a not-so-short introduction to version 0.x

—— Draft ——

IRCAM UMR STMS 9912 – CNRS – UPMC – INRIA/MuTAnt
Document prepared by Jean-Louis Giavitto, Arshia Cont, José Echeveste
and MuTAnt Team Members

Revision April 26, 2016

Antescofo is a coupling of a real-time listening machine with a reactive and timed 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.

This document is a reference for the language 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 at:

- *Antescofo* home page
  `http://repmus.ircam.fr/antescofo`
- on *Antescofo*'s Ircam Forum User Group
  `http://forumnet.ircam.fr/user-groups/antescofo/`
  where you can find tutorials to download with bundles for MAX and PureData
- on Project Development Forge
  `http://forge.ircam.fr/p/antescofo/`
- on the web site of the MuTAnt team-project
  `http://repmus.ircam.fr/mutant`
  where you can find the scientific and technical publications on *Antescofo*.

Contents

1 Understanding *Antescofo* scores **7**
- 1.1 Structure of an *Antescofo* Score . . . 7
- 1.2 Elements of an *Antescofo* Score . . . 10
- 1.3 *Antescofo* keywords 11
- 1.4 @-identifiers: Functions, Macros, and Attributes 12
- 1.5 $-identifiers: Variables 13
- 1.6 ::-identifiers: Processes 14
- 1.7 Comments and Indentation 14

2 Events **17**
- 2.1 Event Specification 17
- 2.2 Events as Containers 18
- 2.3 Event Attributes 20
- 2.4 Importing Scores to Antescofo 21

3 Actions in Brief **23**
- 3.1 Delays 23
- 3.2 Label 25
- 3.3 Action Execution 25

4 Atomic Actions **31**
- 4.1 Message passing to Max/PD 31
- 4.2 OSC Messages 32
- 4.3 Assignments 34
- 4.4 Aborting and Cancelling an Action . 36
- 4.5 I/O in a File 38
- 4.6 Internal Commands 39
- 4.7 Assertion @assert 43

5 Compound Actions **45**
- 5.1 Group 45
- 5.2 If , Switch: Conditional and Alternative 48
- 5.3 Loop: Sequential iterations 50
- 5.4 Forall : Parallel Iterations 53
- 5.5 Curve: Continuous Actions 54
- 5.6 Whenever: Reacting to logical events 64

6 Expressions **69**
- 6.1 Values 69
- 6.2 Variables 71
- 6.3 Internal Antescofo Variables 76
- 6.4 Temporal Variables 78
- 6.5 Operators and Predefined Functions 78
- 6.6 Action as Expressions 79
- 6.7 Structuring Expressions 80
- 6.8 Auto-Delimited Expressions in Actions 80

7 Scalar Values **83**
- 7.1 Undefined Value 83
- 7.2 Boolean Value 83
- 7.3 Integer Value 83
- 7.4 Float Value 84
- 7.5 User-defined Functions 84
- 7.6 Proc Value 84
- 7.7 Exec Value 85

8 Data Structures **87**
- 8.1 String Value 87
- 8.2 Map Value 88
- 8.3 InterpolatedMap Value 91
- 8.4 Tables 97

9 Synchronization and Error Handling Strategies **105**
- 9.1 Synchronization Strategies 105
- 9.2 Missed Event Errors Strategies . . . 112

10 Macros **115**
- 10.1 Macro Definition and Usage 115
- 10.2 Expansion Sequence 116
- 10.3 Generating New Names 117

11 Functions **119**
- 11.1 Function definition 120
- 11.2 Functions as Values 126
- 11.3 Curryfied Functions 126

12 Process **129**
- 12.1 Calling a Process 129
- 12.2 Recursive Process 130
- 12.3 Process as Values 130
- 12.4 Aborting a Process 131
- 12.5 Processes and Variables 131
- 12.6 Process, Tempo and Synchronization 133
- 12.7 Macro *vs.* Function *vs.* Processus . . 133

13 *Antescofo* Workflow **135**
- 13.1 Editing the Score 135
- 13.2 Tuning the Listening Machine 135
- 13.3 Debuging an *Antescofo* Score 136
- 13.4 Dealing with Errors 136
- 13.5 Interacting with MAX 137
- 13.6 Interacting with PureData 138
- 13.7 *Antescofo* Standalone Offline 138
- 13.8 Old Syntax 140
- 13.9 Stay Tuned 140

A Library of Predefined Functions **143**

B Experimental Features **163**
- B.1 Reserved Experimental Keywords . . 163
- B.2 Constant BPM expresssion 163
- B.3 @eval_when_load Clause 164
- B.4 Tracks 164
- B.5 Abort Handler 165
- B.6 Continuations 167
- B.7 Open Scores and Dynamic Jumps . . 169
- B.8 Tracing Function Calls 171
- B.9 Infix notation for function calls . . . 171
- B.10 Methods 172
- B.11 Objects 174
- B.12 Patterns 181
- B.13 Scheduling Priorities 189

C Index **195**

D Detailed Table of Contents **205**

Sidebars

Brief history of *Antescofo* . . . 5

B.1 Some examples of continuation expressions. . . . 167

Figures

1.1 Antescofo Score Excerpt showing basic events and actions . . . 8
1.2 The beginning of *Tensio* (2010) by Philippe Manoury for String Quartet and Live electronics in *AscoGraph* . . . 9
1.3 Example of score attribute affectation (top-down parsing) in *Antescofo* text scores. . . . 11
1.4 Reserved keywords: Event keywords are in red whereas action keywords are in blue . . . 12
1.5 Predefined functions and special forms . . . 13
1.6 Rewrite of Figure 1.6 using a Macro and expressions . . . 14

2.1 Simple score with notes and chords. . . . 18
2.2 TRILL example on notes and chords . . . 19
2.3 MULTI example on chords . . . 19
2.4 Glissandi with Tremolos . . . 19
2.5 Polyphonic TRILL with events . . . 20

3.1 The *Antescofo* system architecture. . . . 26
3.2 Logical instant, physical time frame and relative time frame . . . 27

4.1 Two setvar in a Max patch. . . . 36

5.1 Simplified Curve syntax and its realisation in *Ascograph* . . . 54
5.2 Simplified Curve chain call . . . 55
5.3 Full 2d Curve . . . 57
5.4 Full 2d Curve embedded on Event Score in *Ascograph* . . . 57
5.5 Various interpolation type available in an *Antescofo* Curve and NIM . . . 62
5.6 (cont.) Various interpolation type available in an *Antescofo* Curve and NIM . . . 63

6.1 Example of simple expression and its value realisation in *Ascograph* . . . 70

8.1 The two forms a NIM definition . . . 93
8.2 The effect of @sample (top right), @align_breakpoints (bottom left), and @linearize (bottom right) on the nim pictured top left. . . . 95

9.1 The effect of tempo-only synchronization for accompaniment phrases: illustration for different tempi. . . . 107

9.2 These figures represent temporal evolution of an electronic phrase with several synchronization strategies. The graph shows the relationship between the relative time in beats with respect to absolute time in seconds. The musician events are represented by vectors where the slope correspond to the tempo estimation. The actions are represented by squares and the solid line represents the flow of time in the group enclosing these actions. From left to right and top to bottom, then strategies represented are : @loose, @tight, @target{sync}, @target [2] 110
9.3 Action behavior in case of a missed event for four synchronization and error handling strategies 113

10.1 Example of a Macro and its realisation upon score load 116

12.1 Comparaisons between the mechanisms of macro, function and process. 134

13.1 Help of the standalone offline command line. 139

B.1 Example of an object definition. 176
B.2 *State patterns with* during*,* before *and* @refractory *clauses.* 188

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. This document is mainly syntax and example driven and it supposes that you are familiar with *Antescofo*. On top of the regular document, **Sidebars** provide additional information for scientists or experienced users about the core design.

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 creation. Besides its incremental development with users and artists, the language is highly inspired by *Synchronous Reactive* languages such as *ESTEREL* and *Cyber-Physical Systems*.

Chapter 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 given score) and triggering electronics as written by the artists. *Antescofo* is thus used for computer arts involving live interaction and synchronisation between human and computerised actions. An *Antescofo* score describes both actions, the first or human actions for live recognition and the second as reactions to environmental input. 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 recognised. Actions in *Antescofo* extend the good-old *qlist* object elements in MAX and PD with additional features which will be described in this document.

Figure 1.1 shows a simple example from the *Antescofo* Composer Tutorial on Pierre Boulez' "Anthèmes 2" (1997) for violin and live electronics as seen in *Ascograph*. The left window shows a visual representation of Events and Actions, whereas the right segment shows the raw text score. Events in the score describe expected notes, trills and grace notes from the solo Violin, and actions specify messages to be sent upon the recognition of each event. In this example, we show case actions for four real-time *pitch shifter* (or harmoniser), whose general volume is controlled by the `hr−out−db` parameter, and each shifter parameter separately controlled by `hr1−p` to `hr4−p`. The values for pitch shifters are in pitch-scale factor. Their corresponding musical value is described in the text score as comments (any text proceeding semi-colon ';' is ignored in the score).

