Messing Around with BILL
Custom Scripting Languages for BuboCore

Abstract. BuboCore has been designed so that it is (relatively) simple for a user to define their own scripting language(s) to be used for Live-Coding the steps of a pattern. The general idea is to write a compiler that will translate scripts to a low-level language – BILL – that is interpreted by the BuboCore scheduler. This requires to know BILL and to understand how the BuboCore scheduler works, which is the object of this document. At the end we also give a few guidelines on how to properly integrate a new scripting language into BuboCore.

1 The BuboCore scheduler

1.1 General overview

As show in Figure 1, the scheduler is responsible for emitting (time-stamped) events. These events are mostly sent to the World, the interface between BuboCore and the different devices — hardware or software — that it controls. They can also occasionally be sent to other parts of BuboCore.

For that the scheduler loops forever, executing sequences of steps (each taken into a finite set of steps). The events that shall be emitted at each of these steps are specified as a sequence of instructions (a program) written in the BuboCore Intermediate Low-level Language (BILL). So, each step is associated to a BILL program.

In order to know how and when each step should occur, BuboCore scheduler relies on an environment that provides information on everything else (clocks, devices, etc).

1.2 Lifespan of a BILL program execution

Each step is always associated to a (potentially empty) BILL program. When the environment is such that a new step shall begin, the scheduler is responsible for instantiating a new execution of the BILL program associated to this step (as shown in Figure 2 where programs BP1, BP2 and BP3 respectively correspond to steps 1, 2 and 3). Once a program execution is instantiated, this program is executed by the scheduler until it is finished (that is, until a normal end of the program is reached, or until an error occurs in the program).

Notice that the duration of a program execution is in general not related to the duration of the step in which it started: it may be shorter or longer. It is even possible that the same step occurs again

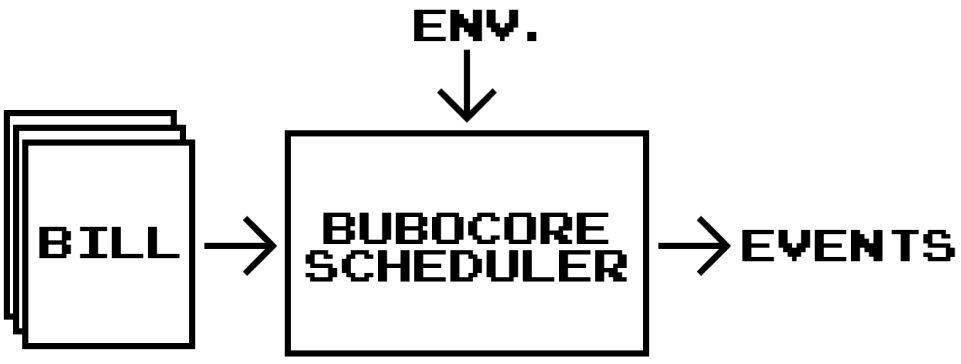

Figure 1: Overview of the BuboCore scheduler

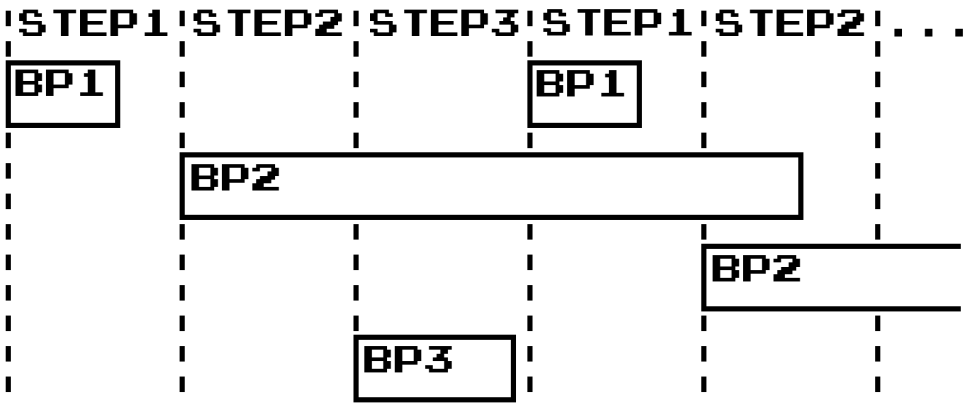

Figure 2: A BILL program execution is instantiated at each step

```
pub enum Instruction {
    Control(ControlASM),
    Effect(Event, Variable),
}
```

Listing 1: Instruction definition

before the end of the corresponding program execution, leading to two instances of the same program running at the same time (as for program BP2 in Figure 2)

1.3 How BILL programs are executed

A BILL program is a sequence of *instructions* (Listing 1) that can either be *control* instructions (a list of all the control instructions is given in Section 2.3) or *effect* instructions (a list of all the effect instructions is given in Section 2.4).

