Antescofo
A not-so-short introduction to version 0.5x

IRCAM UMR STMS 9912 – CNRS – UPMC – INRIA/MuTant

Revision July 22, 2014

Document prepared by Jean-Louis Giavitto
José Echeveste and MuTant Team Members.

Antescofo is a coupling of a real-time listening machine with a reactive synchronous language. The language is used for authoring of music pieces involving live musicians and computer processes, and the real-time system assures its *correct* performance and synchronization despite listening or performance errors. In version 0.5x, the listening machine has been improved and the language accepted by the reactive module of Antescofo has greatly evolved.

This document is a reference for the new architecture starting from version 0.5. The presentation is mainly syntax driven and it supposes that you are familiar with *Antescofo*. The objective is to give enough syntax to upgrade the old *Antescofo* score in the few place where it is needed and to enable the reader to start experimenting with the new features. Please refer to the examples and tutorial to have sensible illustrations of the language.

Additional information on *Antescofo* can be found:

- on the project home page
  `http://repmus.ircam.fr/antescofo`
- on the IrcamForum User Group
  `http://forumnet.ircam.fr/user-groups/antescofo/`
  where you can find tutorials to download with bundles for MAX and PureData
- on the IrcamForge pages of the project
  `http://forge.ircam.fr/p/antescofo/`
- on the web site of the MuTanT project
  `http://repmus.ircam.fr/mutant`
  where you can find the scientific and technical publications on *Antescofo*.

Table of Contents

How to use this document 3

Brief history of Antescofo 3

1 Understanding *Antescofo* scores 4
- 1.1 Structure of an *Antescofo* Score . . 4
- 1.2 Elements of an *Antescofo* Score . . . 6
- 1.3 Identifiers 9
- 1.4 Comments and Indentation 10

2 Events 12
- 2.1 Event Specification 12
- 2.2 Event Parameters 12
- 2.3 Events as Containers 13
- 2.4 Event Attributes 14

3 *Antescofo* Model of Time 16
- 3.1 Logical Instant 17
- 3.2 Time Frame 18
- 3.3 Locating an Action in Time 19

4 Actions in Brief 20
- 4.1 Delays 20
- 4.2 Label 21

5 Atomic Actions 23
- 5.1 Message passing to Max/PD 23
- 5.2 OSC Messages 24
- 5.3 Assignments 26
- 5.4 Aborting and Cancelling an Action . 27
- 5.5 I/O in a File 29
- 5.6 Internal Commands 29
- 5.7 Assertion `@assert` 32

6 Compound Actions 33
- 6.1 `Group` 33
- 6.2 Conditional Actions: `If` 36
- 6.3 Sequential iterations: `Loop` 36
- 6.4 Parallel Iterations: `Forall` 37
- 6.5 Sampling parameters: `Curve` 38
- 6.6 Reacting to logical events: `Whenever` 42

7 Expressions 47
- 7.1 Values 47
- 7.2 Scalar Values 48
- 7.3 Aggregate Values 50
- 7.4 Variables 60
- 7.5 Operators and Predefined Functions 64
- 7.6 Auto-Delimited Expressions in Actions 65

8 Synchronization and Error Handling Strategies 66
- 8.1 Synchronization Strategies 66
- 8.2 Missed Event Errors Strategies . . . 68

9 Process 71
- 9.1 Macro *vs.* Processus 72
- 9.2 Recursive Process 72
- 9.3 Process as Values 72
- 9.4 Aborting a Process 73

10 Macros 74
- 10.1 Macro Definitions 74
- 10.2 Expansion Sequence 75
- 10.3 Generating New Names 75

11 *Antescofo* Workflow 78
- 11.1 Editing the Score 78
- 11.2 Tuning the *Antescofo* Listening Machine 78
- 11.3 Debuging an *Antescofo* Score 78
- 11.4 Dealing with Errors 78
- 11.5 Interacting with MAX 80
- 11.6 Interacting with PureData 81
- 11.7 *Antescofo* Standalone Offline 81
- 11.8 Old Syntax 83

12 Stay Tuned 84

A Library of Predefined Functions 85

B Experimental Features 96
- B.1 Simple Curve *à la* `line~` 96
- B.2 Tracks 96
- B.3 Abort Handler 98
- B.4 Patterns 99
- B.5 Reserved Experimental Keywords . . 109

C Changes in 0.51 110

D Index 111

E Detailed Table of Contents 120

How to use this document

This document is to be used as a reference guide to *Antescofo* language for artists, composers, musicians as well as computer scientists. It describes the new architecture and new language of Antescofo starting version 0.5 and above. The presentation is mainly syntax driven and it supposes that you are familiar with *Antescofo*. Users willing to practice the language are strongly invited to download *Antescofo* and use the additional Max tutorials (with example programs) that comes with it for a sensible illustrations of the language. Available resources in addition to this document are:

- on the project home page
  `http://repmus.ircam.fr/antescofo`
- on the IrcamForum User Group
  `http://forumnet.ircam.fr/user-groups/antescofo/`
  where you can find a tutorials to download with bundles for MAX and PureData
- on the IrcamForge pages of the project
  `http://forge.ircam.fr/p/antescofo/`
- on the web site of the MuTanT project
  `http://repmus.ircam.fr/mutant`
  where you can find the scientific and technical publications on *Antescofo*.

Please, send your comments, typos, bugs and suggestions about this document on the *Antescofo* forum web pages. It will help us to improve the documentation.

Brief history of *Antescofo*

Antescofo project started in 2007 as a joint project between a researcher (Arshia Cont) and a composer (Marco Stroppa) with the aim of composing an interactive piece for saxophone and live computer programs where the system acts as a *Cyber Physical Music System*. It became rapidly a system coupling a simple action language and a machine listening system. The language was further used by other composers such as Jonathan Harvey, Philippe Manoury, Emmanuel Nunes and the system was featured in world-class music concerts with ensembles such as Los Angeles Philharmonics, NewYork Philharmonics, Berlin Philharmonics, BBC Orchestra and more.

In 2011, two computer scientists (Jean-Louis Giavitto from CNRS and Florent Jacquemard from Inria) joined the team and serious development on the language started with participation of José Echeveste (currently a PhD candidate) and the new team *MuTant* was baptized early 2012 as a joint venture between Ircam, CNRS, Inria and UPMC in Paris.

Antescofo has gone through an incremental development in-line with user requests. The current language is highly dynamic and addresses requests from more than 40 serious artists using the system for their system. Besides its incremental development with users and artists, the language is highly inspired by *Synchronous Reactive* languages such as *ESTEREL* and *Cyber-Physical Systems*.

1 Understanding *Antescofo* scores

1.1 Structure of an *Antescofo* Score

An *Antescofo* score is a text file, accompanied by its dedicated GUI *AscoGraph*, that is used for real-time score following (detecting the position and tempo of live musicians in a give score) and triggering electronics as written by the artists. An *Antescofo* score thus has two main elements:

EVENTS are elements to be recognized by the score follower or machine listener, describing the dynamics of the outside environment. They consist of `NOTE`, `CHORD`, `TRILL` and other elements discussed in details in section 2.

ACTIONS are elements to be undertaken once corresponding event(s) or conditions are recognized.

Actions in *Antescofo* extend the good-old *qlist* object elements in MAX and PD with additional *Models of Time* (chapter 3), *Expressions* (chapter 7), *Compound Actions* (chapter 6), *Synchronization Strategies*, *Processes* and more covered in separate chapters of this document.

Figure 1 shows a sample score of *Antescofo* corresponding to first two measures of *Tensio* by composer Philippe Manoury composed in 2010 for string quartet and live electronics as seen in *AscoGraph*. The graphical representation on the left is a visual interpretation of the *Antescofo* text score on the right.

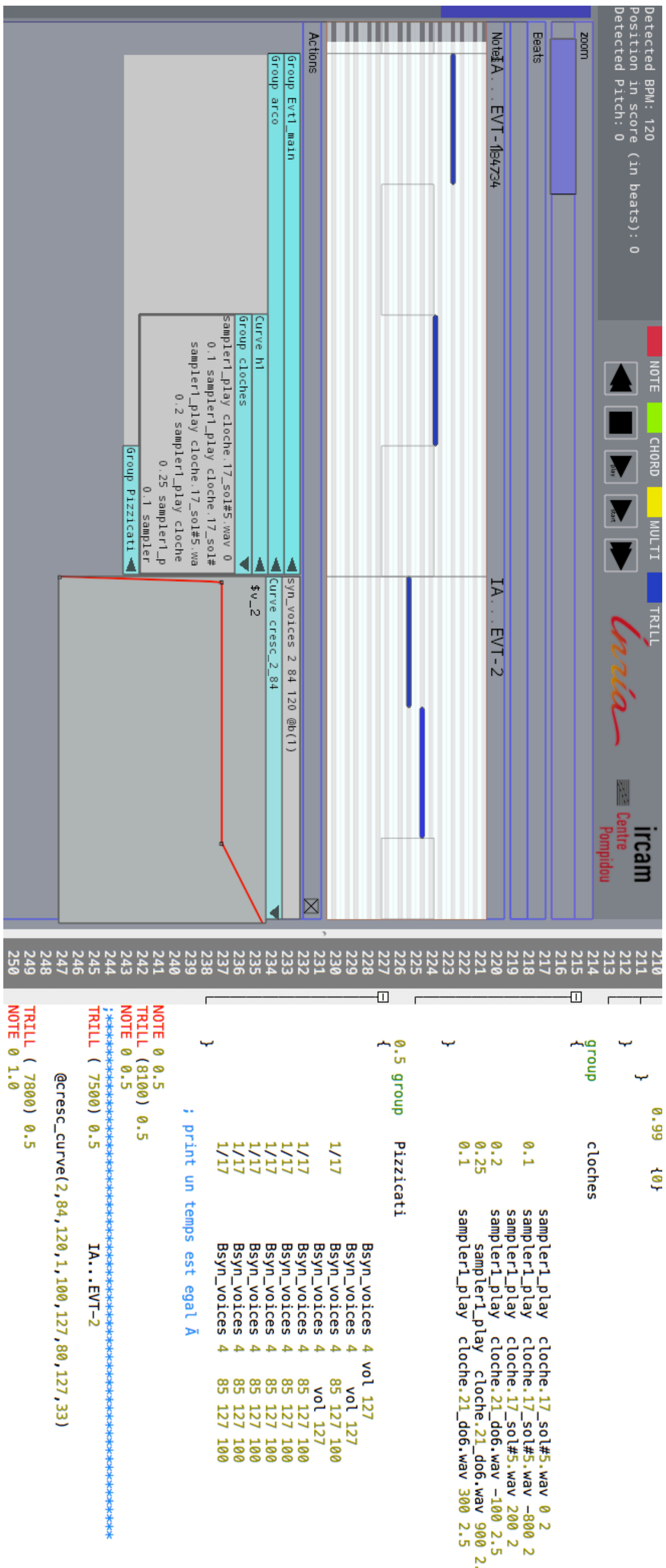
In Figure 1, the score for human musician contains four `TRILL`s (not all visible in text) with a mixture of discrete and continuous compound actions as written by the composer. Particularly, the `TRILL` labeled as `IA...EVT-2` has a continuous action associated as its action generated by a pre-defined macro `cresc_curve` (here, controlling the volume of a sound synthesis program) where as prior to that a compound `group` named `cloches` is supposed to synchronize atomic actions (seen in text and collapsed group box in the visual screen) with the musician.

A textual *Antescofo* score, or program, is written in a file. It can be partitioned into several files, using the `@insert` feature:

```
@insert macro.asco.txt
@insert "file␣name␣with␣white␣space␣must␣be␣quoted"
```

The `@insert` keyword can be capitalized: `@INSERT`, as any other keyword beginning with a `@` sign. An included file may includes (other) files. Macros can be defined to reuse program fragments, see section 10 (see also functions page 7 and processes section 9).

In this document, the *Antescofo* code fragments are colorized. The color code is as follows: keywords related to *file inclusion, function, process and macro definitions* are in **purple**, *event related keywords* are in **red**, keywords related to *actions* are in **blue**, *comments* are in gray, *strings* are in **green**.



```
Detected BPM: 120
Position in score (in beats): 0
Detected Pitch: 0
```

```
sampler1_play cloche.17_sol#5.wav 0
   0.1 sampler1_play cloche.17_sol#
   sampler1_play cloche.17_sol#5.wa
       0.2 sampler1_play cloche
            0.25 sampler1_p
               0.1 sampler
```

```
                0.99   {0}
            }
       }

       group       cloches
       {
                          sampler1_play   cloche.17_sol#5.wav 0 2
                  0.1     sampler1_play   cloche.17_sol#5.wav -800 2
                          sampler1_play   cloche.17_sol#5.wav 200 2
                  0.2     sampler1_play   cloche.21_do6.wav -100 2.5
                  0.25        sampler1_play   cloche.21_do6.wav 900 2.5
                  0.1     sampler1_play   cloche.21_do6.wav 300 2.5
       }

       0.5 group   Pizzicati
       {
                              Bsyn_voices 4 vol 127
                              Bsyn_voices 4    vol 127
                  1/17        Bsyn_voices 4   85 127 100
                              Bsyn_voices 4    vol 127
                  1/17        Bsyn_voices 4   85 127 100
                  1/17        Bsyn_voices 4   85 127 100
                  1/17        Bsyn_voices 4   85 127 100
                  1/17        Bsyn_voices 4   85 127 100
                  1/17        Bsyn_voices 4   85 127 100
                  1/17        Bsyn_voices 4   85 127 100
       }
                ; print un temps est egal Ã

NOTE 0 0.5
TRILL (8100) 0.5
NOTE 0 0.5
;***************************************************************
TRILL ( 7500) 0.5               IA...EVT-2

          @cresc_curve(2,84,120,1,100,127,80,127,33)

TRILL ( 7800) 0.5
NOTE 0 1.0
```

Figure 1: The beginning of *Tensio* (2010) by Philippe Manoury for String Quartet and Live electronics in *AscoGraph*

1.2 Elements of an *Antescofo* Score

The language developed in *Antescofo* can be seen as a domain specific synchronous and timed reactive language in which the accompaniment actions of a mixed score are specified together with the instrumental part to follow.

As a consequence, an *Antescofo* program is a sequence of *statements*, *events* and *actions*. Events are recognized by the listening machine. They are described in section 2. Actions, outlined in in sections 4 and 5, are computations triggered by the occurence of an event or of another action. The model of time of *Antescofo* is described in section 3 and the synchronization between events and actions is described in section 8. Actions are parameterized by *expressions* evaluated during the run of the program; they are described in section 7. Statements parameterize the behavior of the listening machine, of the reactive machine and the interactions between *Antescofo* and its environment. Statements include:

- *Score alterations and management of the listening machine:*
  - `bpm` specification: `bpm 60` or `bpm 50 @modulate` (the pulsation can be given as an integer or a float);
  - `transpose` *t* transposes the specification of the pitches in the following events by *t* (in midicents ; it can also be written `@transpose`);
  - variance specification: `variance 3.7`;
- *Management of the interactions with the environment:*
  - computation of the tempo: `tempo on` and `tempo off` (can also be written `@tempo`);
  - external variable binding: `bind $mouse_position`;
  - additional connection with Max/MSP: `@inlet x`; ;
  - enabling the trace with Notability: `napro_trace`;
- *Definitions for the action language:*
  - function definition;
  - process definition;
  - track definition;
  - pattern definition;
  - macro definition.

REMARK: An Antescofo text score is interpreted from top to bottom. In this sense, *score alteration* commands such as `bpm` or `variance` will affect lines that follow its appearance. Example: Figure 2 shows two simple *Antescofo* scores. In the left score, the second tempo change to 90 BPM will be affected starting on the event with label `Measure2` and as a consequence, the delay 1/2 for its corresponding action is launched with 90 BPM. On the other hand, in the right score the tempo change will affect the chord following that event

```
BPM 60
NOTE C4 1.0 Measure1
CHORD (C4 E4) 2.0
NOTE G4 1.0
BPM 90
NOTE C5 1.0 Measure2
1/2 print action1
CHORD (C5 E5) 2.0
NOTE A4 1.0
```

```
BPM 60
NOTE C4 1.0 Measure1
CHORD (C4 E4) 2.0
NOTE G4 1.0
NOTE C5 1.0 Measure2
1/2 print action1
BPM 90
CHORD (C5 E5) 2.0
NOTE A4 1.0
```

Figure 2: Example of score attribute affectation (top-down parsing) in *Antescofo* text scores.

onwards and consequently, the action delay of $1/2$ beat-time hooked on note $C5$ corresponds to a score tempo of 60 BPM.

Statement cannot appears within an action. They must be at the top level in the file. However, most of these statements correspond also to an internal action, see section 5.6. Macros, processus and functions can be used only after their definition in the score. We suggest to put them at the beginning of the file or to put them in a separate file that will be included at the beginning of the score.

Function Definition. Functions are applied to values to return a value. There is three kind of functions in *Antescofo*:

- predefined functions,
- user-defined intentional functions (specified by an expression),
- user-defined extensional functions (specified by data).

Functions in *Antescofo* are first class values. They are two main operations on this kind of values: they can be applied to arguments (function call) and they can be passed as argument to other functions or process (see next paragraph). A function application can appear everywhere an expression is expected.

Predefined functions are referred through a predefined @-name (case-sensitive). They include logarithmic, exponential and trigonometric functions, simple string manipulations and so on. See section 7 where they are listed in the subsection related to the type of their principal argument.

A user-defined intentional function definition takes the form

```
  @fun_def @factorial($x) { $x < 1 ? 1 : $x * @factorial($x) }
```

(indentation and carriage-return do not matters). The name of a function is an @-identifier as the names of predefined functions. The body of the function is an expression between braces. The parameters are $-identifiers. Such functions can be recursive. See section 7.2.6 for additional information.

A user-defined extensional function is a dictionary defined by giving a list of pairs (`key`, `value`), see section 7.3.1. When bot `key` and `value` are numeric, they corresponding function can be interpolated between the breakpoints, see section 7.3.2).

Process Definition. Process are for actions what functions are for values. A process definition takes the form

```
@proc_def ::Filter($x)
{
              filter on
    $x        filter off
    (2 * $x) ::Filter($x)
}
```

The name of a process is an `::`-identifiers. The parameters are `$`-identifiers. Process can be recursive and the name of a process can be used in expressions (*e.g.*, as the argument of another process). The example shows a recursive process that turn on and off a filter `filter` until *Antescofo* stops.

Processes in *Antescofo* are first class values. The values of this type, *proc*, represent a process definition. They are two main operations on this kind of values: they can be applied to arguments (process instantiation) and they can be passed as argument to other functions call or processes instantiations.

A process application is an action. However, it is also an expression that can be used for instance in the right hand side of an assignment. The returned value is an *exec*. This kind of value represents the running process and should not be confused with *proc*. They are used for instance to kill a specific running process. See section 9 for additional information on processes.

Process are a new feature in *Antescofo* and their use is promoted over macros.

Macro Definition. A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. Functions and processes are usually more convenient than macros. For example, macro-expansion is a purely textual device and occurs before any execution by the system. So, the code fragment corresponding to a macro is not restricted to be an expression (as for functions) or an action (as for process). However, there some constraints apply, *e.g.* macros cannot be recursive. See section 10 for additional information.

Function, Process and Macro Application. The application of a function, a process or a macro is denoted by the juxtaposition of the name of the function, process or macro with the arguments between parenthesis. In case of multiple arguments, they are separated by a comma.

1.3 Identifiers

The four different kinds of identifiers that exist in the language have been mentionned above: *simple identifier*, `@`*-identifiers*, `$`*-identifiers* and `::`*-identifiers*.

Accentuated characters can be used only in simple identifier (labels and MAX name). For the other identifiers, stick to ASCII characters (up to 128).

1.3.1 Simple Identifiers

Simple identifiers, also called *symbols*, like `id`, `id_1`, `id-1` are simple alphabetic characters followed by alphabetic, numeric and special characters. They can start with a digit if the digit is followed by at least two simple alphabetic characters. They cannot start by `@`, `$` or `::`. The special characters include the four arithmetic operators `+ - * /` and latin accentuated characters but the score file must be coded in UTF-8.

Simple identifiers are used for labels, external name (*e.g.*, Max, PD or file names) and some keywords. The current list of simple identifiers that are reserved for keywords is given in Fig. 3. These keyword are *case unsensitive*, that is

```
note NOTE Note NoTe notE
```

all denote the same keyword. But the other simple identifiers (*e.g.*, labels and external names) are *case sensitive*.

There is a long list of reserved keywords and a clash with an external name can happen. In this case, just use a *string* instead to refer to the external symbol.

`abort action assert at`	`let lfwd loop loose`
`before bind bpm`	`map ms multi`
`cfwd chord closefile curve`	`napro_trace note`
`do during`	`of off on openoutfile oscoff`
	`oscon oscrecv oscsend`
`else event expr`	`parfor pattern port`
`false forall`	`s start stop symb`
`gfwd global group guard`	`tab transpose trill true`
`hook`	`until`
`if imap in`	`value variance`
`jump`	`whenever where while`
`kill`	

Figure 3: Reserved keywords

1.3.2 `@`-identifiers

`@`-identifiers like `@id`, `@id_1` are simple identifier prefixed by an *at* sign (`@`). Only `! ? .` and `_` are allowed as special characters.

`@`-identifier are used to name function and macros. In this case, they are *case sensitive*.

They are also some reserved @-identifiers used for various definitions:

```
@fun_def @insert @lid @macro_def @pattern_def @proc_def @track_def
@uid
```

They are also used for the various attributes of an action, namely:

```
@abort @action @coef @date @global @grain @guard @hook @immediate
@jump @kill @label @label @local @map_history @map_history_date
@map_history_rdate @modulate @name @norec @rdate @tempo @tight
@transpose @type
```

These attributes are *case unsensitive*, that is `@tight`, `@TiGhT` and `@TIGHT` are the same keyword. For compatibility reason, there is an overlap between simple identifiers and the @-identifiers (without the @).

1.3.3 $-identifiers

$-identifiers like `$id`, `$id_1` are simple identifier prefixed with a dollar sign. However, only `! ? .` and `_` are allowed as special characters. $-identifier are used to give a name to variables and function and macro parameters.

1.3.4 ::-identifiers

::-identifiers like `::P` or `::q1` are simple identifier prefixed with two semi-columns. ::-identifiers are used to give a name to processus (see section 9).

1.4 Comments and Indentation

Bloc comments are in the C-style and cannot be nested:

```
/*  comment split
    on several lines
*/
```

Line-comment are in the C-style and also in the Lisp style:

```
// comment until the end of the line
; comment until the end of the line
```

Tabulations are handled like white spaces. Columns are not meaningful so you can indent *Antescofo* program as you wish. *However* some constructs must end on the same line as their "head identifier": event specification, internal commands and *external actions* (like Max message or OSC commands).

For example, the following fragment raises a parse error:

```
NOTE
C4 0.5
1.0s print
     "message␣to␣print"
```

(because the pitch and the duration of the note does not appear on the same line as the keyword NOTE and because the argument of print is not on the same line). But this one is correct:

```
Note C4 0.5 "some␣label␣used␣to␣document␣the␣score"
1.0s
print "this␣is␣a␣Max␣message␣(to␣the␣print␣object)"
print "printed␣1␣seconds␣after␣the␣event␣Note␣C4..."
```

Note that the first print is indented after the specification of its delay (1.0s) but ends on the same line as its "head identifier", achieving one of the customary indentations used for cue lists.

A backslash before an end-of-line can be used to specify that the next line must be considered as a continuation of the current line. It allows for instance to split the list of the arguments of a message on several physical rows:

```
print "this␣two" \
      "messages" \
      "are␣equivalent"
print "this␣two" "messages" "are␣equivalent"
```

2 Events

An event in *Antescofo* terminology corresponds to an element of the sequence defining the dynamics of the environment (in this case, a musician interpreting a piece of written music). They are used by the listening machine to detect position and tempo of the musician (along other inferred parameters) which are by themselves used by the reactive and scheduling machine of *Antescofo* to produce synchronized accompaniments.

The listening machine specifically is in charge of real-time automatic alignment of an audio stream played by one or more musicians, into a symbolic musical score described by Events. The *Antescofo* listening machine is polyphonic[1] and constantly decodes the tempo of the live performer. This is achieved by explicit time models inspired by cognitive models of musical synchrony in the brain[2] which provide both the tempo of the musician in real-time and also the *anticipated* position of future events (used for real-time scheduling).

2.1 Event Specification

Events are detected by the listening machine in the audio stream. The specification of an event starts by a keyword defining the kind of event expected and some additional parameters:

```
NOTE pitch  ...
CHORD  (pitch_list)  ...
TRILL  (trill_list)  ...
MULTI  (multi_list)  ...
MULTI  (multi  -> multi)  ...
```

followed by the mandatory specification of a duration and optionally by some attributes. They must be followed by a carriage return (in other word, an event specification is the last thing on a line). There is an additional kind of event

```
EVENT  ...
```

also followed by a mandatory duration, which correspond to waiting a click on the "next event" button on the graphical interface.

The duration of an event is specified by a float, an integer or the ratio of two integers like `4/3`.

2.2 Event Parameters

The parameters of an event are as follows:

[1] Readers curious on the algorithmic details of the listening machine can refer to : A. Cont. A coupled duration-focused architecture for realtime music to score alignment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(6):974–987, 2010.

[2] E. Large and M. Jones. The dynamics of attending: How people track time-varying events. *Psychological review*, 106(1):119, 1999.

pitch is given by a number representing the pitch in midi or midicent, or a note name. A negative pitch means that its corresponding note is tie with the same note of the previous event. For instance

```
pitch D4 3/2
```

represents the occurence of a `D` with a duration of 1.5 beat.

pitch_list is a sequence of *pitch*es:

```
D4b 1200 112 D5#
```

is a list of 4 notes.

trill_list is a sequence of: 1) *pitch*es and 2) sequences of *pitch*s (between parenthesis):

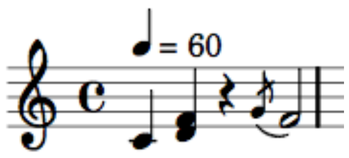
```
D4b (E3 A3) D5
```

multi_list is a sequence of multi and a multi is a *pitch* or a *pitch_list* or a *pitch_list* followed by a quote:

```
(E3 A3)'
```

2.3 Events as Containers

Each event keyword in *Antescofo* in the above listing can be seen as *containers* with specific behavior and given nominal durations. A `NOTE` is a container of *one* pitch. A `chord` contains a vector of pitches. Figure 4 shows an example including simple notes and chords written in *Antescofo*:

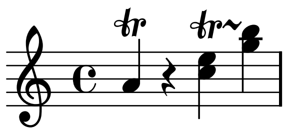


```
BPM 60
NOTE C4 1.0
CHORD (D4 F4) 1.0
NOTE 0   1.0      ; a silence
NOTE G4 0.0       ; a grace note with duration zero
NOTE F4 2.0
```

Figure 4: Simple score with notes and chords.

The two additional keywords `Trill` and `Multi` are also containers with specific extended behaviors:

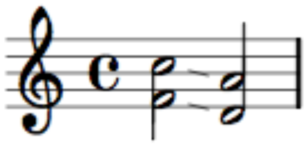
Trill Similar to trills in classical music, a `Trill` is a container of events either as atomic pitches or chords, where the internal elements can happen in any specific order. Additionally, internal events in a `Trill` are not obliged to happen in the environment. This way, `Trill` can be additionally used to notate improvisation boxes where musicians are free to choose elements. A `Trill` is considered as a global event with a nominal relative duration. Figure 5 shows basic examples for Trill.



```
TRILL  (A4 A#4) 1.0
NOTE 0  1.0              ; a silence
TRILL ( (C5 E5) (G5 B5) ) 1.0
```

Figure 5: TRILL example on notes and chords

Multi Similar to Trill, a Multi is a compound event (that can contain notes, chords or event trills) but where the *order* of actions are to be respected and decoded accordingly in the listening machine. They can model continuous events such as *glissando*. Figure 6 shows an example of glissandi between chords written by Multi.



```
MULTI ( (F4 C5) -> (D4 A4) ) 4.0
```

Figure 6: MULTI example on chords

2.4 Event Attributes

They are three kinds of event attributes and they are all optional:

- The keyword hook (or @hook) specifies that this event cannot be missed (the listening machine need to wait the occurrence of this event and cannot presume that it can be missed).
- A simple identifier or a string or an integer acts as a label for this event. They can be several such labels. If the label is a simple identifier, its $-form can be used in a expression elsewhere in the score to denote the time in beat of the onset of the event.
- The keyword jump (or @jump) is followed by a comma separated list of simple identifiers referring to the label of an event in the score. This attribute specifies that this event can be followed by several continuations: the next event in the score, as well as the events listed by the @jump.