The *Antescofo* score of figure 1.1 shows basic use of actions and events. Red text in the text-editor correspond to reserved keyword for Events. For details regarding available Events and their corresponding syntax, see section 2. In this example, actions are basic message-passing to receivers in Max or Pd environments. Since they are isolated and discrete actions, we refer to them as *Atomic Actions*. As will be shown later, actions in *Antescofo* can use delays expressed in various time formats, and further include dynamic (i.e. real-time evaluated) expressions, data-structures and more. Details on action structures are discussed in section 3.

Figure 1.1 Antescofo Score Excerpt showing basic events and actions

```
11  BPM 46
12
13  NOTE 0 1.0
14  NOTE 8100 0.1       Q7
15      ;;;;;;;;;;;;;;;; Harmonizers
16      hr-out-db -12.0 25  ; bring level up to -6db in 25
17      hr1-p 0.9438    ; -1 semi-tones
18      hr2-p 0.7937    ; -4 semi-tones
19      hr3-p 0.6299    ; -8 semi-tones
20      hr4-p 0.5612    ; -10 semi-tones
21
22  NOTE 7300 1.0
23  NOTE 7100 0.1
24  NOTE 7200 1.0
25  NOTE 6600 1/7   Q8
26      hr-out-db -96 300   ; bring level to -96db in 500ms
27
28  NOTE 6900 1/7
29  NOTE 7200 1/7
30  NOTE 7300 1/7
31  NOTE 7400 1/7
32  NOTE 7500 1/7
33  NOTE 7700 1/7
34  TRILL   ( 7900 8000 )   1.5 Q9  ; measure 2 on 3/4
35      ;;;;;;;;;;;;;;;; Harmonizers
36      hr-out-db -12 25
37      hr1-p 0.890899  ; -2 semi-tones
38      hr2-p 0.749154  ; -5 semi-tones
39      hr3-p 0.66742   ; -7 semi-tones
40      hr4-p 0.943873  ; -1 semi-tones
41
42  NOTE 7800 0.5
43  TRILL   ( 7900 8000 )   0.5 Q10
44      ;;; Harmonizers
45      hr1-p 0.629961  ; -8 semi-tones
46      hr2-p 0.594604  ; -9 semi-tones
47      hr3-p 0.445449  ; -14 semi-tones
48      hr4-p 0.3337    ; -19 semi-tones
49
50  NOTE 7300 0.5
51  TRILL   ( 7900 8000 )   2.0 Q11 ; dur should be 1
52      ;;; Harmonizers
53      hr1-p 0.749154  ; -5 semi-tones
54      hr2-p 0.66742   ; -7 semi-tones
55      hr3-p 0.561231  ; -10 semi-tones
56      hr4-p 0.353553  ; -18 semi-tones
57
58  NOTE 7600 1.0
```

Figure 1.2 shows a slightly more complex *Antescofo* score corresponding to first two measures of "Tensio" (2010) by composer Philippe Manoury 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 1.2, the score for human musician contains four TRILLs (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) whereas prior to that a compound group named `cloches` is aimed to synchronize atomic actions (seen in text and collapsed group box in the visual screen) with the musician.

The example in Figure 1.2 makes use of *Compound Actions* which consist of parallel groupings of *Atomic Actions* (Chapter 5), as well as continuous actions defined by Curve keyword (Section 5.5), and macros (Chapter 10).

Figure 1.2 The beginning of *Tensio* (2010) by Philippe Manoury for String Quartet and Live electronics in *AscoGraph*

```
210              0.99   {0}
211          }
212      }
213
214      group       cloches
215      {
216
217                      sampler1_play  cloche.17_sol#5.wav 0 2
218              0.1     sampler1_play  cloche.17_sol#5.wav -800 2
219                      sampler1_play  cloche.17_sol#5.wav 200 2
220              0.2     sampler1_play  cloche.21_do6.wav -100 2.5
221              0.25        sampler1_play  cloche.21_do6.wav 900 2.5
222              0.1     sampler1_play  cloche.21_do6.wav 300 2.5
223      }
224
225
226      0.5 group   Pizzicati
227      {
228                          Bsyn_voices 4 vol 127
229                          Bsyn_voices 4    vol 127
230              1/17        Bsyn_voices 4   85 127 100
231                          Bsyn_voices 4    vol 127
232              1/17        Bsyn_voices 4   85 127 100
233              1/17        Bsyn_voices 4   85 127 100
234              1/17        Bsyn_voices 4   85 127 100
235              1/17        Bsyn_voices 4   85 127 100
236              1/17        Bsyn_voices 4   85 127 100
237              1/17        Bsyn_voices 4   85 127 100
238      }
239              ; print un temps est egal Ã
240
241  NOTE 0 0.5
242  TRILL (8100) 0.5
243  NOTE 0 0.5
244  ;************************************************************
245  TRILL ( 7500) 0.5           IA...EVT-2
246
247          @cresc_curve(2,84,120,1,100,127,80,127,33)
248
249  TRILL ( 7800) 0.5
250  NOTE 0 1.0
```

A textual *Antescofo* score, or program, is written in a single file and loaded from there. The file itself can optionally include pointers to other *Antescofo* score 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. @insert is often used to store definitions and initialisation of the main *Antescofo* score. It will automatically create additional tabs in *Ascograph* text editor. The @insert_once is simlar to the @insert command except that the file is included only once in the current score, when the directive is encountered the first time. The behavior makes possible to include a library of primitives in set of files without the burden to take care of the dependencies.

In this chapter, we briefly introduce main elements in *Antescofo* language and leave details for proceeding dedicated chapters to each concept. 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**.

1.2 Elements of an *Antescofo* Score

Antescofo is a coupling of a listening machine (a score follower, recognising positions and tempo of the musician in a score at real-time) and a real-time programming language to describe the computer interaction as a result of this recognition. As a consequence, an *Antescofo* program is a sequence of *events* and *actions*. Events are recognized by the listening machine described in detail in section 2. Actions, outlined in detail in sections 3 and 4, are computations triggered upon the occurrence of an event or of another action. Actions can be dynamically parametrised by *expressions* and data structures, evaluated in real-time and described in section 6.

Elements of the *Antescofo* language can be categorised into four groups, corresponding to various constructions permitted in the language:

- *Keywords:* are reserved words by *Antescofo* language that specify either Events or Action constructions. Examples include Note (for events) and group (for compound actions).
- *Comments:* Any text proceeding a semi-colon ';' is considered as comment and ignored by parser (inline comment). Block (multi-line) C-Style comments using /∗ ... ∗/ is allowed.
- *@-identifiers:* are words that start with '@' character. They signify either: call to internal *Antescofo* functions, user-defined macros, or action or event attributes.
- *$-identifiers:* are words that start with '$' character. They correspond to user-defined variables or arguments in functions, processes or macro definitions.
- *::-identifiers:* words starting with '::', corresponding to *Process* calls.

REMARK: An Antescofo text score is interpreted from top to bottom. In this sense, *Event Sequence* commands such as bpm or variance will affect lines that follow its appearance.

Example: Figure 1.3 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 onwards and consequently, the action delay of 1/2 beat-time hooked on note $C5$ corresponds to a score tempo of 60 BPM.

Figure 1.3 Example of score attribute affectation (top-down parsing) in *Antescofo* text scores.

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

User defined score elements including Macros, processus and functions can only be employed 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 using the `@insert` command. They will be discussed in proceeding chapters.

1.3 *Antescofo* keywords

The *Antescofo* language comes with a list of pre-defined *keywords* for defining elementary score structures. As usual, they are divided in two groups: *Event Keywords* including Note, Chord, Trill and Multi with specific syntax (see chapter 2) and *Action Keywords* such as group, Loop and more.

Event keywords are used to describe the music score to be recognised whereas Action Keywords are *containers* for basic or atomic actions. Atomic actions are not specified by any keyword. For example in Figure 1.1, lines $16 - 21$ are actions that are hooked to event "NOTE 8100 1.0 Q7", and in Figure 1.3 lines "1/2 print action1" denote an action (sending to a [receive print] in max/pd) with a delay of half-beat time. General syntax for atomic actions is described in section 4.

The current list of reserved *Antescofo* keywords is given in Figure. 1.4. These keyword are *case unsensitive*, that is

```
note NOTE Note NoTe notE
```

all denote the same keyword. Case insensitivity *does not apply* however to user-defined Macros, Functions or event labels.

In case a score requires the user to employ a reserved keyword inside (for example) a message, the user should wrap the keyword in quotes to avoid clash.

Event keywords can not be nested inside Action blocks. Event keywords are always defined at the top-level of the text score. Action keywords and blocks can be nested as will be discussed later.

