440$ of the 12-TET;
2. $(\boldsymbol{t}, u) = ((2, 1), 500)$;
3. $\mathfrak{s}$ is a synthesizer producing a certain sound.

Context changes in phrases give rise to performances in the associated tree patterns.

Changing contexts

Let `p` be a phrase.

- The phrase `put layout = i1 ... ik in p` is the phrase wherein `i1 ... ik` is the layout for `p`.
- The phrase `put root = s k o in p` is the phrase wherein `s k o` is the root for `p`.
- The phrase `put time = m d in p` is the phrase wherein (`m` , `d`) is the time shape for `p`.
- The phrase `put duration = u in p` is the phrase wherein `u` is the unit duration for `p`.
- The phrase `put synthesizer = p r m a d in p` is the phrase wherein each atom is played by the synthesizer specified by the arguments `p` (volume), `r` (harmonics richness), `m` (maximal duration in ms), `a` (attack duration in ms), and `d` (decay duration in ms).

– Example –

```
put layout = 2 2 1 2 2 2 1 in
put root = 0 12 -2 in
put duration = 350 in
put synthesizer = 0.02 0.7 300 10 20 in
```

```
let ch = (0 # 2 # 4) * 4> * 2> * 0> * 2> in
repeat 2
    (repeat 4 ch) * (repeat 4 ch @@ 4)
    * (repeat 2 ch @@ 3)
    * (put layout = 2 1 2 2 1 2 2 in repeat 2 ch @@ 3)
```

Effects

An effect is a transformation on sounds. Four basic effects are proposed. Let `p` be a phrase.

- The phrase `put scale = c in p` is the phrase `p` such that its sound is scaled by `c`.
- The phrase `put clip = c in p` is the phrase `p` such that its sound is clipped with threshold `c`.
- The phrase `put delay = t c in p` is the phrase `p` such that its sound is composed by the sound of `p` added to a version of it shifted by `t` ms and scaled by `c`.
- The phrase `put tremolo = t c in p` is the phrase `p` such that its sounds has a tremolo effect of period `t` ms and having amplitude that can be scaled down by `c`.

– Examples –

```
let p = (1 * 3 * (0 # 4) * 2), in
```

- `(put scale = 0.5 in p) * (put scale = 3.1 in p)`
- `put clip = 0.2 in p`
- `put delay = 250 0.5 in p`
- `put tremolo = 100 0.75 in p`

Sound modifiers

A sound modifier is a phrase wherein only the atom `0` appears.

If `m` is a sound modifier and `p` is a phrase, then `m @@ p` is a phrase wherein the sound of `p` is modified as prescribed by `m`.

Here are three useful modifiers:

```
let octaver =
    put scale = 0.5 in
    0, # 0
```

```
let echoer =
    put delay = 400 0.1 in
    put delay = 300 0.2 in
    put delay = 200 0.2 in
    put delay = 100 0.2 in
    0
```

```
let distorter =
    put clip = 0.15 in
    put scale = 2.0 in
    0
```

– Examples –

```
let p = (1 * 3 * (0 # 4) * 2), in
```

- `octaver @@ p`
- `echoer @@ p`
- `distorter @@ p`
- `octaver @@ distorter @@ p`
- `distorter @@ octaver @@ p`
- `echoer @@ distorter @@ p`
- `echoer @@ octaver @@ distorter @@ p`
- `distorter @@ echoer @@ octaver @@ p`
- `distorter @@ distorter @@ p`

Examples

```
let p1 = (0< * 4 * 2 * 6<<)
    # (0,<<< # 4,<<< # 0<<<)
    # (.<< * 6> * 4> * 5> * 2>
        * 3> * 1> * 0) in
let p2 = 0 * 0 * 1 * 1 * 0 * 3 * 3 * 1 in

let p = p2 @@ p1 in

put root = 5 12 -2 in
put layout = 1 2 2 2 1 2 2 in
put time = 2 1 in
put duration = 300 in
put synthesizer = 0.22 0.51 2000 50 100 in

put delay = 120 0.7 in

repeat 2 p
```

```
let p1 = 0,<< # (0 * 2 * 4 * 0') in
let p2 = 0, * 0,> * 0,> * .  * 0, in
let p3 = p1 # p2 in

let p = (repeat 4 p3)
    * (repeat 2 p3++) * (repeat 4 p3++++)
    * (repeat 2 p3+) * (repeat 4 p3+++)
    * (repeat 2 p3') * (repeat 2 p3'>) in

put root = 5 12 -3 in
put layout = 3 2 2 3 2 in
put time = 2 1 in
put duration = 200 in
put synthesizer = 0.25 0.45 2000 2 90 in

repeat 2 p
```

Examples

```
let p1 = 0 * 0> * 0> in
let p2 = 2 * 0, * 2,> * 0'> in
let p3 = 2<< * 1< * 4,< * 2<<< * 2<<
    * 1< * 4,< * 0<<< in
let p4 = p3 * p3+ in

let p = (repeat 96 p1) # (repeat 64 p2)
    # (repeat 3 p4) in

put layout = 3 2 2 3 2 in
put root = 0 12 -2 in
put time = 2 1 in
put duration = 300 in
put synthesizer = 0.25 0.42 4000 10 200 in

p
```

```
put root = 6 19 -2 in
put layout = 3 2 2 2 3 3 2 2 in
put duration = 200 in
put time = 2 1 in
put synthesizer = 0.3 0.3 6000 10 20 in

let p1 = 0:x' * 0> * 0 * 0> in
let p2 = p1 * (p1 @x -1) * (p1 @x -2) * (p1 @x -3)
    * (p1 @x -4) in
let p3 = 0> * 0 * 0> * 1:x in
let p4 = p3 * (p3 @x 1) * (p3 @x 2) * (p3 @x 3)
    * (p3 @x 4) in
let p5 = p2 * p4 in
let p6 = p5 * p5+ * p5++ * p5- in

let ch = repeat 40 (0,< * 0,) in

repeat 2 p6 # ch
```

Conclusion

- Calimba is designed to be easy to use: both for people with/without experience in music, and for people with/without experience in programming.
- It allows us to easily transcribe composition ideas and offers a live coding environment.
- It is a small language supported by sound theoretical bases (functional programming, theory of operads and of clones).
- New synthesizers and effects and other things are under development.

Feedback is welcome! **Thanks!**

`https://github.com/SamueleGiraudo/Calimba`

`<</^\|_`