These attribute can be given in any order. For instance:

```
Note D4 1 here @jump l1, l2
```

defines an event labeled by here which is potentially followed by the next event (in the file) or the events labeled by l1 and l2 in the score. Note that

```
Note D4 1 @jump l1, l2 here
```

is the same specification: `here` is not interpreted as the argument of the jump but as a label for the event because there is no comma after `12`.

3 *Antescofo* Model of Time

Actions are computations triggered after a delay that elapses starting from the occurrence of an event or another action. In this way, *Antescofo* is both a reactive system, where computations are triggered by the occurrence of an event, and a temporized system, where computations are triggered at some date. The three main components of the *Antescofo* system architecture are sketched in Fig. 7:

- The *scheduler* takes care of the various time coordinate specified in the score and manage all delays, wait time and pending tasks.
- The *environment* handle the memory store of the system: the history of the variables, the management of references and all the notification and event signalization.
- The *evaluation engine* is in charge of parsing the score and of the instantaneous evaluation of the expressions and of the actions.

They are several *temporal coordinate systems*, or *time frame*, that can be used to locate the occurrence of an event or an action and to define a duration.

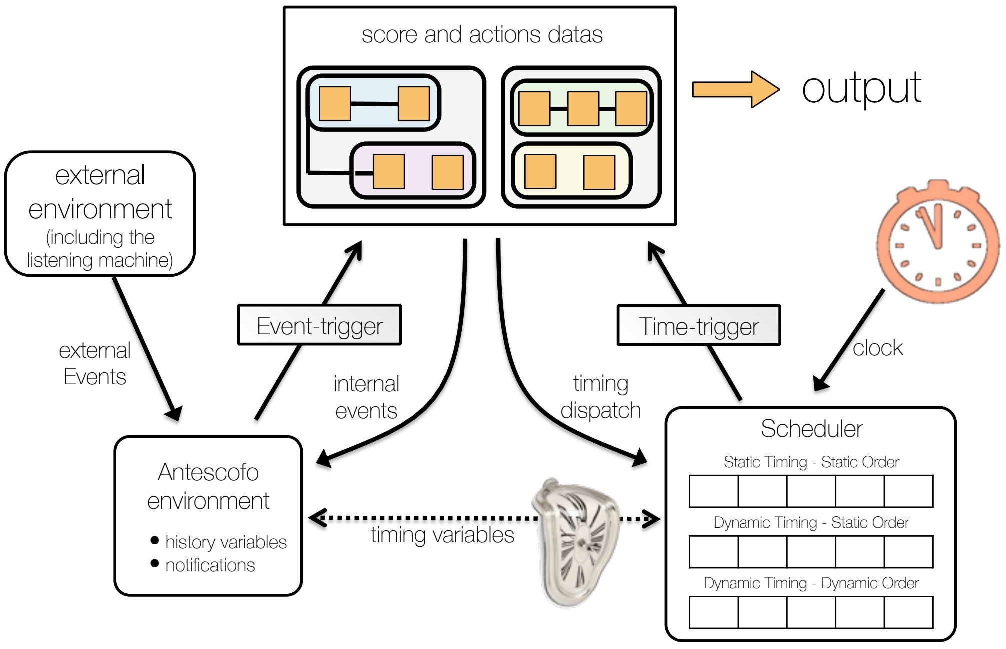


Figure 7: The *Antescofo* system architecture. An action is spanned because the recognition of a musical event, a notification of the external environment (*i.e.*, external assignment of a variable or an OSC message), internal variable assignment by the functioning of the program itself, or the expiration of a delay. Actions are launched after a delay which can be expressed in various time frame.

3.1 Logical Instant

A *logical instant* is an instant in time distinguished because it corresponds to:

- the recognition of a musical event;
- the assignment of a variable by the external environment (*e.g.* through an OSC message or a MAX/PD binding);
- the expiration of a delay.

Such instant has a date (*i.e.* a coordinate) in each time frame. The notion of logical instant is instrumental to maintain the synchronous abstraction of actions and to reduce temporal approximation. Whenever a logical instant is started, the internal variables `$NOW` (current date in the physical time frame) and `$RNOW` (current date in the relative time frame) are updated, see section 7.4. Within the same logical instant, synchronous actions are performed sequentially in the same order as in the score.

Computations are supposed to take no time and thus, atomic actions are performed inside one logical instant of zero duration. This abstraction is a useful simplification to understand the scheduling of actions in a score. In the real world, computations take time but this time can be usually ignored and do not disturb the scheduling planned at the score level. In figure 8, the sequence of synchronous actions appears in the vertical axis. So this axis corresponds to the dependency between simultaneous computations. Note for example that

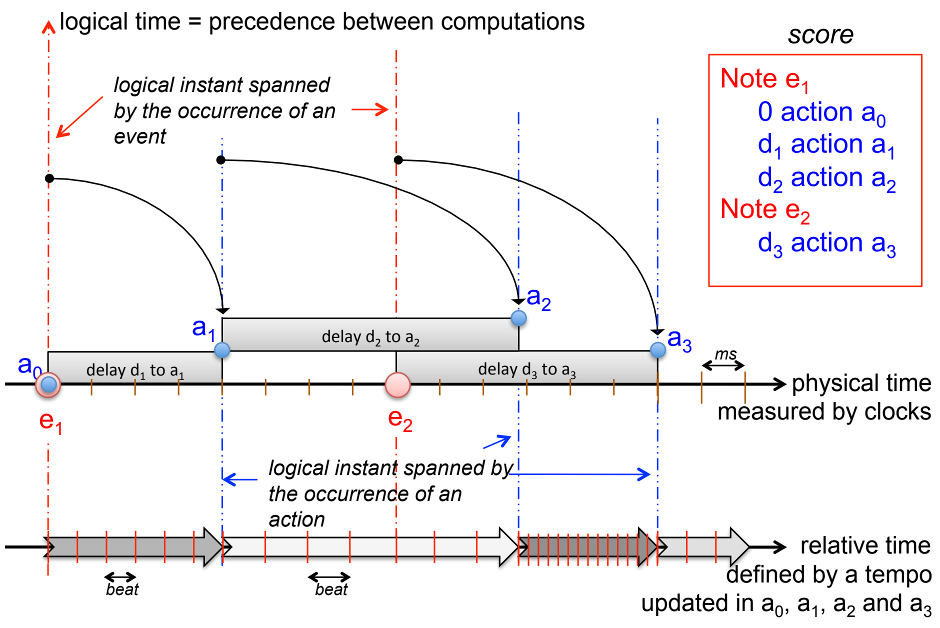

logical time = precedence between computations
score
Note e_1
0 action a_0
d_1 action a_1
d_2 action a_2
Note e_2
d_3 action a_3
logical instant spanned by the occurrence of an event
a_0
a_1
a_2
a_3
delay d_1 to a_1
delay d_2 to a_2
delay d_3 to a_3
ms
physical time measured by clocks
e_1
e_2
logical instant spanned by the occurrence of an action
relative time defined by a tempo updated in a_0, a_1, a_2 and a_3
beat
beat

Figure 8: Logical instant, physical time frame and relative time frame corresponding to a computed tempo. Notice that the (vertical) height of a box is used to represent the logical dependencies while the (horizontal) length of a box represents a duration in time.

even if $\mathtt{d}_1$ and $\mathtt{d}_2$ are both zero, the execution order of actions $\mathtt{a}_0$, $\mathtt{a}_1$ and $\mathtt{a}_2$ is the same as the appearance order in the score.

Two different logical instants are located at two distinct points in the physical time, in the horizontal axis. They are several ways to locate these instants.

3.2 Time Frame

Frames of reference, or *time frames* are used to interpret delays and to give a date to the occurrence of an event or to the launching of an action. Two frames of reference are commonly used:

- the physical time $\mathcal{P}$ expressed in seconds and measured by a clock (also called *wall clock time*),
- and the relative time which measure the progression of the performance in the score measured in beats.

More generally, a frame of reference $\mathcal{T}$ is defined by a *tempo* $T_\mathcal{T}$ which specifies the "passing of time in $\mathcal{T}$" relatively to the physical time[3]. In short, a tempo is expressed as a number of beats per minutes. The tempo $T_\mathcal{T}$ can be any *Antescofo* expression. The date $t_\mathcal{P}$ of the occurrence of an event in the physical time and the date $t_\mathcal{T}$ of the same event in the relative time $\mathcal{T}$ are linked by the equation:

$$t_\mathcal{T} = \int_0^{t_\mathcal{P}} T_\mathcal{T} \tag{1}$$

Variable updates are discrete in *Antescofo*; so, in this equation, $T_\mathcal{T}$ is interpreted as a piece-wise constant function.

Programmers may introduce their own frames of reference by specifying a tempo local to a group of actions using a dedicated attribute, see section 6.1. This frame of reference is used for all relative delays and datation used in the actions within this group. The tempo expression is evaluated continuously in time for computing dynamically the relationships specified by equation (1).

Antescofo provides a predefined dynamic tempo variable through the system variable `$RT_TEMPO`. This tempo is refered as "*the* tempo" and has a tremendous importance because it is the time frame naturally associated with the musician part of the score[4]. This variable is extracted from the audio stream by the listening machine, relying on cognitive model of

[3] Mazzola, G., & Zahorka, O. (1994). *Tempo curves revisited: Hierarchies of performance fields.* Computer Music Journal, 18(1), 40-52.

[4]The `$RT_TEMPO` is computed by *Antescofo* to mimics the tracking of the tempo by a human, and implements an idea of smooth tempo fluctuation, rather than trying to satisfy exactly equation (1) at any moment. So, for *the* relative time frame, equation (1) is only an approximation. As a consequence, the current position in the score is explicitly given by the variable `$BEAT_POS` which is more accurate than the integration of `$RT_TEMPO`. See paragraph 7.4.3.

musician behavior[5]. The corresponding frame of reference is used when we speak of "relative time" without additional qualifier.

3.3 Locating an Action in Time

Given a time frame, there are several ways to implement the specification of the occurrence of an action. For instance, consider action a_2 in figure 8 and suppose that $d_1 + d_2$ is greater than `1.5` beat (the duration of the event NOTE e_1). Then action a_2 can be launched either:

- $(d_1 + d_2)$ beats after the occurrence of the event NOTE e_1,
- *or* $(d_1 + d_2 - 1.5)$ beats after the occurrence of the event NOTE e_2

(several other variations are possible).

In the "ideal interpretation of the score", these two ways of computing the launching date of action a_2 are equivalent because the event NOTE e_2 occurs *exactly* after 1.5 beat after event NOTE e_1. *But* this is not the case in an actual performance.

Antescofo allows a composer to choose the right way to compute the date of an action in a time frame, to best match the musical context. This is the purpose of the *synchronization strategy*. They are described in section 8.

[5] A. Cont. *A coupled duration-focused architecture for realtime music to score alignment.* IEEE Transaction on Pattern Analysis and Machine Intelligence, Juin 2010, Vol. 32, n○6, pp 974–987.

4 Actions in Brief

Think of *actions* as what *Antescofo* undertakes as a result of arriving at an instant in time. In traditional practices of interactive music, actions are *message passing* through *qlist* object in Max/Pd (or alternatively message boxes or *COLL, PATTR* objects in MAX). Actions in *Antescofo* allow more explicit organisation of computer reactions over time and also with regards to themselves. See section 5.1 for a detailed description of message passing mechanism.

Actions are divided into *atomic actions* performing an elementary computation or simple message passing, and *compound actions*. Compound actions group others actions allowing *polyphony*, *loops* and *interpolated curves*. An action is triggered by the event or the action that immediately precedes it.

In the new syntax, an action, either atomic or compound, starts with an optional *delay*, as defined hereafter. The old syntax for compound action, where the delay is after the keyword, is still recognized.

Action Attributes. Each action has some optional attributes which appear as a comma separated list:

```
atomic_action @att1 , @att2 := value
compound_action @att1 , @att2 := value { ... }
```

In this example, `@att1` is an attribute limited to one keyword, and `@att2` is an attribute that require a parameter. The parameter is given after the optional sign `:=`.

Some attributes are specific to some kind of actions. There is however one attribute that can be specified for all actions: *label*. It is described in sections 4.2. The attributes specific to a given kind of action are described in the section dedicated to this kind of action.

4.1 Delays

An optional specification of a *delay* d can be given before any action a. This delay defines the amount of time between the previous event or the previous action in the score and the computation of a. At the expiration of the delay, we say that the action is *fired* (we use also the word *triggered* or *launched*). Thus, the following sequence

```
NOTE C 2.0
   d₁ action1
   d₂ action2
NOTE D 1.0
```

specifies that, in an ideal performance that adheres strictly to the temporal constraint specified in the score, `action1` will be fired d1 after the recognition of the C note, and `action2` will be triggered d2 after the launching of `action1`.

A delay can be any expression. This expression is evaluated when the preceding event is launched. That is, expression d2 is evaluated in the logical instant where `action1` is computed. If the result is not a number, an error is signaled.

Zero Delay. The absence of a delay is equivalent to a zero delay. A zero-delayed action is launched synchronously with the preceding action or with the recognition of its associated event. Synchronous actions are performed in the same logical instant and last zero time, cf. paragraph 3.1.

Absolute and Relative Delay. A delay can be either absolute or relative. An absolute delay is expressed in seconds (respectively in milliseconds) an refer to wall clock time or physical time. The qualifier `s` (respectively `ms`) is used to denote an absolute delay:

```
            a0
       1s   a1
  (2*$v)ms  a2
```

Action *a*1 occurs one seconds after *a*0 and *a*2 occurs `2*$v` milliseconds after *a*1. If the qualifier `s` or `ms` is missing, the delay is expressed in beat and it is relative to the tempo of the enclosing group (see section 6.1.1).

Evaluation of a Delay. In the previous example, the delay for *a*2 implies a computation whose result may depend of the date of the computation (for instance, the variable `$v` may be updated somewhere else in parallel). So, it is important to know when the computation of a delay occurs: it takes place when the previous action is launched, since the launching of this action is also the start of the delay. And the delay of the first action in a group is computed when the group is launched.

A second remark is that, once computed, the delay itself is not reevaluated until its expiration. However, the delay can be expressed in the relative tempo or relatively to a computed tempo and its mapping into the physical time is reevaluated as needed, that is, when the tempo changes.

Synchronization Strategies. Delays can be seen as temporal relationships between actions. There are several ways, called *synchronization strategies*, to implement these temporal relationships at runtime. For instance, assuming that in the first example of this section *action*2 actually occurs *after* the occurrence of `NOTE D`, one may count a delay of $d_1 + d_2 - 2.0$ starting from `NOTE D` after launching *action*2. This approach will be for instance more tightly coupled with the stream of musical events. Synchronization strategies are discussed in section 8.2.

4.2 Label

Labels are used to refers to an action. As for events, the label of an action can be

- a simple identifier,

- a string,
- an integer.

The label of an action are specified using the `@name` keyword:

```
... @name := somelabel
... @name somelabel
```

They can be several label for the same action. Contrary to the label of an event, the $-identifier associated to the label of an action cannot be used to refer to the relative position of this action in the score[6].

Compound actions have an optional identifier (section 6). This identifier is a simple identifier and act as a label for the action.

[6]There is no useful notion of position of an action in the score because the same action may be fired several times (actions inside a `loop` or a `whenever` or associated to a `curve`).

5 Atomic Actions

An atomic action corresponds to

- message passing: to MAX/PD receives or an OSC message,
- an assignment,
- the abort of another action;
- an internal command,
- an assertion.

5.1 Message passing to Max/PD

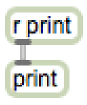
The simplest form of action in *Antescofo* is send some values to a *receive* object in MAX or PD. This way, *Antescofo* acts as a coordinator between multiple tasks (machine listening and actions themselves) attempting to deliver actions deterministically as they have been authored despite changes from musicians or controllers. These actions are simply equivalent to *message boxes* and their usage is similar to *qlist* object in MAX/PD with the extension of the notion of Delay (see section 4.1). They take the familiar form of:

```
<optional-delay> <receiver-name> <message-content>
```

Since such actions are destined for interaction with external processes (in MAX/PD), we refer to them as *external actions*. They are currently two main mechanisms to interact with external tasks: OSC messages described in section 5.2 and MAX/PD messages[7].

A MAX/PD message starts by an optional delay followed by a symbol identifier referring to a MAX or PD receiver. This identifier must be different from the simple identifiers listed in section 1.3 page 9. They should correspond to a *receiver* object in MAX/PD with the same identifier[8]. For example, the following action attempts to send its message to a receiver called "print" in MAX/PD whose patch might look like the figure on its right:

```
NOTE C4 1.0
    print I will be printed upon recognition of C4
    0.5 print I will be printed next, after 0.5 beats
            print Comma separated mess as in MAX
```



What follows the receiver identifier can be a sequence of expressions, simple identifiers and @-identifiers that are the arguments of the message.

[7]The interaction with MAX or PD is asymmetric: inlet and outlet are used to interact with the rest of a patch, but provide a fixed interface, see sections 5.6 and 11.5. On the contrary, arbitrary messages can be sent from an *Antescofo* score to external MAX objects with appropriate receivers.

[8]As you go on, you will notice that everything in *Antescofo* can be dynamic. The receiver identifier can also be calculated using *string concatenation* (see section 7.2.5). In this case, use `@command()`. For example, if the value of $num is 1, `@command("spat"+$num)` builds the "spat1" receiver.

The message ends with a carriage return (the end of the line) or a closing brace. A message can span several lines, but the intermediate lines must end with a backslash \.

For instance,

```
$a := 1   ; This is an assignment! see section 3 of this chapter
print "the␣value␣of␣the␣variable␣a␣is␣" $a
print and here is \
      a second message \
      (2 * $a) "specified␣on␣3␣lines␣(note␣the␣\\)"
```

will print

```
the value of the variable a is 1
and here is a second message 2 specified on 3 lines (note the \)
```

Antescofo expressions are evaluated to give the argument of the message. For the first `print`, there are two arguments: a string and a variable which evaluates to `1`. Each *Antescofo* value is converted into the appropriate MAX/PD value (*Antescofo* string are converted into MAX/PD symbols, *Antescofo* float into MAX/PD float, etc.). In the second `print` message there are 8 arguments: the first six are simple identifiers converted into the corresponding symbol, the seventh argument is evaluated into an integer and the last is a string. The backslash character has a special meaning and must be "backslashed" to appear in the string, see sect. 7.2.5.

When an *Antescofo* string is converted into a MAX/PD string, the delimiters (the quote `"`) do not appear. If one want these delimiters, you have to introduce it explicitly in the string, using an escaped quote `\"`:

```
print "\"this␣string␣will␣appear␣quoted\""
```

prints the following to MAX/PD console

```
"this␣string␣will␣appear␣quoted"
```

5.2 OSC Messages

Many people have been using *Antescofo* message passing strategy as defined above to interact with processes living outside MAX/PD (such as CSound, SuperCollider, etc.). To make their life easier, *Antescofo* comes with a builtin OSC host. The OSC protocol[9] can be used to interact with external processes using the UDP protocol. It can also be used to make two *Antescofo* objects interact within the same patch. Contrary to MAX or PD messages, OSC message can be sent and received at the level of the *Antescofo* program. The embedding of OSC in *Antescofo* is done through 4 primitives.

[9] `http://opensoundcontrol.org/`

5.2.1 OSCSEND

This keyword introduces the declaration of a named OSC output channel of communication. The declaration takes the form:

```
oscsend name host : port msg_prefix
```

After the OSC channel has been declared, it can be used to send messages. Sending a message takes a form similar to sending a message to MAX or PD:

```
name arg1 ... arg_n
```

The idea is that this construct and send the osc message

```
msg_prefix arg1 ... arg_n
```

where *msg_prefix* is the OSC address declared for *name*. *Note that to handle different message prefixes, different output channels have to be declared.* The character / is accepted in an identifier, so the usual hierarchical name used in message prefixes can be used to identify the output channels. For instance, the declarations:

```
oscsend extprocess/start test.ircam.fr : 3245 "start"
oscsend extprocess/stop  test.ircam.fr : 3245 "stop"
```

can be used to invoke later

```
0.0 extprocess/start "filter1"
1.5 extprocess/stop "filter1"
```

The arguments of an `oscsend` declaration are as follow:

- *name* is a simple identifier and refers to the output channel (used later to send messages).
- *host* is the optional IP address (in the form *nn.nn.nn.nn* where *nn* is an integer) or the symbolic name of the host (in the form of a simple identifier). If this argument is not provided, the `localhost` (that is, IP 127.0.0.1) is assumed.
- *port* is the mandatory number of the port where the message is routed.
- *msg_prefix* is the OSC address in the form of a string.

A message can be send as soon as the output channel has been declared. Note that sending a message before the definition of the corresponding output channel is interpreted as sending a message to MAX.

5.2.2 OSCRECV

This keyword introduces the declaration of an input channel of communication. The declaration takes the form:

```
oscrecv name port msg_prefix $v_1 ... $v_n
```

where:

- *name* is the identifier of the input channel, and its used later to stop or restart the listening of the channel.
- *port* is the mandatory number of the port where the message is routed.
- On the previous port, the channel accepts messages with OSC address *msg_prefix*. Note that for a given input channel, the message prefixes have to be all different.
- When an OSC message is received, the argument are automatically dispatched in the variables $\$v_1 \ldots \$v_n$. If there is less variables than arguments, the remaining arguments are simply thrown away . Otherwise, if there is less arguments than variables, the remaining variables are set to their past value .

  Currently, *Antescofo* accepts only OSC int32, int64, float and string. These value are converted respectively into *Antescofo* integer, float and string.

A `whenever` can be used to react to the reception of an OSC message: it is enough to put one of the variables $\$v_i$ as the condition of the whenever (see below).

The reception is active as soon as the input channel is declared.

5.2.3 `OSCON` and `OSCOFF`

These two commands take the name of an input channel. Switching off an input channel stops the listening and the message that arrives after, are ignored. Switching on restarts the listening. These commands have no effect on an output channel.

5.3 Assignments

The assignment of a variable by the value of an expression is an atomic action:

```
let $v := expr
```

The `let` keyword is optional but make more clear the distinction between the delay and the assigned variable:

```
$d $x := 1 ; is equivalent to
$d let $x := 1
```

and the `let` keyword it is mandatory in vector assignment with a delay (see sect.??):

```
$d let $t[$index] := $val
```

Expressions in the right hand side of := are described in section 7. A variable as a value before its first assignment: its value is the `undefined` value (sect. 7.2.1).

The assignment of a value to a variable may triggers some activity:

- the evaluation of a `whenever` that depends on this variable (see section 6.6);

- the reevaluation of the delays that depends on a relative tempo that depends on this variable[10];

System variables cannot be assigned: `$RT_TEMPO`, `$PITCH`, `$BEAT_POS`, `$LAST_EVENT_LABEL`, `$DURATION`, `$NOW`, `$RNOW`. These variables are *read-only* for the composer: they are assigned by the system during the performance. However, like usual variables, their assignment may trigger some activities. For instance delays expressed in the relative time are updated on `$RT_TEMPO` changes. Tight actions waiting on a specific event (cf. section 8.1.2) are notified on `$BEAT_POS` changes. Etc. Refer to section 7.4.4 for additional information.

5.4 Aborting and Cancelling an Action

An atomic action takes "no time" to be processed. So, *aborting* an atomic action is irrelevant: the action is either already fired or has not already been fired. On the other hand, compound actions described in section 6 act as containers for others actions and thus span over a duration. We say that a compound action is *active* when it has been fired itself but some of its nested actions are still waiting to be fired. Compound actions can be aborted while they are active.

Cancelling an action refers to another notion: the suppression of an action from the score. Both atomic and compound action can be cancelled.

5.4.1 Abort of an Action

After a compound action has been launched, it can be aborted, meaning that the nested actions not already fired, will be aborted. They are two possible syntax:

```
kill delay name
delay abort name
```

where *name* is the label of an action. If the named action is atomic or not active, the command has no effect. If the named action is an active compound action, the nested remaining actions are aborted.

Beware that distinct actions may share the same label: all active actions labeled by *name* are aborted together. Also one action can have several occurrences (*e.g.* the body of a `loop` or the body of a `whenever` see section 6.6). All occurrences of an action labeled by *name* are aborted.

The `abort` command accepts also the name of a process as argument. In this case, all active instances of this process are aborted.

[10]As mentioned in section 4.1, the expression specifying a delay is evaluated only once, when the delay is started. It is not re-evaluated after that, even if the variable in the expression are assigned to new values. However, if the delay is expressed in a relative time, its conversion in physical time must be adjusted when the corresponding tempo changes.

`Abort` and the hierarchical structure of compound actions. By default, the abort command applies recursively on the whole hierarchical structure of actions (cf. section 6). Notice that the actions launched by a process call in a context C are considered as descendants of C.

The attribute `@norec` can be used to abort only the top level actions of the compound. Here is an example:

```
1  group G1 {
2    1 a1
3    1 group G2 {
4        0.2 b1
5        0.5 b2
6        0.5 b3
7      }
8    1 a2
9    1 a3
10 }
11 2.5 abort G1
```

The action `abort` takes place at `2.5` beats after the firing of `G1`. At this date, actions *a*1 and *b*1 have already been fired. The results of the abort is to suppress the future firing of *a*2, *a*3, *b*2 and *b*3. If line 11 is replaced by

```
2.5 abort G1 @norec
```

then, actions *a*2 and *a*3 are aborted but not actions *b*2 and *b*3.

5.4.2 Cancelling an Action

The action

```
 kill delay nameA of nameG
delay abort nameA of nameG
```

cancels the action labeled *nameA* in the group labeled *nameG*. *Cancelling* an action make sense only if the action has not been already fired. For example, if the action is in a loop, the cancellation has an effect only on the firing of the action that are in the future of the cancellation.

The effect of cancelling an action is similar to its syntactic suppression from the score. Here is an example

```
group G1 {
  1 a1
  0 abort action_to_suppress of G1
  1 a2 @name := action_to_supress
  1 a3
}
```

The cancelling of the action at line 4 by the abort ... of at line 3 results in firing action $a1$ at date 1 and action $a3$ at date 2.

Notice that this behavior departs in two ways from the previous abort command: (1) you can inhibit an atomic action, and (2) the following actions are fired earlier because the delay of the inhibited action is suppressed. This second point also distinguish the behavior of inhibited action from the behavior of a conditional action when the condition evaluates to false (compare with the example in section ??).

5.5 I/O in a File

Actually it is only possible to write an output file. The schema is similar to OSC messages: a first declaration opens and binds a file to a symbol. This symbol is then used to write out in the file. Then the file is eventually closed. Here is a typical example:

```
openoutfile out "/tmp/tmp.txt" opt_int
...
out "\n\tHello␣Wolrd\n\n"
...
closefile out
```

After the command openoutfile, the symbol out can be used to write in file /tmp/tmp.txt. In command, out is followed by a list of expressions, as for OSC or MAX/PD commands. Special characters in strings are interpreted as usual.

The optional integer *opt_int* at the end of the openoutfile is interpreted as follow: if negative or null, the associated buffer is shrink to zero and the outputs are always flushed immediately to the file. If positive, this number is used as a multiplier of the default file buffer size. Factors greater than one increase the size of the buffer and thus reduce the number of effective i/o. The effect is usually negligible[11].

The file is automatically closed at *Antescofo* exit. Beware that because file buffering, the content of the file may be not entirely written on disk before closing it. If not explicitly closed, the file remains open between program load, start and play. Currently, there is only one possible mode to open a file: if it does not exists, it is created. If it already exists, it is truncated to zero at opening.