Figure 1.4 Reserved keywords: Event keywords are in red whereas action keywords are in blue

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

1.4 @-identifiers: Functions, Macros, and Attributes

A word preceded immediately with a '@' character is called a @-identifier. They have five purposes in *Antescofo* language:

1. when loading a file to insert another file @insert or to generate fresh identifiers @uid and @lid;
2. to introduce new definitions in a score file: @fun_def (see section 11), @macro_def (see section 10), @pattern_def (see section B.12), @proc_def (see section 12), @track_def (see section B.4);
3. to introduce various attributes of an event or an action:

   @abort @action @ante @back @back_in @back_in_out @back_out @bounce @bounce_in @bounce_in_out @bounce_out @circ @circ_in @circ_in_out @circ_out @coef @command @conservative @cubic @cubic_in @cubic_in_out @cubic_out @date @dsp_channel @dsp_cvar @dsp_inlet @dsp_link @dsp_outlet @dump @elastic @elastic_in @elastic_in_out @elastic_out @eval_when_load @exclusive @exp @exp_in @exp_in_out @exp_out @fermata @global @grain @guard @hook @immediate @inlet @is_undef @jump @kill @label @latency @linear_in @linear_in_out @linear_out @local @loose @modulate @name @norec @pizz @plot @post @progressive @quad @quad_in @quad_in_out @quad_out @quart @quart_in @quart_in_out @quart_out @quint @quint_in @quint_in_out @quint_out @rdate @refractory @rplot @sine @sine_in @sine_in_out @sine_out @staccato @staticscope @sync @target @tempo @tempovar @tight @transpose @type

4. to call internal functions that comes with *Antescofo* language as listed in Figure 1.5 and detailed in Annexe A.
5. and to call user-defined functions or macros (*case sensitive*).

Only ! ? . and _ are allowed as special characters after the @. Note that in the first three cases, @-identifiers are *case unsensitive*, that is @tight, @TiGhT and @TIGHT are the same keyword. Users can define their own functions as shown in section 7.5.

Figure 1.5 Predefined functions and special forms

@+ @− @∗ @/ @%
@< @<= @> @>=
@== @!=
@|| @&&

@abs @acos @add_pair
@approx @arch_darwin
@arch_linux
@arch_windows @asin
@atan

@between
@bounded_integrate_inv
@bounded_integrate

@car @cdr @ceil @clear
@concat @cons @copy
@cosh @cos @count

@dim @domain @dump
@dumpvar

@empty @exp @explode

@find @flatten @flatten
@floor

@gnuplot @gnuplot
@gnuplot @gnuplot
@gnuplot @gshift_map

@history_length

@insert @insert
@integrate @iota
@is_bool @is_defined
@is_fct @is_float
@is_function
@is_integer_indexed
@is_interpolatedmap
@is_int @is_list
@is_map @is_numeric
@is_prefix @is_prefix
@is_prefix @is_prefix
@is_string @is_subsequence
@is_subsequence
@is_subsequence
@is_subsequence @is_suffix
@is_suffix @is_suffix
@is_suffix @is_symbol
@is_undef @is_vector

@lace @last @listify
@loadvalue @loadvar
@log10 @log2 @log

@make_duration_map
@make_label_pos
@make_label_bpm
@make_label_duration
@make_label_pitches
@make_score_map @map
@map_compose
@map_concat
@map_history
@map_history_date
@map_history_rdate
@map_normalize
@map_reverse @mapval
@max_key @max_key
@max_val @max
@member @merge
@min_key @min_key
@min_val @min

@normalize

@occurs

@permute @plot @pow
@push_back @push_back
@push_back @push_front

@rand_int @random
@rand @reduce @range
@remove @remove
@remove_duplicate
@replace @reshape
@resize @reverse
@rnd_bernoulli
@rnd_binomial
@rnd_exponential
@rnd_gamma
@rnd_geometric
@rnd_normal
@rnd_uniform_int
@rnd_uniform_float
@rotate @round @rplot

@savevalue @scan
@scramble @select_map
@shape @shift_map @sinh
@sin @size @sort @sort
@sputter @sqrt
@string2fun @string2proc
@stutter @system

@tab_history
@tab_history_date
@tab_history_rdate @tan
@Tracing @Tracing

@UnTracing @UnTracing

1.5 $-identifiers: Variables

$-identifiers like $id, $id_1 are simple identifier prefixed with a dollar sign. Only ! ? . and _ are allowed as special characters. $-identifier are used to give a name to variables during assignments (sec 4.3) and for function, process and macro definition arguments. *They are case-sensitive.*

Figure 1.6 shows a rewrite of the score in Fig. 1.1 using a simple Macro and employing basic @ and $ identifiers. The harmoniser command is here defined as a Macro (section 10) for convenience and since it is being repeated through the same pattern. The content of the “hr1−p” to “hr4−p” actions inside the Macro use a mathematical expression using the internal function @pow to convert semi-tones to pitch-scale factor. As a result the *Antescofo*

score is shorter and musically more readable. Variables passed to the Macro definitions are $-identifiers as expected.

You can learn more on expressions and variables in chapter 6 onwards.

Figure 1.6 Rewrite of Figure 1.6 using a Macro and expressions

1.6 ::-identifiers: Processes

::-identifiers like ::P or ::q1 are simple identifier prefixed with two semi-columns. ::-identifiers are used to give a name to processus (see section 12).

1.7 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"
```

Chapter 2

Events

An event in *Antescofo* terminology corresponds to elements defining what will probably happen outside your computers for real-time detection and recognition. In regular usage, they describe the *music score* to be played by the musician. 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 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).

This section describes *Events* and their syntax in *Antescofo* language. In a regular workflow, they can come from pre-composed music scores using *MusicXML* or *MIDI* import (see section 2.4). They can also be composed directly into the *Antescofo* text program.

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 duration [label]
CHORD (pitch_list) duration [label]
TRILL (*(pitch_list)) duration [label]
MULTI (*(pitch_list)) duration [label]
MULTI ((pitch_list) -> (pitch_list)) duration [label]
```

(the '*' sign means "zero or more" repetition of what follows.

Parameters for events specification is described 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 A *pitch* (used in NOTE) can take the following forms: MIDI number (e.g. 69 and 70), MIDI cent number (e.g. 6900 and 7000), or Standard Pitch Name (e.g. A4 and A#4). For microtonal notations, one can use either MIDI cent (e.g. 6900) or Pitch Name standard and MIDI cent deviations using '+' or '-' (e.g. A4+50 and A#4+50 or B4-50).

pitch_list is a set containing one or more pitches (used to define content of a CHORD). For example, the following line defines a C-Major chord composed of `C4, E4, G4`:

```
CHORD ( C4 64 6700)
```

Trill and Multi are examples of *compound events*, meaning that they can accept one or several *pitch_lists*. *pitch_lists* in Trill and Multi are distinguished by their surrounding parenthesis. See next section for a more musically subtle explanation.

duration is a mandatory specification for all events. The duration of an event is specified in beats either by a float (1.0), an integer (1) or the ratio of two integers (4/3).

label Optionally, users can define *labels* on events as a single *string*, useful for browsing inside the score and for visualisation purposes. For example `measure1` is an accepted label. If you intend to use *space* or *mathematical symbols* inside your string, you should surround them with quotations such as `"measure 1"` or `"measure-1"`

Events specification can be optionally followed by some attributes as discussed in section 2.3. Events must end by a carriage return. In other word, you are allowed to define one event per line.

There is an additional kind of event

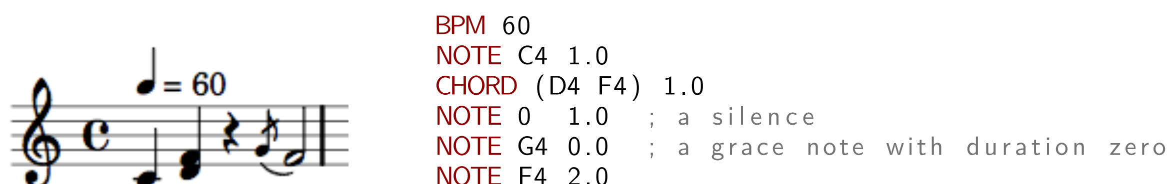
```
EVENT ...
```

also followed by a mandatory duration, which correspond to a fake event triggered manually by "nextevent" button on the graphical interface.

2.2 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 2.1 shows an example including simple notes and chords written in *Antescofo*:

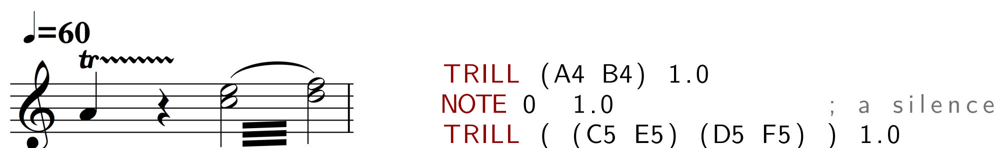
Figure 2.1 Simple score with notes and chords.



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 2.2 shows basic examples for Trill.

Figure 2.2 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*. Additionally, a Multi contents can be trills. To achieve this, it suffices to insert a ' character after the *pitch_list* closure. Figure 2.3 shows an example of glissandi between chords written by Multi.

Figure 2.3 MULTI example on chords

Compound Structures Events can be combined and correspond to specific music notations. For example, a classical *tremolo* can be notated as a Trill with one event (note or chord) inside. Figure 2.4 shows a glissando whose internal elements are tremolo. In this case, the *prime* next to each chord group indicate that the elements in side the MULTI are TRILLs instead of regular notes or chords.

Figure 2.4 Glissandi with Tremolos

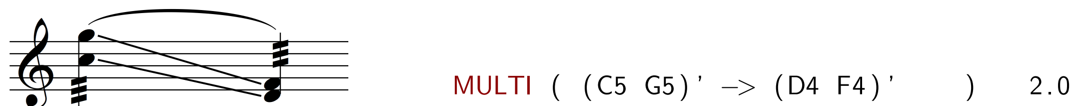


Figure 2.5 shows a typical polyphonic situation on piano where the right-hand is playing a regular trill, and the left hand regular notes and chords. In this case, the score is to be segmented at each event onset as Trill s whose elements would become the trill element plus the static notes or chords in the left-hand.