The effect instructions are the ones that generate emissions of events to the World. Any effect instruction contains two parts: an event e and a duration d (in Listing 1 this last part is represented by a Variable, this will be explained later). In the following, such an instruction is denoted by (e, d).

The control instructions are all the other instructions: they are silent from the point of view of the World. In particular, the kind of instructions that one would expect to find in any assembly language (arithmetic and logic operations, control-flow management) are control instructions.

1.3.1 Execution of a single program

In order to execute a program, the scheduler maintains a time counter that states when the next event can be emitted. This counter is initialized at the current time (so it is possible to emit an event at the very beginning of the program execution).

The scheduler then executes the program instructions one after the other in order. Depending of the kind of instruction reached by the scheduler, the execution is different:

- a control instruction is executed as soon as it is reached;
- when an effect instruction (e, d) is reached, the scheduler waits until the current time is equal or above the value of its time counter. As soon as this is the case the event e is emitted and the time counter value is set to the sum of the current time and the duration d.

This means that control instructions are executed as fast as possible but that the delay between two effect instructions is at least equal to the duration of the first one (notice that this delay could be larger if the duration of the first effect instruction is shorter than the time needed to execute all the control instructions in between the two effect instructions).

1.3.2 Execution of several programs in parallel

When several programs execute in parallel (as in step 3 in Figure 2) each runs as described in Section 1.3.1. The scheduler executes, in turn, one instruction from each program. The order in which the programs are considered is the order in which they started their execution. In case a program shall execute an effect instruction but the time for the event emission has not yet been met, its turn is skipped (so it does not pause all the other program executions).

1.4 Pattern, sequences, steps and some vocabulary

For the moment, we abstracted the exact way in which BuboCore scheduler handles steps. The idea is that there is an object that we call a *pattern* which is an array of objects called *sequences*. Each of these sequences is itself an array of *steps*. A step is constituted of a BILL program (that we call the program *associated* to this step) and a duration.

The BuboCore scheduler executes all the sequences in the pattern in parallel. For executing a sequence it starts at the first step in the array. Each steps is occurring for a time corresponding to its duration. At the end of a step, the scheduler switches to the next step in the same sequence. At the end of a sequence, the scheduler goes back to the start of this sequence. At the beginning of any step, the scheduler starts an execution of the corresponding BILL program. We call this execution an *instance* of the program.

1.5 How variables are handled

BILL programs can manipulate variables with control instructions and use them in effect instructions. These variables are of five kinds: environment variables, global variables, sequence variables, step variables and instance variables (Listing 2).

1.5.1 Environment variables

From the point of view of BILL programs, environment variables are read-only variables. Their values are set by the environment (think of time informations, random values, etc). A list of these variables is given in Section 2.5.

1.5.2 Global variables

Global variables are shared among all the BILL program executions.

1.5.3 Sequence variables

Sequence variables are shared among all the BILL programs of a given sequence (the sequence in which they are declared). They cannot be seen by programs associated to steps from other sequences.

1.5.4 Step variables

Step variables are shared among all the instances of the BILL program in which they are declared but are not seen by other programs.

```
pub enum Variable {
    Environment(String),
    Global(String),
    Sequence(String),
    Step(String),
    Instance(String),
    Constant(VariableValue),
}
```

Listing 2: Kinds of variables

```rust
pub enum VariableValue {
    Integer(i64),
    Float(f64),
    Bool(bool),
    Str(String),
    Func(Program),
    Dur(TimeSpan),
}
```

Listing 3: Types

1.5.5 Instance variables

Instance variables are local to the instance of the BILL program in which they are declared. So, if several instances (parallel or not) of the same program exist, each of them has its own version of these variables.

1.5.6 Variables with similar names

In BILL programs one refers to variables by their name but also has to explicitly state their kind. Therefore, there is no issue with variables of different kinds having the same name.

2 BILL: BuboCore Intermediate Low-level Language

In this section we describe all the control instructions (Section 2.3) and all the effect instructions (Section 2.4) available in the BILL language. These instructions use variables and durations and we explain how they behave in Section 2.1 and Section 2.2 respectively. We also list the environment variables (Section 2.5).

2.1 Types of variables

Each variable (being environment, global, sequence, step, or instance) and constant has a type.

2.1.1 Existing types

The possible types are given in Listing 3, which is an extract of the file `src/lang/variable.rs`.

Integers, Float, Bool, Str and Dur variables are used to store values that can be read or written by the instructions of a program.

Func variables are programs themselves, they can be executed by calling them with the CallFunction control instruction.

2.1.2 Type casting

Instructions arguments are typed: each instruction expects a particular type for each of its input arguments (unless specified otherwise) and has to respect the type of its (potential) output argument when writing to it.

In order to avoid errors, values that have not the expected type will be casted to the correct type, following the rules given in Table 1. In this table, $\perp$ denotes a function that does nothing (the program is a single return instruction).

