

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. This document presents the grammar of this language and gives insights on its semantics.

1 The grammar

1.1 The grammar itself

In the below grammar, a **Number** is any sequence of one or more digits (ASCII characters 48 to 57). 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.

```
<Program> ::= <Program> <Time-Statement> | <Time-Statement>
<Time-Statement> ::= (> <Concrete-Fract> <Program>) | (>> <Program> )
    | (< <Concrete-Fract> <Program>) | (<< <Program> )
    | (loop Number <Concrete-Fract> <Program> )
    | <Control-Effect>
<Control-Effect> ::= (seq <Control-List>) | (if <Bool-Expr> <Control-List> )
    | (for <Bool-Expr> <Control-List>) | <Effect>
<Control-List> ::= <Control-List> <Control-Effect> | <Control-Effect>
<Effect> ::= (def Identifier <Arithm-Expr> )
    | (note <Arithm-Expr> <Arithm-Expr> <Arithm-Expr> <Abstract-Fract>)
    | (prog <Arithm-Expr> <Arithm-Expr> )
    | (control <Arithm-Expr> <Arithm-Expr> <Arithm-Expr> )
<Concrete-Fract> ::= // Number Number ) | Number
<Abstract-Fract> ::= // <Arithm-Expr> <Arithm-Expr> ) | <Arithm-Expr>
<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 | Number
```

1.2 Reserved identifiers

A few identifiers are reserved.

Musical notation. All the identifier of the following form are reserved¹: X, XY, Xb, XbY, XYb, 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.

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

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

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.

Moreover, in (`note` n v c d), arguments v and c are optional: one can write (`note` n v d) and (`note` n d). In these cases, c and v (if needed) will have default values.

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 step (and thus a sequence and a pattern) in BuboCore. Each timing information used in BaLi is relative to this step.

2.1 Number and Identifier

A **Number** is any 8 bits number (so, in [0, 128[). In case a number n out of this range is used in a program the actual number that will be considered is $n \bmod 128$.

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 number. They hold only numbers in [0, 128[. In case a number n out of this range is stored in a variable the actual number that will be used is $n \bmod 128$. 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 $\langle\text{Arithm-Expr}\rangle$

An $\langle\text{Arithm-Expr}\rangle$ represents an arithmetic calculus over integer numbers in [0, 128[. The result is always in [0, 128[. If needed, a modulo is performed.

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

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

2.3 $\langle\text{Bool-Expr}\rangle$

A $\langle\text{Bool-Expr}\rangle$ 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 $a \text{ op } b$, that is `(get a b)` corresponds to $a \geq b$.

2.4 $\langle\text{Concrete-Fract}\rangle$ and $\langle\text{Abstract-Fract}\rangle$

A $\langle\text{Concrete-Fract}\rangle$ or an $\langle\text{Abstract-Fract}\rangle$ 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 – a single number or arithmetic expression d – represents a fraction with a numerator of 1 and a denominator d (except if $d = 0$ in which case the numerator is 0 and the denominator is 1).

A $\langle\text{Concrete-Fract}\rangle$ represents a fraction that will be computed at compile time. It is defined from numbers only.

An $\langle\text{Abstract-Fract}\rangle$ represents a fraction that will be computed at execution time. It can be defined from $\langle\text{Arithm-Expr}\rangle$.

2.5 $\langle\text{Effect}\rangle$

An $\langle\text{Effect}\rangle$ changes the state of the program or impacts the external world. At the moment there are four effects.

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

`(note n v c d)` asks the default Midi device to play the note n with velocity v on channel c for duration d . The duration is an $\langle\text{Abstract-Fract}\rangle$.

`(prog p c)` sends a program change message to default Midi device. With program p on channel c .

`(control con v c)` sends a control change message to default Midi device. With control con , value v , and on channel c .

2.6 $\langle\text{Control-Effect}\rangle$ and $\langle\text{Control-List}\rangle$

A $\langle\text{Control-Effect}\rangle$ allows to perform $\langle\text{Effect}\rangle$ (or $\langle\text{Control-Effect}\rangle$) in sequence (`seq`), in loop (`for`), or conditionally (`if`). A $\langle\text{Control-List}\rangle$ is simply an ordered set of $\langle\text{Control-Effect}\rangle$.

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

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

`(for cond s)` will execute all the elements in s 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.

2.7 $\langle\text{Time-Statement}\rangle$

A $\langle\text{Time-Statement}\rangle$ allows to perform some (list of) $\langle\text{Control-Effect}\rangle$ at a given point in time. The time is expressed as a $\langle\text{Concrete-Fract}\rangle$ because having variables here would lead to execution orders that cannot be decided at compile time. The time is relative to the length of the step in which the program is executed. It is possible to have nested $\langle\text{Time-Statement}\rangle$, in which case times are added. The default time for executing something, when there is no $\langle\text{Time-Statement}\rangle$ is 0 (so, right at the beginning of the step).

`(> frac p)` executes p at a point in time $frac$ after what was expected.

(`< frac p`) executes *p* at a point in time *frac* before what was expected. In case *p* should be executed at a negative time *t*, it will be executed at time 0 but before any other thing that should be executed at time 0 or at a time negative but larger than *t*.

(`>> p`) executes *p* at the expected time point, but just after everything else that should occur at this time point.

(`<< p`) executes *p* at the expected time point, but just before everything else that should occur at this time point.

For example, the program (`> 5 p1 << p2 >> p3`) will execute *p1*, *p2* and *p3* all at $\frac{1}{5}$ of the step, but in the following order: *p2*, then *p1*, then *p3*.

Finally (`loop n frac p`) executes *n* times *p*. First at the expected time point, then *frac* after this point, then *frac* later, and so on.