Figure 2.5 Polyphonic TRILL with events


```
TRILL ( (A4  A2) (B4  A2) )           1/2
TRILL ( (A4  D3) (B4  D3) )           1/2
TRILL ( (A4  C3 E3) (B4  C3 E3) )     1/2
TRILL ( (A4  D3) (B4  D3) )           1/2
TRILL ( A4 B4 )       2.0
```

2.3 Event Attributes

Attributes in *Antescofo* are keywords following a '@' character after the definition of the event. There are four kinds of event attributes and they are all optional.

- The keyword fermata (or @fermata) specifies that this event has a *fermata* signature. A Fermata event can last longer and arriving and leaving it does not contribute to the tempo decoding of the performance.
- The keyword pizz (or @pizz) specifies the event is a string *pizzicato*. This usually helps Score Follower stability.
- 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).
- 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  @fermata @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. It has a *fermata*. 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 l2.

Event Label. 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 @modulate Attribute. The @modulate attribute an be used on a BPM specification, not on an event. It specify that the tempo must be modulated to the *pro rata* of the actual tempo of the performer. For example, if a BPM 60 is specified in the score, and the actual tempo of the performance if 70, then an indication of BPM 80 @modulate reset the tempo expected by the listening machine to $80.\frac{70}{60} \simeq 93.3$.

2.4 Importing Scores to Antescofo

It is possible to automatically import MIDI or MusicXML scores to *Antescofo* format. This feature is available by drag and dropping MIDI or MusicXML files into *Ascograph*. For multiple instrument score, care should be taken to extract required *Antescofo* part in a separate MIDI or MusicXML file.

Users employing these features should pay attention to the following notes:

2.4.1 Importing MIDI scores to Antescofo

The major problem with MIDI format is the absence of *grace notes*, *trills*, and *glissandi*. Such events will be shown as raw NOTE and CHORD event elements in the *Antescofo* score.

Another major issue with MIDI import is the fact that in most cases, timing of note-offs are not decoded correctly (based on where the MIDI is coming from). Bad offset timing creates additional NOTE or CHORDs with linked pitches (negative notes) with short durations. To avoid this, we recommend users to *quantize* their MIDI files using available software. We do not quantise durations during import.

2.4.2 Importing MusicXML scores to Antescofo

MusicXML is now the standard inter-exchange score format file between various score editing and visualisation software. It includes high-level vocabulary for events such as *trills*, *grace notes* and *glissandi* which can be converted to equivalent *Antescofo* event. However, decoding and encoding MusicXML is not necessarily unique for the same score created by different software!

The *Ascograph* MusicXML import is optimised for MusicXML exports from *FINALE* software. Before converting MusicXML score to *Antescofo*, users are invited to take into account the following notes and correct their score accordingly, especially for complex contemporary music scores:

- Avoid using *Layers*: Merge all voices into one staff/voice before converting to MusicXML and dragging to *Ascograph*. XML parsers sometimes generate errors and suppress some events when conflicts are detected between layers.
- Avoid using *Graphical Elements* in score editors. For example, *Trills* can only be translated to *Antescofo* if they are non-graphical.
- If possible, avoid non-traditional note-heads in your editor to assure correct parsing for *Antescofo* events.
- Avoid *Hidden* elements in your scores (used mostly to create beautiful layouts) as they can lead to unwanted results during conversion. Verify that durations in your score correspond to what you see and that they are not defined as hidden in the score.
- Verify your *Trill* elements after conversion as with some editors they can vary.

This feature is still experimental and we encourage users encountering problems to contact us through the *Antescofo* Online User Group.

Chapter 3

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 4.1 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 3.2. The attributes specific to a given kind of action are described in the section dedicated to this kind of action.

3.1 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 3.3.

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 `a1` occurs one seconds after `a0` and `a2` occurs 2*$v milliseconds after `a1`. 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 5.1.1).

Evaluation of a Delay. In the previous example, the delay for `a2` 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 `action2` 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 `action2`. This approach will be for instance more tightly coupled with the stream of musical events. Synchronization strategies are discussed in section 9.2.

3.2 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[1].

Compound actions have an optional identifier (section 5). This identifier is a simple identifier and act as a label for the action.

3.3 Action Execution

We write at the beginning of this chapter that *actions* are performed when arriving at an instant in time. But the specification of this date can take several forms. It can be

- the occurrence of a musical event;
- the occurrence of a logical event (see the whenever construction page 64 and the pattern specification 181);
- the loading of the score (cf. the @eval_when_load construct at page 164);
- the signal spanned by an @abort action (see abort handler at page 165);
- the sampling of a curve construct (page 54);
- the instance of an iterative construct (page 50 and page 53);
- or the expiration of a delay starting with the triggering of another action.

Before digging into the details of the actions, we sketch the *Antescofo* notion of time and date.

[1]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).

Antescofo Model of Time

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. *Antescofo* thus takes care timely delivery, coordination and synchronisation of actions with regards to the external environment (musicians) using machine listening.

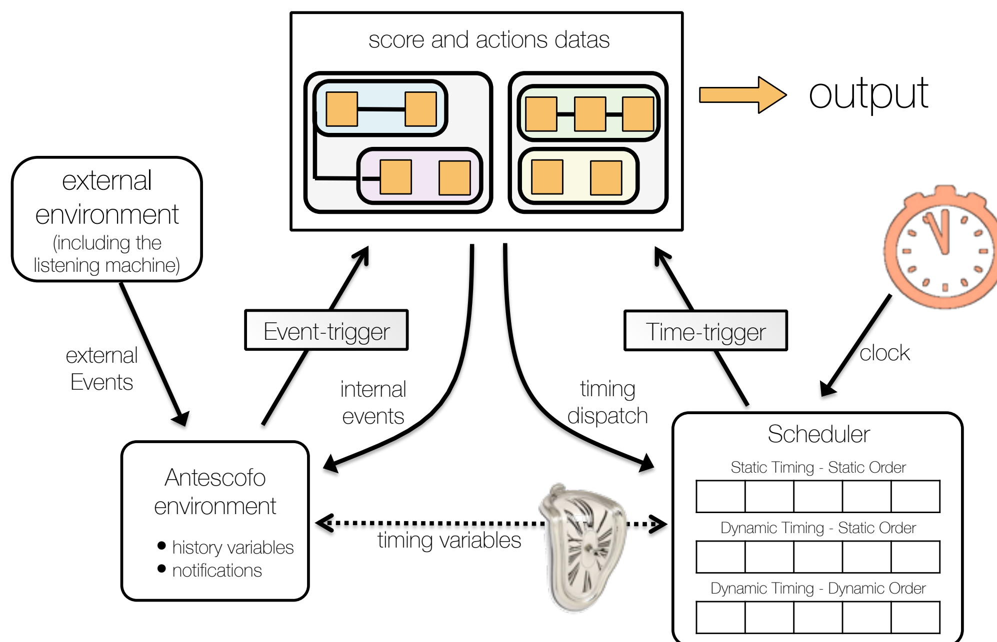
Experienced users should note that *Antescofo* is delivered with its own *Real-time Scheduler*. This is mainly to reduce utility costs of using internal Max and Pd *Timers* and to significantly reduce their interference with other actions in Max/Pd schedulers themselves. The *Antescofo* internal scheduler is new since version 0.5 onwards. It is explained briefly in this Sidebar.

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. 3.1:

- 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 3.1 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.



Logical Instant

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

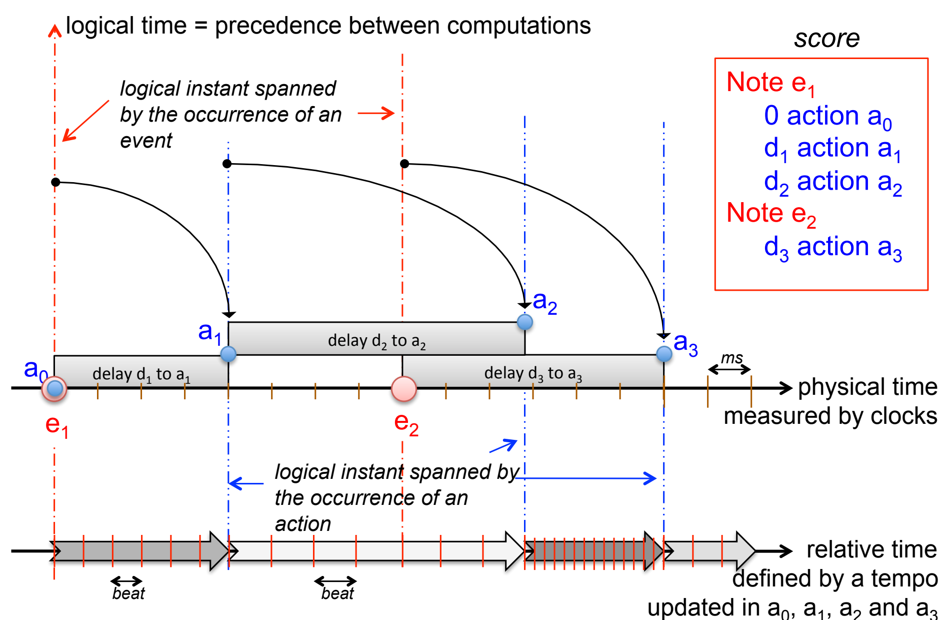
- the recognition of a musical event;
- the assignment of a variable by the external environment (*e.g.* through an OSC message or a MAX/PD binding);
- the expiration of a delay.