2.2 Dealing with durations

According to Listing 4 (which is an extract of the file `src/clock.rs`), variables representing durations can hold three kinds of values: microseconds, beats, and steps. A duration expressed as microseconds is an absolute time. A duration expressed as beats is a relative time: the exact duration depends on the number of microseconds in a beat. A duration expressed as steps is a relative time as well: the exact duration depends on the number of beats in the step associated to the BILL program in which

From\To	Int	Float	Bool	Str	Func	Dur
Int		Represented as float	$0 \rightarrow$ `false` $\neq 0 \rightarrow$ `true`	Decimal representation	$\perp$	Absolute value as microseconds
Float	Rounded to int		$0 \rightarrow$ `false` $\neq 0 \rightarrow$ `true`	Decimal representation	$\perp$	Absolute value rounded to int as microseconds
Bool	`false` $\rightarrow 0$ `true` $\rightarrow 1$	`false` $\rightarrow 0.0$ `true` $\rightarrow 1.0$		`false` $\rightarrow$ "False" `true` $\rightarrow$ "True"	$\perp$	`false` $\rightarrow 0\mu s$ `true` $\rightarrow 1\mu s$
Str	Parsed as int (0 if error)	Parsed as float (0 if error)	"" $\rightarrow$ `false` $\neq$"" $\rightarrow$ `true`		$\perp$	Parsed as time duration (0 if error)
Func	$\perp \rightarrow 0$ $\neq \perp \rightarrow 1$	$\perp \rightarrow 0.0$ $\neq \perp \rightarrow 1.0$	$\perp \rightarrow$ `false` $\neq \perp \rightarrow$ `true`	Name of the function		$\perp \rightarrow 0\mu s$ $\neq \perp \rightarrow 1\mu s$
Dur	microseconds as int	microseconds represented as float	0ms $\rightarrow$ `false` $\neq$ 0ms $\rightarrow$ `true`	Time as string	$\perp$	

Table 1: Type casting rules.

```
pub enum TimeSpan {
    Micros(u64),
    Beats(f64),
    Steps(f64),
}
```

Listing 4: TimeSpan definition

the duration is used (that is, the step at which the program execution started). The duration of a beat or a step can be changed by BILL programs and by the environment.

Concrete durations are always expressed in microseconds. So, when a time-stamp must be associated to an event or when a delay must be applied the corresponding durations are converted to microseconds if needed. Before that, durations are always kept as general as possible: when an arithmetic operation is performed between two durations, the most concrete one is converted to the kind of the most general, as show in Table 2.

Sometimes, one may want the result of a computation on durations not to be as general as possible, e.g to be evaluated as microseconds immediately, to prevent the duration to change with changes to the beat duration or to a step duration. For that, we provide operations to change the concreteness of a duration in Section 2.3.

	microseconds	beats	steps
microseconds	microseconds	beats	steps
beats		beats	steps
steps			steps

Table 2: Result kinds in arithmetic operations between durations

```rust
pub enum ControlASM {
    // Arithmetic operations
    Add(Variable, Variable, Variable),
    Div(Variable, Variable, Variable),
    Mod(Variable, Variable, Variable),
    Mul(Variable, Variable, Variable),
    Sub(Variable, Variable, Variable),
    // Boolean operations
    And(Variable, Variable, Variable),
    Not(Variable, Variable),
    Or(Variable, Variable, Variable),
    Xor(Variable, Variable, Variable),
    // Comparisons
    LowerThan(Variable, Variable, Variable),
    LowerOrEqual(Variable, Variable, Variable),
    GreaterThan(Variable, Variable, Variable),
    GreaterOrEqual(Variable, Variable, Variable),
    Equal(Variable, Variable, Variable),
    Different(Variable, Variable, Variable),
    // Bitwise operations
    BitAnd(Variable, Variable, Variable),
    BitNot(Variable, Variable),
    BitOr(Variable, Variable, Variable),
    BitXor(Variable, Variable, Variable),
    ShiftLeft(Variable, Variable, Variable),
    ShiftRightA(Variable, Variable, Variable),
    ShiftRightL(Variable, Variable, Variable),
    // String operations
    Compile(Variable, String, Variable),
    Concat(Variable, Variable, Variable),
    Format(String, Vec<Variable>, Variable),
    // Time manipulation
    AsBeats(Variable, Variable),
    AsMicros(Variable, Variable),
    AsSteps(Variable, Variable),
    BeatsToNum(Variable, Variable),
    MicrosToNum(Variable, Variable),
    StepsToNum(Variable, Variable),
    FloatAsBeats(Variable, Variable),
    FloatAsSteps(Variable, Variable),
    // Memory manipulation
    DeclareGlobale(String, Variable),
    DeclareInstance(String, Variable),
    DeclareSequence(String, Variable),
    DeclareStep(String, Variable),
    Mov(Variable, Variable),
    // Stack operations
    Push(Variable),
    Pop(Variable),
    // Jumps
    Jump(usize),
    JumpIf(Variable, usize),
    JumpIfNot(Variable, usize),
    JumpIfDifferent(Variable, Variable, usize),
    JumpIfEqual(Variable, Variable, usize),
    JumpIfLess(Variable, Variable, usize),
    JumpIfLessOrEqual(Variable, Variable, usize),
    // Calls and returns
    CallFunction(Variable),
    CallProcedure(usize),
    Return,
}
```

