The grammar of BaLi

Abstract. BaLi is a small language inspired by the syntax of Lisp. It has been developed in order to test BuboCore in Live-Coding situations. Using BaLi one may feel that obvious things are missing. Sometimes this is because they are difficult to add due to the fact that BaLi has been developed without designing it before. Sometimes this is also just because we forgot. This document presents the grammar of this language and gives insights on its semantics.

1 The grammar

1.1 The grammar itself

This is a simplified version of the actual grammar (mainly for readability concerns).

In the below grammar, a **Number** is any sequence of one or more digits (ASCII characters 48 to 57) and a **Decimal** is any sequence of one or more digits and at most one dot which is not the last character (so 27 is a **Number** and a **Decimal** and 2.7 and .27 are **Decimal**). An **Identifier** is any sequence of one or more letters (ASCII characters 65 to 90 and 97 to 122) and - and # characters, starting with a letter. A **Literal** is any sequence of characters starting end ending with double quotes.

⟨*Program*⟩	::=	ε | ⟨*Program*⟩ ⟨*T-Statement*⟩ | ⟨*Program*⟩ ⟨*Function-Declaration*⟩
⟨*Function-Declaration*⟩	::=	(fun **Identifier Identifier*** ⟨*Control-Effect*⟩+ ⟨*Arithm-Expr*⟩)
⟨*Context*⟩	::=	⟨*Context-Element*⟩+
⟨*Context-Element*⟩	::=	ch: ⟨*Arithm-Expr*⟩ | dev: ⟨*Arithm-Expr*⟩ | dur: ⟨*Arithm-Expr*⟩ | v: ⟨*Arithm-Expr*⟩
⟨*Timing-Info*⟩	::=	⟨*Concrete-Fract*⟩ | ⟨*Concrete-Fract*⟩.f
⟨*T-Statement*⟩	::=	(> ⟨*Timing-Info*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+) | (>> ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(< ⟨*Timing-Info*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+) | (<< ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(spread ⟨*Timing-Info*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(ramp **Identifier Number Number Number Literal** ⟨*Timing-Info*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(loop **Number** ⟨*Timing-Info*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(eucloop **Number Number** ⟨*Timing-Info*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(binloop **Number Number** ⟨*Timing-Info*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(pick ⟨*Arithm-Expr*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(? ⟨*Number*⟩ ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(alt ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	(with ⟨*Context*⟩ ⟨*T-Statement*⟩+)
	|	⟨*Control-Effect*⟩
⟨*Control-Effect*⟩	::=	(seq ⟨*Context*⟩ ⟨*Control-Effect*⟩+)
	|	(if ⟨*Bool-Expr*⟩ ⟨*Context*⟩ ⟨*Control-Effect*⟩+)
	|	(for ⟨*Bool-Expr*⟩ ⟨*Context*⟩ ⟨*Control-Effect*⟩+)
	|	(pick ⟨*Arithm-Expr*⟩ ⟨*Context*⟩ ⟨*Control-Effect*⟩+)

```
                      | (? ⟨Number⟩ ⟨Context⟩ ⟨Control-Effect⟩+ )
                      | (alt ⟨Context⟩ ⟨Control-Effect⟩+ )
                      | (with ⟨Context⟩ ⟨Control-Effect⟩+)
                      | ⟨Effect⟩
⟨Effect⟩           ::= (def Identifier ⟨Arithm-Expr⟩ )
                      | (note ⟨Arithm-Expr⟩ ⟨Context⟩)
                      | (prog ⟨Arithm-Expr⟩ ⟨Context⟩ )
                      | (control ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ⟨Context⟩ )
                      | (at ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ⟨Context⟩ )
                      | (chanpress ⟨Arithm-Expr⟩ ⟨Context⟩ )
                      | (osc Literal ⟨Arithm-Expr⟩* ⟨Context⟩ )
                      | (dirt Literal ⟨Dirt-Param⟩* ⟨Context⟩ )
                      | ()
⟨Dirt-Param⟩       ::= :Identifier ⟨Arithm-Expr⟩
⟨Concrete-Fract⟩   ::= (// Number Number ) | Number | Decimal
⟨Bool-Expr⟩        ::= (and ⟨Bool-Expr⟩ ⟨Bool-Expr⟩ )
                      | (or ⟨Bool-Expr⟩ ⟨Bool-Expr⟩ )
                      | (not ⟨Bool-Expr⟩ )
                      | (lt ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ) | (leq ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ )
                      | (gt ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ) | (geq ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ )
                      | (== ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ) | (!= ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ )
⟨Arithm-Expr⟩      ::= (+ ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ) | (* ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ )
                      | (- ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ) | (/ ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ )
                      | (% ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ )
                      | Identifier | Decimal
                      | (Identifier ⟨Arithm-Expr⟩* )
                      | (rand ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ )
                      | (scale ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ⟨
                        Arithm-Expr⟩)
                      | (clamp ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩)
                      | (min ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩)
                      | (max ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩)
                      | (quantize ⟨Arithm-Expr⟩ ⟨Arithm-Expr⟩)
                      | (sine ⟨Arithm-Expr⟩)
                      | (saw ⟨Arithm-Expr⟩)
                      | (triangle ⟨Arithm-Expr⟩)
                      | (isaw ⟨Arithm-Expr⟩)
                      | (randstep ⟨Arithm-Expr⟩)
                      | (ccin ⟨Arithm-Expr⟩)
```