5.6 Internal Commands

... *This section is still to be written*

Internal commands correspond to the MAX or PD messages accepted by the antescofo object in a patch. The "internalization" of these messages as *Antescofo* primitive actions makes possible the control of the MAX or the PD antescofo object from within an *Antescofo* score itself.

[11] If the i/o's interfere with the scheduling, consider to use the host environment to implement them (*i.e.* rely on Max or PD buffer to minimize the impact on time sensitive resources).

Internal commands are named `antescofo::`*xxx* where the suffix *xxx* is the head of the corresponding MAX/PD message (cf. section 11.5):

- `antescofo::actions` *string* :
- `antescofo::analysis` *int* :
- `antescofo::tempo` *float* :
- `antescofo::before_nextlabel` (no argument) :
- `antescofo::bpmtolerance` *float* :
- `antescofo::calibrate` *int int int* :
- `antescofo::clear` (no argument) :
- `antescofo::gamma` *float* :
- `antescofo::getcues` (no argument) :
- `antescofo::getlabels` (no argument) :
- `antescofo::gotobeat` *float* :
- `antescofo::gotocue` *string* :
- `antescofo::gotolabel` *string* :
- `antescofo::harmlist` *float* ... (a list of floats corresponding to a vector) :
- `antescofo::info` (no argument) :
- `antescofo::jumptocue` *string* :
- `antescofo::jumptolabel` *string* :
- `antescofo::killall` (no argument) :
- `antescofo::mode` *int* :
- `antescofo::mute` *string* :
- `antescofo::nextaction` (no argument) :
- `antescofo::nextevent` (no argument) :
- `antescofo::nextfwd` (no argument) :
- `antescofo::nextlabel` (no argument) :
- `antescofo::nofharm` *int* :
- `antescofo::normin` *float* :

- `antescofo::obsexp` *float* :
- `antescofo::pedalcoeff` *float* :
- `antescofo::pedaltime` *float* :
- `antescofo::pedal` *int* :
- `antescofo::piano` *int* :
- `antescofo::playfrombeat` *float* :
- `antescofo::playfrom` *string* :
- `antescofo::play` (no argument) :
- `antescofo::preload` *string string* : preloads a score and store it under a name (the second argument) for latter use;
- `antescofo::preventzigzag` *string* :
- `antescofo::previousevent` (no argument) :
- `antescofo::previouslabel` (no argument) :
- `antescofo::printfwd` (no argument) :
- `antescofo::printscore` (no argument) :
- `antescofo::read` *string* : loads the corresponding *Antescofo* score;
- `antescofo::report` (no argument) :
- `antescofo::setvar` *string numeric* : assign the value given by the second argument to the variable named by the first argument. Using this command, the environment may notify *Antescofo* some information. For instance, *Antescofo* may react because the variable is in the logical condition of a `whenever`).
- `antescofo::score` *string* : loads the corresponding *Antescofo* score;
- `antescofo::start` *string* :
- `antescofo::stop` (no argument) :
- `antescofo::suivi` *int* :
- `antescofo::tempoinit` *int* :
- `antescofo::temposmoothness` *float* :
- `antescofo::tune` *float* :
- `antescofo::unmute` *string* :

- `antescofo::variance` *float* :
- `antescofo::verbosity` *int* :
- `antescofo::verify` *int* :
- `antescofo::version` (no argument) : print the version on the MAX console;

As for MAX/PD or OSC message, there is no other statement, action or event defined after the internal command until the end of the line..

5.7 Assertion `@assert`

The action `@assert` checks that the result of an expression is `true`. If not, the entire program is aborted. This action is provided as a facility for debugging and testing, especially with the standalone version of *Antescofo* (in the Max or PD version, the embedding host is aborted as well).

6 Compound Actions

Compound actions act as containers for others actions. The actions “inside” a container (we say also “nested in”) inherits some of the attribute of the container.

The nesting of actions can be explicit. This is the case for a (sub-)group nested in a group (see below): the fragment of the score that defines the sub-group is part of the score fragment that defines the enclosing group. But the nesting of action can be also implicit. This is the case for the action launched by a process call: they are “implicitly nested” in the caller.

The actions of a container are spanned in a *parallel thread*: their timing does not impact the sequence of actions in which the container is embedded.

The nesting of containers creates a hierarchy which can be visualized as an inclusion tree. The *father* of an action A is its immediately enclosing container F, if it exists, and A is a *child* of F.

We present first the `group` structure which is the basic container: all other compound actions are variations on this structure.

6.1 `Group`

The group construction gathers several actions logically within a same block that share common properties of tempo, synchronization and errors handling strategies in order to create polyphonic phrases.

```
delay group name attributes { actions_list }
```

The specification of the *delay*, *name* and *attributes* are optional. The *name* is a simple identifier that acts as a label for the action.

There is a short notation for a group without delay, attribute and name: its actions can be written between braces. For example:

```
action1
{ 1 action2 }
action3
```

is equivalent to

```
action1
Group {
  1 action2
}
action3
```

The action following an event are members of an implicit group named `top_gfwd_`*xxx* where *xxx* is a number unique to the event.

6.1.1 Local Tempo.

A local tempo can be defined for a group using the attribute:

```
group G @tempo := expr ...
```

expr is an arbitrary expression that defines the passing of time for the delay of the action of `G` that are expressed in relative time, see section 3.2.

6.1.2 Attributes of `Group` and Compound Actions

Synchronization (cf. section 8.1)

```
group ... @loose ...
group ... @tight ...
```

and error strategies (cf. section 8.2)

```
group ... @global ...
group ... @local ...
```

can be specified for `group` bt also for every compound actions (`loop`, `curve`, etc.) using the corresponding attributes. If they are not explicitly defined, the attributes of an action are *inherited* from the enclosing action. Thus, using compound actions, the composer can create easily nested hierarchies (groups inside groups) sharing an homogeneous behavior.

6.1.3 Instances of a `Group`

A group `G` is related to an event or another action. When the event occurs or the action is triggered, *Antescofo* waits the expiration of its delay before launching the actions composing the group. We say that an *instance* of the group is created and launched. The instance is said *alive* while there is an action of the group waiting to be launched. In other word, an instance expires when the last action of the group is performed.

We make a distinction between the group and its instances because several instances of the same group can exists and can even be alive simultaneously. Such instances are created by `loop`, parallel iterations `forall`, reactions to logical conditions `whenever` and processes `proc`. These constructions are described below.

Note that when the name of a group is used in an `abort` action, all alive instances of this group are killed[12].

6.1.4 Aborting a `group`

There are several ways to provoque the premature end of a group, or more generally, of any compound action:

[12] It is possible to kill a specific instance using the `exec` that refers to this instance, see 9.4.

- using an `abort` action, see 6.1.4,
- using a `until` (or a `while`) *logical clause*,
- using a `during` *temporal clause*.

The `until` Clause. The specification of a `group` may include an optional until clause that is checked before the triggering of an action of the group:

```
$x := false
Group G {
  1 $x := true
  1 print DONE
} until ($x)
```

There is a dual of the `until` keyword:

```
group ... { ... } until (exp)
```

is equivalent to

```
group ... { ... } while (!exp)
```

The `during` Clause. A `during` clause specify a *temporal scope*, *i.e.* the time a group is active. When this time is exhausted, the group is aborted. This time can be specified in beats (relative time) or in (milli-)seconds (absolute time). For instance:

```
Group G {
  1 $x := true
  1 print DONE
} during [1.5]
```

will launch the assignment 1 beat after the launching of `G` but the `print` action is never executed because `G` is aborted 1.5 beats after its start.

The `[ ]` notation follows the notation used for the access to the history of a variable (cf. sect. 7.4.1 pp. 60). So

```
Group G {
  ; ...
} during [1.5 s]
```

will execute the actions specified by the group, up to 1.5 seconds after its start. And

```
Group G {
  ; ...
} during [1 #]
```

will execute the group only 1 times. This last logical duration may seems useless for a group, but is very convenient to specify the number of iterations of a loop or the maximal number of triggering of a `whenever`.

6.2 Conditional Actions: `If`

A conditional action is a construct that performs different actions depending on whether a programmer-specified boolean condition evaluates to true or false. A conditional action takes the form:

```
if (boolean condition)
{
  actions launched if the condition evaluates to true
}
```

or

```
if (boolean condition)
{
  actions launched if the condition evaluates to true
}
else
{
  actions launched if the condition evaluates to false
}
```

As the other actions, a conditional action can be prefixed by a delay. Note that the actions in the *if* and in the *else* clause are evaluated as if they are in a group. So the delay of these actions does not impact the timing of the actions which follows the conditional. For example

```
if ($x) { 5 print HELLO }
1 print DONE
```

will print `DONE` one beat after the start of the conditional independently of the value of the condition.

The actions of the “true” (resp. of the “else”) parts of a condition are members of an implicit group named *xxx*`_true_body` (resp. *xxx*`_false_body`) where *xxx* is the label of the conditional itself.

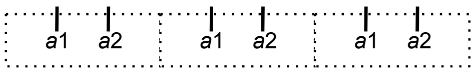
They exist also conditional expressions, cf. sect. 7.5 page 64 that share a similar syntax.

6.3 Sequential iterations: `Loop`

The `loop` construction is similar to group where actions in the loop body are iterated depending on a period specification. Each iteration takes the same amount of time, a period.

Stopping a `Loop`. The optional `until` or `while` clause is evaluated at each iteration and eventually stops the `loop` For instance, the declarations on the left produce the timing of the action’s firing figured in the right:

```
let $cpt := 0
loop L 1.5
{
  let $cpt := $cpt + 1
  0.5 a1
  0.5 a2
}
until ($cpt >= 3)
```



If an `until` condition is not provided, nor a `during` condition, the loop will continue forever but it can be killed by an `abort` command:

```
     loop ForEver 1
     {
       print OK
     }
3.5 abort ForEver
```

will print only three `OK`.

6.4 Parallel Iterations: `Forall`

The previous construction spans a group sequentially (one after the other, with a given period). The `forall` action (for *parallel iteration*) instantiates in parallel a group for each elements in an iteration set. The simplest example is the iteration on the elements of a vector (`tab`) :

```
$t := tab [1, 2, 3]
forall $x in $t
{
  (3 - $x) print OK $x
}
```

will trigger in parallel a group for each element in the vector referred by `$t`. The *iterator variable* `$x` takes for each group the value of its corresponding element in the vector. The result of this example is to `print` in sequence

```
OK 3     ; at time 0 = (3 - 3)
OK 2     ; at time 1 = (3 - 2)
OK 1     ; at time 2 = (3 - 1)
```

The general form of a parallel iteration is:

```
forall variable in expression
{
  actions...
}
```

where *expression* evaluates to a vector or a `proc`. In this case, the iteration variable takes an `exec` value corresponding to the active instances of the `proc`.

Parallel iterations accepts also maps[13] using two variables to refers to the keys and values in the map:

```
$m := map { (1, "one"), (2, "two"), (3, "three") }
forall $k, $v in $m
{
  print $k "␣=>␣" $v
}
```

will print:

```
1 => one
2 => two
3 => three
```

6.5 Sampling parameters: Curve

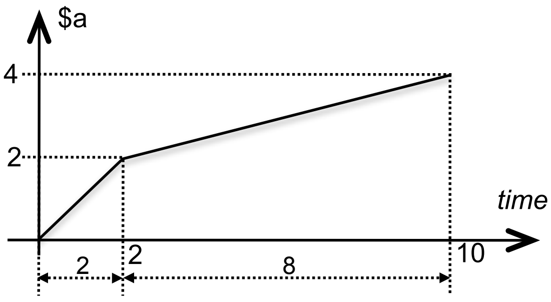
The curve construction samples a set of predefined curves to fire an action repeatedly with the sampled points. The predefined curves are defined by a sequence of points and a sequence of interpolation methods. When time passes, the curve is traversed and the corresponding action fired at the sampling point.

We introduce the syntax[14] starting with the simple case of one curve. Then we detail the complete features of this construction.

6.5.1 A Simple Curve

The curve is defined by a breakpoint function. In the example, the curve starts at 0. Two beats later, the curve reaches 2 and 8 additional beats later, the curve finishes at 4. Between the breakpoints, the interpolation is linear, as indicated by the keyword @linear.

```
curve C
  @action := { print $y } ,
  @grain := 0.1
{
  $y
  {
         { 0 } @type "linear"
       2 { 2 } @type "linear"
       8 { 4 }
  }
}
```



This curve is parameterized by the variable $y. The interpolated value of the curve is assigned to $y at each time step. The time step is defined by the @grain attribute. The specification of the values of $y when the time passes, is called a *parameter clause*.

[13]Interpolated maps are planned.

[14] Curve can be edited graphically using the *Ascograph* editor.

6.5.2 Actions Fired by a `Curve`

Each time the parameter `$y` is assigned, the action specified by the attribute `@action` is also fired. This action can be a simple message without attributes or any kind of action. In the latter case, a pair of braces must be used to delimit the action to perform. With this declaration:

```
curve C
  action := {
      group G {
          print $y
        2 action1 $y
        1 action2 $y
      }
  }
{ ... }
```

at each sampling point the value of `$y` is immediately printed and two beats later *action*1 will be fired and one additional beat later *action*2 will be fired.

If the attribute `@action` is missing, the curve construct simply assign the variables specified in its body. This can be useful in conjunction with an `whenever` statement or because the variables appears elsewhere in some expression.

6.5.3 Step, Durations and Parameter Specifications

Here, the step and the duration between breakpoints is expressed in relative time. But they can be also expressed in absolute time and arbitrarily mixed (*e.g.* the time step in second and duration in beats, and there also is possible to mix duration in beats and in seconds).

Step, duration, as well as the parameters, can be arbitrary expressions. These expressions are evaluated when the `curve` is fired.

The sampling rate can be as small as needed to achieve perceptual continuity. However, in the MAX environment, one cannot go below 1ms.

6.5.4 Interpolation Methods

The specification of the interpolation between two breakpoints is given by an optional string. By default, a linear interpolation is used. *Antescofo* offers a rich set of interpolation methods, mimicking the standard *tweeners* used in flash animation[15]. There are 10 different types:

- linear, quad, cubic, quart, quint: which correspond to polynomial of degree respectively one to five;
- expo: exponential, *i.e.* $\alpha e^{\beta t+\delta} + \gamma$

[15]See http://wiki.xbmc.org/?title=Tweeners

- sine: sinusoidal interpolation $\alpha \sin(\beta t + \delta) + \gamma$
- back: overshooting cubic easing $(\alpha + 1)t^3 - \alpha t^2$
- circ: circular interpolation $\alpha\sqrt{(\beta t + \delta)} + \gamma$
- bounce: exponentially decaying parabolic bounce
- elastic: exponentially decaying sine wave

Most of the interpolations types comes in three “flavors” called traditionally *ease*: *in* (the default) which means that the derivative of the curve is increasing with the time (usually from zero to some value), *out* when the derivative of the curve is decreasing (usually to zero), and *in_out* when the derivative first increase (until halfway of the two breakpoints) and then decrease. See figure 9. The corresponding interpolation keyword are listed below. Note that the interpolation can be different for each successive pair of breakpoints.

`"linear"`

`"back"` or `"back_in"`	`"exp"` or `"exp_in"`	`"elastic"` or `"elastic_in"`
`"back_out"`	`"exp_out"`	`"elastic_out"`
`"back_in_out"`	`"exp_in_out"`	`"elastic_in_out"`
`"bounce"` or `"bounce_in"`	`"quad"` or `"quad_in"`	`"sine"` or `"sine_in"`
`"bounce_out"`	`"quad_out"`	`"sine_out"`
`"bounce_in_out"`	`"quad_in_out"`	`"sine_in_out"`
`"cubic"` or `"cubic_in"`	`"quart"` or `"quart_in"`	
`"cubic_out"`	`"quart_out"`	
`"cubic_in_out"`	`"quart_in_out"`	
`"circ"` or `"circ_in"`	`"quint"` or `"quint_in"`	
`"circ_out"`	`"quint_out"`	
`"circ_in_out"`	`"quint_in_out"`	

Programming its Own Interpolation Method. If one need an interpolation method not yet implemented, it is easy to program it. The idea is to apply a user defined function to the value returned by a simple linear interpolation, as follows:

```
@FUN_DEF @f($x) { ... }
...
curve C action := print @f($x), grain := 0.1
{
  $x
  {       { 0 } @linear
       1s { 1 }
  }
}
```

The curve `C` will interpolate function `@f` between `0` and `1` after its starts, during one second and with a sampling rate of `0.1` beat.

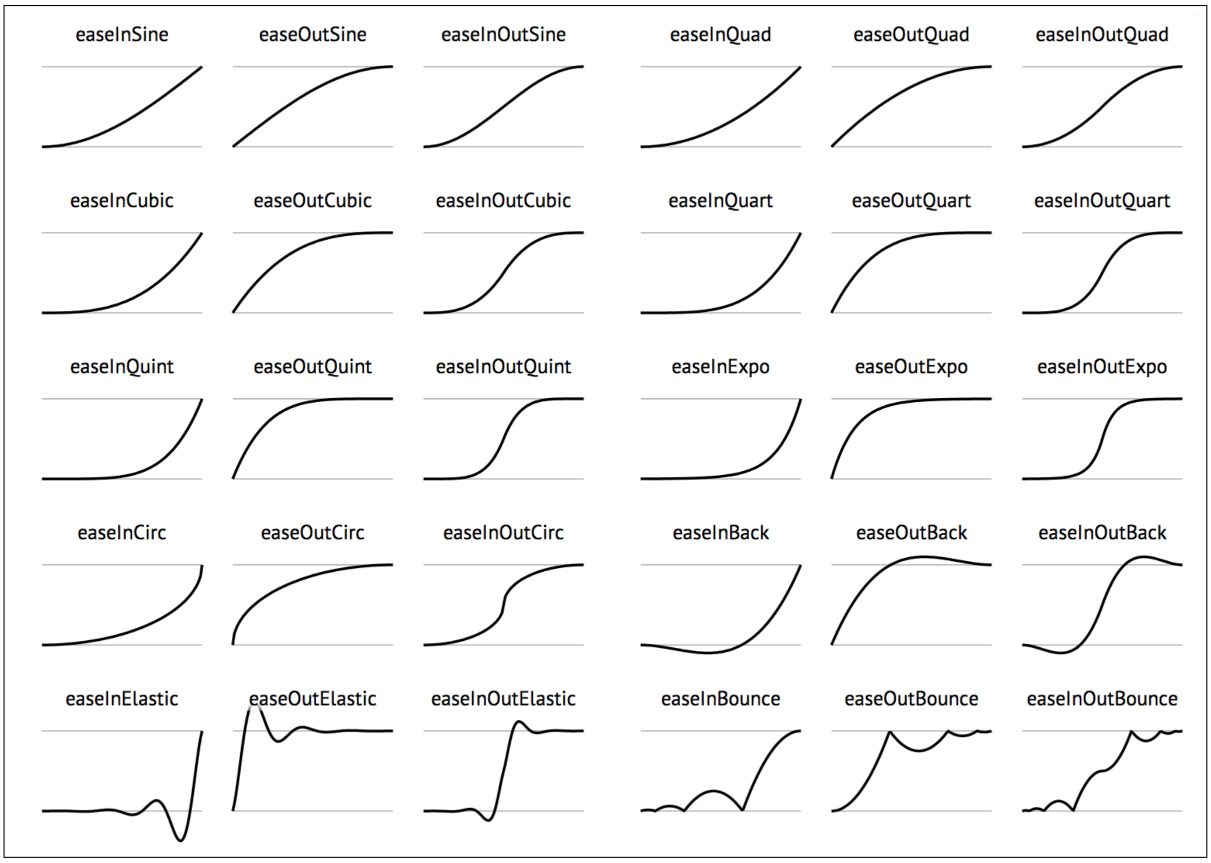
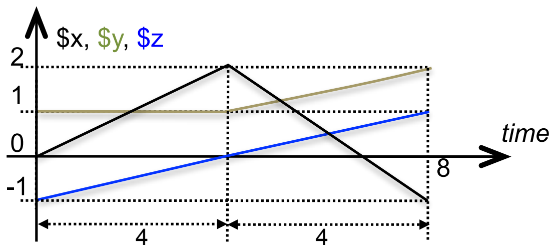


Figure 9: *Various interpolation type available in an* Antescofo *curve.*

6.5.5 Managing Multiple Curves Simultaneously

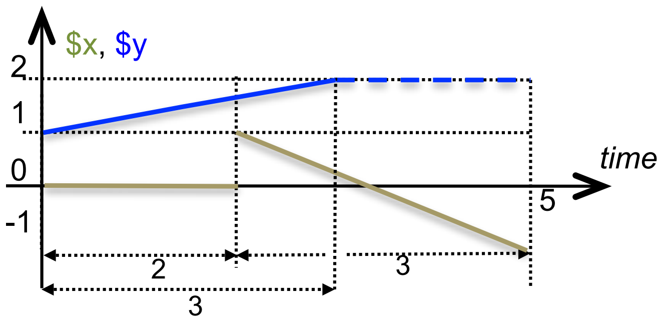
To make easier the simultaneous sampling of several curves, it is possible to define multiples parameters together in the same clause:

```
curve C
{
  $x, $y, $z
  {
       { 0, 1, -1 } @linear
     4 { 2, 1,  0 } @linear
     4 { -1, 2,  1 }
  }
}
```



Note that with the previous syntax it is not possible to define simultaneous curves with breakpoints at different time. This is possible using multiple parameter clauses, as in:

```
curve C
{
  $x
  {
       { 0  } @constant
     2 { 1  } @linear
     3 { -1 }
  }
  $y
  {
       { 1 } @linear
      3 { 2 }
  }
}
```



In this example, the parameters `$x` and `$y` do not share their breakpoints. The first two breakpoints for `$x` defines a constant function. And the second and the last breakpoints define a linear function. Incidentally note that the result is not a continuous function on $[0, 5]$. The parameter `$y` is defined by only one pair of breakpoints. The last breakpoint has its time coordinate equal to 3, which ends the function before the end of `$x`. In this case, the last value of the function is used to extend the parameters "by continuity".

6.5.6 Curve with a `NIM`

A `NIM` value can be used as an argument of the `Curve` construct which allows to build dynamically the breakpoints as the result of a computation.

```
Curve ... { $x : e }
```

defines a curve where the breakpoints are taken from the value of the expression e. This expression is evaluated when the curve is triggered and must return a `NIM` value. This value is used as a specification of the breakpoints of the curve.

6.6 Reacting to logical events: `Whenever`

The `whenever` statement allows the launching of actions conditionally on the occurrence of a logical condition:

```
whenever optional_label (boolean_expression1)
{
  actions_list
} until (boolean_expression2)
```

The label and the `until` clause are optional. A `during` clause can be used in place of the `until`.

The behavior of this construction is the following: The `whenever` is active from its firing until its end, as specified by the `until` or the `during` clauses (see next paragraph 6.6.1). After the firing of the whenever, and until its end, each time the variables of the *boolean_expression* 1 are updated, *boolean_expression* 1 is re-evaluated. We stress the fact that only the variables that appear explicitly in the boolean condition are tracked. If the condition evaluates to true, the body of the whenever is launched.

Note that the boolean condition is not evaluated when the whenever is fired: only when one of the variables that appears in the Boolean expression is updated by an assignment elsewhere.

Notice also the difference with a conditional action (section 6.2): a conditional action is evaluated when the flow of control reaches the condition while the whenever is evaluated as many time as needed, from its firing, to track the changes of the variables appearing in the condition.

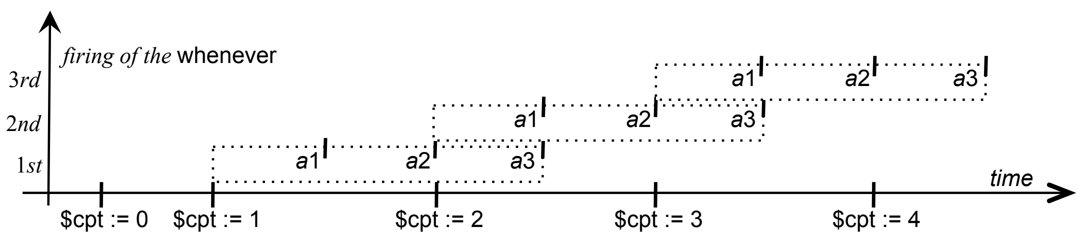
The whenever is a way to reduce and simplify the specification of the score particularly when actions have to be executed each time some condition is satisfied. It also escapes the sequential nature of traditional scores. Resulting actions of a whenever statement are not statically associated to an event of the performer but dynamically satisfying some predicate, triggered as a result of a complex calculation, launched by external events, or any combinations of the above.

Because the action in the body of a whenever are not bound to an event or another action, synchronization and error handling attributes are irrelevant for this compound action.

Nota Bene that multiple occurrence of the body of the same whenever may be active simultaneously, as shown by the following example:

```
      let $cpt := 0
  0.5 loop 1 {
        let $cpt := $cpt + 1
      }
      whenever ($cpt > 0) {
        0.5 a1
        0.5 a2
        0.5 a3
      } until ($cpt <= 3)
```

This example will produce the following schedule:



6.6.1 Stopping a whenever

A during and/or an until clause can be defined for a whenever (see sect. 6.1.4). These clauses are evaluated each time the logical condition of the whenever must be evaluated, irrespectively of its false or true value. For example,

```
$X := false
whenever ($X)
{ print "OK" $ } during [2 #]
1.0 $X := false
1.0 $X := true
1.0 $X := true
```

will print only one "OK" because at (relative) time 1.0 the body of the logical condition is false, at time 2.0 the logical condition is true, the body is launched and the whenever is stopped because it has been "activated" two times, *i.e.* [2 #].

Using a duration in relative time [2.0] or in absolute time [2000 ms] gives the whenever a *temporal scope* during which it is active. When the duration is elapsed, the whenever cannot longer fire its body.

The previous example with logical time [2 #] shows how to stop the whenever after two changes of $X (whatever is the change). It is easy to stop it after a given number of body's fire, using a counter in the condition:

```
$X := false
$cpt := 0
whenever (($cpt < 1)   $X)
{  $cpt := $cpt + 1
   print "OK" $
}
1.0 $X := false
1.0 $X := true
1.0 $X := true
```

will print only one "OK" at relative time 1.0. Then the counter $cpt is set to 1 and the condition will always be false in the future.

Another option is to give the whenever a label and to abort it, see sect. 6.1.4.

6.6.2 Causal Score and Temporal Shortcuts

The actions triggered when the body of a whenever W ... is fired, may fire others whenever, including directly or indirectly W itself. Here is an example:

```
let $x := 1
let $y := 1
whenever ($x > 0) @name W1
{
  let $y := $y + 1
}
```

```
whenever ($y > 0) @name W2
{
  let $x := $x + 1
}
let $x := 10 @name Start
```

When action `Start` is fired, the body of `W1` is fired in turn in the same logical instant, which leads to the firing of the body of `W2` which triggers `W1` again, etc. So we have an infinite loop of computations that are supposed to take place *in the same logical instant*:

```
Start → W1 → W2 → W1 → W2 → W1 → W2 → W1 → W2 → W1 →   ...
```

This infinite loop is called a *temporal shortcuts* and correspond to a *non causal score*. The previous score is non-causal because the variable `$x` depends *instantaneously* on the updates of variable `$y` and variable `$y` depends instantaneously of the update of the variable `$x`.

The situation would have been much different if the assignments had been made after a certain delay. For example:

```
let $x := 1
let $y := 1
whenever ($x > 0) @name W1
{
  1 let $y := $y + 1
}
whenever ($y > 0) @name W2
{
  1 let $x := $x + 1
}
let $x := 10 @name Start
```

also generate an infinite stream of computations but with a viable schedule in time. If `Start` is fired at 0, then `W1` is fired at the same date but the assignment of `$y` will occurs only at date 2. At this date, the body of `W2` is subsequently fired, which leads to the assignement of `$x` at date 3, etc.

```
  0: Start → W1
→ 1: $y := 1+1 → W2
→ 2: $x := 1+1 → W1
→ 3: $y := 2+1 → W2
→ 4: $x := 2+1 → W1
→ 5: ...
```

Automatic Temporal Shortcut Detection. *Antescofo* detects automatically the temporal shortcuts and stops the infinite regression. No warning is issued although temporal shortcuts are considered as bad programming.