Listing 5: Control instructions

2.3 Control instructions

Control instructions allow to perform basic operations (boolean and arithmetic) over variables. They also can change the control-flow of a program.

Concretely, a BILL program is a vector of instructions. At any time, the next instruction to be executed is given by a position in this vector (think of the program counter for a processor) that the scheduler stores. After executing an instruction, by default this position is increased by one. To alter the control-flow, a few instructions allow to arbitrarily change this position (jump instructions) or even to change the vector that represents the current program (call and return instructions).

The existing control instructions are given in Listing 5, which is an extract of the file `src/lang/control_asm.rs`.

2.3.1 Arithmetic operations

These instructions are all of the form `Op(x, y, z)`. Arguments x and y are inputs and z is an output. It is expected that x and y are two numbers of the same type (Int, Float or Dur). If this is not the case:

- if x is a number y will be casted to the type of x,
- else if y is a number x will be casted to the type of y,
- else they will both be casted to Int.

The result of the operation will be casted to the type of z (if needed).

Each instruction performs a different operation, as shown in Table 3.

2.3.2 Boolean operations

These instructions are all of the form `Op(x, y, z)` or `Op(x, z)`. Arguments x and y are inputs and will be casted to bool (if needed). Argument z is an output. The result of the operation will be casted to the type of z (if needed).

Op	Semantics	Remark
Add	$z \leftarrow x + y$	
Div	$z \leftarrow x/y$	$z \leftarrow 0$ if $y = 0$
Mod	$z \leftarrow x \bmod y$	$z \leftarrow x$ if $y = 0$ $z \leftarrow 0$ if y is a float
Mul	$z \leftarrow x \times y$	
Sub	$z \leftarrow x - y$	

Table 3: Arithmetic operations semantics

Op	Semantics	Remark
And	$z \leftarrow x \wedge y$	
Not	$z \leftarrow \neg x$	
Or	$z \leftarrow x \vee y$	
Xor	$z \leftarrow x \oplus y$	

Table 4: Boolean operations semantics

Each instruction performs a different operation, as shown in Table 4.

2.3.3 Comparisons

These instructions are all of the form `Op(x, y, z)`. Arguments x and y are inputs and z is an output. It is expected that x and y are two numbers of the same type (Int, Float or Dur). If this is not the case:

- if x is a number y will be casted to the type of x,
- else if y is a number x will be casted to the type of y,
- else they will both be casted to Int.

The result of the comparison is a boolean and will be casted to the type of z if needed (i.e. if z is not a boolean).

Each instruction performs a different comparison, as shown in Table 5.

2.3.4 Bitwise operations

These instructions are all of the form `Op(x, y, z)` or `Op(x, z)`. Arguments x and y are inputs and will be casted to int (if needed). Argument z is an output. The result of the operation will be casted to the type of z (if needed).

Each instruction performs a different operation, as shown in Table 6.

2.3.5 String operations

Compile(p, c, z). Cast p to a string. Try to compile it as a function with compiler c (if the compilation fail, the result is $\perp$). Store this function in z (after casting it to the type of z if needed).

Op	True if	Remark
LowerThan	$x < y$	
LowerOrEqual	$x \leq y$	
GreaterThan	$x > y$	
GreaterOrEqual	$x \geq y$	
Equal	$x = y$	
Different	$x \neq y$	

Table 5: Comparisons semantics

Concat(x, y, z). Arguments x and y are inputs and will be casted to str (if needed). Argument z is an output. The result of the operation will be casted to the type of z (if needed). Concat build the concatenation of x and y ($x.y$) and stores the result in z.

Format(f, x_1, ..., x_n, z). The first argument f is a format string. In other words it is a string containing placeholders for variable values (as in the C printf function for example) that will be replaced, in order, by the values of the x_1 to x_n arguments (which are variables in a vector). Each value is casted to the type associated with its placeholder. The result of this instruction is a string that will be stored in z (after appropriate cast if needed). Possible placeholders are: `%d` (Integer), `%f` (Float), `%b` (Bool), `%s` (Str), `%p` (Func), and `%t` (Dur).

2.3.6 Time manipulation

These instructions allow to perform conversions on durations.