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 6.2. 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 3.2, the sequence of synchronous actions appears in the vertical axis. So this axis corresponds to the dependency between simultaneous computations. Note for example that even if d_1 and d_2 are both zero, the execution order of actions a_0, a_1 and 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.

Figure 3.2 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.



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[2]. 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:

$$t_\mathcal{T} = \int_0^{t_\mathcal{P}} T_\mathcal{T} \tag{3.1}$$

Variable updates are discrete in *Antescofo*; so, in this equation, $T_\mathcal{T}$ is interpreted as a piecewise 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 5.1. 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 (3.1).

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[3]. This variable is extracted from the audio stream by the listening machine, relying on cognitive model of musician behavior[4]. The corresponding frame of reference is used when we speak of "relative time" without additional qualifier.

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 3.2 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 9.

---

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

[3] 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 (3.1) at any moment. So, for *the* relative time frame, equation (3.1) 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.

[4] 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.

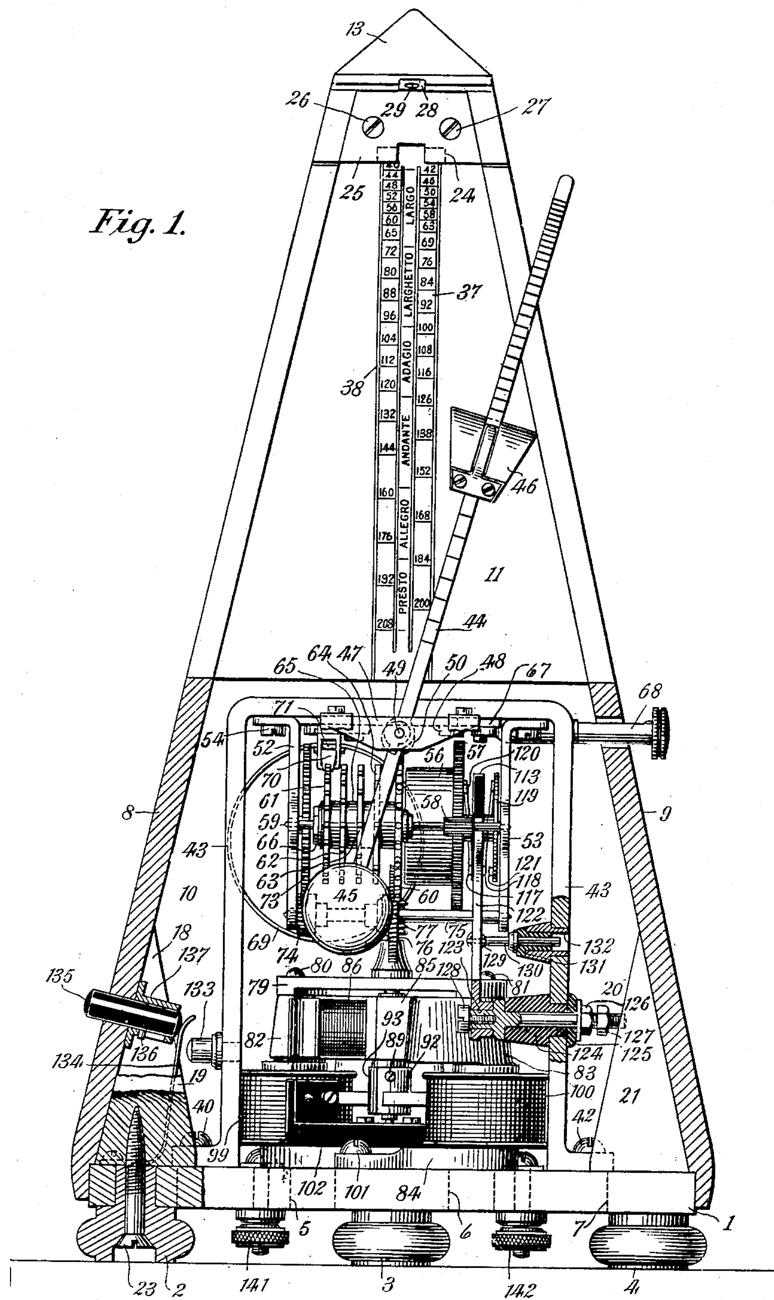


Fig. 1.

Chapter 4

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.

4.1 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 3.1). They take the familiar form of:

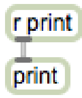
```
<optional-delay> <receiver-name> <message-content>
```

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 4.2 and MAX/PD messages[1].

A MAX/PD message starts by an optional delay followed by a symbol referring to a MAX or PD receiver. This identifier must be different from *Antescofo* reserved keywords listed in section 1.3 page 11. They should correspond to a *receiver* object in MAX/PD with the same identifier[2]. 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 left:

[1]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 4.6 and 13.5. On the contrary, arbitrary messages can be sent from an *Antescofo* score to external MAX objects with appropriate receivers.

[2]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 8.1). In this case, use `@command()`. For example, if the value of `$num` is 1, `@command("spat"+$num)` builds the "spat1" receiver.




```
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. 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. 8.1.

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"
```

4.2 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[3] 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.

[3] http://opensoundcontrol.org/

4.2.1 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:

$name\ arg1\ \ldots\ arg_n$

The idea is that this construct and send the osc message

$msg_prefix\ arg1\ \ldots\ arg_n$

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.

4.2.2 OSCRECV

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

$oscrecv\ name\ port\ msg_prefix\ \$v_1\ \ldots\ \$v_n$

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 $\$v_1 \dots \$v_n$. 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.

4.2.3 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.

4.3 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 makes more clear the distinction between the delay and the assigned variable:

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

In the previous example, the delay is specified by an expression, the $d variable, and the let outlines that the assigned variable is $x and not $d.

A variable has a value before its first assignment: the undefined value (sect. 7.1).

Expressions e in the right hand side of := are described in section 6.

The identifier of the assigned variable in the left hand side can be replaced by an underscore _ which is usefull to spare a variable when the result of the expression in the right hand side is not needed. This is the case if the expression is evaluated for its side-effects, like dumping values in a file. This action

```
_ := exp
```

simply evaluates the right hand side and discard the result.

Antescofo variable can be assigned outside *Antescofo*, using the setvar message in Max or PureData, or an OSC message, see below.

Assignment to Vector Elements and to Local variables. The left hand side of := is not restricted to a variable. As a matter of fact, they are three kind of assignment:

1. variable assignment: `$x` :=e
2. the assignment of an element in an tab:

   `let` e'`[`i_1`,` i_2`, ...] :=` e

   where e' is an expression that evaluates to a tab and i_1, i_2... evaluate to integers (sect. 8.4 p. 99);
3. the assignment of a local variable in an *exec*:

   `let` e'`.$x :=` e

   where e' is an expression that evaluates to an *exec* (see sect. 7.7 p. 85 and sect. 6.2.4).

The `let` keyword it is mandatory when e' is more complex than a variable.

Activities Triggered by Assignments. The assignment of a value to a variable[4] may triggers some activities:

- the evaluation of a `whenever` that depends on this variable (see section 5.6);
- the reevaluation of the delays that depends on a relative tempo that depends on this variable[5].

System variables and special variable cannot be assigned: `$RT_TEMPO`, `$PITCH`, `$BEAT_POS`, `$LAST_EVENT_LABEL`, `$DURATION`, `$NOW`, `$RNOW`, `$MYSELF`. These variables are *read-only* for the composer: they are assigned by the system during the performance. However, like usual variables, their assignment (by the system) may trigger some activities. For instance delays expressed in the relative time are updated on `$RT_TEMPO` changes. Tight actions waiting on a specific event (cf. section 9.1.2) are notified on `$BEAT_POS` changes. Etc. Refer to section 6.3.2 for additional information.

External Assignments. A global variable may be assigned "from outside *Antescofo*" in two ways:

1. using the `setvar` message in Max or PureData,
2. using an OSC message.

[4] and not to a tab element nor to a local variable, see paragraph 5.6 page 65

[5] As mentioned in section 3.1, 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.

Section 4.2.2 describes the assignment of variables upon the reception of an OSC message.

A simple patch using the setvar message is pictured fig.4.1. The setvar message take the name of the *Antescofo* variable to assign as its first argument. If there is only a second numeric argument, this argument becomes the value of the variable. If there are several numeric remaining arguments, these arguments are put in a tab (see sect 8.4) and the the variable is assigned with this value.

External assignments trigger the whenevers that may watch the externally assigned variables, cf. sect. 5.6. For example, with the patch in Fig. 4.1, the *Antescofo* program:

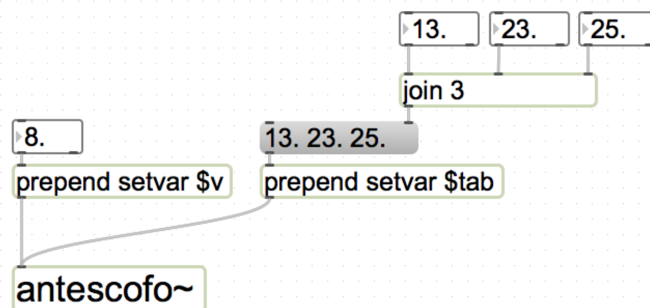
```
whenever ($tab)
{
  print "I just received the vector " $tab
}
```

will write

```
I just received the vector 13 23 25
```

on the console when the "prepend setvar . . . " is launched.