As a matter of fact, a temporal shortcut indicates that some variable is updated multiple time synchronously (in the same logical instant). If these updates are specified in the same group, they are well ordered. But if they are issued from “parallel” groups, their order is undetermined, which lead to non-deterministic results.

7 Expressions

Expressions can be used to compute delay, `loop` period, `group` local tempo, breakpoints in `curve` specification, and arguments of internal commands end external messages sent to the environment.

7.1 Values

Expression are evaluated into values. They are two kind of values:

- *scalar* or *atomic values* include the undefined value and the booleans, the integers, the floats (IEEE double), the strings, the symbols (a representation of the simple identifiers), the function definitions, the process definitions and the running processes (*exec*);
- *aggregate values* or *compound values* are data structures like tabs (vectors), maps (dictionaries), and interpolated functions. Such data structures can be arbitrarily nested, to obtain for example a dictionary of vector of interpolated functions.

Predefined functions can be used to combine values to build new values. The programmer can defines its own functions, see paragraph 1.2.

Dynamic Typing. From a programming language perspective, *Antescofo* is a dynamically typed programming language: the type of values are checked during the performance and this can lead to an error at run-time.

When a bad argument is provided to an operator or a predefined function, an error message is issued on the console and most of the time, the returned value is a string that contains a short description of the error. In this way, the error is propagated and can be traced back. See section 11.3 for useful hints on how to debug an *Antescofo* score.

Compound values are not necessarily homogeneous : for example, the first element of a vector (tab) can be an integer, the second a string and the third a boolean.

Note that each kind of value can be interpreted as a boolean or as a string. The string representation of a value is the string corresponding of an *Antescofo* fragment that can be used to denote this value.

Checking the Type of a Value. Several predicates check if a value is of some type: `@is_undef`, `@is_bool`, `@is_string`, `@is_symbol`, `@is_int`, `@is_float`, `@is_numeric` (which returns true if the argument is either `@is_int` or `@is_float`), `@is_map`, `@is_interpolatedmap`, `@is_nim`, `@is_tab`, `@is_fct` (which returns true if the argument is an intentional function), `@is_function` (which returns true if the argument is either an intentional function or an extensional one), `@is_proc`, and `@is_exec`.

Value Comparison. Two values can always be compared using the relational operators

```
<  <=  =  !=  =>  >
```

or the `@min` and `@max` operators. The comparison of two values of the same type is as expected: arithmetic comparison for integers and floats, lexicographic comparison for strings, etc. When an integer is compared against a float, the integer is first converted into the corresponding float. Otherwise, comparing two values of two different types is well defined but implementation dependent.

7.2 Scalar Values

7.2.1 Undefined Value

There is only one value of type `Undefined`. This value is the value of a variable before any assignment. It is interpreted as the value `false` if needed.

The undefined value is used in several other circumstances, for example as a return value for some exceptional cases in some predefined functions.

7.2.2 Boolean Value

They are two boolean values denoted by the two symbols `true` and `false`. Boolean values can be combined with the usual operators:

- the negation `!` written prefix form: `!false` returns `true`
- the logical disjunction `||` written in infix form: `$a || $b`
- the logical conjunction `&&` written in infix form: `$a && $b`

Logical conjunction and disjunction are "lazy": `a && b` does not evaluate `b` if `a` is `false` and `a || b` does not evaluate `b` if `a` is `true`.

7.2.3 Integer Value

Integer values are written as usual. The arithmetic operators `+`, `-`, `*`, `/` and `%` (modulo) are the usual ones with the usual priority. Integers and float values can be mixed in arithmetic operations and the usual conversions apply. Similarly for the relational operators. In boolean expression, a zero is the false value and all other integers are considered to be `true`.

7.2.4 Float Value

Float values are handled as IEEE double (as in the C language). The arithmetic operators, their priority and the usual conversions apply.

Float values can be implicitly converted into a boolean, using the same rule as for the integers.

For the moment, there is only a limited set of predefined functions:

@abs	@acos	@asin	@atan	@cos	@cosh	@exp
@floor	@log10	@log2	@log	@max	@min	@pow
@ceil	@sinh	@sin	@sqrt	@tan		

These functions correspond to the usual IEEE mathematical functions.

There are additional functions like `@rand`, used to generate a random number between 0 and d: `@rand`(d). See the functions of the `@rnd_`*xxx* family in the annex A.

7.2.5 String Value

String constant are written between quote. To include a quote in a string, the quote must be escaped:

```
print "this␣is␣a␣string␣with␣a␣\"␣inside"
```

Others characters must be escaped in string: `\n` is for end of line (or carriage-return), `\t` for tabulation, and `\\` for backslash.

The + operator corresponds to string concatenation:

```
$a := "abc" + "def"
print $a
```

will output on the console `abcdef`. By extension, adding a value `a` to a string concatenate the string representation of `a` to the string:

```
$a = 33
print ("abc" + $a)
```

will output `abc33`.

7.2.6 Intentional Functions

Intentional functions f are defined by rules (*i.e.* by an expression) that specify how an image $f(x)$ is associated to an element x. Intentional functions can be defined and associated to an @-identifier using the `@fun_def` construct introduced in section 1.2 page 7. Some intentional functions are predefined and available in the initial *Antescofo* environment like the IEEE mathematical functions. See annex A for a description of the available functions.

There is no difference between predefined intentional functions and user's defined intentional functions except that in a Boolean expression, a user's defined intentional function is evaluated to `true` and a predefined intentional function is evaluated to `false`.

In an *Antescofo* expression, the @-identifier of a function denotes a functional value that can be used for instance as an argument of a higher-order functions (see examples of higher-order predefined function in section 7.3.1 for map building and map transformations).

7.2.7 Proc Value

The ::-name of a processus can be used in an expression to denote the corresponding process definition, in a manner similar of the @-identifier used for intensionnal functions (see 7.2.6). Such value are qualified as *proc value*. Like intensionnal functions, proc value are first class value. They can be passed as argument to a function or a procedure call.

The main operation on proc value is "calling the corresponding process", see section 9.

7.2.8 Exec Value

An *exec value* refers to a running process. Such value are created when a process is instantiated, see section 9. This value can be used to kill the corresponding process instanciation. It is also used to access the values of the local variables of the process.

Warning: These features are still experimental.

7.3 Aggregate Values

7.3.1 Map Value

A map is a dictionary associating a value to a key. The value can be of any kind, as well as the key:

```
map{ (k1,v1), (k2,v2), ... }
```

A map is an ordinary value and can be assigned to a variable to be used latter. The usual notation for function application is used to access the value associated to a key:

```
$dico = map{ (1, "first"), (2, "second"), (3, "third") }
...
print ($dico(1)) ($dico(3.14))
```

will print

```
first  "<Map:␣Undefined>"
```

The string `"<Map:␣Undefined>"` is returned for the second call because there is no corresponding key.

Extensional Functions. A `map` can be seen as a function defined by extension: an image (the value) is explicitly defined for each element in the *domain* (*i.e.*, the set of keys). *Interpolated maps* and *NIM* are also *extensional functions*.

Extensional function are handled as values in *Antescofo* but this is also the case for *intentional functions*, see the previous section 7.2.6.

In an expression, extensional function or intentional function can be used indifferently where a function is expected. In other words, you can apply an extensional function to get a value, in teh same way you apply a predefined or a user-defined intensionnal function:

```
@fun_def @factorial($x) { ($x <= 0 ? 1 : $x * @factorial($x - 1)) }
$f := MAP{ (1,2), (2,3), (3,5), (4,7), (5,11), (6,13), (7,17) }
$v := $f(5) + @factorial(5)
```

Domain, Range and Predicates. One can test if a map `m` is defined for a given key `k` using the predicate `@is_defined(m, k)`.

The predefined `@is_integer_indexed` applied on a map returns true if all key are integers. The predicate `@is_list` returns true if the keys form the set $\{1, \dots, n\}$ for some n. The predicate `@is_vector` returns true if the predicate `@is_list` is satisfied and if every element in the range satisfies `@is_numeric`.

The functions `@min_key`, resp. `@max_key`, computes the minimal key, resp. the maximal key, amongst the key of its map argument.

Similarly for the functions `@min_val` and `@max_val` for the values of its map argument.

In boolean expression, an empty map acts as the value `false`. Other maps are converted into the `true` value.

Constructing Maps. These operations act on a whole `map` to build new maps:

- `@select_map` restricts the domain of a map: `select_map(`m`, `P`)` returns a new map m' such that $m'(x) = m(x)$ if $P(x)$ is true, and undefined elsewhere. The predicate P is an arbitrary function (*e.g.*, it can be a user-defined function or a dictionary).
- The operator `@add_pair` can be used to insert a new `(`*key*`, `*val*`)` pair into an existing map:

  ```
  @add_pair($dico, 33, "doctor")
  ```

  enriches the dictionary referred by `$dico` with a new entry (no new map is created).
- `@shift_map(`m`, `n`)` returns a new map m' such that $m'(x + n) = m(x)$
- `@gshift_map(`m`, `f`)` generalizes the previous operator using an arbitrary function f instead of an addition and returns a map m' such that $m'(f(x)) = m(x)$
- `@map_val(`m`, `f`)` compose f with the map m: the results m' is a new map such that $m'(x) = f\big(m(x)\big)$.
- `@merge` combines two maps into a new one. The operator is asymmetric, that is, if $m = \text{merge(a, b)}$, then:

$$m(x) = \begin{cases} \mathsf{a}(x) & \text{if } \texttt{@is\_defined(a,}x\texttt{)} \\ \mathsf{b}(x) & \text{elsewhere} \end{cases}$$

Extension of Arithmetic Operators. Arithmetic operators can be used on maps: the operator is applied "pointwise" on the intersection of the keys of the two arguments. For instance:

```
$d1 := MAP{ (1, 10), (2, 20), (3, 30) }
$d2 := MAP{ (2, 2), (3, 3), (4, 4) }
$d3 := $d1 + $d2
print $d3
```

will print

```
MAP{ (2, 22), (3, 33) }
```

If an arithmetic operators is applied on a map and a scalar, then the scalar is implicitly converted into the relevant map:

```
$d3 + 3
```

computes the map `MAP{ (2, 25), (3, 36) }`.

Maps Transformations.

- `@clear`: erase all entries in the map.
- `@compose_map`: given

  ```
  $p := map{ ($k_1$, $p_1$),  ($k_2$, $p_2$), ...,  ($k_n$, $p_n$),  }
  $q := map{ ($k'_1$, $q_1$),  ($k'_2$, $q_2$), ...,  ($k'_m$, $q_m$),  }
  ```

  `@compose_map($p, $q)` construct the map:

  ```
  map{  ..., ($p_k$, $q_k$), ... }
  ```

  if it exists a k such that

  ```
  $p($k$) = $p_k$ and   $q($k$) = $q_k$
  ```

- `@listify` applied on a map `m` builds a new map where the key of `m` have been replaced by their rank in the ordered set of keys. For instance, given

  ```
  $m := map{ (3, 3), ("abc", "abc"), (4, 4)}
  ```

  `@listify($m)` returns

  ```
  map{ (1, 3), (2, 4), (3, "abc") }
  ```

  because we have `3 < 4 < "abc"`.

- `@map_reverse` applied on a list reverse the list. For instance, from:

  ```
  map{ (1, $v_1$),  (2, $v_2$), ...,  ($p$, $v_p$),  }
  ```

  the following list is build:

  ```
  map{ (1, $v_p$),  (2, $v_{p-1}$), ...,  ($p$, $v_1$),  }
  ```

Score reflected in a Map. Two functions can be used to reflect the events of a score into a map[16]:

- `@make_score_map(start, stop)` returns a map where the key is the event number (its rank in the score) and the associated value, its position in the score in beats (that is, its date in relative time). The map contains the key corresponding to events that are in the interval [`start`,`stop`] (interval in relative time).
- `@make_duration_map(start, stop)` returns a map where the key is the event number (its rank in the score) and the associated value, its duration in beats (relative time). The map contains the key corresponding to events that are in the interval [`start`,`stop`] (interval in relative time).

The corresponding maps are *vectors*.

History reflected in a map. The sequence of the values of a variable is keep in an history. This history can be converted into a map, see section 7.4.1 pp. 60.

7.3.2 InterpolatedMap Value

Interpolated map are values representing a piecewise interpolated function. They have been initally defined as piecewise linear functions. They are now superseded by `NIM` (an acronym for *new interpolated map*) which extend the idea to all the interpolation kinds available in the `curve` construct.

`NIM` interpolated Map. A `NIM` is an aggregate data structure that defines an interpolated function: the data represent the breakpoints of the piecewise interpolation (as in a curve) and a `NIM` can be applied to a numerical value to returns the corresponding image.

`NIM` can be used as an argument of the `Curve` construct which allows to build dynamically the breakpoints as the result of a computation. See section 6.5.6.

There are two ways of defining a `NIM`. *Continuous* `NIM` are defined by an expression of the form:

```
NIM { x0 y0, d1 y1 "cubic"
           , d2 y2              // no type = "linear"
           , d3 y3 "bounce"
           , ...
           , dn yn "typen"
    }
```

which specifies a piecewise function f: between x_i and $x_{i+1} = x_i + d_{i+1}$, function f is an interpolation of type type_{i+1} from y_i to y_{i+1}.

[16]Besides `@make_score_map` and `@make_duration_map`, recall that the label of an event in $-form, can be used in expressions as the position of this event in the score in relative time.

The function f is extended outside $[\mathrm{x}_0, \mathrm{x}_n]$ such that

$$f(x) = \begin{cases} \mathrm{y}_0 & \text{for } x \leq \mathrm{x}_0 \\ \mathrm{y}_n & \text{for } x \geq \mathrm{x}_n = \mathrm{x}_0 + \sum_{i=0}^{n} \mathrm{d}_i \end{cases}$$

The type of the interpolation is either a constant string or an expression. But in this case, the expression must be enclosed in the `EXPR { . }` construct

Note that the previous definition specifies a continuous function because the value of f at the beginning of $[\mathrm{x}_i, \mathrm{x}_{i+1}]$ is also the value of f at the end of the previous interval.

The second syntax to define a `NIM` allows to define a discontinuous function by specifying a different y value for the end of an interval and the beginning of the next one:

```
NIM { x₀, y₀ d₀ Y₀ "cubic"
        , y₁ d₁ Y₁
        , y₂ d₂ Y₂ "bounce"
        , ...
    }
```

The definition is similar to the previous form except that on the interval $[\mathrm{x}_i, \mathrm{x}_{i+1}]$ the function is an interpolation between y_i and Y_i.

Interpolated Map. This type of value is now superseded by `NIM` and there usage is deprecated.

Interpolated map functions are piecewise linear functions defined by a set of points (x_i, y_i):

```
$f := imap{ (0, 0), (1.2, 2.4), (3.0, 0) }
print $f(2.214)
```

defines a kind of "triangle" function. The value of `$f` at a point x between x_j and x_{j+1} is the linear interpolation of y_j and y_{j+1}.

The linear function on $[x_0, x_1]$ is naturally extended on $]-\infty, x_0]$ by prolonging the line that passes through (x_0, y_0) and (x_1, y_1) and similarly for $[x_{\max}, +\infty[$. This natural extension defines the value of the interplated map on a point x outside $[x_0, x_{\max}]$.

There exist a few predefined function to manipulate interpolated maps: `@integrate` to integrate the piecewise linear function on $[x_0, x_{\max}]$ and `@bounded_integrate` to integrate the function on an arbitrary interval.

7.3.3 Tab Value

Tab values are simple vectors. They can be defined by giving the list of its element, using a syntax similar to maps:

```
$t := tab [0, 1, 2, 3]
```

this statement assign a tab with 4 elements to the variable `$t`. The `tab` keyword is optional. Elements of a tab can be accessed through the usual square bracket $\cdot[\cdot]$ notation: `$t[`n`]` refers to the $(n+1)$th element of tab `$t` (elements indexing starts at 0).

Multidimensional tab. Elements of a tab are arbitrary, so they can be other tabs. Nesting of tabs can be used to represent matrices and multidimensional arrays. For instance:

```
[ [1, 2], [3, 4], [4, 5] ]
```

is a 3×2 matrix. The function `@dim` can be used to query the dimension of a tab, that is, the maximal number of tab nesting found in the tab. If `$t` is a multidimensional array, the function `@shape` returns a tab of integers where the element i represents the number of elements in the ith dimension. For example

```
@shape( [ [1, 2], [3, 4], [4, 5] ] )    ⟶ [3, 2]
```

The function `@shape` returns `0` if the argument is not an well-formed array. For example

```
@shape( [1, 2, [3, 4]] )    ⟶ 0
```

Note that for this argument, `@dim` returns `2` because there is a tab nested into a tab, but it is not an array because the element of the top-level tab are not homogeneous. The tab

```
[ [1, 2], [3, 4, 5] ]
```

fails also to be an array, despite that all elements are homogeneous, because these elements have not the same size.

A `Forall` construct can be used to refer to all the elements of a tab in an action see sect. 6.4. A tab comprehension can be used to build new tab by filtering and mapping tab elements. They are also several predefined functions to transform a tab.

`Tab` **Comprehension.** The definition of a tab by giving the list of its elements: the definition is said *in extension*. A tab can also be defined *in comprehension*. A tab comprehension is construct for creating a tab based on existing tab or on some iterators. It follows the form of the mathematical set-builder notation (set comprehension). The general form is the following:

```
[ e | $x in e′ ]
```

which generates a tab of the values of the output expression e for `$x` running through the elements specified by the input set e'. If e' is a tab, then `$x` takes all the values in the tab. For example:

```
[ 2*$x | $x in [1, 2, 3] ]   ⟶   [2, 4, 6]
```

The input set e' may also evaluates to a numeric value n: in this case, `$x` take all the numeric values between 0 and n by unitary step:

```
[ $x | $x in (2+3) ]    ⟶   [0, 1, 2, 3, 4]
[ $x | $x in (2 - 4) ] ⟶   [0, -1]
```

Note that the variable `$x` is a local variable visible only in the tab comprehension: its name is not meaningful and can be any variable identifier (but beware that it can mask a regular variable in the output expression, in the input set or in the predicate).

The input set can be specified by a range giving the starting value, the step and the maximal value:

```
[ e | $x in start .. stop : step ]
```

If the specification of the step is not given, it value is $+1$ or -1 following the sign of `(stop - start)`. The specification of `start` is also optional: in this case, the variable will start from 0. For example:

```
[ @sin($t) | $t in -3.14 .. 3.14 : 0.1 ]
```

generates a tab of 62 elements.

In addition, a predicate can be given to filter the members of the input set:

```
[$u | $u in 10, $x % 3 == 0]   ⟶   [0, 3, 6, 9]
```

filters the multiple of 3 in the interval $[0, 10)$. The expression used as a predicate is given after a comma, at the end of the comprehension.

Tab comprehensions are ordinary expressions. So they can be nested and this can be used to manipulate tab of tabs. Suc data structure can be used to make matrices:

```
[ [$x + $y | $x in 1 .. 3] | $y in [10, 20, 30] ]
 ⟶  [ [11, 12], [21, 22], [31, 32] ]
```

Here are some additional examples of tab comprehensions showing the syntax:

```
[ 0 | (100) ]         ; builds a vector of 100 elements, all zeros
[ @random() | (10) ] ; build a vector of ten random numbers
[ $i | $i in 40, $i % 2 == 0 ] ; lists the even numbers from 0 to 40
[ $i | $i in 40 : 2]           ; same as previous
[ 2*$i | $i in (20) ]          ; same as previous

; equivalent to ($s + $t) assuming arguments of the same size
[ $s[$i] + $t[$i] | $i in @size($t) ]

; transpose of a matrix $m
[ [$m[$j, $i] | $j in @size($m)] | $i in @size($m[0])]

; scalar product of two vectors $s and $t
@reduce(@+, $s * $t)

; matrice*vector product
[ @reduce(@+, $m[$i] * $v) | $i in @size($m) ]

; squaring a matrix $m, i.e. $m * $m
[ [ @reduce(@+, $m[$i] * $m[$j]) | $i in @size($m[$j]) ]
 | $j in @size($m) ]
```

`Tab` operators. Usual arithmetic and relational operators are *listable* (cf. also listable functions in annex A).

When an operator *op* is marked as *listable*, the operator is extended to accepts `tab` arguments in addition to scalar arguments. Usually, the result of the application of *op* on tabs is the tab resulting on the point-wise application of *op* to the scalar elements of the tab. But for predicate, the result is the predicate that returns true if the scalar version returns true on all the elements of the tabs. If the expression mixes scalar and tab, the scalar are extended pointwise to produce the result. So, for instance:

```
[1, 2, 3] + 10    ⟶  [11, 12, 13]
2 * [1, 2, 3]     ⟶  [2, 4, 6]
[1, 2, 3] + [10, 100, 1000]   ⟶  [11, 102, 1003]
[1, 2, 3] < [4, 5, 6]  ⟶  true
0 < [1, 2, 3]  ⟶ true
[1, 2, 3] < [0, 3, 4]  ⟶  false
```

`Tab` **manipulation.** Several functions exists to manipulate tabs *intentionally*, *i.e.*, without referring explicitly to the elements of the tab.

`@car(t)` returns the first element of `t` if `t` is not empty, else it returns an empty tab.

`@cdr(t)` returns a new tab corresponding to `t` deprived from its first element. If `t` is empty, `@cdr` returns an empty tab.

`@clear(t)` shrinks the argument to a zero-sized tab (no more element in `t`).

`@concat(t1, t2)` returns the concatenation of `t1` and `t2`.

`@cons(v, t)` returns a new tab made of `v` in front of `t`.

`@count(t, v)` returns the number of occurrences of `v` in the elements of `t`.

`@dim(t)` returns the dimension of `t`, i.e. the maximal number of nested tabs. If `t` is not a tab, the dimension is 0.

`@find_index(t, f)` returns the index of the first element of `t` that satisfies the predicate `f`.

`@flatten(t)` build a new tab where the nesting structure of `t` has been flattened. For example, `@flatten([[1, 2], [3], [[], [4, 5]]])` returns `[1, 2, 3, 4, 5, 6]`.

`@flatten(t, l)` returns a new tab where `l` levels of nesting has been flattened. If `l == 0`, the function is the identity. If `l` is strictly negative, it is equivalent to `@flatten` without the level argument.

`@insert(t, i, v)` inserts "in place" the value `v` into the tab `t` after the index `i`. If `i` is negative, the insertion take place in front of the tab. If `i` $\leq$ `@size(t)` the insertion takes place at the end of the tab. Notice that the form `@insert "file"` is also used to include a file at parsing time.

`@lace(t, n)` returns a new tab whose elements are interlaced sequences of the elements of the `t` subcollections, up to size `n`. The argument is unchanged. For example:

```
@lace([[1, 2, 3], 6, ["foo", "bar"]], 12)
⟶ [ 1, 6, "foo", 2, 6, "bar", 3, 6, "foo", 1, 6, "bar" ]
```

`@map(t, f)` computes a new tab where element `i` has the value `f(t[i])`.

`@max_val(t)` returns the maximal elements among the elements of `t`.

`@member(t, v)` returns true if `v` is an element of `t`.

`@min_val(t)` returns the maximal elements among the elements of `t`.

`@normalize(t, min, max)` returns a new tab with the elements normalized between `min` and `max`. If `min` and `max` are omitted, they are assumed to be 0 and 1.

`@occurs(t, v)` returns the number of elements of `t` equal to `v`.

`@permute(t, n)` returns a new tab which contains the `n`th permutations of the elements of `t`. They are factorial `s` permutations, where `s` is the size of `t`. The first permutation is numbered 0 and corresponds to the permutation which rearranges the elements of `t` in an array t_0 such that they are sorted increasingly. The tab t_0 is the smallest element amongst all tab that can be done by rearranging the element of `t`. The first permutation rearranges the elements of `t` in a tab t_1 such that $t_0 < t_1$ for the lexicographic order and such that any other permutation gives an array t_k lexicographicaly greater than t_0 and t_1. Etc. The last permutation (factorial `s` - 1) returns a tab where all elements of `t` are in decreasing order.

`@push_back(t, v)` pushes `v` at the end of `t` and returns the updated tab (`t` is modified in place).

`@push_front(t, v)` pushes `v` at the beginning of `t` and returns the updated tab (`t` is modified in place and the operation requires the reorganization of all elements).

`@rank(t)` if `t` is an array

`@reduce(f, t)` computes `f(... f(f(t[0], t[1]), t[2]), ... t[n])`. If `t` is empty, an undefined value is returned. If `t` has only one element, this element is returned. In the other case, the binary operation `f` is used to combine all the elements in `t` into a single value. For example, `@reduce(@+, t)` returns the sum of the elements of `t`.

`@remove(t, n)` removes the element at index `n` in `t` (`t` is modified in place).

`@remove_duplicate(t)` keep only one occurrence of each element in `t`. Elements not removed are kept in order and `t` is modified in place.

`@replace(t, find, rep)` returns a new tab in which a number of elements have been replaced by another. See full description page 92.

`@reshape(t, s)` builds an array of shape `s` with the element of tab `t`. These elements are taken circularly one after the other. For instance

```
@reshape([1, 2, 3, 4, 5, 6], [3, 2])
⟶ [ [1, 2], [3, 4], [5, 6] ]
```

`@resize(t, v)` increases or decreases the size of `t` (`v` is used if the size is increased to give a value to the new elements).

`@reverse(t)` computes the tab with the elements of `t` in the reverse order.

`@rotate(t, n)` build a new array which contains the elements of `t` circularly shifted by `n`. If `n` is positive the element are right shifted, else they are left shifted.

`@scan(f, t)` returns the tab `[ t[0], f(t[0],t[1]), f(f(t[0], t[1]),t[2]), ...]`. For example, the tab of the factorials up to 10 can be computed by:

```
@scan(@*, [$x : $x in 1 .. 10])
```

`@size(t)` returns the number of elements of `t`.

`@scramble(t)` : returns a new tab where the elements of `t` have been scrambled. The argument is unchanged.

`@sort(t)` : sorts in-place the elements into ascending order using <.

`@sort(t, cmp)` sorts in-place the elements into ascending order. The elements are compared using the function `cmp`. This function must accept two elements of the tab `t` as arguments, and returns a value converted to bool. The value returned indicates whether the element passed as first argument is considered to go before the second.

`@sputter(t, p, n)` : returns a new tab of length `n`. This tab is filled as follows: for each element, a random number between 0 and 1 is compared with `p` : if it is lower, then the element is the current element in `t`. If it is greater, we take the next element in `t` which becomes the current element. The process starts with the first element in `t`.

`@stutter(t, n)` : returns a new tab whose elements are each elements of `t` repeated n times. The argument is unchanged.

Lists and Tabs. *Antescofo*'s tabs may be used to emulate lists and the operators `@cons` and `@cdr` can be used to define recursive functions on tabs in a manner similar to recursive function on list. Note however that `@cdr` builds a new tab (contrary to the operation *cdr* on list in Lisp). A tab comprehension is often more convenient and usually more efficient. Several functions on tab are similar to well known functions on lists:

- Notice that `car`, `cdr` and `@map` are similar to the corresponding functions on list that exists in Lisp.
- `@concat(a, b)` returns the concatenation (*append*) of two lists.
- Arithmetic operations on vectors are done pointwise.

7.4 Variables

Antescofo variables are *imperative* variables: they are like a box that holds a value. The assignment of a variable consists in changing the value in the box:

```
$v  := expr
```

An assignment is an action, and as other action, it can be done after a delay. See sect. 5.3.

Variables are named with a $-identifier. By default, a variable is global, that is, it can be referred in an expression everywhere in a score.

Note that variables are not typed: the same variable may holds an integer and later a string.