Op	Semantics	Remark
BitAnd	$z \leftarrow x \& y$	
BitNot	$z \leftarrow \sim x$	
BitOr	$z \leftarrow x \mid y$	
BitXor	$z \leftarrow x \hat{} y$	
ShiftLeft	$z \leftarrow x \ll y$	$z \leftarrow x$ if $y < 0$
ShiftRightA	$z \leftarrow x \gg y$	arithmetic shift $z \leftarrow x$ if $y < 0$
ShiftRightL	$z \leftarrow x \gg y$	logical shift $z \leftarrow x$ if $y < 0$

Table 6: Bitwise operations semantics (C-like syntax)

AsBeats(d, v). Casts d to a duration. Set this duration to beats, cast it to the type of v, and then store it in v.

AsMicros(d, v). Casts d to a duration. Set this duration to microseconds, cast it to the type of v, and then store it in v.

AsSteps(d, v). Casts d to a duration. Set this duration to steps, cast it to the type of v, and then store it in v.

BeatsToNum(d, v). Casts d to a duration. Get the corresponding number of beats as a float, cast it to the type of v, and then store it in v.

MicrosToNum(d, v). Casts d to a duration. Get the corresponding number of microseconds as an int, cast it to the type of v, and then store it in v.

StepsToNum(d, v). Casts d to a duration. Get the corresponding number of steps as a float, cast it to the type of v, and then store it in v.

FloatAsBeats(f, v). Casts f to a float, then consider this float as a number of beats. Cast this number of beats to the type of v, and then store it in v.

FloatAsSteps(f, v). Casts f to a float, then consider this float as a number of steps. Cast this number of steps to the type of v, and then store it in v.

2.3.7 Memory manipulation

The four variable declaration instructions (DeclareGlobal, DeclareInstance, DeclareSequence, DeclareStep) are of the form `Declare(name, value)` and will create a new (Global, Instance, Sequence or Step) variable named `name` and initialize its value to `value`. The type of the new variable is the type of `value`.

Notice that, in any program instruction arguments, if a variable that does not exist is read this will give a 0 value. If a variable that does not exist is written, the variable will be created. (except if it is an environment variable, writing to environment variables has no effect).

The `mov(x, z)` instruction semantics is $z \leftarrow x$. If needed, the value of `x` will be casted to the type of `z`.

2.3.8 Stack operations

The BILL execution model provides a stack that can be used to store and retrieve variable values.

Push(v). The value of v is added on top of the stack. No type casting is performed so this value retains its type.

Pop(v). The value on top of the stack is removed from the stack and stored in v. The value is casted to the type of v.

2.3.9 Jumps

By default, the instructions of a BILL program are executed one after the other in the order in which they are stored in the vector representing the program. At each time, the position of the instruction to be executed is stored by the scheduler (think of a program counter for a processor). Assume that the place where this position is stored is called `pc`. By default, after executing an instruction, the scheduler increases `pc`: $\text{pc} \leftarrow \text{pc} + 1$. Jump instructions allow to replace this standard update of `pc` by something else, potentially based on a condition.

The semantics of the different jump instructions is given in Table 7. In each case, if the condition is `true` then $\text{pc} \leftarrow d \bmod n$ (where n is the number of instructions in the program). Else, $\text{pc} \leftarrow \text{pc} + 1$.

Instruction	Cond.	Remark
Jump(d)	`true`	
JumpIf(x, d)	x	x casted to Bool
JumpIfNot(x, d)	$\neg x$	x casted to Bool
JumpIfDifferent(x, y, d)	$x \neq y$	y casted to the type of x
JumpIfEqual(x, y, d)	$x = y$	y casted to the type of x
JumpIfLess(x, y, d)	$x < y$	y casted to the type of x
JumpIfLessOrEqual(x, y, d)	$x \leq y$	y casted to the type of x

Table 7: Jumps semantics

2.3.10 Calls and returns

A jump instruction always jumps to the same position in a program. Hence, one cannot use them to simulate procedure calls (the return position from a procedure depends on the point in code at which the jump to the procedure happened).

Calls are jumps that store, in a stack, the position from which they jumped. Returns are jumps that read in this stack to determine the position to which they jump. This stack will be called *return stack*.

CallFunction(f). Cast f to a program, then replace the current program p with f. Push $(p, \text{pc} + 1)$ into the return stack. Set `pc` to 0 (the start of the new program).

CallProcedure(pos). Push $(p, \text{pc} + 1)$ (where p is the current program) into the return stack. Set `pc` to pos.

Return. Pop (p, pos) from the return stack. Replace the current program with p (if needed) and set `pc` to pos. If the return stack is empty when using return, the program will end.

2.4 Effect instructions

Effect instructions are constituted of an *Event* and a *TimeSpan* (Listing 1). The Event describes the effect of the instruction on the World and the TimeSpan tells how much time shall elapse after the event occurs.