Figure 4.1 Two setvar in a Max patch.



4.4 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 5 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.

4.4.1 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 5.6). 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.

Abort and the hierarchical structure of compound actions. By default, the abort command applies recursively on the whole hierarchical structure of actions (cf. section 5). 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 *a1* and *b1* have already been fired. The results of the abort is to suppress the future firing of *a2*, *a3*, *b2* and *b3*. If line 11 is replaced by

```
2.5 abort G1 @norec
```

then, actions *a2* and *a3* are aborted but not actions *b2* and *b3*.

4.4.2 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

```
1  group G1 {
2     1 a1
3     0 abort action_to_suppress of G1
4     1 a2 @name := action_to_supress
5     1 a3
6  }
```

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.

4.5 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[6].

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.

It is possible to save a value in a file to be read somewhere else, or to dump the value of some variables to be restored later (or in another program execution). See functions @savevalue, @loadvalue, @dump, @dumpvar and @loadvar.

[6] 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).

4.6 Internal Commands

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.

Internal commands are named antescofo :: *xxx* where the suffix *xxx* is the head of the corresponding MAX/PD message (cf. section 13.5):

- antescofo::actions *string* : inhibits ("off") or trigger ("on") the launch of actions on events recognition. This is different of muting a track (p. 164): muting a track inhibit the sending of some Max/PD messages while this command inhibit all actions.
- antescofo::add_completion_string *string* : specify a new completion string to a running and connected *Ascograph* (Mac only).
- antescofo::analysis *int int* : specify a new FFT windows length and a new hop size for the audio analysis.
- antescofo :: ascographwidth_set *int* : specify the width of the *Ascograph* window. Look also some parameters of the interface in the inspector of the *Antescofo* object).
- antescofo :: ascographheight_set *int* : specify the height of the *Ascograph* window. Look also some parameters of the interface in the inspector of the *Antescofo* object).
- antescofo :: ascographxy_set *int int* : specify the x and y position of the *Ascograph* window. Look also some parameters of the interface in the inspector of the *Antescofo* object).
- antescofo::asco_trace *int* : turn on (1) or off (0) the *Ascograph* tracking of the score position when *Antescofo* is runng in following mode.
- antescofo::before_nextlabel (no argument) : force the progression of the score following up-to the next label, launching the actions between the current point and the next label, but still waiting its occurence.
- antescofo::bpmtolerance *float* : reserved command.
- antescofo::calibrate *int* : turn calibration mode on (1) or off (0).
- antescofo::clear (no argument) : clear all preloaded scores.
- antescofo::decodewindow *int* : changes the length of the decoding window used in the inference of the position.
- antescofo::filewatchset *string* : (Max only) watch the file whose path is given by the argument, to reload it when changed on disk (the file is suppose dto be the source of the current score).
- antescofo::gamma *float* : change an internal parameter of the score following engine.
- antescofo::get_current_score (no argument) : send to a running and connected *Ascograph* the source of the current score.

- antescofo::get_patch_receivers (no argument) : reserved command.
- antescofo::getlabels (no argument) : send to the first outlet the list of events label. Correspond to the `get_cues` message accepted by the object.
- antescofo::gotobeat *float* : position the follower at a position specified in beat without doing anything else. See below for moving in score commands.
- antescofo::gotolabel *string* : position the follower on an event specified by its label without doing anything else.
- antescofo::harmlist *float ...* (a list of floats corresponding to a vector) : specify the list of harmonics used in the audio observation.
- antescofo::info (no argument) : print on the console output various informations on the *Antescofo* version, and the current status of the object. Usefull when reporting a problem.
- antescofo::killall (no argument) : abort all running processes and actions.
- antescofo::mode *int* : reserved command.
- antescofo::mute *string* : mute (inhibits the sending) of the messages matched by a track.
- antescofo::nextaction (no argument) : forces the follower to wait for the next event that has some associated actions. The actions triggered between the current position and the new one, are launched.
- antescofo::nextevent (no argument) : forces the follower to wait for the first event that appears after the current position. The actions between the current position and the next event are launched.
- antescofo::nextlabel (no argument) : forces the follower to wait for the next event that has a label. The actions between the current position and the next event are launched. The infered tempo is kept unchanged.
- antescofo::nextlabel tempo (no argument) : same as antescofo::nextlabel but the elapsed time is used to adjust the tempo.
- antescofo::nofharm *int* : number of harmonics to compute for the audio analysis.
- antescofo::normin *float* : set some internal parameter of the score following.
- antescofo::obsexp *float* : set some internal parameter of the score following.
- antescofo::pedal *int* : enables (1) or disables (0) the use of a resonnance model in the audio observation.
- antescofo::pedal coeff *float* : attenuation coefficient of the pedal model.
- antescofo::pedal time *float* : attenuation time of the pedal model.
- antescofo::piano *int* : turn the follower in the piano mode (corresponding to a set of parameters adjusted to optimize piano observation).

- antescofo::play (no argument) : simulates the score (instrumental+electronics) from the beginning until the end or until STOP.
- antescofo::play frombeat `float` : executes the score from current position upto the position given by the argument in accelerated more WITHOUT sending messages, then PLAYs (simulates) the score from thereon.
- antescofo::play fromlabel `string` : executes the score from current position upto then event specified by the argument in accelerated more WITHOUT sending messages, then PLAYs (simulates) the score from thereon.
- antescofo::play string `string` : interpret the string argument as an *Antescofo* sequence of actions and perform it. This command is used by *Ascograph* to evaluate a text region highlighted in the editor. It can be used to evaluate on-the-fly actions that have been dynamically generated in an improvisation scenario. Due to OSC limitation, the string size cannot be greather than 1000 characters. But see next command.
- antescofo::play string_append `string` : this command is used to evaluate on-the-fly a string of size greather than 1000 characters. The sequence of actions must be broken in a sequence of strings each of size less than 1000. Each of these pieces are processed in turn by this command, except for the last one which uses antescofo::play string
- antescofo::play tobeat `float` : PLAY (simulate) score from current position up to position specified by the argument.
- antescofo::play tolabel `string` : PLAY (simulate) score from current position up to the event specified by the argument.
- antescofo::preload `string string` : preloads a score and store it under a name (the second argument) for latter use.
- antescofo::preventzigzag `string` : allow or disallow zig-zag in the follower. In non-zig-zag mode, the default, the follower infer only increasing positions (except in the case of a jump). In zig-zag mode, the follower may revise a past inference. This ability does not impact the reactive engine : in any case, actions are performed without revision.
- antescofo::previousevent (no argument) : similar to antescofo::nextevent but looking backward in the score.
- antescofo::previouslabel (no argument) : similar to antescofo::nextlabel but looking backward in the score.
- antescofo::printfwd (no argument) : output the formatted print of the current program in a new window.
- antescofo::printscore (no argument) : output the formatted print of the current program in a new window.
- antescofo::read `string` : loads the corresponding *Antescofo* score from the file specified by the argument.
- antescofo::report (no argument) : reserved command

- antescofo::scrubtolabel `string` : Executes the score from current position to the event specified by the argument, in accelerated more WITH sending messages. Waits for follower (or user input) right before this position.
- antescofo::scrubtobeat `string` : Executes the score from current position to the position given by the argument, in accelerated more WITH sending messages. Waits for follower (or user input) right before this position.
- 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). See section 4.3
- antescofo::score `string` : loads the corresponding *Antescofo* score (an alias of antescofo::read).
- antescofo::start `string` : Sends initialization actions (before first event) and wait for follower.
- antescofo::start fromlabel `string` : Executes the score from current position to position corresponding to in accelerated more WITHOUT sending messages. Waits for follower (or user input) right before this position.
- antescofo::start frombeat `int` : Executes the score from current position to the given position in accelerated more WITHOUT sending messages. Waits for follower (or user input) right before this position.
- antescofo::static_analysis (no argument) : reserved command
- antescofo::stop (no argument) : stop the follower and abort the runing actions.
- antescofo::suivi `int` : enables or disables the follower. Even if the follower is off, actions may run and can be spanned and interaction with the environment may happens through antescofo :: set_var and whenever.
- antescofo::tempo `float` : specify an arbitrary tempo.
- antescofo::tempo init `int` : reserved command.
- antescofo::temposmoothness `float` : adjust a parameter of the tempo inference algorithm.
- antescofo::tune `float` : set the tuning base (default 440.0 Hz
- antescofo::unmute `string` : unmute (allows the sending) of the messages matched by a track.
- antescofo::variance `float` : set the variance parameter of the inference algorithm.
- antescofo::verbosity `int` : specify the level of system messages emitted during execution.
- antescofo::version (no argument) : print the version of the object and various build information on the 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..