7.4.1 Accessing Variable Histories

Variable are managed in a imperative manner. The assignment of a variable is seen as an internal event that occurs at some date. Such event is associated to a logical instant. Each *Antescofo* variable has a time-stamped history. So, the value of a variable at a given date can be recovered from the history, achieving the notion of *stream of values*. Thus, `$v` corresponds to the last value (or the current value) of the stream. It is possible to access the value of a variable at some date in the past using the *dated access*:

```
[date]:$v
```

returns the value of variable `$v` at date *date*. The date can be expressed in three different ways:

- as an update count: for instance, expression `[2#]:$v` returns then antepenultimate value of the stream;
- as an absolute date: expression `[3s]:$v` returns the value of `$v` three seconds ago;
- and as a relative date: expression `[2.5]:$v` returns the value of `$v` 2.5 beats ago.

For each variable, the programmer may specify the size n of its history, see next section. So, only the n "last values" of the variable are recorded. Accessing the value of a variable beyond the recorded values returns an undefined value.

User variables are assigned within an augmented score using Assignment Actions (see section 5.3). However, they can also be assigned by the external environment, using a dedicated API.

History reflected in a map. The history of a variable may be accessed also through a map. Three special functions are used to build a map from the history of a variable:

- `@history_map($x)` returns a list where element n is the $n-1$ to the last value of `$x`. In other word, the element associated to 1 in the map is the current value, the previous value is associated to element 2, etc. The size of this list is the size of the variable history, see the paragraph *History Length of a Variable* below. However, if the number of update of the variable is less than the history length, the corresponding undefined values are not recorded in the map.

- `@history_map_date($x)` returns a list where element n is the date (physical time) of $n - 1$ to the last update of `$x`. The previous remark on the map size applies here too.
- `@history_map_rdate($x)` returns a list where element n is the relative date of $n - 1$ to the last update of `$x`. The previous remark on the map size applies here too.

These three function are special forms: they accept only a variable as an argument. These functions build a snapshot of the history at the time they are called. Later, the same call will build eventually different maps. beware that the history of a variable is managed as a ring buffer: when the buffer is full, any new update takes the place of the oldest value.

7.4.2 Variables Declaration

Antescofo variables are global by default, that is visible everywhere in the score or they are declared local to a group which limits its scope and constraints its life. For instance, as common in scoped programming language, the scope of variable declared local in a `loop` is restricted to one instance of the loop body, so two loop body refers to two different instances of the local variable. This is also the case for the body of a `whenever` or of a process.

Local Variables. To make a variable local to a scope, it must be explicitly declared using a `@local` declaration. A scope is introduced by a `group`, a `loop`, a `whenever` or a process statement, see section 6. The `@local` declaration, may appear everywhere in the scope and takes a comma separated list of variables:

```
@local $a, $i, $j, $k
```

They can be several `@local` declaration in the same scope but all local variables can be accessed from the beginning of the scope, irrespectively of the location of their declaration.

A local variable may hide a global variable and there is no warning. A local variable can be accessed only within its scope. For instance

```
$x := 1
group {
  loc $x
  $x := 2
  print "local␣var␣$x:␣" $x
}
print "global␣var␣$x:␣" $x
```

will print

```
local var $x 2
global var $x 1
```

History Length of a Variable. For each variable, *Antescofo* records only an history of limited size. This size is predetermined, when the score is loaded, as the maximum of the history sizes that appears in expressions and in variable declarations.

In a declaration, the specification of an history size for the variable `$v` takes the form:

```
n:$v
```

where n is an integer, to specify that variable `$v` has an history of length at least n.

To make possible the specification of an history size for global variables, there is a declaration

```
@global  $x, 100:$y
```

similar to the `@local` declaration. Global variable declarations may appear everywhere an action may appear. Variables are global by default, thus, the sole purpose of a global declaration, is to specify history lengths.

The occurence of a variable in an expression is also used to determine the length of its history. In an expression, the n^{th} past value of a variable is accessed using the *dated access* construction (see 7.4):

```
[n#]:$v
```

When n is an integer (a constant), the length of the history is assumed to be at least n.

When there is no declaration and no dated access with a constant integer, the history size has an implementation dependant default size.

Lifetime of a Variable. A local variable can be referred as soon as its nearest enclosing scope is started but it can persist beyond the enclosing scope lifetime. For instance, consider this example :

```
Group G  {
  @local $x
  2 Loop L {
    ... $x ...
  }
}
```

The loop nested in the group run forever and accesses to the local variable `$x` after "the end" of the group `G`. This use of `$x` is perfectly legal. *Antescofo* manages the variable environment efficiently and the memory allocated for `$x` persists as long as needed but no more.

7.4.3 System Variables

There are several variables which are updated by the system. Composers have read-only access to these variables:

- `$BEAT_POS` is the current position in the score. This position is updated between two events, as the time progress: after the occurence of an event,`$BEAT_POS` increases until reaching the position in the score of the next waited event. The $BEAT_POS is stuck until the occurrence of this event or the detection of a subsequent event (making this one missed).

- $DURATION is the duration of the last detected event.
- $LAST_EVENT_LABEL is the label of the last event seen. This variable is updated on the occurence of an event only if the event has a label.
- $NOW corresponds to the absolute date of the "current instant" in seconds. The "current instant" is the instant at which the value of $NOW is required.
- $PITCH is the pitch of the current event. This value is well defined in the case of a Note and is not meaningful[17] for the other kinds of event.
- $RNOW is the date in relative time (in beats) of the "current instant".
- $RT_TEMPO is the tempo infered by the listening machine from the input audio stream.

Note that when an event occurs, several system variables are susceptible to change simultaneously.

7.4.4 Variables and Notifications

Notification of events from the machine listening module drops down to the more general case of variable-change notification from an external environment. The Reactive Engine maintains a list of actions to be notified upon the update of a given variable.

Actions associated to a musical event are notified through the $BEAT_POS variable. This is also the case for the group, loop and curve constructions which need the current position in the score to launch their actions with loose synchronization strategy. The whenever construction, however, is notified by all the variables that appear in its condition.

The *Antescofo* scheduler must also be globally notified upon any update of the tempo computed by the listening module and on the update of variables appearing in the local tempi expressions.

Temporal Shortcuts. The notification of a variable change may trigger a computation that may end, directly or indirectly, in the assignment of the same variable. This is known as a "temporal shortcut" or a "non causal" computation. The Event Manager takes care of stopping the propagation when a cycle is detected. See section 6.6.2. Program resulting in temporal shortcuts are usually considered as bad practice and we are developing a static analysis of augmented scores to avoid such situations.

[17]in the current version of *Antescofo*

7.4.5 Dates functions

Two functions let the composer know the date of a logical instant associated to the assignment of a variable `$v`: `@date([n#]:$v)` returns the date in the absolute time frame of the nth to last assignement of `$v` and `@rdate([n#]:$v)` returns the date in the relative time frame.

These functions are special forms: they accept only a variable or the dated access to a variable.

7.5 Operators and Predefined Functions

The main operators and predefined functions have been sketched in the section 7.2 and 7.3. We sketch here some operators or functions that are not linked to a specific type. The predefined functions are listed in annex A p. 85.

Conditional Expression. An important operator is the conditional *à la* C:

```
(bool_exp ? exp1 : exp2)
```

or equivalently

```
if (bool_exp) { exp1 } else { exp2 }
```

returns the value *exp* 1 if *bool_exp* evaluates to `true` and else *exp* 2. The parenthesis are mandatory. As usual, the conditional operators is a special function: it does not evaluates all of its arguments. If *bool_exp* is true, only *exp* 1 is evaluated, and similarly for false and *exp* 2.

They are also conditonal actions, cf. sect. 6.2. Conditional actions have a syntax similar to the `if`... `else`... form, but beware not to confuse them: a conditional expression appears only where an expression is expected. Another difference is that the true and the false branch of a conditional expression are both mandatory (because an expression must always have a value, irrespectively of the value of the condition).

`@size`. Function `@size` accepts any kind of argument and returns:

- for aggregate values, the "size" of the arguments; that is, for a `map`, the number of entries in the dictionary and for an `imap`, the number of breakpoints;
- for scalar values, `@size` returns a strictly negative number. This negative number depends only on the type of the argument, not on the value of the argument.

The "size" of an undefined value is `-1`, and this can be used to test if a variable refers to an undefined value or not (see also the predicates of the `@is_`*xxx* family, sect. 7.1 p. 47).

The addition is heavily overloaded in *Antescofo*. If one of the arguments of `+` is a string, then the other argument is converted (if needed) into a string and the result of the operation is the concatenation of the two strings.

7.6 Auto-Delimited Expressions in Actions

Expressions appear everywhere to parameterize the actions and this may causes some syntax problems. For example when writing:

```
print @f (1)
```

there is an ambiguity: it can be interpreted as the message `print` with two arguments (the function `@f` and the integer `1`) or it can be the message `print` with only one argument (the result of the function `@f` applied to the argument `1`). This kind of ambiguity appears in other places, as for example in the specification of the list of breakpoints in a curve.

The cause of the ambiguity is that we don't know where the expression starting by `@f` finishes. This leads to distinguish a subset of expressions: *auto-delimited expressions* are expressions that cannot be "extended" with what follows. For example, integers are auto-delimited expressions and we can write

```
print 1 2
```

without ambiguity (this is the message `print` with two arguments and there is no other possible interpretation). Variables are others examples of auto-delimited expressions.

Being auto-delimited is an involved syntactic property. *Antescofo* accepts a simple subset of auto-delimited expressions to avoid possible ambiguities in the places where this is needed, *i.e.*:

- in the specification of a delay,
- in the arguments of a message,
- in the arguments of an internal command,
- in the specification list of breakpoints in a curve,
- in the specification of an attribute value
- in the specification of a `when` or `until` clause,

If an expression is provided where an auto-delimited expression is required, a syntax error is declared. But, *every expression between braces is an auto-delimited expression.*

So, a rule of thumb is to put the expressions in the listed contexts between braces, when this expression is more complex than a constant or a variable.

8 Synchronization and Error Handling Strategies

The musician's performance is subject to many variations from the score. There are several ways to adapt to this musical indeterminacy based on specific musical context. The musical context that determines the correct synchronization and error handling strategies is at the composer or arranger's discretion.

8.1 Synchronization Strategies

8.1.1 Loose Synchronization

By default, once a group is launched, the scheduling of its sequence of relatively-timed actions follows the real-time changes of the tempo from the musician. This synchronization strategy is qualified as `loose`.

Figure 10 attempts to illustrate this within a simple example: Figure 10(a) shows the *ideal performance* or how actions and instrumental score is given to the system. In this example, an accompaniment phrase is launched at the beginning of the first event from the human performer. The accompaniment in this example is a simple group consisting of four actions that are written parallel (and thus synchronous) to subsequent events of the performer in the original score, as in Figure 10(a). In a regular score following setting (*i.e.*, correct listening module) the action group is launched synchronous to the onset of the first event. For the rest of the actions however, the synchronization strategy depends on the dynamics of the performance. This is demonstrated in Figures 10(b) and 10(c) where the performer hypothetically accelerates or decelerate the consequent events in her score. In these two cases, the delays between the actions will grows or decreases until converge to the performer tempo.

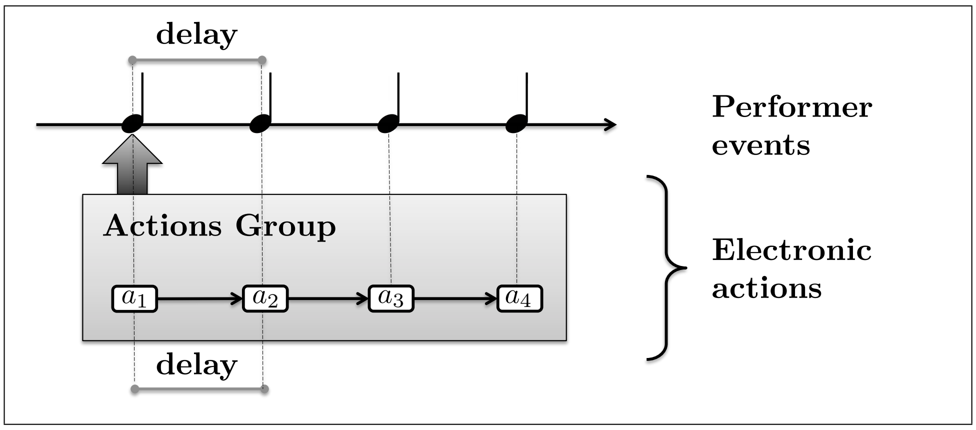
The `loose` synchronization strategy ensures a fluid evolution of the actions launching but it does not guarantee a precise synchronization with the events played by the musician. Although this fluid behavior is desired in certain musical configurations, there is an an alternative synchronization strategy where the electronic actions will be launched as close as possible to the events detection.

8.1.2 Tight Synchronization

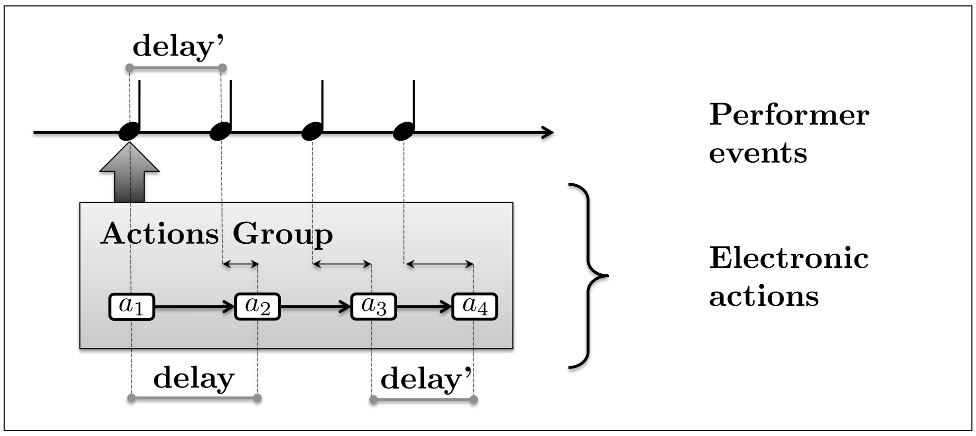
If a group is `tight`, its actions will be dynamically analyzed to be triggered not only using relative timing but also relative to the nearest event in the past. Here, the nearest event is computed in the ideal timing of the score.

This feature evades the composer from segmenting the actions of a group to smaller segments with regards to synchronization points and provide a high-level vision during the compositional phase. A dynamic scheduling approach is adopted to implement the `tight` behavior. During the execution the system synchronize the next action to be launched with the corresponding event.

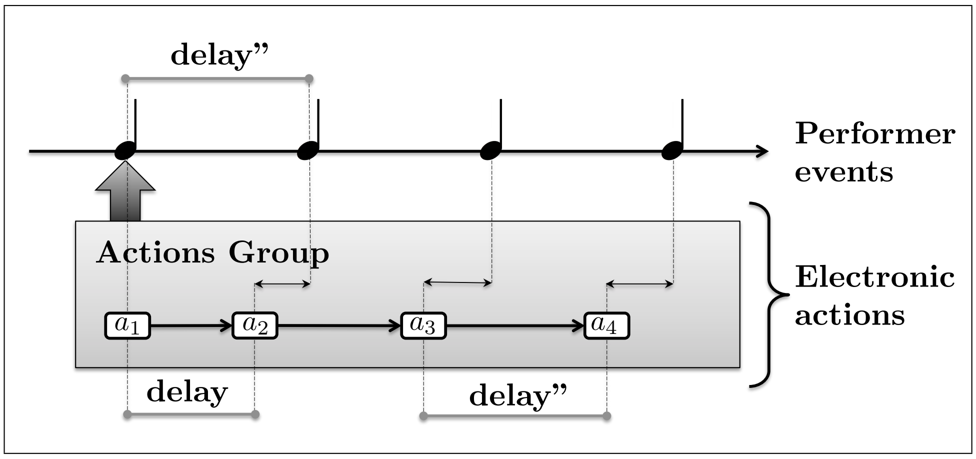
Note that the arbitrary nesting of groups with arbitrary synchronization strategies do not



(a) Ideal performance



(b) Faster performance (delay' < delay)



(c) Slower performance (delay" > delay)

Figure 10: *The effect of tempo-only synchronization for accompaniment phrases: illustration for different tempi.* In the score, the actions are written to occur simultaneously with the notes, cf. fig. (a). Figure (b) and (c) illustrate the effect of a faster or a slower performance. In these cases, the tempo inferred by the listening machine converges towards the actual tempo of the musicians. Therefore, the delays, which are relative to the inferred tempo, vary in absolute time to converge towards the delay between the notes observed in the actual performance.

always make sense: a group `tight` nested in a group `loose` has no well defined triggering event (because the start of each action in the `loose` group are supposed to be synchronized dynamically with the tempo). All other combinations are meaningful. To acknowledge that, groups nested in a `loose` group, are `loose` even if it is not enforced by the syntax.

8.2 Missed Event Errors Strategies

Parts but not all of the errors during the performance are handled directly by the listening modules (such as false-alarms and missed events by the performer). The critical safety of the accompaniment part is reduced to handling of missed events (whether missed by the listening module or human performer). In some automatic accompaniment situations, one might want to dismiss associated actions to a missed event if the scope of those actions does not bypass that of the current event at stake. On the contrary, in many live electronic situations such actions might be initializations for future actions to come. It is the responsability of the composer to select the right behavior by attributing relevant *scopes* to accompaniment phrases and to specify, using an attribute, the corresponding handling of missed events.

A group is said to be `local` if it should be dismissed in the absence of its triggering event during live performance; and accordingly it is `global` if it should be launched in priority and immediately if the system recognizes the absence of its triggering event during live performance. Once again, the choice of a group being `local` or `global` is given to the discretion of the composer or arranger.

Combining Synchronization and Error Handling. The combination of the synchronization attributes (`tight` or `loose`) and error handling attributes (`local` or `global`) for a group of accompaniment actions give rise to four distinct situations. Figure 11 attempts to showcase these four situations for a simple hypothetical performance setup similar to Figure 10.

Each combination corresponds to a musical situation encountered in authoring of mixed interactive pieces:

- `local` and `loose`: A block that is both local and loose correspond to a musical entity with some sense of rhythmic independence with regards to synchrony to its counterpart instrumental event, and strictly reactive to its triggering event onset (thus dismissed in the absence of its triggering event).
- `local` and `tight`: Strict synchrony of inside actions whenever there's a spatial correspondence between events and actions in the score. However actions within the strict vicinity of a missing event are dismissed. This case corresponds to an ideal concerto-like accompaniment system.
- `global` and `tight`: Strict synchrony of corresponding actions and events while no actions is to be dismissed in any circumstance. This situation corresponds to a strong musical identity that is strictly tied to the performance events.

- `global` and `loose`: An important musical entity with no strict timing in regards to synchrony. Such identity is similar to integral musical phrases that have strict starting points with *rubato* type progressions (free endings).

The Antescofo behavior during an error case is shown in Figure 11. To have a good understanding of the picture note that:

- An **action** (a_i), associated with a delay, can be an atomic action, a group, a loop or a curve.
- The **triggers**, defining when an action is fired (*i.e.*, at an event detection, at another action firing, at a variable update...), are represented with plain arrows in the figure and detail mainly the schedule of the next action delay or the direct firing of an action. A black arrow signals a normal triggers whereas a red arrow is for the error case (*i.e.*, a missed, a too late or a too early event).

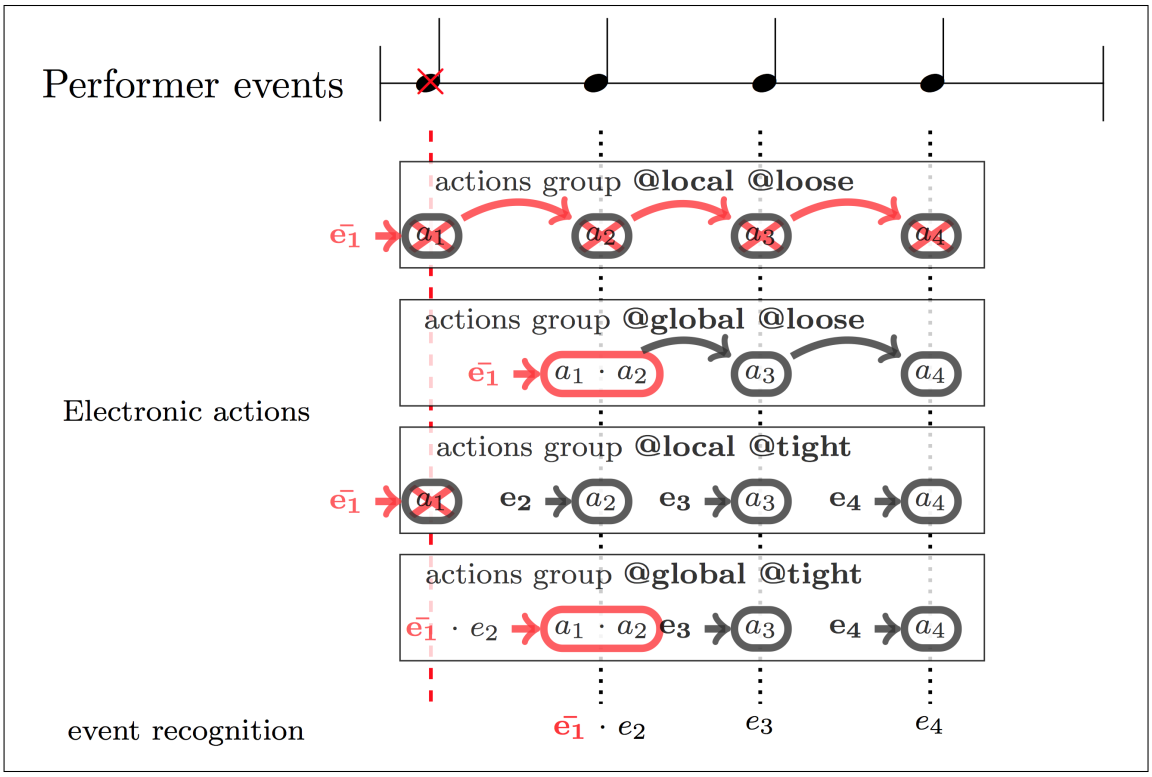


Figure 11: *Action behavior in case of a missed event for four synchronization and error handling strategies.* In this example, the score is assumed to specify four consecutive performer events (e_1 to e_4) with associated actions gathered in a group. Each action is aligned in the score with an event. The four groups correspond to the four possible combinations of two possible synchronization strategies with the two possible error handling attributes. This diagram illustrates the system behavior in case event e_1 is missed and the rest of events detected without tempo change. Note that e_1 is detected as missed (in real-time) once of course e_2 is reported. The signaling of the missing e_1 is denoted by $\bar{e}_1$.

Remarks:

- A sequence of actions following an event in an *Antescofo* score correspond to a phantom group with attributes **@global** and **@loose**. In other words, the two following scores are similar.

```
NOTE C 2.0
d₁ action1
d₂ group G1
{
    action2
}
NOTE D 1.0
```

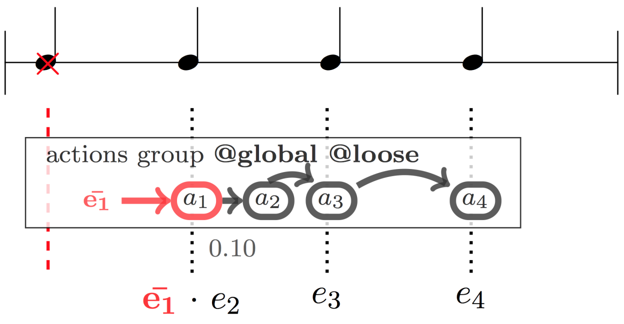
```
NOTE C 2.0
Group @global @loose
{
    d₁ action1
    d₂ group G1
    {
        action2
    }
}
NOTE D 1.0
```

  As a consequence, if `G1` is `@loose` and `@local` and the first note (C 2.0) is missed, the group is fired if $d_1 + d_2 >= 2.0$ and not otherwise.

- During a performance, even in case of errors, if an action has to be launch, the action is fired at a date which as close as possible from the date specified in the score. This explain the behavior of a `@global @loose` group when its event trigger is recognized as missed. In this case, the action that are still in the future, are played at their "right" date, while the actions that should have been triggered, are launched immediately (as a tight group strategy). In the previous example, we remark delays variations (a_2 is directly fired for the @loose @global case and not 1.0 after a_1). This 'tight' re-schedulation is important if the a_2 action has a delay of 1.10, the action should effectively be fired at 0.10 beat after a_1 (next figure) :

```
NOTE C 1.0 e1
group G1 @global @loose
{
          a1
    1.10  a2
    1.0   a3
    1.0   a4
}

NOTE D 1.0 e2
NOTE D 1.0 e3
NOTE D 1.0 e4
```



9 Process

Process are similar to functions: after its definition, a function `@f` can be called and computes a value. After its definition, a process `::P` can be called and generates actions that are run as a group. This group is called the “instanciation of the process”. They can be several instanciations of the same process that run in parallel.

Process can be defined using the `@proc_def` construct. For instance,

```
@proc_def ::trace($note, $d)
{
          print begin $note
     $d print end $note
}
```

The name of the process denotes a proc value, see 7.2.7, and it is used for calling the process. A process call, is similar to a function call: arguments are between parenthesis:

```
NOTE C4 1.3
   ::trace("C4", 1.3)
   action1
NOTE D3 0.5
   ::trace("D3", 0.5)
```

In the previous code we know that `::trace("C4", 1.3)` is a process call because the name of functions and the name of processes differs.

A process call is an atomic action (it takes no time and does not introduces a new scope for variables). The result of the call is evaluated as a group. So the previous code fragment behave similarly as:

```
NOTE C4 1.3
    group {
          print begin "C4"
      1.3 print end "C4"
    }
    action1
NOTE D3 0.5
     group {
          print begin "D3"
      0.5 print end "D3"
    }
```

A process can also be called in an expression and the instanciation mechanism is similar: a group is started and run in parallel. However, an *exec value*, is returned as the result of the process call, see section 7.2.8. This value refers to the group lauched by the process instanciation and is eventually used in the computation of the surrounding expression.

9.1 Macro *vs.* Processus

From the point of view of the process instanciation, a process call can be seen as a kind of macro expansion (see section 10). However, contrary to macros:

- the expression that are arguments of a call are computed only once at call time, and call time is when the application is fired (not when the file is read);
- the actions that are launched consequently to the firing of a process application are computed when the process is applied,
- process can be recursive;
- process are higher-order values: a process `::P` can be used as the value of the argument of a function or a process call. This enable the parameterization of process, an expressive and powerful construction to describe complex compositional schemes.

Let give some examples of higher order recursive processes.

9.2 Recursive Process

A infinite loop

```
Loop L 10
{
  ... action_i ...
}
```

is equivalent to a call of the recursive process `::L` defined by:

```
@proc_def ::L()
{
  Group repet {
    10 ::L()
  }
  ... action_i ...
}
```

The group `repet` is used to launch recursively the process without disturbing the timing of the actions in the loop body. In this example, the process has no parameters.

9.3 Process as Values

A process can be the argument of another process. For example:

```
@Proc_def ::Tic($x) {
  $x print TIC
}
@proc_def ::Toc($x) {
  $x print TOC
}
@proc_def ::Clock($p, $q) {
  :: $p(1)
  :: $q(2)
  3 ::Clock($q, $p)
}
```

A call to `Clock(::Tic, ::Toc)` will print `TIC` one beat after the call, then `TOC` two beats latter, and then `TIC` again at date 4, `TOC` at date 6, etc.

In the previous code a `::` is used in the first two lines of the `::Clock` process to tell *Antescofo* that the value of arguments `$p` and `$q` must be processes and that this is a process call and not a function call. This indication is mandatory because in this case, there is no way to know for sure that `$p(1)` is a function call or a process call.

9.4 Aborting a Process

The actions spanned by a process call constitute a group. It is possible to abort all groups spanned by the calls to a given process using the process name:

```
  abort ::P
```

will abort all the active instances of `::P`.

It is possible to kill a specific instance of the process `::P`, provided that the *exe*

10 Macros

Macro can be defined using the `@macro_def` construct. The macro-expansion is a syntactic replacement that occurs during the parsing but before any evaluation. The body of a macro can call (other) macros. When a syntax error occurs in the expansion of a macro, the location given refers to the text of the macro and is completed by the location of the macro-call site (which can be a file or the site of another macro-expansion).