The existing events are given in Listing 6, which is an extract of the file `src/lang/event.rs`.

In this section we give the semantics of these events.

2.4.1 Meta events

Nop. Does nothing.

List(e). Performs all the events in e as fast as possible (that is, kind of simultaneously it there are not too much events in e), in the order in which they are given.

2.4.2 Music events

Music events are the events that actually allow to play sound on a given device. Not all devices accept all events.

```rust
pub enum Event {
    // Meta
    Nop,
    List(Vec<Event>),
    // Music
    MidiNote(Variable, Variable, Variable, Variable, Variable),
    MidiControl(Variable, Variable, Variable, Variable),
    MidiProgram(Variable, Variable, Variable),
    MidiAftertouch(Variable, Variable, Variable, Variable),
    MidiChannelPressure(Variable, Variable, Variable),
    MidiSystemExclusive(Vec<Variable>, Variable),
    MidiStart(Variable),
    MidiStop(Variable),
    MidiReset(Variable),
    MidiContinue(Variable),
    MidiClock(Variable),
    // Time handling
    SetBeatDuration(Variable),
    SetCurrentStepDuration(Variable),
    SetStepDuration(Variable, Variable),
    // Program starting
    Continue,
    ContinueInstance(Variable),
    ContinueOldest(Variable),
    ContinueSequence(Variable),
    ContinueSequenceOldest(Variable),
    ContinueSequenceYoungest(Variable),
    ContinueStep(Variable),
    ContinueStepOldest(Variable, Variable),
    ContinueStepYoungest(Variable, Variable),
    ContinueYoungest(Variable),
    Start(Variable, Variable),
    // Program halting
    Pause,
    PauseInstance(Variable),
    PauseOldest(Variable),
    PauseSequence(Variable),
    PauseSequenceOldest(Variable, Variable),
    PauseSequenceYoungest(Variable, Variable),
    PauseStep(Variable),
    PauseStepOldest(Variable, Variable),
    PauseStepYoungest(Variable, Variable),
    PauseYoungest(Variable),
    Stop,
    StopInstance(Variable),
    StopOldest(Variable),
    StopSequence(Variable),
    StopSequenceOldest(Variable, Variable),
    StopSequenceYoungest(Variable, Variable),
    StopStep(Variable),
    StopStepOldest(Variable, Variable),
    StopStepYoungest(Variable, Variable),
    StopYoungest(Variable),
}
```

Listing 6: Event definition

At the moment, only Midi events are available. Each MidiXXX event takes as last parameter the name of the Midi device to which the event should be sent. This parameter is casted to a string if needed. All the other parameters are casted to integers, a modulo is then performed (modulo 128 in general, modulo 16 for the channel arguments) and the result is used in the Midi message resulting from the event. This is process is detailed for the MidiNote event below and is similar for all other events.

MidiNote(n, v, c, dur, dev). Plays note n (casted to int and modulo 128 used as a midi value) with velocity v (casted to int and modulo 128) on channel c (casted to int and modulo 16) for *dur* (casted to a duration and set to microseconds) microseconds on device *dev* (casted to a string, and changed to the special log device if this string does not identifies a known device).

MidiControl(ctr, v, c, dev). Sends control message *ctr* with value v on channel c to device *dev*.

MidiProgram(prg, c, dev). Sends program message *prg* on channel c to device *dev*.

MidiAftertouch(n, p, c, dev). Sends after touch message for note n and pressure p on channel c to device *dev*.

MidiChannelPressure(p, c, dev). Sends channel pressure message with pressure p on channel c to device *dev*.

MidiSystemExclusive(data, dev). Sends a system exclusive message with data *data* to device *dev*.

MidiStart(dev), MidiStop(dev), MidiReset(dev), MidiContinue(dev), MidiClock(dev). Sends the corresponding midi message to device *dev*.

2.4.3 Time handling events

Time handling events allow to manage the relations between beats, step duration, and absolute time.

SetBeatDuration(t). Sets the duration of one beat to t (casted to a duration). This duration is set in microseconds (absolute time) by first evaluating t in microseconds. The standard use is to give t in microseconds to setup a tempo. However, one could give t in beats for relative change of tempo (if t is 3 beats the tempo is divided by 3 as the duration of a beat is multiplied by 3).

SetCurrentStepDuration(t). Sets the duration of the step associated to the program instance calling this instruction to t (casted to a duration). This duration is set in beats if possible or, else, it is set in microseconds. The standard use is to give t in beats, so that if beat duration changes step duration changes accordingly. However one could give t in microseconds to avoid this side effect.

SetStepDuration(n, t). Same as SetCurrentStepDuration but for step n (casted to an int). See Section 2.5 for knowing how to get step numbers.

2.4.4 Program starting events

Starting events allow to initiate new program instances (*start*) and to resume execution of program instances that were previously paused (*continue*). How program instances can be paused is described in Section 2.4.5.