Moving in score commands

These commands are message (from the Max or PD patch) to the *Antescofo* object. However, they can also be issued as actions in the score itself (using the antescofo :: *xxx* syntax).

start (no argument): Sends initialization actions (before first event) and wait for follower.

play (no argument): Simulates the score (instrumental+electronics) from the beginning until the end or until stop.

startfromlabel (string) or **startfrombeat** (float): Executes the score from current position to position specified by the argument in accelerated more WITHOUT sending messages. Then waits for follower (or user input) right before this position.

scrubtolabel (string) or **scrubtobeat** (float): Executes the score from current position to position specified by the argument in accelerated more WITH sending messages up to (and not including) the specified position. And then waits for follower (or user input).

playfromlabel (string) or **playfrombeat** (float): executes the score from current position to position specified by the argument in accelerated more WITHOUT sending messages, then PLAYs (simulates) the score from thereon.

playtolabel (string) or **playtobeat** (float): PLAY (simulate) score from current position up to position specified by the argument.

gotolabel (string) or **gotobeat** (float): Position yourself on position specified by the argument without doing anything else.

4.7 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).

Chapter 5

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 the 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.

5.1 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.

5.1.1 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 3.3.

With the local tempo, you can create, for example, an accelerando. In the next example, we use a variable as a local tempo and we control this variable with a curve (see section 5.5). With that, we can write a group where all durations are equal. It's the variation of the local tempo variable who create the accelerando.

```
curve tempVariation @grain := 0.05s
      { $localtemp
            {           { 60 }
                      1 { 120 }
            }
      }

group G @tempo := $localtemp
      {
            action1
            1/4 action2
            1/4 action3
            1/4 action4
            1/4 action5
            1/4 action6
            1/4 action7
            1/4 action8
      }
```

5.1.2 Attributes of Group and Compound Actions

Synchronization (cf. section 9.1)

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

and error strategies (cf. section 9.2)

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

can be specified for group but 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.

5.1.3 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[1].

5.1.4 Aborting a group

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

- using an abort action, see 4.4.1,
- 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:

[1] It is possible to kill a specific instance using the exec that refers to this instance, see 7.7 and 12.4.

```
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. 6.2.1 pp. 71). 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 (see below).

5.2 If, Switch: Conditional and Alternative

5.2.1 If: Conditional Actions

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. 6.5 page 78 that share a similar syntax.

5.2.2 Switch: Alternative Actions

Alternative actions extend conditional actions to handle several alternative. At most one of the alternative will be performed. They are two forms of alternative actions, without and with selector, which differs by the way the alternative to perform is chosen.

Alternative Action without Selector. An alternative action without selector is simply a sequence of cases guarded by expressions. The guards are evaluated in the sequence order and the action performed is the first case whose guard evaluates to true:

```
switch
{
  case e1: a1
  ...
  case en: an
}
```

can be rewritten in:

```
if (e1)  { a1 }
else {
  switch
  {
    case e2: a2
    ...
    case en: an
  }
}
```

If no guard e_i is true, then no action is performed. Notice that several actions can be associated to a case: they are launched as a group.

Here is an example where the evaluation order matter: the idea is to rank the value of the variable $PITCH. The following code

```
whenever ($PITCH)
{
   switch
   {
     case $PITCH < 80: $octave := 1
     case $PITCH < 92: $octave := 2
     case $PITCH < 104: $octave := 3
   }
}
```

uses a switch to set the variable $octave for some value each time $PITCH is updated for a value below 104 (for the whenever construction, see below).

Note that the actions associated to a `case` 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 alternative. And as the other actions, an alternative action can be prefixed by a delay.

Alternative Action with a Selector. In this form, a selector is evaluated and checked with each guard of the cases:

```
switch (s)
{
  case e1: a1
  ...
  case en: an
}
```

The evaluation proceeds as follow: the selector s is evaluated and then, the result is checked in turn with the result of the evaluation of the e_i:

- If e_i evaluates to a function, this function is assumed to be a unary predicate and is applied to s. If the application returns a true value, the action a_i is performed.
- If e_i is not a function, the values of s and e_i are compared with the == operator. If it returns a true value, the action a_i is performed.

The evaluation start with e_0 and stops as soon as an action is performed for one of the e_i. If no guard checks true, no action is performed.

For example:

```
switch ($x)
{
  case 0:
    $zero := true
  case @size:
    $empty := false
    $zero := false
}
```

checks a variable $x and sets the variable $zero to true if $x equals 0 or 0.0 (because 0.0 == 0) and sets the variable $empty and $zero to false if $x refers to an empty tab, to an empty map or to a scalar value (because function @size returns an integer which is 0 only if its argument is an empty tab or an empty map).

5.3 Loop: Sequential iterations

The `loop` construction

```
loop optional_label period { loop_body }
```

is similar to group but instead of being performed once, the actions in the loop body are iterated depending on a period specification giving the time elapsed between two loop iterations:

```
Loop L 0.5
{ print $NOW }
```

will print 0 0.5 1 1.5 ...

If the period is shorter than the duration of the body of the loop, successive iterations will overlap (the instances of the loop body are evaluated as independent groups):

```
$i := 0
Loop L1 1
{
  @local $j
  $j := $i
  $i := $i+1
  print "start" $j
  2 print "stop" $j
}
```

will print:

```
start 0
start 1
stop 0
start 2
stop 1
stop 2
```

Here, when the body of the loop L is instantiated, the global variable $i is copied in the local variable $j: $i can be updated without affecting the current loop bodies. The loop period is 1 and the duration of the body is 2. So the printing are interleaved. Notice that the local variable j is local to a loop body instantiation (they are as many $j as concurrent loop bodies).

The overlapping of two iterations of the loop body can be avoided, see below p. 52.

Loop Period. The period of a loop is an expression evaluated at each iteration. So the duration between two iterations can change as the time progress.

The period expression is a duration, *i.e.*, it can be absolute or relative.

```
$period := 1
Loop $period s
{
    print $NOW
    0.5 $period := $period + 1
}
```

will print 0 1 3 6 10 15 ...

When the loop is launched at time 0 second, the body is also launched for the first time and, in parallel, the next iteration is scheduled with the current value of the period (which at this time is 1 second). A 0 is printed. After 0.5 beat, the variable $period is incremented. At date 1 second, the period for the next iteration is evaluated (to 2) and the second iteration is launched (printing a 1). So after 1+2 seconds, the third iteration takes place and print a 3, etc.

In addition, the period expression can evaluate to a vector: in this case, the elements of the vector are the successive periods of the loop:

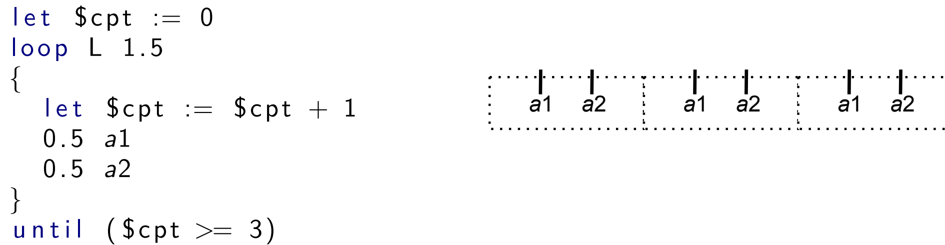
```
$p := [100, 200, 400, 800]
Loop $p ms
{   print $NOW }
```

will print 0 0.1 0.3 0.7 1.5 1.6 ...

Note that the periods are taken cyclically in the vector. The ms specification after the period indicates that the period is given in millisecond.

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.

Avoiding Overlapping Iterations of a Loop body. As mentioned above, two iterations of a loop body may overlap. In some case this is not the intended behavior: the previous iteration must be stopped before starting the new iteration of the loop body. This is achieved by specifying the @exclusive attribute for the loop: with this attribute, the previous iteration and its eventual childs are aborted (see 36). For instance, the program

```
$i := 0
loop 1 @exclusive
{
    @local $id
    $i := $i + 1
    $id := $i

    loop 0.25 { print iteration $id at $NOW }
}

2 antescofo::killall
```

will print the trace at the left. Without the @exclusive attribute, the trace is given on the right:

```
iteration 1 at 0.0
iteration 1 at 0.25
iteration 1 at 0.5
iteration 1 at 0.75
iteration 2 at 1.0
iteration 2 at 1.25
iteration 2 at 1.5
iteration 2 at 1.75
iteration 2 at 2.0
```

```
iteration 1 at 0.0
iteration 1 at 0.25
iteration 1 at 0.5
iteration 1 at 0.75
iteration 2 at 1.0
iteration 1 at 1.0
iteration 1 at 1.25
iteration 2 at 1.25
iteration 1 at 1.5
iteration 2 at 1.5
iteration 1 at 1.75
iteration 2 at 1.75
iteration 2 at 2.0
```

Notice that without the @exclusive attribute, there are two iterations of the loop body that execute the print command at the same date. With the @exclusive attribute, each iteration of the loop body occurs at disjoint time interval.

See also section B.13 page 189 for the management of actions that takes place at the same date.

5.4 Forall: Parallel Iterations

The loop 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 map 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
```

5.5 Curve: Continuous Actions

Many computer music controls are by nature continuous. Curves in *Antescofo* allow users to define such actions and to delegate the rest of the hard work to *Antescofo* taking care of correct arrival and interpolations between parameters. The curve construction allows the definition of continuously sampled actions on break-points and detailed control of the interpolation between them. Curves are defined by a sequence of break points and their interpolation methods along with specific attributes. As time passes, the curve is traversed and the corresponding action fired at the sampling point. Curves can be scalar (one-dimensional) or vectoriel (multi-dimensional).