A process can be often used in place of a macro with several advantages. See paragraph 9.1 for a comparison of macros and processes. The use of processes must be privileged over macros but in rare place a macro can be the solution (*e.g.* to generate `@`*xxx*`_def` constructs).

10.1 Macro Definitions

Macro name are `@`-identifier. For compatibility reason, a simple identifier can be used but the `@`-form must be used to call it.

```
@macro_def @f($a, $x, $b)  { $a * $x + $b }
```

Macro parameter names are `$`-identifiers. The body of the macro is between braces. The white spaces and tabulation and carriage-returns immediately after the open brace and immediately before the closing brace are not part of the macro body.

Notice that in a macro-call, the white-spaces and carriage-returns surrounding an argument are removed. But "inside" the argument, one can use it:

```
@macro_def @delay_five($x)
{
  5 group {
    $x
  }
}
@delay_five(
   1 print One
   2 print Two
)
```

is expanded as:

```
5 group {
    1 print One
    2 print Two
  }
```

Macro have been extended to accept zero argument:

```
@macro_def @PI { 3.1415926535 }
let $x := @sin($t * @PI)
```

10.2 Expansion Sequence

The body of a macro `@m` can contain calls to others macro, but they will be expanded after the expansion of `@m`. Similarly, the arguments of a macro may contain calls to other macros, but beware that their expansion take place only *after* the expansion of the top-level call. So one can write:

```
@macro_def apply1($f,$arg) { $f($arg) }
@macro_def concat($x, $y)  { $x$y }
let $x := @apply1(@sin, @PI)
print @concat(@concat(12, 34), @concat(56, 78))
```

which results in

```
let $x := @sin(3.1415926535)
print 1234 5678
```

The expression `@sin(3.1415926535)` results from the expansion of `@sin(@PI)` while `1234 5678` results from the expansion of `@concat(12, 34)@concat(56, 78)`. In the later case, we don't have `12345678` because after the expansion the first of the two remaining macro calls, we have the text `1234@concat(56, 78)` which is analyzed as a number followed by a macro call, hence two distinct tokens.

10.3 Generating New Names

The use of macro often requires the generation of new name. Consider using *local variables* (see below) that can be introduced in groups. Local variables enable the reuse of identifier names, as in modern programming languages.

Local variable are not always a solution. They are two special macro constructs that can be used to generates fresh identifiers:

```
@UID(id)
```

is substituted by a unique identifier of the form id*xxx* where *xxx* is a fresh number (unique at each invocation). `id` can be a simple identifier, a `$`-identifier or an `@`-identifier. The token

```
@LID(id)
```

is replaced by the id*xxx* where *xxx* is the number generated by the last call to `@UID`. For instance

```
LFWD 2 @name = @UID(loop)
{
  let @LID($var) := 0
  ...
  superVP speed @LID($var) @name = @LID(action)
}
...
kill @LID(action) of @LID(loop)
...
```

```
kill @LID(loop)
```

is expanded in (the number 33 used here is for the sake of the example):

```
LFWD 2 @name = loop33
{
  let $var33 := 0
  ...
  superVP speed $var33 @name = action33

}
...
kill action33 of loop33
...
kill loop33
```

The special constructs `@UID` and `@LID` can be used everywhere (even outside a macro body).

If the previous constructions are not enough, they are some tricks that can be used to concatenate text. For example, consider the following macro definition:

```
@macro_def @Gen($x, $d, $action)
{
  GFWD @name = Gengroup$x
  {
    $d $action
    $d $action
  }
}
```

Note that the character `$` cannot be part of a simple identifier. So the text `Gengroup$x` is analyzed as a simple identifier immediately followed by a `$`-identifier. During macro-expansion, the text `Gengroup$x` will be replaced by a token obtained by concatenating the actual value of the parameter `$x` to `Gengroup`. For instance

```
@Gen(one, 5, print Ok)
```

will expand into

```
GFWD @name = Gengroupone
{
  5 print Ok
  5 print Ok
}
```

Comments are removed during the macro-expansion, so you can use comment to concatenate thing after an argument, as for the C preprocessor:

```
@macro_def @adsuffix($x) { $x/**/suffix }
@macro_def concat($x, $y) { $x$y }
```

With these definition,

```
@addsuffix($yyy)
@concat( 3.1415 , 9265 )
```

is replaced by

```
$yyysuffix
3.14159265
```

11 *Antescofo* Workflow

This chapter is still to be written...

11.1 Editing the Score

- From score editor (Finale, Notability, Sibelius) to *Antescofo* score.
- Conversion using *Ascograph* (import of Midi files and of MusicXML files).
- Direct editing using *Ascograph* .
- Latex printing of the score using the package `lstlisting`
- Syntax coloring for `TextWrangler` and `emacs` :-)

Ascograph is a graphical tool that can be used to edit and control a running instance of *Antescofo* through OSC messages.

Ascograph and *Antescofo* are two independent applications but the coupling between *Ascograph* and an *Antescofo* instance running in MAX appears transparent for the user: a double-click on the *Antescofo* object launches *Ascograph*, saving a file under *Ascograph* will reload the file under the *Antescofo* object, loading a file under the *Antescofo* object will open it under *Ascograph*, etc.

Ascograph is available in the same bundle as *Antescofo* on the IRCAM Forum.

...

11.2 Tuning the *Antescofo* Listening Machine

...

11.3 Debuging an *Antescofo* Score

11.4 Dealing with Errors

Errors, either during parsing or during the execution of the *Antescofo* score, are signaled on the MAX console.

The reporting of syntax errors includes a localization. This is generally a line and column number in a file. If the error is raised during the expansion of a macro, the file given is the name of the macro and the line and column refers to the begining of the macro definition. Then the location of the call site of the macro is given.

See the paragraph 11.8 for additional information on the old syntax.

11.4.1 Monotoring with Notability

...

11.4.2 Monotoring with *Ascograph*

.

...

11.4.3 Tracing an *Antescofo* Score

They are several alternative features that make possible to trace a running *Antescofo* program.

Printing the Parsed File. Using *Ascograph*, one has a visual representation of the parsed *Antescofo* score along with the textual representation.

The result of the parsing of an *Antescofo* file can be listed using the `printfwd` internal command. This command opens a text editor. Following the verbosity, the listing includes more or less information.

...

Verbosity. The verbosity can be adjusted to trace the events and the action. A verbosity of n includes all the messages triggered for a verbosity $m < n$. A verbosity of:

- 1: prints the parsed files on the shell console, if any.
- 3: trace the parsing on the shell console. Beware that usually MAX is not launched from a shell console and the result, invisible, slowdown dramatically the parsing. At this level, all events and actions are traced on the MAX console when they are recognized of launched.
- 4: traces also all audio internals.

The `TRACE` Outlet. If an *outlet* named `TRACE` is present, the trace of all event and action are send on this outlet. The format of the trace is

```
EVENT label ...
ACTION label ...
```

Tracing the Updates of a Variable. If one want to trace the updates of a variable `$v`, it is enough to add a corresponding `whenever` at the begining of the scope that defines `$v`:

```
whenever ($v = $v)
{
    print Update "$v:␣" $v
}
```

The condition may seems curious but is needed to avoid the case where the value of `$v` if interpreted as `false` (which will prohibit the triggering of the `whenever` body).

11.5 Interacting with MAX

When embedded in MAX, the *Antescofo* systems appears as an `antescofo~` object that can be used in a patch. This object presents a fixed interface through its inlets and outlets.

11.5.1 Inlets

The main inlet is dedicated to the audio input. *Antescofo*'s default observation mode is “audio” and based on pitch and can handle multiple pitch scores (and audio). But it is also capable of handling other inputs, such as control messages and user-defined audio features. To tell *Antescofo* what to follow, you need to define the type of input during object instantiation, after the `@inlets` operator. The following hardcoded input types are recognized:

- `KL` is the (default) audio observation module based on (multiple) pitch.
- `HZ` refers to raw pitch input as control messages in the inlet (e.g. using fiddleõr yinõbjects).
- `MIDI` denotes to midi inputs.

You can also define your own inlets: by putting any other name after the '@inlets' operator you are telling *Antescofo* that this inlet is accepting a LIST. By naming this inlet later in your score you can assign *Antescofo* to use the right inlet, using the `@inlet` to switch the input stream.

11.5.2 Outlets

By default, they are three *Antescofo*'s outlets:

- `Main outlet` (for note index and messages),
- `tempo` (BPM / Float),
- `score label` (symbol) plus an additional BANG sent each time a new score is loaded.

`Main outlet` `tempo` `score label` Additional (predefined) outlet can be activated by naming them after the `@outlets` operator. The following codenames are recognized

- `ANTEIOI` Anticipated IOI duration in ms and in runtime relative to detected tempo
- `BEATNUM` Cumulative score position in beats
- `CERTAINTY` *Antescofo*'s live certainty during detections $[0, 1]$
- `ENDBANG` Bang when the last (non-silence) event in the score is detected
- `MIDIOUT`
- `MISSED`
- `NOTENUM` MIDI pitch number or sequenced list for trills/chords
- `SCORETEMPO` Current tempo in the original score (BPM)
- `TDIST`
- `TRACE`
- `VELOCITY`

`ANTEIOI BEATNUM CERTAINTY ENDBANG MIDIOUT MISSED NOTENUM SCORETEMPO TDIST TRACE VELOCITY`

11.5.3 Predefined Messages

The *Antescofo* object accepts predefined message sent to `antescofo-mess1`. These messages corresponds to the internal commands described in section 5.6.

11.6 Interacting with PureData

...

11.7 *Antescofo* Standalone Offline

...

A standalone offline version of *Antescofo* is available. By "standalone" we mean that *Antescofo* is not embedded in Max or PD. It appears as an executable (command line). By "offline" we means that this version does not accept a real-time audio input but an audio file. The time is then managed virtually and goes as fast as possible. This standalone offline version is the machine used for the "simulation" feature in *Ascograph*.

The help of the command line is given in Fig. 12.

```
Usage : antescofo [options...] [scorefile]
  Syntax of options: --name or --name value or --name=value
  Some options admit a short form (-x) in addition to a long form (--uvw)

Offline execution Modes:
  --full : This is the default mode where an Antescofo score file is
           aligned against an audio file and the actions are triggered.
           A score file (--score) and an audio file (--audio) are both needed.
  --play : Play mode where the audio events are simulated from the
           score specification (no audio recognition) (-p).
           A score file (--score) is needed.
  --recognition : Audio recognition-only mode (no action is triggered) (-r)
                  An audio file is needed.

Inputs/Output files:
  --score filename : input score file (-s)
          alternatively, it can be specified as the last argument of the command line
  --audio filename : input audio file (-a)
  --output filename : output file for the results in recognition mode (default standard output) (-o)
  --lab : output format in ecognition mode (default, alternative --mirex)
  --mirex : output format in ecognition mode  (alternative --lab)
  --trace filename : trace all events and actions (use 'stdout' for standard output) (-t)

Listening module options:
  --fftlen samples : fft window length (default 2048) (-F)
  --hopsize samples : antescofo resolution in samples (default: 512) (-S)
  --gamma float : Energy coefficient (default: -2.0) (-G)
  --pedal (0|1) : pedal on=1/off=0 (default: 0)
  --pedaltime float : pedaltime in milliseconds (default: 600.0) (-P)
  --nofharm n : number of harmonics used for recognition (default: 10) (-H)

Reactive module options:
  --message filename : write messages in filename (use 'stdout' for standard output) (-m)
  --strict : program abort when an error is encountered

Others options
  --verbosity level : verbosity (default 0) (-v)
  --version : current version (-V)
  --help : print this help (-h)
```

Figure 12: Help of the standalone offline command line.

11.8 Old Syntax

...

The old syntax for several constructs is still recognized but is deprecated. Composers are urged to use the new one.

```
KILL delay name
KILL delay name OF name
GFWD delay name attributes { ... }
LFWD delay name period attributes { ... }
CFWD  delay name step attributes { ... }
```

where:

- `KILL` and `KILL OF` correspond to `abort` and `abort of`. The specification of *delay* and *attributes* are optional, *name* is mandatory.
- `GFWD` corresponds to `group`. The specification of *delay* and *attributes* are optional, *name* is mandatory.
- `LFWD` corresponds to `loop`. The argument *period* is mandatory and correspond to the period of the loop.
- `CFWD` corresponds to `curve`. The parameter *step* is the step used in the sampling of the curve.

12 Stay Tuned

Antescofo is in constant improvement and evolution. Several directions are envisioned; to name a few:

- temporal regular expressions,
- modularization of the listening machine,
- multimedia listening,
- graphical editor and real-time control board,
- standalone version,
- richer set of values and libraries,
- static analysis and verification of scores,
- multi-target following,
- extensible error handling strategies,
- extensible synchronization strategies,
- parallel following,
- distributed coordination,
- tight coupling with audio computation.

Your feedback is important for us. Please, send your comments, questions, bug reports, use cases, hints, tips *&* wishes using the Ircam Forum *Antescofo* discussion group at

`http://forumnet.ircam.fr/discussion-group/antescofo/?lang=en`

A Library of Predefined Functions

Antescofo includes a set of predefined functions. They are described mostly in the section 7. For the reader convenience, we give here a list of these functions.

The sequence of name after the function defines the type of the arguments accepted by a function. For example, "`numeric`" is used when an argument must satisfy the predicate `@is_numeric`, that is, `@is_int` or `@is_float`. In addition we use the terms *value* when the function accepts any kind of arguments.

Listable Functions. When a function f is marked as *listable*, the function is extended to accepts `tab` arguments in addition to scalar arguments. Usually, the result of the application of f on a tab is the tab resulting on the point-wise application of f to the scalar elements of the tab. But for predicate, the result is the predicate that returns true if the scalar version returns true on all the elements of the tabs.

For example, `@abs` is a listable function on numerics: so it can be applied to a tab of numerics. The results is the tab of the absolute value of the elements of the tab argument. The function `@approx` is a listable predicate and `@approx(`*u*`, `*v*`)` returns true if `@approx(`*u*`[`*i*`],` *v*`[`*i*`])` returns true for all elements *i* of the tabs *u* and *v*.

Side-Effect. The majority of functions are "pure function", that is, they do not modify their argument and build a new value for the result. In some case, the function work by a side-effect. Such function are marked as *impure*.

`@+` (*value*, *value*), *listable*: prefix form of the infix `+` binary operator: `@+(x, y)` ≡ `x+y`. The function form is useful to pass the operator to a high-order function as in `@reduce(@+, v)` which sums up all the elements of the tab `v`.

The addition of an `int` and a `float` returns a float.

The addition of two `string` corresponds to the concatenation of the arguments. The addition of a string and any other value convert this value into its string representation before the concatenation.

`@-` (numeric, numeric), *listable*: prefix form of the infix `-` arithmetic operator. Coercions between numeric apply when needed.

`@*` (numeric, numeric), *listable*: prefix form of the infix `*` arithmetic operator. Coercions between numeric apply when needed.

`@/` (numeric, numeric), *listable*: prefix form of the infix `/` arithmetic operator. Coercions between numeric apply when needed.

`@%` (numeric, numeric), *listable*: prefix form of the infix `%` binary operator. Coercions between numeric apply when needed.

`@<` (*value*, *value*), *listable*: prefix form of the infix `<` relational operator. This is a total order: value of different type can be compared and the order between unrelated type is *ad hoc*. Note however that coercion between numeric applies if needed.

`@>=` (*value*, *value*), *listable*: prefix form of the infix `>=` relational operator. Same remarks as for `@<`.

`@==` (*value*, *value*), *listable*: prefix form of the infix `==` relational operator. Same remarks as for `@<`. So, beware that `1 == 1.0` evaluates to `true`.

`@!=` (*value*, *value*), *listable*: prefix form of the infix `!=` relational operator. Same remarks as for `@<`. needed.

`@<=` (*value*, *value*), *listable*: prefix form of the infix `<=` relational operator. Same remarks as for `@<`.

`@<` (*value*, *value*), *listable*: prefix form of the infix `<` relational operator. Same remarks as for `@<`.

`@&&` (*value*, *value*), *listable*: functional form of the infix `&&` logical conjunction. Contrary to the functional form is *not lazy*, cf. sect. 7.2.2 p. 48: so `@&&(a, b)` evaluates `b` irrespectively of the value of `a`

`@||` (*value*, *value*), *listable*: prefix form of the infix `||` logical disjunction. Same remarks as for `&&`.

`@abs` (numeric), *listable*: absolute value

`@acos` (numeric), *listable*: arc cosine

`@add_pair` (map, key:*value*, *value*) or (imap, key:numeric, numeric): add a new entry to a dictionary or a breakpoint in a interpolated map.

`@approx` (x:numeric, y:numeric), *listable*. The function call can also be written with the special syntax `(x ~ y)` (the parenthesis are mandatory).

This predicate returns true if `abs((x - y)/max(x, y)) < $APPROX_RATIO`. The variable `$APPROX_RATIO` is initalized to `0.1` so `(x ~ y)` means `x` and `y` differ by less than 10%. By changing the value of the variable `$APPROX_RATIO`, one changes the level of approximation for the further calls to `@approx`.

If one argument is a tab, the other argument u is extended to tab if it is scalar (all elements of the extension is equal to u) and the predicate returns true if it hold point-wise for all element of the tabs. For example `(tab[1, 2] ~ 1.02)` returns `false` because we don't have `(2 ~ 1.02)`.

`@asin` (numeric), *listable*: arc sine

`@atan` (numeric), *listable*: arc tangente

`@between` (a:numeric, x:numeric, b:numeric), *listable*. This function admits two special syntax and can be written `(x in a .. b)` (the parenthesis are mandatory).

This predicate is true if `a < x < b`. If one argument is a tab, each scalar argument u is extended into a tab whose all elements are equal to u and the predicate returns true if it hold point-wise for all element of the tabs. For example:

```
([1, 2] in 0 .. 3)
```

returns true because `0 < 1 < 3` and `1 < 2 < 3`.

`@bounded_integrate_inv`

`@bounded_integrate`

`@car` t:tab: returns the first element of tab `t` if `t` is not empty, else an empty tab.

`@cdr` t:tab: if `t` is not empty, it returns a copy of `t` but deprived of its first element, else it returns an empty tab.

`@ceil` (numeric), *listable*: This function returns the smallest integral value greater than or equal to its argument.

`@clear` (tab *or* map), *impure*: clear all elements in the tab (resp. map) argument, resulting in a vector (resp. a dictionary) of size zero.

`@concat` (tab, tab): returns a new tab made by the concatenation of the two tab arguments.

`@cons` (v, t:tab): returns a new tab which is like `t` but has `v` prepended.

`@copy` (*value*): returns a fresh copy of the argument.

`@cosh` (numeric), *listable*: computes the hyperbolic cosine of its argument.

`@cos` , *listable*: computes the cosine of its argument.

`@count` (tab, *value*): computes the occurrences of its second argument as element in its first argument.

`@dim` (t) returns the dimension of `t`, i.e. the maximal number of nested tabs. If `t` is not a tab, the dimension is `0`.

`@exp` (x:numeric), *listable*: the base-e exponential of `x`.

`@find_index` (t:tab, f:function) returns the index of the first element of `t` that satisfies the predicate `f`.

`@flatten` (t) build a new tab where the nesting structure of `t` has been flattened. For example,

```
@flatten([[1, 2], [3], [[], [4, 5]]]) ⟶ [1, 2, 3, 4, 5, 6]
```

`@flatten` (t:tab, l:numeric) returns a new tab where `l` levels of nesting has been flattened. If `l == 0`, the function is the identity. If `l` is strictly negative, it is equivalent to `@flatten` without the level argument.

```
@flatten([1, [2, [3, 3], 2], [[[4, 4, 4]]]], 0)
⟶ [1, [2, [3, 3], 2], [[[4, 4, 4]]]]
@flatten([1, [2, [3, 3], 2], [[[4, 4, 4]]]], 1)
⟶ [1, 2, [3, 3], 2, [[4, 4, 4]]]
@flatten([1, [2, [3, 3], 2], [[[4, 4, 4]]]], 2)
⟶ [1, 2, 3, 3, 2, [4, 4, 4]]
@flatten([1, [2, [3, 3], 2], [[[4, 4, 4]]]], 3)
⟶ [1, 2, 3, 3, 2, 4, 4, 4]
@flatten([1, [2, [3, 3], 2], [[[4, 4, 4]]]], 4)
⟶ [1, 2, 3, 3, 2, 4, 4, 4]
@flatten([1, [2, [3, 3], 2], [[[4, 4, 4]]]], -1)
⟶ [1, 2, 3, 3, 2, 4, 4, 4]
```

`@floor` (x:numeric), *listable*: returns the largest integral value less than or equal to `x`.

`@gshift_map` (a:map, f:function): returns a new map `b` such that `@b(f(x)) = a(x)` where `f` can be a map, an interpolated map or an intentional function.

`@insert` (t:tab, i:numeric, v:val), *impure*: inserts “in place” the value `v` into the tab `t` after the index `i` (tab’s elements are indexed starting with `0`). If `i` is negative, the insertion take place in front of the tab. If `i` $\leq$ `@size(t)` the insertion takes place at the end of the tab. Notice that the form `@insert "file"` is also used to include a file at parsing time.

`@integrate`

`@iota` (n:numeric): return `[ $x | $x in n]`, that is, a tab listing the integers from `0` to `n` excluded.

`@is_bool` (*value*): the predicate returns true if its argument is a boolean value.

`@is_defined` (*value*): the predicate returns true for any argument that is not of type `Undefined`.

`@is_fct` (*value*): the predicate returns true if its argument is an intentional function.

`@is_float` (*value*): the predicate returns true if its argument is a decimal number.

`@is_function` (*value*): the predicate returns true if its argument is a map, an interpolated map or an intentional function.

`@is_integer_indexed` (*value*): the predicate returns true if its argument is a map whose domain is a set of integers.

`@is_interpolatedmap` (*value*): the predicate returns true if its argument is an interpolated map.

`@is_int` (*value*): the predicate returns true if its argument is an integer.

`@is_list` (*value*): the predicate returns true if its argument is a map whose domain is the integers $\{0, \ldots, n\}$ for some n.

`@is_map` (*value*): the predicate returns true if its argument is a map.

`@is_numeric` (*value*): the predicate returns true if its argument is an integer or a decimal.

`@is_string` (*value*): the predicate returns true if its argument is a string.

`@is_symbol` (*value*): the predicate returns true if its argument is a symbol.

`@is_undef` (*value*): the predicate returns true if its argument is the undefined value.

`@is_vector` (*value*): the predicate returns true if its argument is a list and its range is a set of numeric.

`@lace` (t:tab, n:numeric) returns a new tab whose elements are interlaced sequences of the elements of the `t` subcollections, up to size `n`. The receiver is unchanged.

```
@lace([[1, 2, 3], 6, ["foo", "bar"]], 9) == [1, 6, "foo", 2, 6, "bar", 3, 6
```

`@listify` (map): returns the range of its argument as a list, *i.e.* the returned map is obtained by replacing the keys in the arguments by consecutive integers starting from `0`.

`@log10` (numeric), *listable*: computes the value of the logarithm of its argument to base 10.

`@log2` (numeric), *listable*: computes the value of the logarithm of its argument to base 2.

`@log` (numeric), *listable*: computes the value of the natural logarithm of its argument.

`@make_duration_map` (numeric, numeric): `@make_duration_map(a, b)` returns a map where the integer i is associated to the duration (in beat) of the ith event of the score, for $\texttt{a} \leq i \leq \texttt{b}$.

`@make_score_map` (numeric, numeric): `@make_score_map(a, b)` returns a map where the integer i is associated to the position (in beat) of the ith event of the score, for $\texttt{a} \leq i \leq \texttt{b}$.

`@map` (f:function, t:tab) returns a tab such that element i is the result of applying `f` to element `t[`i`]`. Note that

```
@map(f, t) ≡ [f($x) | $x in t]
```

`@map_compose` (a:map, b:map): returns a map `c` such that `c(x) = b(a(x))`.

`@map_concat` (a:map, b:map): returns a map `c` which contains the (key, value) pairs of `a` and a pair (n, e) for each pair (k, v) in `b` with n ranging from `@size(a)` to `@size(a) + @size(b)`. If `a` and `b` are "vectors" (*i.e.* the range is an interval $[0, p]$ for some p), then `@map_compose` is the vector concatenation.

`@map_normalize` (map)

`@map_reverse` (map): `@map_reverse(m)` reurns a new map `p` such that `p(i) = m(sz - i)` with `sz = @size(m)`.

`@mapval` (map, function): `mapval(m, f)` return a map `p` such that `p(x) = f(m(x))`.

`@max_key` (map): returns the maximal element in the domain of the argument.

`@max_val` (tab *or* map): returns the maximal element in the tab if it is a map, the maximal element in the domain of the map if the argument is a map.

`@max` (*value*, *value*); return the maximum of its arguments. Values in *Antescofo* are totally ordered. The order between two elements of different type is implementation dependent. However, the order on numeric is as expected (numeric ordering, the integers are embedded in the decimals). For two argument of the same type, the ordering is as expected (lexicographic ordering for string, etc.).

`@member` (tab, *value*): returns true if the secon argument is an element of the first.

`@merge` (map, map): asymetric merge of the two argument maps. The result of `@merge(a, b)` is a map `c` such that `c(x) = a(x)` if `a(x)` is defined, and `b(x)` elsewhere.

`@min_key` (map): return the minimal element in the domain of its argument.

`@min_val` (tab *or* map): returns the minimal element present in the tab or the minimal element in the domain (if the argument is a map).

`@min` (*value*, *value*): returns the minimal elements in its arguments.

`@normalize` (tab, min:numeric, max:numeric): returns a new tab with the elements normalized between min and max. If they are omitted, they are assumed to be 0 and 1.

`@occurs` (tab, *value*): return the number of times the second argument is present as element of the first.

`@permute` (t:tab, n:numeric) returns a new tab which contains the `n`th permutations of the elements of `t`. They are factorial `s` permutations, where `s` is the size of `t`. The first permutation is numbered 0 and corresponds to the permutation which rearranges the elements of `t` in an array t_0 such that they are sorted increasingly. The tab t_0 is the smallest element amongst all tab that can be done by rearranging the element of `t`. The first permutation rearranges the elements of `t` in a tab t_1 such that $t_0 < t_1$ for the lexicographic order and such that any other permutation gives an array t_k lexicographicaly greater than t_0 and t_1. Etc. The last permutation (factorial `s` - 1) returns a tab where all elements of `t` are in decreasing order.

```
$t := [1, 2, 3]
@permute($t, 0) == [1, 2, 3]
@permute($t, 1) == [1, 3, 2]
@permute($t, 2) == [2, 1, 3]
@permute($t, 3) == [2, 3, 1]
@permute($t, 4) == [3, 1, 2]
@permute($t, 5) == [3, 2, 1]
```

`@pow` (numeric, numeric), *listable*: `@pow(x, y)` compute `x` raised to the power `y`.

`@push_back` (tab, *value*), *impure*: add the second argument at the end of the first argument. The first argument, modified by side-effect, is the returned value.

`@push_front` (tab, *value*), *impure*: add the second argument at the beginning of its first argument. The first argument, modified by side-effect, is the returned value.

`@rand_int` (int), *impure*: returns a random integer between 0 and its argument (excluded). This is not a pure function because two calls with the same argument are likely to return different results.

`@random` (), *impure*: returns a random number between 0 and 1 (included). The resolution of this random number generator is $\frac{1}{2^{31}-1}$, which means that the minimal step between two numbers in the images of this function is $\frac{1}{2^{31}-1}$. This is not a pure function because two successive calls are likely to return different results.

`@rand` (), *impure*: similar to `@random` but rely on a different algorithm to generate the random numbers.

`@reduce` (f:function, t:tab): if `t` is empty, an undefined value is returned. If `t` has only one element, this element is returned. In the other case, the binary operation `f` is used to combine all the elements in `t` into a single value `f(... f(f(t[0], t[1]), t[2]), ... t[n])`. For example, if `v` is a vector of booleans, `@reduce(@||, v)` returns the logical disjunction of the `t`'s elements.