1.2 Reserved identifiers

A few identifiers are reserved.

Musical notation. All the identifier of the following form are reserved[1]: X, XY, Xb, XbY, XYb, X#, X#Y, XY# with X a letter in $\{c, d, e, f, g, a, b\}$ and Y a natural number in $[-2, 8]$. For example, identifiers c, eb, f#, gb7 and a-1# are reserved.

Global variables. The identifiers A, B, C, D, W, X, Y, and Z are reserved.

[1]With the exception of cb-2, c-2b, g#8, g8#, a8, ab8, a8b, a#8, a8#, b8, bb8, b8b, b#8, b8#.

Environment variables. The identifiers T and R are reserved.

1.3 Syntax simplifications

For fractions one can always write (X // Y) instead of `(// X Y)` in any BaLi program.

⟨*Context*⟩ can be empty in all constructions using it, except for `(with ...)`.

1.4 Comments

At any point in a program, the symbol ; will start a comment. This comment ends at the end of the line.

2 The semantics

A BaLi program is associated to a frame (and thus a line and a scene) in BuboCore. Each timing information used in BaLi is relative to this frame.

2.1 Number and Identifier

The **musical notation** reserved identifiers represent notes as handled by Midi, that is numbers: c-2 is 0, g8 is 127, c3 is 60, c#3 (or c3#) is 61, cb3 (or c3b) is 59. The letter gives the note in alphabetical notation. The number gives the octave. Omitting the number is similar to using 3: c is c3, eb is eb3. They can be used exactly as numbers, they cannot be redefined.

The **environment variables** reserved identifier represent values that can change over time and are set by BuboCore. Environment variable T represents the current beats per minute. Environment variable R is a random number (in [0, 128[) determined by BuboCore each time R is used. It is not possible to redefine these variables (with def, see below) and trying to do so will fail silently.

Appart from that, an **Identifier** is a name for a variable that will hold a decimal number. A variable is private to one program (in BuboCore several programs can execute at the same time) except for the **global variables** (reserved identifiers) that are shared between all programs.

2.2 ⟨*Arithm-Expr*⟩

An ⟨*Arithm-Expr*⟩ represents an arithmetic calculus over decimal numbers. More generally, all numbers manipulated by BaLi are decimals (so integers are in fact decimals with denominator 1).

Available operators are: + (addition), * (multiplication), - (subtraction), / (division), % (modulo).

The expression `(op a b)` corresponds to the calculus a `op` b, that is `(% a b)` corresponds to $a \bmod b$.

A few additional utility functions are available:

- `(rand min max)`: returns a random value between *min* and *max* (*min* can be omitted, in which case the random value is taken between 0 and *max*).
- `(scale val old_min old_max new_min new_max)`: Linearly maps *val* from the range [*old_min*, *old_max*] to the range [*new_min*, *new_max*]. The result is clamped to the new range.
- `(clamp val min max)`: Clamps *val* to be within the range [*min*, *max*].
- `(min a b)`: Returns the smaller of *a* and *b*.
- `(max a b)`: Returns the larger of *a* and *b*.
- `(quantize val step)`: Rounds *val* to the nearest multiple of *step*.

Several stateful oscillator functions generate periodic signals commonly used in LFOs (Low-Frequency Oscillators). They return MIDI-compatible integer values in the range [1, 127]. Their *speed* argument determines the frequency in cycles per beat. They maintain internal state (phase and last update time) across calls within the same script instance, ensuring smooth, continuous oscillation based on the elapsed beats.

- `(sine speed)`: Generates a sine wave.

- `(saw speed)`: Generates a sawtooth wave (ramping up).
- `(triangle speed)`: Generates a triangle wave.
- `(isaw speed)`: Generates a reverse sawtooth wave (ramping down).
- `(randstep speed)`: Generates a stepped random signal. A new random value (1-127) is chosen at the beginning of each cycle (determined by *speed*) and held constant until the next cycle begins.

Finally, it is possible to call a user defined function *f* by writing `(f arg1 arg2 …)` where *argi* is an arithmetic expression used as the i^{th} argument of *f*.