Continue. Resumes all currently paused program instances.

ContinueInstance(n). Resumes the program instance with number n (casted to an int). See Section 2.5 for knowing how to get instance numbers.

ContinueOldest(k). Resumes the k (casted to an int) program instances that were paused the longest time ago.

ContinueSequence(n). Resumes all currently paused program instances corresponding to steps in sequence n (casted to an int). See Section 2.5 for knowing how to get sequence numbers.

ContinueSequenceOldest(n, k). Resumes the k (casted to an int) program instances corresponding to steps in sequence n (casted to an int) that were paused the longest time ago. See Section 2.5 for knowing how to get sequence numbers.

ContinueSequenceYoungest(n, k). Resumes the k (casted to an int) program instances corresponding to steps in sequence n (casted to an int) that were paused the shortest time ago. See Section 2.5 for knowing how to get sequence numbers.

ContinueStep(n). Resumes all currently paused program instances corresponding to step n (casted to an int). See Section 2.5 for knowing how to get step numbers.

ContinueStepOldest(n, k). Resumes the k (casted to an int) program instances corresponding to step n (casted to an int) that were paused the longest time ago. See Section 2.5 for knowing how to get step numbers.

ContinueStepYoungest(n, k). Resumes the k (casted to an int) program instances corresponding to step n (casted to an int) that were paused the shortest time ago. See Section 2.5 for knowing how to get step numbers.

ContinueYoungest(k). Resumes the k (casted to an int) program instances that were paused the shortest time ago.

Start(p, i). Starts a new instance of program p. If p is a function, then this function is used as a program. Else the program corresponding to step p (casted to an int) is used. The number of the new instance is recorded in i (after casting it to the type of i). Remark that such a program instance is associated to the step and the sequence to which the program instance in which Start was called is associated.

2.4.5 Program halting events

Halting events are of two kinds: *stop* events and *pause* events. Stop events will end the execution of a (set of) program(s) instance(s). Pause events will pause the execution of a (set of) program(s) instance(s) allowing to continue their execution from the point at which they where paused using program starting events (Section 2.4.4).

We describe here the stop events as the corresponding pause events have the same behavior.

Stop. Stops all the program instances currently running.

StopInstance(n). Stops the program instance with number n (casted to an int). See Section 2.5 for knowing how to get instance numbers.

StopOldest(k). Stops the k (casted to an int) oldest program instances (that started the longest time ago).

StopSequence(n). Stops all the program instances corresponding to steps in sequence number n (casted to an int). See Section 2.5 for knowing how to get sequence numbers.

StopSequenceOldest(n, k). Stops the k (casted to an int) oldest program instances (that started the longest time ago) corresponding to steps in sequence number n (casted to an int). See Section 2.5 for knowing how to get sequence numbers.

StopSequenceYoungest(n, k). Stops the k (casted to an int) youngest program instances (that started the shortest time ago) corresponding to steps in sequence number n (casted to an int). See Section 2.5 for knowing how to get sequence numbers.

StopStep(n). Stops all the program instances corresponding to step number n (casted to an int). See Section 2.5 for knowing how to get step numbers.

StopStepOldest(n, k). Stops the k (casted to an int) oldest program instances (that started the longest time ago) corresponding to step number n (casted to an int). See Section 2.5 for knowing how to get step numbers.

StopStepYoungest(n, k). Stops the k (casted to an int) youngest program instances (that started the shortest time ago) corresponding to step number n (casted to an int). See Section 2.5 for knowing how to get step numbers.

StopYoungest(k). Stops the k (casted to an int) youngest program instances (that started the shortest time ago).

2.5 Environment variables

TODO: on aurait envie d'avoir des variables d'environnement qui sont des ensembles, comment faire ? Ça demande sans doute d'ajouter un type de variable ? On voudrait aussi paramétrer les variables d'environnement mais en l'état ce n'est pas trop possible (par exemple pour obtenir le nombre de pas dans la séquence n), la version actuelle ne fonctionne pas vraiment car on ne peut pas construire les noms de variable dans un programme BILL. Il faut peut-être que les variables en question soient remplacées par des évènements (getters)

TODO: à ajouter dans l'outil (mais pas tout de suite, il faut d'abord voir comment ça devrait marcher exactement)

The environment variables provided by BuboCore are given below. Some of them are parameterized for simplicity. Parameters are depicted here between dollars signs, they should be replaced by integers. For example, Sequence$n$NumSteps corresponds to the variables Sequence1NumSteps, Sequence2NumSteps, and so on.