`@remove` (t, n), *impure*: removes the element at index `n` in `t` (`t` is modified in place). Note that building a new tab by removing elements satisfying some property P is easy with a comprehension:

```
[ $x | $x in $t, !P ]
```

`@remove_duplicate` (t), *impure*: keep only one occurrence of each element in `t`. Elements not removed are kept in order and `t` is modified in place.

`@replace` (t:tab, find:value, rep:value) returns a new tab in which a number of elements have been replaced by another. The argument `find` represents a sub-sequence to be replaced: if it is not a tab, then all the occurrences of this value at the top-level of `t` are replaced by `rep`:

```
@replace([1, 2, 3, [2]], 2, 0]) ⟶ [1, 0, 3, [2]]
```

If `find` is a tab, then the replacement is done on sub-sequence of `t`:

```
@replace([1, 2, 3, 1, 2], [1, 2], 0) ⟶ [0, 3, 0]
```

Note that the replacement is done eagerly: the first occurrence found is replaced and the replacement continue on the rest of the tab. Thus, there is no ambiguity in case of overlapping sub-sequences, only the first is replaced:

```
@replace([1, 1, 1, 2], [1, 1], 0) ⟶ [0, 1, 2]
```

If the `rep` argument is a tab, then it represents at sub-sequence to be inserted in place of the occurrences of `find`. So, if the replacement is a tab, it must be wrapped into a tab:

```
@replace([1, 2, 3], 2, [4,5]) ⟶ [1, 4, 5, 3]
@replace([1, 2, 3], 2, [[4, 5]]) ⟶ [1, [4, 5], 3]
```

`@reshape` (t, s) builds an array of shape `s` with the element of tab `t`. These elements are taken circularly one after the other. For instance

```
@reshape([1, 2, 3, 4, 5, 6], [3, 2])
⟶ [ [1, 2], [3, 4], [5, 6] ]
```

`@resize` (tab, int), *impure*: resize its argument and returns the results. If the second argument is smaller that the size of the first argument, it effectively shrinks the first argument. If it is greater, undefined values are used to extend the tab.

`@reverse` (tab): return a tab where the element are given in the reverse order.

`@rnd_bernouilli` (p:float), *impure*. The members of the `@rnd_`*`distribution`* family return a random generator in the form of an impure function f taking no argument. Each time f is called, the value of a random variable following the *`distribution`* distribution is returned. The arguments of the `@rnd_`*`distribution`* are the parameters of the distribution. Two successive calls to `@rnd_`*`distribution`* returns two different random generators for the same distribution, that is, generators with unrelated seeds.

`@rnd_bernouilli`(p) returns a boolean random generator with a probability p to have a `true` value. For example

```
$bernouilli := @rnd_bernouilli(0.6)
$t := [ $bernouilli() | (1000) ]
```

produces a tab of 1000 random boolean values with a probability of 0.6 to be true.

`@rnd_binomial` (t:int, p:float) returns a random generator that produces integers according to a *binomial discrete distribution* P:

$$P(i, |t, p) = \binom{t}{i} p^i \, (1-p)^{t-i}, \quad i \geq 0.$$

`@rnd_exponential` (λ:float) returns a random generator that produces floats x according to an *exponential distribution* P:

$$P(x|\lambda) = \lambda e^{-\lambda x}, \quad x > 0.$$

`@rnd_gamma` (α:float): returns a random generator that produces floating-point values according to a *gamma distribution* P:

$$P(x|\alpha) = \frac{1}{\Gamma(\alpha)} \, x^{\alpha-1}, \quad x \geq 0.$$

`@rnd_geometric` (p:int) returns a random generator that produces integers following a *geometric discrete distribution*:

$$P(i|p) = p(1-p)^i, \quad ,i \geq 0.$$

`@rnd_normal` (μ:float, σ:float) returns a random generator that produces floating-point values according to a *normal distribution* P:

$$P(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

`@rnd_uniform_int` (a:int, b:int) returns a generator giving integer values according to a *uniform discrete distribution*:

$$P(i|a, b) = \frac{1}{b-a+1}, \quad a \leq i \leq b.$$

`@rnd_uniform_float` (a:int, b:int) returns a generator giving float values according to a *uniform distribution*:

$$P(x|a, b) = \frac{1}{b-a}, \quad a \leq x < b.$$

`@rotate` (t:tab, n:int) build a new array which contains the elements of `t` circularly shifted by `n`. If `n` is positive the element are right shifted, else they are left shifted.

```
@rotate([1, 2, 3, 4], -1) == [2, 3, 4, 1]
@rotate([1, 2, 3, 4], 1) == [4, 1, 2, 3]
```

`@round` (numeric), *listable*: returns the integral value nearest to its argument rounding half-way cases away from zero, regardless of the current rounding direction.

`@scan` (f:function, t:tab) returns the tab `[ t[0], f(t[0],t[1]), f(f(t[0], t[1]),t[2]), ...]`. For example, the tab of the partial sums of the integers between `0` (included) and `10` (excluded) is computed by the expression:

```
@scan(@+, [$x | $x in (10)])  ⟶  [0,1,3,6,10,15,21,28,36,45]
```

`@scramble` (t:tab) : returns a new tab where the elements of `t` have been scrambled. The argument is unchanged.

`@select_map`

`@shape` (t:value) returns 0 if `t` is not an array, and else returns a tab of integers each corresponding to the size of one of the dimensions of `t`. Notice that the elements of an array are homogeneous, *i.e.* they have all exactly the same dimension and the same shape.

`@shift_map`

`@sinh` (x:numeric), *listable*: computes the hyperbolic sine of `x`.

`@sin` (x:numeric), *listable*: computes the sine of `x` (measured in radians).

`@size` (x:*value*): if `x` is a scalar value, it return a strictly negative integer related to the type of the argument (that is, two scalar values of the same type gives the same result). If it is a map, an interpolated map or a tab, it returns the number of element in its argument (which is a positive integer).

`@sort` (t:tab) : sorts in-place the elements into ascending order using `<`.

`@sort` (t:tab, cmp:fct) sorts in-place the elements into ascending order. The elements are compared using the function `cmp`. This function must accept two elements of the tab `t` as arguments, and returns a value converted to bool. The value returned indicates whether the element passed as first argument is considered to go before the second.

`@sputter` (t:tab, p:float, n:numeric), *impure*: returns a new tab of length `n`. This tab is filled as follows: for each element, a random number between 0 and 1 is compared with `p` : if it is lower, then the element is the current element in `t`. If it is greater, we take the next element in `t` which becomes the current element. The process starts with the first element in `t`.

```
@sputter([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 0.5, 16)
⟶ [ 1, 1, 1, 1, 1, 2, 3, 4, 5, 6, 6, 6, 7, 8, 8, 9 ]
⟶ [ 1, 2, 3, 3, 4, 5, 6, 7, 8, 8, 9, 9, 9, 9, 10 ]
⟶ [ 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5 ]
```

`@sqrt` (x:numeric), *listable*: computes the non-negative square root of `x`.

`@stutter` (t:tab, n:numeric), *impure*: returns a new tab whose elements are repeated n times. The receiver is unchanged. The argument is unchanged.

```
@stutter([1, 2, 3, 4, 5, 6], 2)
⟶ [ 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 ]
```

`@tan` (x:numeric), *listable*: computes the tangent of `x` (measured in radians).

B Experimental Features

WARNING: in this section we sketch some experimental *features. The purpose is to have some feedback on them. However, notice that an experimental feature means that the implementation is in alpha version, the syntax and the functionalities may changes at any time and/or it can possibly be removed from future versions.*

B.1 Simple Curve *à la* `line~`

There is a simple form for curve where there is only a starting and an ending breakpoints:

```
Curve msg begin , end dur
```

corresponding to a curve sending the message *msg* x for x sampling a line from *begin* to *end* in *dur* times. The grain is `0.03s` if the duration is in absolute time and `0.2` beat if the duration is expressed in relative time.

A following

```
Curve msg stop dur
```

will start the sampling from the end value of the previous form to the specified *stop* value.

This behavior if similar to the `line~` object in Max or PD.

These two simple forms are internally expanded in a usual curve and make use of two global variables whose identifiers start by `$__` and contains the string *msg*. Such name must not be used to avoid interference.

B.2 Tracks

A *track* refers to all actions that have a label of some form and to the message whose head has some form. A track is defined using a `@track_def` statement:

```
@track_def track::T {
  print, "synth.*"
}
```

refers to all actions that: (1) have a label `print` or a label that matches `synth.*` (*i.e.* any name that starts with the prefix `synth`) *and* (2) all Max or PD messages whose receivers satisfy the same constraints *and* (3) the children of these actions (recursively).

More generally,

- a track definition is a list of token separated by a comma;

- a token is either a symbol (an identifier without double-quote) or a string;
- a symbol refers to labels or receivers equal to this symbol;
- a string denotes a regular expressions[18] (without the double quote) used to match a label or a receiver name;
- an action belongs to a track if there is a symbol in the track equals to the label of the action or if there is a regular expression that matches the label;
- in addition, a Max or PD message belongs to the track if the receivers name fulfills the same constraint;
- in addition, an action nested in a compound action belonging to the track, also belongs to the track;
- an action may belong to several tracks (or none);
- there is a maximum of 32 definable tracks.

Tracks can be muted or unmuted:

```
antescofo::mute track::T
antescofo::unmute track::T
```

A string can be also used for the name of the track:

```
antescofo::mute "track::T"
antescofo::unmute "track::T"
```

Track are muted/unmuted independently. An action is muted if it belongs to a track that is muted, else it is unmuted. A muted action has the same behavior of an unmuted action, *except* for messages: there arguments are evaluated as usual but the final shipping to Max or PD ins inhibited. It is important to note that muting/unmuting a track has no effect on the *Antescofo* internal computations.

For example, to inhibit the sending of all messages, one can defines the track:

```
@track_def track::all { ".*" }
```

and mute it:

```
antescofo::mute track::all
```

[18]The syntax used to define the regular expression follows the *posix* extended syntax as defined in IEEE Std 1003.2, see for instance http://en.wikipedia.org/wiki/Regular_expression

B.3 Abort Handler

An `abort` command can be used to stop a compound action before its natural end, cf. section 6.1.4. It is then often convenient to execute some dedicated actions at the premature end, actions that are not needed when the compound action reaches its natural end[19].

A direct implementation of this behavior is provided by `@abort` *handlers*. An abort handler is a group of actions triggered when a compound action is aborted. Abort handlers are specified using an `@abort` clause with a syntax similar to the syntax of the `@action` clause of a `curve`.

An `@abort` handler can be defined for all compound actions[20]. The scope of the handler is the scope introduced by the compound actions (if any): local variables introduced eventually by the compound action are accessible in the handler.

Wen an handler associated to a compound action `G` is spanned by an `@abort G` command, the handler cannot be killed by further `@abort G` command.

Notice that `@abort` commands are usually recursive, killing also the subgroups spanned by `G`. If these groups have themselves `@abort` handlers, they will be triggered when killing `G` but the order of activation is not specified and can differ from one execution to another.

Example. A good example is given by a curve that samples some parameter controlling the generation of a sound. On some event, the sound generation must be stopped, but this cannot be done abruptly: the parameter must go from its current value to some predetermined value, *e.g.* 0, in one beat. This is easily written:

```
Curve C
  @grain := 0.5
  @action := { print "curve:␣" $x }
  @abort := {
     print "Curve␣C␣aborted␣at␣" $x
     Curve AH
       @grain := 0.2
       @action := { print "handler␣curve:␣" $x }
     {
       $x { { $x } 1 { 0.0 } }
     }
  }
{
  $x { { 0.0 } 10 { 10.0 } 10 { 0.0 } }
}
```

[19] This effect can be achieved by wrapping the actions to perform in case of an `abort` in a `whenever` that watches a dedicated variable. The whenever is then triggered by setting this variable to the boolean value `true` immediately after the `abort` command. This approach becomes cumbersome when the actions launched on abort have to access variables that are local to the aborted action, when dealing with multiple compound actions and furthermore, scatter the code in several places.

These drawbacks motivates the introduction of *handlers*. Other kind of handlers are envisioned to handle other kind of exceptional conditions.

[20] with the current exception of `whenever`

When an `abort C` is emitted, the curve `C` is stopped and the actions associated to the `@abort` attribute are launched. These action spans a new curve `AH` with the same variable `$x`, starting from the current value of `$x` to `0.0` in one beat. A typical trace is (the `abort` command is issued at `1.5` beats):

```
print: curve:  0.
print: curve:  0.5
print: curve:  1.
print: curve:  1.5
print: Curve Aborted at  1.5
print: handler curve:  1.5
print: handler curve:  1.2
print: handler curve:  0.9
print: handler curve:  0.6
print: handler curve:  0.3
print: handler curve:  0.
```

B.4 Patterns

Patterns are a simple way to define complex logical conditions to be used in a `whenever`. A pattern is a sequence of *atomic patterns*. They are three kinds of atomic patterns: `Note`, `Event` and `State`.

Such a sequence can be used as the condition of a `whenever` to trigger some actions every time the pattern matches. It can represent a *neume*, that is a melodic schema defining a general shape but not necessarily the exact notes or rhythms involved. It can also be used in broader contexts involving not only the pitch detected by the *Antescofo* listening machine, but also arbitrary variables.

Warning: The notion of pattern used here is very specific and the recognition algorithm departs from the recognition achieved by the listening machine. Patterns defines an exact variation in time of variables while the listening machine recognizes the most probable variation from a given dictionary of musical events. The latter relies on probabilistic methods. The former relies on algorithms like those used for recognizing regular expressions. So the pattern matching available here is not relevant for the audio signal, even if it can have some applications[21].

B.4.1 `Note`: Patterns on Score

The basic idea is to react to the recognition of a musical phrase defined in a manner similar to event's specification. For example, the statement:

[21] Note also that the pattern matching is running asynchronously on variables supposed to be updated at most at the rate of control.

```
@pattern_def pattern::P
{
   Note C4 0.5
   Note D4 1.0
}
```

defines a `pattern::P` that can be used later in a `whenever`:

```
whenever pattern::P
{
   print "found␣pattern␣P"
}
```

The `Note` pattern is an *atomic pattern* and the `@pattern_def` defines and gives a name to a sequence of atomic patterns.

In the current version, the only event recognized are `Note`: `Trill`, `Chord`, etc., cannot be used (see however the other kinds of atomic patterns below). Contrary to the notes in the score, the duration may be omitted to specify that any duration is acceptable.

Pattern Variables. To be more flexible, patterns can be specified using local variables that act as wildcards:

```
@pattern_def pattern::Q
{
  @Local $h

  Note $h
  Note $h
}
```

This pattern defines a repetition of two notes of the same pitch (and their respective duration do not matter). The wildcard, or *pattern variable* `$h`, is specified in the `@Local` clause at the beginning of the pattern definition. Every occurrence of a pattern variable must refer to the same value. Here, this value is the pitch of the detected note (given in midicents).

Pattern variables are really local variables and their scope extends to the body of the `whenever`s that use that uses this pattern. So they can be used to parametrize the actions to be triggered. For example:

```
whenever pattern::Q
{
  print "detection␣of␣the␣repetition␣of␣pitch␣" $h
}
```

Specifying Duration. A pattern variable can also be used to constraint durations in the same manner. The value of a duration is the value given in the score (and *not* the actual duration played by the musician).

Specifying Additional Constraints. The pitch of a pattern Note can be an integer or a ratio of two integers (bot corresponding to midicents); a symbolic midi pitch; a pattern variable or a variable. The duration of a pattern Note can be an integer, a pattern variable or a variable. For example,

```
@pattern_def pattern::R
{
  Note $X
  Note C4 $Y
}
```

specifies a sequence of two notes, the first one must have a pitch equal to the value of the variable $X (at the time where the pattern is checked) and the pitch of the second one is C4, and the duration of the first is irrelevant while the duration of the second must be equal to the value of $Y (as for $X, this variable is updated elsewhere and the value considered is its value at the time where the pattern is checked).

An additional where clause can be used to give finer constraints:

```
@pattern_def pattern::R
{
  @Local $h, $dur1, $dur2

  Note $h $dur1 where $h > 6300
  Note $h $dur2 where $dur2 < $dur1
}
```

pattern::R specifies a sequence of two successive notes such that:

- their pitch is equal and this value in midicents is the value of the local variable $h;
- $h is higher than 6300 midicents;
- and the duration of the second note must be lower than the duration of the first note.

Pattern Causality. In a where clause, all variables used must have been set before. For example, it is not possible to refer to $dur2 in the where clause of the first note: the pattern recognition is *causal* which means that the sequence of pattern is recognized "on-line" in time from the first to the last without guessing the future.

A Complete Exemple. It is possible to refer in the various clause of a pattern to variables (or expression for the where clause) computed elsewhere. For example

```
@pattern_def pattern::M
{
  @Local $h, $dur

  Note $X $dur
  Note $h $dur where $dur > $Y
```

```
    Note C4
}
```

defines a sequence of 3 notes. The first note has a pitch equal to `$X` (at the moment where the pattern is checked); the second note as an unknown pitch referred by `$h` and its duration `$dur`, which is the same as the duration of the first note, must be greater than the current value of `$Y`; and finally, the third note as a pitch equal to `C4`.

B.4.2 `Event` on Arbitrary Variables

From the listening machine perspective, a `Note` is a complex event to detect in the input audio stream. But from the pattern matching perspective, a `Note` is an atomic event that can be detected looking only on the system variables `$PITCH` and `$DURATION` managed by the listening machine.

It is then natural to extend the pattern-matching mechanism to look after any *Antescofo* variable. This generalization from `$PITCH` to any variable is achieved using the `Event` pattern:

```
@pattern_def pattern::Gong
{
    @Local $x, $y, $s, $z

    Event $S value $x
    Event $S value $y at $s where $s > 110
    Before [4]
      Event $S value $z where [$x < $z < $y]
}
```

The keyword `Event` is used here to specify that the event we are looking for is an *update* in the value of the variable `$S`[22]. We say that `$S` is the *watched variable* of the pattern.

An `Event` pattern is another kind of atomic pattern. `Note` and `Event` patterns can be freely mixed in a `@pattern_def` definition.

Four optional clauses can be used to constraint an `event` pattern:

1. The `before` clause is used to specify a temporal scope for looking the pattern.
2. The `value` clause is used to give a name or to constraint the value of the variable specified in the `Event` at matching time.
3. The `at` clause can be used to refer elsewhere to the time at which the pattern occurs.
4. The `where` clause can be used to specify additional logical constraint.

[22] A variable may be updated while keeping the same value, as for instance when evaluating `$S := $S`. Why `$S` is updated or what it represents does not matter here. For example, `$S` can be the result of some computation in *Antescofo* to record a rhythmic structure. Or `$S` is computed in the environment using a pitch detector or a gesture follower and its value is notified to *Antescofo* using a `set_var` message.

The `before` clause must be given before the `Event` keyword. The last three clauses can be given in any order after the specification of the watched variable.

Contrary to the `Note` pattern, there is no “duration” clause because an event is point wise in time: it detects the update of a variable, which is instantaneous.

The `value` Clause. The `value` clause used in an `event` is more general that the the `value` clause in a `note` pattern: it accepts a pattern variable or an arbitrary expression. An arbirary expression specify that the value of the watched variable must be equal to the value of this expression. A pattern variable is bound to the value of the watched variable. This pattern variable can be used elsewhere in the pattern.

Note that to both bind the pitch or the duration of a note to a pattern variable and to constraint its value, you need to use a where clause. If you do not need to bind the pitch or the duration of a note, you can put the expression defining its expected value directly in the right place.

The `at` Clause. An `at` clause is used to bind a local variable to the value of the `$NOW` variable when the match occurs. This variable can then be used in a `where` clause, *e.g.* to assert some properties about the time elapsed between two events or in the body of the `whenever`.

Contrary to a `value` clause, it is not possible to specify directly a value for the `at` clause but this value can be tested in the `where` clause:

```
@pattern_def pattern::S
{
  @Local $s, $x, $y

  Event $S at $s where $s==5   ; cannot be written: Event $S at 5
  Event $S at $x
  Event $S at $y where ($y - $x) < 2
}
```

Note that it is very unluckily that the matching time of a pattern is exactly “5”. Notice also that the `at` date is expressed in absolute time.

The `where` Clause. As for `Note` patterns, a `where` clause is used to constraint the parameters of an event (value and occurrence time). It can also be used to check a “parallel property”, that is, a property that must hold at the time of matching. For example: in the `where` clause:

```
@pattern_def pattern::S
{
  Event $S where $ok
}
```

will match an update of `$S` only when `$ok` is true.

The `before` Clause. For a pattern p that follows another pattern, the `before` clause is used to relax the temporal scope on which *Antescofo* looks to match p.

When *Antescofo* is looking to match the pattern $p =$ `Event $X...` it starts to watch the variable `$X` right after the match of the previous pattern. Then, at the *first value change* of `$X`, *Antescofo* check the various constraints on p. If the constraints are not meet, the matching fails. The `before` clause can be used to shrink or to extend the temporal interval on which the pattern is matched beyond the first value change. For instance, the pattern

```
@pattern_def pattern::twice[$x]
{
    Event $V value $x
    Before [3s] Event $V value $x
}
```

is looking for two updates of variable `$V` for the same value `$x` in less than 3 seconds. *Nota bene* that other updates for other values may occurs but `$V` must be updated for the same value before 3 seconds have elapsed for the pattern to match.

If we replace the temporal scope `[3s]` by a logical count `[3#]`, we are looking for an update for the same value that occurs in the next 3 updates of the watched variable. The temporal scope can also be specified in relative time.

When the temporal scope of a pattern is extended beyond the first value change, it is possible that several updates occurring within the temporal scope satisfy the various patterns's constraints[23]. *However* the *Antescofo* pattern matching stops looking for further occurrences in the same temporal scope, after having found the first one. This behavior is called the *single match* property.

For instance, if the variable `$V` takes the same value three times within 3 seconds, say at the dates $t_1 < t_2 < t_3$, then `pattern::twice` occurs three times as (t_1, t_2), (t_1, t_3), and (t_2, t_3). Because *Antescofo* stops to look for further occurrences when a match starting at a given date is found, only the two matches (t_1, t_2) and (t_2, t_3) are reported.

Finally, notice that the temporal scope defined on an event starts with the preceding event. So a `before` clause on the first `Event` of a pattern sequence is meaningless and actually forbidden by the syntax.

Watching Multiple Variables Simultaneously. It is possible to watch several variables simultaneously: the event occurs when one of the watched variable is updated (and if the constraints are fulfilled). For instance:

```
@pattern_def pattern::T
{
  @Local $s1, $s2

  Event $X, $Y at $s1
  Event $X, $Y at $s2 where ($s2 - $s1) < 1
```

[23]If there is no `before` clause, the temporal scope is "the first value change" which implies that there is *at most* one match.

```
}
```

is a pattern looking for two successive updates of either `$X` or `$Y` in less than one second.

Notice that when watching multiple variables, it is not possible to use a `value` clause.

A Complex Example. As mentioned, it is possible to freely mix `Note` and `Event` patterns, for example to watch some variables after the occurrence of a musical event:

```
@pattern_def pattern::T
{
  @Local $d, $s1, $s2, $z

  Note D4 $d
  Before [2.5] Event $X, $Y at $s1
  Event $Z value $z at $s2 where ($z > $d)  $d > ($s2 - $s1)
}
```

Note that different variables are watched after the occurrence of a note `D4` (6400 midicents). This pattern is waiting for an assignment to variable `$X` or `$Y` in an interval of `2.5` beats after a note `D4`, followed by a change in variable `$Z` for a value `$s` such that the duration of the `D4` is greater also the interval between the changes in `$X` (or `$Y`) and `$Z` and such that the value `$z` is greater than this interval.

B.4.3 `State` Patterns

The `Event` pattern corresponds to a logic of *signal*: each variable update is meaningful and a property is checked on *a given point in time*. This contrasts with a logic of *state* where a property is looked *on an interval of time*. The `State` pattern can be used to face such case.

A Motivating Example. Suppose we want to trigger an action when a variable `$X` takes a given value v for at least 2 beats. The following pattern:

```
Event $X value v
```

does not work because the constraint "at least 2 beats" is not taken into account. The pattern matches every times `$X` takes the value `v`.

The pattern sequence

```
@Local $start, $stop
Event $X value v at $start
Event $X value v at $stop where ($stop - $start) >= 2
```

is not better: it matches two successive updates of `$X` that span over 2 seconds. Converting the absolute duration in relative time is difficult because it would imply to track all the tempo changes in the interval. More importantly, it would not match three consecutive updates of `$X` for the same value v, one at each beat, a configuration that should be recognized.

This example shows that is not an easy task to translate the specification of a state that lasts over an interval into a sequence of instantaneous events. This is why, a new kind of atomic pattern to match states has been introduced. Using a `State` pattern, the specification of the previous problem is easy:

```
State $X where $X == v during 2
```

matches an interval of 2 beats where the variable `$X` constantly has the value v (irrespectively of the variable updates).

Four optional clauses can be used to constraint a `state` pattern:

1. The `before` clause is used to specify a temporal scope for looking the pattern.
2. The `start` clause can be used to refer elsewhere to the time at which the matching of the pattern has started.
3. The `stop` clause can be used to refer elsewhere to the time at which the matching of the pattern stops.
4. The `where` clause can be used to specify additional logical constraint.
5. The `during` clause can be used to specify the duration of the state.

The `before` clause must be given before the `state` keyword. The others can be given in any order after the specification of the watched variable. There is no `value` clause because the value of the watched variable may change during the matching of the pattern, for instance when the state is defined as "being above some threshold".

The first three clauses are similar to those described for an `event` pattern, except that the `at` is split into the `start` and the `stop` clauses because here the pattern is not "point wise" but spans an interval of time.

The initiation of a `state` Pattern. Contrary to `note` and `event`, the `state` pattern is not driven solely by the updates of the watched variables. So the matching of a `state` is initiated immediately after the end of the previous matching.

The `during` Clause. The optional `during` clause is used to specify the time interval on which the various constraints of the pattern must hold. If this clause is not provided, the `state` finishes to match as soon as the constraint becomes false. Figure 13 illustrates the behavior of the pattern

```
@Refractory r

State $X during ℓ where $X > a
Before [d]
  State $X where $X > b
```

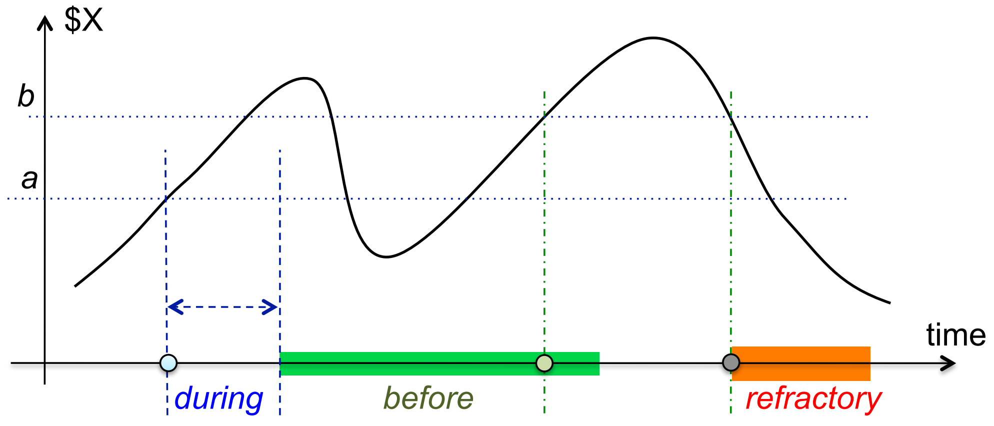


Figure 13: *State patterns with* `during`*,* `before` *and* `@refractory` *clauses.*

The schema assumes that variable `$X` is sampling a continuous variation.

The first `state` pattern is looking for an interval of length ℓ where constantly `$X` is greater than a.

The second `state` pattern must start to match before d beats have elapsed since the end of the previous pattern (the allowed time zone is in green). The match starts as soon as `$X` is greater than b.