2.3 ⟨*Bool-Expr*⟩

A ⟨*Bool-Expr*⟩ represents a boolean calculus over booleans and integer numbers in [0, 128[. As expressed by the grammar: such an expression can be used only as a condition for a for loop or an if conditional. In particular, the value resulting of the calculus corresponding to such an expression cannot be stored in a variable.

Available operators on booleans are: and, or, not.

Available operators on integers are: lt (strictly lower than), leq (lower or equal), gt (strictly greater than), geq (greater or equal), == (equal), != (not equal).

The expression `(op a b)` corresponds to the calculus a `op` b, that is `(get a b)` corresponds to $a \geq b$.

2.4 ⟨*Concrete-Fract*⟩

A ⟨*Concrete-Fract*⟩ is a fraction used for expressing time durations. The fraction is converted to a floating point value at the last possible moment (that is, when BuboCore has to compute a timestamp).

In practice `(// n d)` represents a fraction with numerator n and denominator d. The alternative definition of a fraction as a single number or arithmetic expression n represents a fraction with a numerator of n and a denominator 1.

A ⟨*Concrete-Fract*⟩ represents a fraction that will be computed at compile time. It is defined from numbers only. An alternative definition exists for ⟨*Concrete-Fract*⟩ as a decimal number f. It represents a fraction with numerator n and denominator d such that $f = \frac{n}{d}$. A ⟨*Concrete-Fract*⟩ can always be omitted, in which case it will be considered as $\frac{1}{1}$.

2.5 ⟨*Effect*⟩

An ⟨*Effect*⟩ changes the state of the program or impacts the external world. At the moment there are nine effects.

`(def v e)` sets the value of variable v to e. Any variable has value 0 by default.

`(note n c)` Sends a MIDI Note On message followed by a corresponding Note Off message after a specified duration. It targets a specific MIDI device. n is the note number. A velocity, a MIDI channel and a duration and the target device are obtained from the context c if they are defined in it or, else, from the context in which this effect is used.

`(prog p c)` Sends a MIDI Program Change message to a specific MIDI device. p is the program number. A MIDI channel and the target device are obtained from the context c if they are defined in it or, else, from the context in which this effect is used.

`(control con v c)` Sends a MIDI Control Change message to a specific MIDI device. *con* is the control number and v is the control value. A MIDI channel and the target device are obtained from the context c if they are defined in it or, else, from the context in which this effect is used.

`(at note value c)` After Touch **TODO** `(chanpress value c)` Channel Pressure **TODO**

`(osc address arg1 arg2 … c)` Send OSC message **TODO**

`(dirt sound param1 param2 … c)` Send Dirt message **TODO**

`()` Does nothing. It can be used as a place order for futur effects, or to add silences in rhythms for example.

2.6 ⟨*Control-Effect*⟩

A ⟨*Control-Effect*⟩ allows to perform ⟨*Effect*⟩ (or ⟨*Control-Effect*⟩) in sequence (seq), in loop (for), conditionally (if), etc.

`(seq c s)` will execute in order the elements of s in the context c.

`(if cond c s)` will execute the elements of s (not necessarily in order) in the context c if the condition *cond* is evaluated to `true`.

`(for cond c s)` will execute all the elements in s in the context c as long as the condition *cond* is evaluated to `true`. One should avoid making infinite loops as this will mess with the timing requirements (see next section) due to BuboCore program execution model.

`(pick expr c s)` will evaluate the expression *expr* to get a number e. Then, the element number e (modulo the length of s) in s will be executed.

`(? n c s)` will execute n randomly chosen elements of s in the context c. The n elements will be different (if n is greater than the number of elements, then each element is executed once). It is possible to omit n, in which case it will be set to 1.

`(alt c s)` will execute one element of s each time this effect is reached. The elements of s are executed in order. This is also valid between scripts instances. So, the script `(alt (note c) (note d))` will play a C at its first execution, a D at its second execution, then a C, and so on.

`(with c s)` will execute the elements of s (not necessarily in order) in the context c.

2.7 ⟨*T-Statement*⟩

A ⟨*T-Statement*⟩ allows to perform some (list of) ⟨*Control-Effect*⟩ at a given time point. The time is expressed as a ⟨*Concrete-Fract*⟩ because having variables here would lead to execution orders that cannot be decided at compile time. The time points calculations are relative to the duration of a time window. The time point is initially set to the beginning of the frame in which the program is executed. The time window is initially set to the duration of the frame in which the program is executed. The time point and the time window evolve with some of the statements.

2.7.1 Basic semantics

In the below descriptions of statements, we consider that *TW* is the duration of the time window when the statement is used and *TP* is the time point when the statement is used.