- **InstanceID.** ID of this program instance.
- **Instance$n$SequenceID.** ID of the sequence containing the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$SequenceBeats.** Number of beats in the sequence containing the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$SequenceMicros.** Number of microseconds in the sequence containing the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$StepID.** ID of the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$StepBeats.** Number of beats in the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$StepMicros.** Number of microseconds in the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$SequenceNumInstances.** Same as NumInstances but only for instances corresponding to the sequence containing the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$SequenceNumRunning.** Same as NumRunning but only for instances corresponding to the sequence containing the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$SequenceNumPaused.** Same as NumPaused but only for instances corresponding to the sequence containing the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).

- **Instance$n$StepNumInstances.** Same as NumInstances but only for instances corresponding to the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$StepNumRunning.** Same as NumRunning but only for instances corresponding to the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **Instance$n$StepNumPaused.** Same as NumPaused but only for instances corresponding to the step associated to the program instance number n (or the instance of the program using this variable if n is omitted).
- **NumInstances.** Number of instances currently running or paused.
- **NumPaused.** Number of instances currently paused.
- **NumRunning.** Number of instances currently running.
- **NumSequences.** Number of sequences.
- **Sequence$n$NumSteps.** Number of steps in sequence number n.
- **SequenceID.** ID of the sequence containing the step corresponding to this program.
- **Sequence$n$Beats.** Number of beats in the sequence number n (or the sequence containing the step associated to the program using this variable if n is omitted).
- **Sequence$n$Micros.** Number of microseconds in the sequence number n (or the sequence containing the step associated to the program using this variable if n is omitted).
- **Sequence$n$NumInstances.** Same as NumInstances but only for instances corresponding to the sequence number n (or the sequence containing the step associated to the program using this variable if n is omitted).
- **Sequence$n$NumRunning.** Same as NumRunning but only for instances corresponding to the sequence number n (or the sequence containing the step associated to the program using this variable if n is omitted).
- **Sequence$n$NumPaused.** Same as NumPaused but only for instances corresponding to the sequence number n (or the sequence containing the step associated to the program using this variable if n is omitted).
- **StepID.** ID of the step corresponding to this program.
- **Step$n$SequenceID.** ID of the sequence containing the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$SequenceBeats.** Number of beats in the sequence containing the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$SequenceMicros.** Number of microseconds in the sequence containing the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$Beats.** Number of beats in the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$Micros.** Number of microseconds in the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$NumInstances.** Same as NumInstances but only for instances corresponding to the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$NumRunning.** Same as NumRunning but only for instances corresponding to the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$NumPaused.** Same as NumPaused but only for instances corresponding to the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$SequenceNumInstances.** Same as NumInstances but only for instances corresponding to the sequence containing the step number n (or the step associated to the program using this variable if n is omitted).

- **Step$n$SequenceNumRunning.** Same as NumRunning but only for instances corresponding to the sequence containing the step number n (or the step associated to the program using this variable if n is omitted).
- **Step$n$SequenceNumPaused.** Same as NumPaused but only for instances corresponding to the sequence containing the step number n (or the step associated to the program using this variable if n is omitted).
- **TotalBeats.** Number of beats since the launch of BuboCore.
- **TotalMicros.** Number of microseconds since the launch of BuboCore. This cannot be computed from TotalBeats as the duration of a beat may have changed over time.
- **BeatMicros.** Number of microseconds in a beat.

3 Guidelines for building a custom scripting language

In order to build a custom scripting language one needs to be able to compile it to BILL. There are two possibilities for writing a compiler compatible with BuboCore:

- compilers written in RUST can be integrated as modules of BuboCore, and
- compilers built with any technology can be provided as binaries that BuboCore will call.

The first method is preferred as it allows to directly build BILL programs as RUST data-structures and ensures a better integration into BuboCore. It also allows to easily distribute new scripting languages: a simple pull request on our Github repository[1] will let us integrate any scripting language into the next versions of BuboCore.

3.1 Compiler integration into BuboCore

Building a custom scripting langage requires to know how to build BuboCore, please refer to the appropriate document[2] for that part.

In order to add a compiler for a new script language one has to comply with the following guidelines:

- create a directory with the name of the language in `src/compiler/` and implement the `compiler` trait by providing a `compile` function that given a script in the language (provided as a string) produces the corresponding BILL code,
- create a `.rs` file with the name of the language in `src/compiler/` and export (`pub use`) the `compiler` implementation,
- declare the new module (`pub mod`) in the `src/compiler.rs` file.

As an example, one can have a look at the *dummylang* language that has been created for testing purposes while developing BuboCore:

- the compiler trait is implemented in `src/compiler/dummylang/dummycompiler.rs`,
- it is exported in `src/compiler/dummylang/dummylang.rs` by the line `pub use dummycompiler::DummyCompiler;`,
- it is declared in `src/compiler.rs` by the line `pub mod dummylang;`.

3.2 Compiler as a standalone binary

TODO: regarder comment ça marche et faire un exemple

[1] https://github.com/Bubobubobubobubo/deep-bubocore

[2] TODO (quand la doc pour construire BuboCore sera écrite il faudra la référencer ici)