There is no specification of a duration, so the second pattern finishes its matching as soon as `$X` becomes smaller than b.

With the sketched curve, there are many other possible matches corresponding to delaying in time the start of the first `state` while still maintaining `$X` $> b$. Because the start time of these matches are all different, they are not ruled out by the *single match* property. A `@refractory` period is used to restrict the number of successful (reported) matches.

B.4.4 Limiting the Number of Matches of a Pattern

A `@Refractory` clause specify the period after a successful match during which no other matches may occur. This period is counted starting from the end of the successful match. The refractory period is represented in red in Figure 13. The net effect of a `@refractory` period is to restrict the number of matching per time interval.

The refractory period is defined for a pattern sequence, not for an atomic pattern. The `@refractory` clause must be specified at the beginning of the pattern sequence just before or after an eventual `@Local` clause. The period is given in absolute time.

B.4.5 Pattern Compilation

Patterns are not a core feature of the *Antescofo* language: internally they are compiled in a nest of `whenever`, conditionals and local variables. If verbosity is greater than zero, the `printfwd` commands reveals the result of the pattern compilation in the printed score.

Two properties of the generated code must be kept in mind:

1. *Causality:* The pattern compiler assumes that the various constraints expressed in a pattern are free of side-effect and the pattern matching is achieved on-line, that is, sequentially in time and without assumption about the future.

2. *Single match property:* When a pattern sequence occurs several times starting at the same time t, only one pattern occurrence is reported[24].

[24] Alternatives behaviors may be considered in the future.

B.5 Reserved Experimental Keywords

. . .

Some keywords are reserved for current and future experimentations. You cannot use these keyword as function names or symbol: `@ante`, `at`, `antescofo::mute`, `antescofo::unmute`, `@dsp_channel`, `@dsp_inlet`, `@dsp_outlet`, `@faust_def`, `patch`, `@pattern_def`, `pattern::`, `@post`, `@refractory`, `start`, `stop`, `@target`, `@track_def`, `track::`, `where`.

C Changes in 0.51

With respect to the previous official release:

- the listening machine has been improved in many ways;
- new syntax for groups and loops;
- delays and tempi can be defined by an expression;
- expressions have been extended with conditionals, global and local variables, histories, recursive functions, etc.;
- the types of values managed by *Antescofo* has been greatly extended to include: boolean, integer, float, string, symbol, function, process, vector, dictionary (map) and interpolated function;
- extended and new atomic actions (assert, abort, erase of an action in a group, process call, assignment);
- new compound actions have been introduced:
  - curve,
  - whenever,
  - conditionals,
  - processes,
  - parallel iterations;
- the new synchronization strategies (loose and tight) and the new error handling strategies (local or global) can be used uniformly on each compound actions;
- improved error messages including the location in the score file;
- management of OSC communications;
- output to a file;
- trace, communication with Notability, integration with Ascograph;
- internalization of Max or PD messages as *Antescofo* atomic actions;
- standalone (offline) and 64 bit version (Mac, Linux).

D Index

— Symbols —

@* 85
* 48
@- 85
- 48
@/ 85
/ 48
: 64
@< 86
< 48
@<= 86
<= 48
= 48
@== 86
> 48
@>= 86
>= 48
? 64
@ 48
@% 86
% 48
@&& 86
Antescofo
  bug report 1, 3
  discussion group 84
  scientific publications 1, 3
  web page 1, 3
$-identifiers 10
::-identifiers 10
@-identifiers 9

— A —

abort 27
  flat 28
  handler 98
  processes 27
  recursive 28
abort 9
@abs 49, 86
@acos 49, 86
action 4, 6, 20, 23, 33
  abort 27
  active (compound action) 27
  cancel 28
  child 33
  compound 33
  conditional 36
  external 23
  father 33
  fired 20
  triggered 20
@action 10
action 9
actions command 30
@add_pair 51, 86
aggregate values 47
analysis command 30
ANTEIOI 81
antescofo-mess1 81
antescofo::actions 30
antescofo::analysis 30
antescofo::before_nextlabel 30
antescofo::bpmtolerance 30
antescofo::calibrate 30
antescofo::clear 30
antescofo::gamma 30
antescofo::getcues 30
antescofo::getlabels 30
antescofo::gotobeat 30
antescofo::gotocue 30
antescofo::gotolabel 30
antescofo::harmlist 30
antescofo::info 30
antescofo::jumptocue 30
antescofo::jumptolabel 30
antescofo::killall 30
antescofo::mode 30
antescofo::mute 97
antescofo::mute 30
antescofo::nextaction 30
antescofo::nextevent 30
antescofo::nextfwd 30
antescofo::nextlabel 30
antescofo::nofharm 30
antescofo::normin 30
antescofo::obsexp 31
antescofo::pedal 31
antescofo::pedalcoeff 31
antescofo::pedaltime 31
antescofo::piano 31
antescofo::play 31
antescofo::playfrom 31
antescofo::playfrombeat 31
antescofo::preload 31
antescofo::preventzigzag 31
antescofo::previousevent 31
antescofo::previouslabel 31

antescofo::printfwd 31
antescofo::printscore 31
antescofo::read 31
antescofo::report 31
antescofo::score 31
antescofo::setvar 31
antescofo::start 31
antescofo::stop 31
antescofo::suivi 31
antescofo::tempo 30
antescofo::tempoinit 31
antescofo::temposmoothness 31
antescofo::tune 31
antescofo::unmute 97
antescofo::unmute 31
antescofo::variance 32
antescofo::verbosity 32
antescofo::verify 32
antescofo::version 32
application 8
@approx 86
arithmetic operators 48
array 55
Ascograph
  monitoring 79
  score conversion 78
  score edition 78
Ascograph
  curve edition 38
@asin 49, 87
assert 9
assignment 26
at 103
at 9
@atan 49, 87
atomic values 47
auto-delimited expression 65

— B —

"back" interpolation 40
"back_in" interpolation 40
"back_in_out" interpolation 40
"back_out" interpolation 40
backslash 24, 49
\ 24
BEATNUM 81
$BEAT_POS 63
before 104
before 9
before_nextlabel command 30
@between 87
bind 6
bind 9
boolean value 48
"bounce" interpolation 40
"bounce_in" interpolation 40
"bounce_in_out" interpolation 40
"bounce_out" interpolation 40
@bounded_integrate 54, 87
@bounded_integrate_inv 87
BPM 7
bpm 6, 9
bpmtolerance command 30
breakpoint function 38

— C —

calibrate command 30
@car 57, 87
@car 59
carriage-return 49
causal score 45, 63
causality 44
@cdr 57, 87
@cdr 59
@ceil 49, 87
CERTAINTY 81
CFWD 83
cfwd 9
childs of a compound action 33
CHORD 4, 7, 12
chord 9, 13
"circ" interpolation 40
"circ_in" interpolation 40
"circ_in_out" interpolation 40
"circ_out" interpolation 40
@clear 52, 57, 87
clear command 30
closefile 9
@coef 10
column 10
@command 23
comments 10
comparison 48
@compose_map 52
compound action 33
  abort 34
  instance 34
  premature end 34
compound values 47
comprehension 55
  predicate 56
@concat 57, 59, 87

conditional
  action . . . 36
  expression . . . 64
conjunction . . . 48
@cons . . . 57, 87
@cons . . . 59
containers . . . 33
@copy . . . 87
@cos . . . 49, 87
@cosh . . . 49, 87
@count . . . 57, 88
"cubic" interpolation . . . 40
"cubic_in" interpolation . . . 40
"cubic_in_out" interpolation . . . 40
"cubic_out" interpolation . . . 40
curve
  simple syntax . . . 96
curve . . . 38
curve . . . 9
  graphical edition with *Ascograph* . . . 38
curve
  and NIM . . . 42

— D —

@date . . . 10
dated access . . . 60, 62
@defined . . . 51
delay . . . 20
  absolute . . . 21
  beat . . . 21
  relative . . . 21, 34
  second . . . 21
  zero . . . 21
dictionary . . . 50
@dim . . . 57, 88
@dim . . . 55, 94
dimension . . . 55, 94
disjunction . . . 48
do . . . 9
domain . . . 50
duration . . . 12
$DURATION . . . 63
during . . . 35, 106
during . . . 9

— E —

"elastic" interpolation . . . 40
"elastic_in" interpolation . . . 40
"elastic_in_out" interpolation . . . 40
"elastic_out" interpolation . . . 40
if . . . 64
else . . . 9, 36
end of line . . . 10, 24, 32, 49
ENDBANG . . . 81
error (in predefined functions) . . . 47
error handling strategy . . . 68
EVENT . . . 12
event . . . 4, 6, 12
  attributes . . . 14
  parameter . . . 12
event . . . 9
exec value . . . 50
@exp . . . 49, 88
"exp" interpolation . . . 40
"exp_in" interpolation . . . 40
"exp_in_out" interpolation . . . 40
"exp_out" interpolation . . . 40
expr . . . 9
expression
  auto-delimited . . . 65
  conditional . . . 64
external actions . . . 23

— F —

false . . . 9
father . . . 33
file . . . 29
@find_index . . . 57, 88
@flatten . . . 57, 88
float value . . . 48
@floor . . . 49, 88
forall . . . 37
forall . . . 9
forum . . . 84
frames of reference . . . 18
@fun_def . . . 10
function . . . 49, 50, 53
  application . . . 8
  definition . . . 6
  domain . . . 50
  extensional definition . . . 50
  intentional definition . . . 49, 50
  interpolated map . . . 53
  listable . . . 85
  map . . . 50
  predefined . . . 49
  pure . . . 85
  side-effect . . . 85

— G —

gamma command 30
getcues command 30
getlabels command 30
GFWD 83
gfwd 9
@global 10, 34
global 9
gotobeat command 30
gotocue command 30
gotolabel command 30
@grain 10
group 4, 9
@gshift_map 51, 88
@guard 10
guard 9

— H —

handler 98
harmlist command 30
hierarchy
  of actions 28, 33
  OSC names 25
history
  as a map 53
@history_map($x) 60
@history_map_date($x) 61
@history_map_rdate($x) 61
@hook 10, 14
hook 9, 14
HZ 80
HZ 80

— I —

identifier
  clash with keywords 9
if 64
if 9, 36
imap 9
@immediate 10
import
  midi file 78
  MusicXML file 78
in 9
indentation 10
info command 30
inlet
  HZ 80
  KL 80
  MIDI 80
inlet 6
@inlet x 6
@insert 10, 57, 88
instances of a group 34
integer value 48
@integrate 54, 88
internal command ..... *see* antescofo::*xxx*
interpolated map value 53
interpolation
  linear 39
  see also curve 39
  step function 39
  type 40
  user defined 40
IO 29
@iota 89
@is_bool 47, 89
@is_defined 89
@is_exec 47
@is_fct 47, 89
@is_float 47, 89
@is_function 47, 89
@is_int 47, 89
@is_integer_indexed 51, 89
@is_interpolatedmap 47, 89
@is_list 51, 89
@is_map 47, 89
@is_nim 47
@is_numeric 47, 89
@is_proc 47
@is_string 47, 89
@is_symbol 47, 89
@is_tab 47
@is_undef 47, 89
@is_vector 51, 89
iteration ..... *see* loop, *see* forall

— J —

@jump 10, 14
jump 9, 14
jumptocue command 30
jumptolabel command 30

— K —

KILL 83
kill 27
kill 9
KILL OF 83
killall command 30
KL 80
KL 80

— L —

@label 10
@lace 57, 89
$LAST_EVENT_LABEL 63
let 9, 26
LFWD 83
lfwd 9
@lid 10
"linear" interpolation 40
list
  as tab 59
listable 56, 85
@listify 52, 90
@local 10, 34
@Local 100
@log 49, 90
@log10 .... 49, 90
@log2 .... 49, 90
logical and 48
logical instant 17
logical operator, lazy 48
logical or 48
loop 36
loop 9
@loose 34
loose 9

— M —

macro
  application 8
  definition 8
macro definition 6
@macro_def 10
Main outlet 80
@make_duration_map 90
@make_duration_map(start, stop) 53
@make_score_map 90
@make_score_map(start, stop) 53
@map 58, 90
map
  arithmetic extension 52
  as an extensional function 50, 51
  construction 50, 51
  domain 51
  history representation 53
  iteration 37
  range 51
  score representation 53
  value 50
map 9
@map_compose 90
@map_concat 90
@map_history 10
@map_history_date 10
@map_history_rdate 10
@map_normalize 90
@map_reverse 52, 90
@map_val 51
@mapval 90
matrix 55
matrix computations 56
MAX 23
  message 23
  symbol 24
@max 48, 49, 90
@max_val 58
@max_key 51, 90
@max_val 51, 90
@member 58, 91
@merge 51, 91
MIDI 80
MIDI 80
midi file (import) 78
MIDIOUT 81
@min 48, 49, 91
@min_val 58
@min_key 51, 91
@min_val 51, 91
MISSED 81
mode command 30
model of time 16
@modulate 10
ms 9, 21
MULTI 12, 14
Multi 13, 14
multi 9
multi 13
multi_list 12, 13
multi_list 13
MusicXML file (import) 78
mute 97
mute command 30

— N —

@name 10
napro_trace 6, 9
negation 48
nesting groups 68
next event 12
nextaction command 30
nextevent command 30
nextfwd command 30
nextlabel command 30
NIM 53
  continuous 53
  discontinuous 54
nofharm command 30
@norec 10
@normalize 58, 91
normin command 30
NOTE 4, 7, 12, 13
note 99
note 9
NOTENUM 81
$NOW 63

— O —

obsexp command 31
@occurs 58, 91
of 9
off 9
offline 81
on 9
open 29
openoutfile 29
openoutfile 9
operator
  listable 56
OSC 24
oscoff 9
oscon 9
oscrecv 9
OSCSEND 25
oscsend 9
outlet 80
  ANTEIOI 81
  BEATNUM 81
  CERTAINTY 81
  ENDBANG 81
  score label 80
  Main outlet 80
  MIDIOUT 81
  MISSED 81
  NOTENUM 81
  SCORETEMPO 81
  TDIST 81
  tempo 80
  TRACE 81
  VELOCITY 81

— P —

parameter clause 38
parfor 9
pattern 99
  @Local 100
  @refractory 107
  at 103
  atomic 100
  before 104
  causality 101, 108
  duration 100
  during 106
  event 102
  on arbitrary variables 102
  score 99
  sequence 100
  single match property 104, 107, 108
  start 106
  state 105
  stop 106
  temporal scope 104
  value 103
  variables 100
  watched variables 102, 104
  where 101, 103
pattern 9
pattern definition 6
@pattern_def 10
PD 23
  message 23
  symbol 24
pedal command 31
pedalcoeff command 31
pedaltime command 31
@permute 58, 91
physical time 18
piano command 31
pitch 13
pitch 12
pitch 12
$PITCH 63
pitch_list 12, 13
pitch_list 13
play command 31
playfrom command 31
playfrombeat command 31

port . . . 9
@pow . . . 49, 91
preload command . . . 31
preventzigzag command . . . 31
previousevent command . . . 31
previouslabel command . . . 31
printfwd command . . . 31
printscore command . . . 31
proc value . . . 50
@proc_def . . . 10
process . . . 71
  abort . . . 27
  abort . . . 73
  application . . . 8
  as value . . . 72
  definition . . . 8
  iteration . . . 37
  recursive . . . 72
process definition . . . 6
processus . . . 50
Pure Data . . . *see* PD
@push_back . . . 58
@push_front . . . 58
@push_back . . . 91
@push_front . . . 91

— Q —

"quad" interpolation . . . 40
"quad_in" interpolation . . . 40
"quad_in_out" interpolation . . . 40
"quad_out" interpolation . . . 40
"quart" interpolation . . . 40
"quart_in" interpolation . . . 40
"quart_in_out" interpolation . . . 40
"quart_out" interpolation . . . 40
"quint" interpolation . . . 40
"quint_in" interpolation . . . 40
"quint_in_out" interpolation . . . 40
"quint_out" interpolation . . . 40

— R —

@rand . . . 49, 92
@rand_int . . . 92
@random . . . 92
random number . . . 49
@rank . . . 58
@rdate . . . 10
reactive system . . . 16
read command . . . 31
@reduce . . . 58, 92
@refractory . . . 107
relational operators . . . 47, 48
relative time . . . 18, 19
@remove . . . 58, 92
@remove_duplicate . . . 58, 92
@replace . . . 58, 92
report command . . . 31
@reshape . . . 58, 93
@resize . . . 59, 93
@reverse . . . 59, 93
@rnd_bernouilli . . . 93
@rnd_binomial . . . 93
@rnd_exponential . . . 93
@rnd_gamma . . . 93
@rnd_geometric . . . 94
@rnd_normal . . . 94
@rnd_uniform_float . . . 94
@rnd_uniform_int . . . 94
$RNOW . . . 63
@rotate . . . 59, 94
@round . . . 94
$RT_TEMPO . . . 63

— S —

s . . . 9, 21
scalar . . . 47
scalar product . . . 56
@scan . . . 59, 94
score
  as a map . . . 53
  pattern . . . 99
score command . . . 31
score label . . . 80
SCORETEMPO . . . 81
@scramble . . . 59, 94
@select_map . . . 51, 94
set_var . . . 102
setvar command . . . 31
shape . . . 94
@shape . . . 94
@shape . . . 55
@shift_map . . . 51, 95
simple identifier . . . 9
@sin . . . 49, 95
"sine" interpolation . . . 40
"sine_in" interpolation . . . 40
"sine_in_out" interpolation . . . 40
"sine_out" interpolation . . . 40
@sinh . . . 49, 95

size
- non atomic value . . . 64
- scalar value . . . 64
- undefined value . . . 64

@size . . . 59, 64, 95
@sort . . . 59, 95
@sputter . . . 59, 95
@sqrt . . . 49, 95
[] . . . 54
standalone . . . 81
start . . . 106
start . . . 9
start command . . . 31
state . . . 105
statement . . . 6
stop . . . 106
stop . . . 9
stop command . . . 31
strategy
- error handling . . . 68
- synchronization . . . 66

string value . . . 49
@stutter . . . 59, 95
suivi command . . . 31
symb . . . 9
symbols . . . 9
synchronization strategy . . . 21, 66
system variables
- assignment . . . 27
- $BEAT_POS . . . 63
- $DURATION . . . 63
- $LAST_EVENT_LABEL . . . 63
- $NOW . . . 63
- $PITCH . . . 63
- $RNOW . . . 63
- $RT_TEMPO . . . 63

— T —

tab . . . 54
- comprehension . . . 55
- extension . . . 55

tab
- and list . . . 59
- iteration . . . 37

tab . . . 9
tab value . . . 54
tabulation . . . 49
@tan . . . 49, 95
TDIST . . . 81
tempo . . . 18
- *the* . . . 18
- local . . . 18
- tracking . . . 18

@tempo . . . 10, 34
tempo . . . 80
tempo command . . . 30
tempo off . . . 6
tempo on . . . 6
tempoint command . . . 31
temporal clause . . . 35
temporal coordinate systems . . . 16
temporal scope . . . 35
temporal shortcuts . . . 45, 63
temporized system . . . 16
temposmoothness command . . . 31
the tempo . . . 18
@tight . . . 10, 34
time
- absolute . . . 18
- logical instant . . . 17
- model of . . . 16
- physical . . . 18
- relative . . . 18, 34
- shortcuts . . . 45, 63
- wall clock . . . 18

time frame . . . 16
time frames . . . 18
TRACE . . . 81
track . . . 96
- definition . . . 6
- mute . . . 97
- unmute . . . 97

@track_def . . . 10
transpose . . . 9
TRILL . . . 4, 12, 14
Trill . . . 13, 14
trill . . . 9
trill_list . . . 12, 13
trill_list . . . 13
true . . . 9
tune command . . . 31
@type . . . 10
type of a value . . . 47

— U —

@uid . . . 10
undefined value . . . 48
unmute . . . 97
unmute command . . . 31
until . . . 35
until . . . 9
UTF-8 . . . 9

— V —

value
  atomic . . . 47
  boolean . . . 48
  exec . . . 50
  float . . . 48
  integer . . . 48
  interpolated map . . . 53
  map . . . 50
  non atomic . . . 47
  ordering . . . 90
  proc . . . 50
  scalar . . . 47
  string . . . 49
  tab . . . 54
  types of . . . 47
  undefined . . . 26, 48
value . . . 103
value . . . 9
variable
  assignment . . . 60
  dated access . . . 60
  declaration . . . 61
  global . . . 62
  history . . . 60, 62
  lifetime . . . 62
  local . . . 61
  local to a pattern . . . 100
  scope . . . 61
  stream of values . . . 60
  system . . . 27, 62
  watched . . . 102
variance . . . 6, 9
variance command . . . 32
VELOCITY . . . 81
verbosity command . . . 32
verify command . . . 32
version command . . . 32

— W —

wall clock time . . . 18
whenever . . . 42
  during . . . 44
  pattern . . . 99
  temporal scope . . . 44
  temporal shortcuts . . . 44
  until . . . 44
whenever . . . 9
where . . . 101, 103
where . . . 9
while . . . 35
while . . . 9
writing in a file . . . 29

E Detailed Table of Contents

How to use this document 3

Brief history of Antescofo 3

1 Understanding *Antescofo* scores 4
- 1.1 Structure of an *Antescofo* Score . . . 4
- 1.2 Elements of an *Antescofo* Score . . . 6
  - Function Definition. . . . 7
  - Process Definition. . . . 8
  - Macro Definition. . . . 8
  - Function, Process and Macro Application. . . . 8
- 1.3 Identifiers . . . 9
  - 1.3.1 Simple Identifiers . . . 9
  - 1.3.2 @-identifiers . . . 9
  - 1.3.3 $-identifiers . . . 10
  - 1.3.4 ::-identifiers . . . 10
- 1.4 Comments and Indentation . . . 10

2 Events 12
- 2.1 Event Specification . . . 12
- 2.2 Event Parameters . . . 12
- 2.3 Events as Containers . . . 13
- 2.4 Event Attributes . . . 14

3 *Antescofo* Model of Time 16
- 3.1 Logical Instant . . . 17
- 3.2 Time Frame . . . 18
- 3.3 Locating an Action in Time . . . 19

4 Actions in Brief 20
  - Action Attributes. . . . 20
- 4.1 Delays . . . 20
  - Zero Delay. . . . 21
  - Absolute and Relative Delay. . . . 21
  - Evaluation of a Delay. . . . 21
  - Synchronization Strategies. . . . 21
- 4.2 Label . . . 21

5 Atomic Actions 23
- 5.1 Message passing to Max/PD . . . 23

5.2 OSC Messages . . . 24
5.2.1 OSCSEND . . . 25
5.2.2 OSCRECV . . . 25
5.2.3 OSCON and OSCOFF . . . 26
5.3 Assignments . . . 26
5.4 Aborting and Cancelling an Action . . . 27
5.4.1 Abort of an Action . . . 27
Abort and the hierarchical structure of compound actions. . . . 28
5.4.2 Cancelling an Action . . . 28
5.5 I/O in a File . . . 29
5.6 Internal Commands . . . 29
5.7 Assertion @assert . . . 32

6 Compound Actions **33**
6.1 Group . . . 33
6.1.1 Local Tempo. . . . 34
6.1.2 Attributes of Group and Compound Actions . . . 34
6.1.3 Instances of a Group . . . 34
6.1.4 Aborting a group . . . 34
The until Clause. . . . 35
The during Clause. . . . 35
6.2 Conditional Actions: If . . . 36
6.3 Sequential iterations: Loop . . . 36
Stopping a Loop. . . . 36
6.4 Parallel Iterations: Forall . . . 37
6.5 Sampling parameters: Curve . . . 38
6.5.1 A Simple Curve . . . 38
6.5.2 Actions Fired by a Curve . . . 39
6.5.3 Step, Durations and Parameter Specifications . . . 39
6.5.4 Interpolation Methods . . . 39
Programming its Own Interpolation Method. . . . 40
6.5.5 Managing Multiple Curves Simultaneously . . . 41
6.5.6 Curve with a NIM . . . 42
6.6 Reacting to logical events: Whenever . . . 42
6.6.1 Stopping a whenever . . . 44
6.6.2 Causal Score and Temporal Shortcuts . . . 44
Automatic Temporal Shortcut Detection. . . . 45

7 Expressions **47**
7.1 Values . . . 47
Dynamic Typing. . . . 47

Checking the Type of a Value. 47
Value Comparison. 47
7.2 Scalar Values 48
7.2.1 Undefined Value 48
7.2.2 Boolean Value 48
7.2.3 Integer Value 48
7.2.4 Float Value 48
7.2.5 String Value 49
7.2.6 Intentional Functions 49
7.2.7 Proc Value 50
7.2.8 Exec Value 50
7.3 Aggregate Values 50
7.3.1 Map Value 50
Extensional Functions. 50
Domain, Range and Predicates. 51
Constructing Maps. 51
Extension of Arithmetic Operators. 52
Maps Transformations. 52
Score reflected in a Map. 53
History reflected in a map. 53
7.3.2 InterpolatedMap Value 53
NIM interpolated Map. 53
Interpolated Map. 54
7.3.3 Tab Value 54
Multidimensional tab. 55
Tab Comprehension. 55
Tab operators. 56
Tab manipulation. 57
Lists and Tabs. 59
7.4 Variables 60
7.4.1 Accessing Variable Histories 60
History reflected in a map. 60
7.4.2 Variables Declaration 61
Local Variables. 61
History Length of a Variable. 62
Lifetime of a Variable. 62
7.4.3 System Variables 62
7.4.4 Variables and Notifications 63
Temporal Shortcuts. 63
7.4.5 Dates functions 64
7.5 Operators and Predefined Functions 64

Conditional Expression. 64
@size. 64
7.6 Auto-Delimited Expressions in Actions 65

8 Synchronization and Error Handling Strategies **66**
8.1 Synchronization Strategies 66
8.1.1 Loose Synchronization 66
8.1.2 Tight Synchronization 66
8.2 Missed Event Errors Strategies 68
Combining Synchronization and Error Handling. 68

9 Process **71**
9.1 Macro *vs.* Processus 72
9.2 Recursive Process 72
9.3 Process as Values 72
9.4 Aborting a Process 73

10 Macros **74**
10.1 Macro Definitions 74
10.2 Expansion Sequence 75
10.3 Generating New Names 75

11 *Antescofo* Workflow **78**
11.1 Editing the Score 78
11.2 Tuning the *Antescofo* Listening Machine 78
11.3 Debuging an *Antescofo* Score 78
11.4 Dealing with Errors 78
11.4.1 Monotoring with Notability 79
11.4.2 Monotoring with *Ascograph* 79
11.4.3 Tracing an *Antescofo* Score 79
Printing the Parsed File. 79
Verbosity. 79
The TRACE Outlet. 79
Tracing the Updates of a Variable. 79
11.5 Interacting with MAX 80
11.5.1 Inlets 80
11.5.2 Outlets 80
11.5.3 Predefined Messages 81
11.6 Interacting with PureData 81
11.7 *Antescofo* Standalone Offline 81
11.8 Old Syntax 83

12 Stay Tuned **84**

A Library of Predefined Functions **85**

B Experimental Features **96**

B.1 Simple Curve *à la* `line~` 96
B.2 Tracks 96
B.3 Abort Handler 98
Example. 98
B.4 Patterns 99
B.4.1 `Note`: Patterns on Score 99
Pattern Variables. 100
Specifying Duration. 100
Specifying Additional Constraints. 101
Pattern Causality. 101
A Complete Exemple. 101
B.4.2 `Event` on Arbitrary Variables 102
The `value` Clause. 103
The `at` Clause. 103
The `where` Clause. 103
The `before` Clause. 104
Watching Multiple Variables Simultaneously. 104
A Complex Example. 105
B.4.3 `State` Patterns 105
A Motivating Example. 105
The initiation of a `state` Pattern. 106
The `during` Clause. 106
B.4.4 Limiting the Number of Matches of a Pattern 107
B.4.5 Pattern Compilation 107
B.5 Reserved Experimental Keywords 109

C Changes in 0.51 **110**

D Index **111**

E Detailed Table of Contents **120**