`(> frac c p)` executes p in context c at a point in time *frac* × *TW* after *TP*. So, intuitively, this shifts p by a fraction of the time window.

`(< frac c p)` executes p in context c at a point in time *frac* × *TW* before *TP*. In case p should be executed before the start of the current frame, it will be executed at the start of this frame but before any other thing that should be executed at a later time (even before the start of the frame).

`(>> c p)` executes p in context c at *TP*, but just after everything else that should occur at this time point.

`(<< c p)` executes p in context c at *TP*, but just before everything else that should occur at this time point.

For example, the program `(> 0.5 p1 (<< p2) (>> p3)` will execute *p1*, *p2* and *p3* all at $\frac{1}{2}$ of the frame, but in the following order: *p2*, then *p1*, then *p3*.

`(spread frac c p)` sets *TW* as *frac* $\times$ *TW*. Then executes the statements of p (in context c) distributed fairly in *TW* starting from *TP*. Each statement is executed with a new time window equal to $\frac{TW}{len(p)}$ where *len(p)* is the number of statements in p.

`(loop n frac c p)` sets *TW* as *frac* $\times$ *TW*. Then executes n times, distributed fairly in *TW* starting from *TP*, the program p in context c. Each execution of the program has a time window set to $\frac{TW}{n}$.

`(ramp variable granularity min max distribution frac c p)` is similar to `(loop granularity frac c p)` excepted that at each iteration the value of *variable* is set to a new value in the interval [min max] according to the *distribution*. At the moment the only possible distribution is *"linear"*, meaning that *variable* increases linearly from *min* to *max*.

`(eucloop steps beats frac c p)` sets *TW* as *frac* $\times$ *TW*. Then executes *steps* times the program p in context c. This executions take place on the time points corresponding to an euclidean rhythm whose beats are fairly distributed in *TW* starting from *TP*. Each execution of the program has a time window set to $\frac{TW}{beats}$.

`(binloop val beats frac c p)` sets *TW* as *frac* $\times$ *TW*. Then considers the time points obtained by distributing *beats* points fairly in *TW* starting from *TP*. At each of these time points the program p (in context c) is executed or not, depending on the binary representation of *val* over 7 bits. For example, if $val = 6$ the representation of *val* is 0000110. Then, if $beats = 7$, p will be executed at the fifth and the sixth time points, but not at the other time points. If *beats* is smaller than 7 then only the *beats* most significant bits of *val* are considered. In the previous example, if $beats = 5$, we consider 00001. If *beats* is larger than 7 then we loop over the representation of *val*. In the previous example, if $beats = 12$, we consider 000011000001.

`(pick expr c p)` is similar to the pick $\langle$*Control-Effect*$\rangle$. The only thing to notice is that *expr* is evaluated once and for all at the first *TP* at which a statement of p could be executed.

`(? n c p)` is similar to the ? $\langle$*Control-Effect*$\rangle$.

`(alt c p)` is similar to the alt $\langle$*Control-Effect*$\rangle$.

`(with c p)` is similar to the with $\langle$*Control-Effect*$\rangle$.

2.7.2 Alternative semantics

The statements that admit a $\langle$*Timing-Info*$\rangle$ in their arguments can have their time points computed relatively to the duration of the frame in which the script is executed rather than relatively to the duration of their time window. For that, the timing information shall be given with a trailing *.f* (for example `(> 0.5.f p)` instead of `(> 0.5 p)`).

The statements which change the time window (spread, loop, eucloop, binloop) can be used by giving the duration of a step rather than the duration of the full statement by adding a `:step` argument. For example `(loop 4 0.5:step p)` will execute p 4 times, each time for the duration of half the frame and `(loop 4 0.5 p)` will execute p 4 times in half a frame (so each execution of p will be over $\frac{1}{8}$ of the frame).

2.8 ⟨*Function-Declaration*⟩

It is possible to declare new functions in a program. Functions must be declared at the first level of the program. Functions may have several arguments but always have exactly one return value. Functions can be called only in ⟨*Arithm-Expr*⟩ (see Section 2.2).

A function declaration has the following syntax: `(fun name arg_name1 arg_name2 … p return)` where:

- *name* is the name of the function (that will be used to call it).
- *arg_namei* is the name of the i^{th} argument of the function, that is the name of a variable that exists only in the function body.
- *p* and *return* constitute the function body and are such that:
  - *return* is an ⟨*Arithm-Expr*⟩ and its value is returned by the function,
  - *p* is a set of ⟨*Control-Effect*⟩ that will be executed in sequence before computing the value of *return*.

At compile time it is verified that each function is defined at most once (i.e. each identifier is used at most once as a function name) and that each call to a function has the correct number of arguments.