We introduce the Curves[2] starting with a simplified and familiar syntax of linear interpolation and move on to the complete syntax and showcase details of Curve construction.

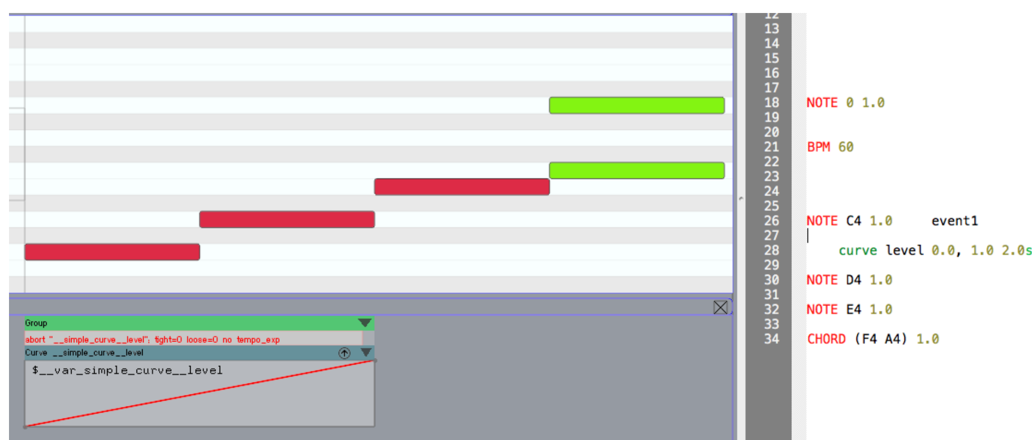
5.5.1 Simplified Curve Syntax

The simplest continuous action to imagine is the linear interpolation of a scalar value between a starting and ending point with a duration, similar to *line* objects in Max and Pd. The time-step for interpolation in the simplified curve is 30 milli-seconds and hard-coded. This can be achieved using the simplified Curve syntax as shown in Figure 5.1 below.

```
Curve level 0.0, 1.0 2.0s
```

In this example, the curve command constructs a line starting at 0.0, going to 1.0 in 2.0 seconds and sending the results to the receiver object "level". The initial point 0.0 is separated by a comma from the destination point. Destination point consists of a destination value (1.0) and the time to achieve it (2.0*s* in this case).

Figure 5.1 Simplified Curve syntax and its realisation in *Ascograph*



Another facility of *Simplified Curves* is their ability to be chained. The score excerpt in Figure 5.2 shows the score in Figure 5.1 where a second call to curve is added on the third note. This new call does *not* have a starting point and only has a destination value:

```
Curve level 0.5 1.0
```

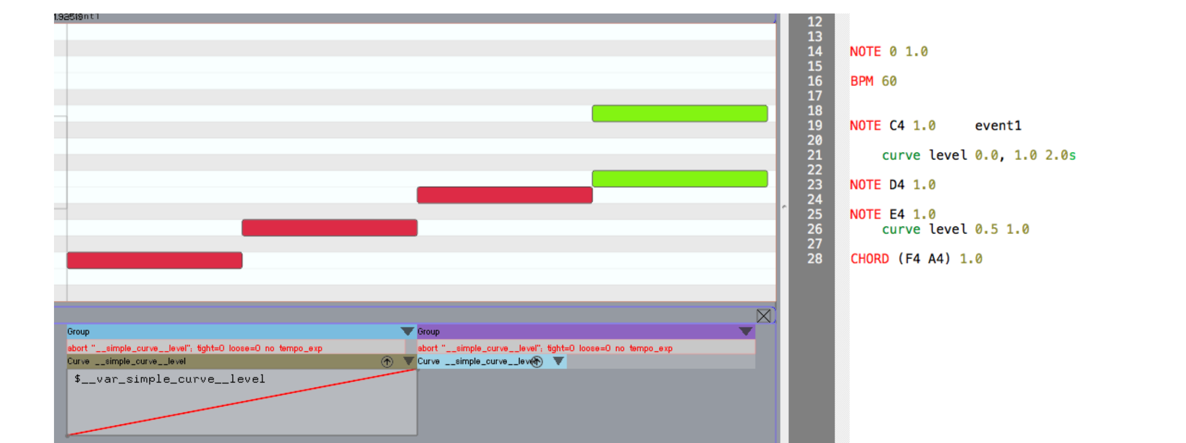
[2] Curve can be edited graphically using the *Ascograph* editor.

Both Curves also act on the same receiver "level". This means that during performance, the second curve will take on from whatever value of the prior curve and arrives to its destination (here 0.5) at the given time (here 1.0 beat).

Note that the second curve in Figure 5.2 can not be visualised by *Ascograph*. This is because its starting point is a variable whose value is unknown and depends on where and when the prior curve arrives during performance. Moreover, by calling simplified curves as above you can make sure that the first curve does not continue while the second is running. This is because of the way *Simplified Curves* are hard-coded. A new call on the same receiver/action will cancel the previous one before taking over.

The reason for the malleability of *Simplified Curves* is because they store their value as a variable. A new call on the same receiver aborts prior call and takes the latest stored value as departing point if no initial point is given. You can program this yourself using the complete curve syntax.

Figure 5.2 Simplified Curve chain call



The simplified curve is thus very similar to `line` object in Max or PD. This said, it is important (and vital) that the first call to *Simplified Curve* has an initial value otherwise the departing point is unknown and you risk receiving *NaN* value until the first destination point!

The simplified Curve command hides several important properties of Curves from users and are there to accelerate calls for simple linear and scalar interpolation. For example, the time-step for interpolation in the above curve is 30 milli-seconds and hard-coded. A complete curve allows adjusting such parameters, having multi-dimensional interpolations, complex actions, and more. We detail the complete curve syntax hereafter.

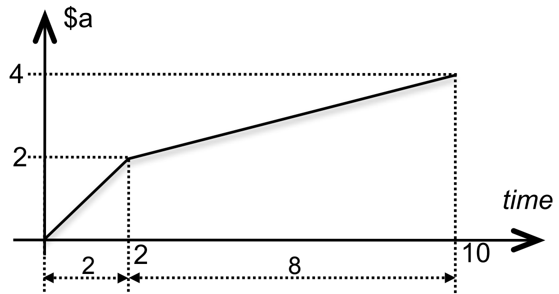
5.5.2 Full Curve Syntax

Curves are defined by breakpoint functions (BPFs) over a variable which is by itself used inside the @Action attribute. The Action attribute determines what the Curve must do upon each interpolation point. The BPFs themselves are consist of a delay-point, destination value or vector and interpolation type. An additional @grain attribute defines the global interpolation time-step of the curve and takes general time values of *Antescofo*.

The example below shows a simple non-linear curve. The Curve has the name "C" and

starts at a value of 0. Two beats later, the curve reaches 2 and ends on 4 after 8 additional beats. Between the breakpoints, the interpolation is linear, as indicated by the string " linear " after the keyword @type. Linear interpolation is the default behaviour of a curve (hence it can be dismissed).

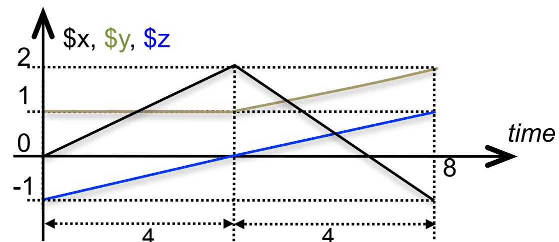
```
curve C
   @action := { level $a } ,
   @grain := 0.1
{
   $a
   {
           { 0 } @type "linear"
         2 { 2 } @type "linear"
         8 { 4 }
   }
}
```



In the above example, the curve is defined over variable $a. This value is updated at a time-rate defined by attribute @grain (can be absolute time or relative). Each time $a is updated, the @@action block is triggered which can make use of $a.

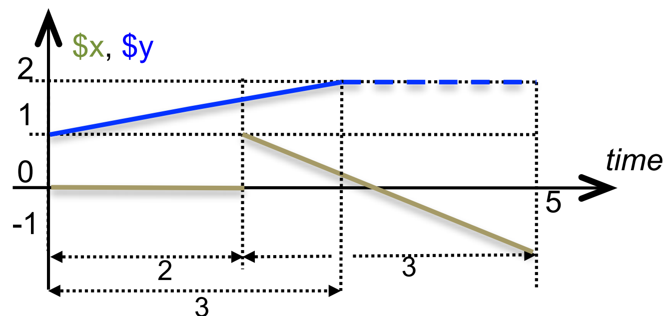
It is easy to apply curves on multi-dimensional vectors as shown in the following example:

```
curve C
{
   $x , $y , $z
   {
        {  0 , 1 , −1 }
      4 {  2 , 1 ,  0 }
      4 { −1 , 2 ,  1 }
   }
}
```



In the above example, all values in the three-dimensional vector share the same break-point and the same interpolation type. It is also possible to split the curve to multiple parameter clauses as below:

```
curve C
{
   $x
   {
        { 0  }
     2  { 0 }
     0  { 1  }
     3  { −1 }
   }
   $y
   {
        { 1 }
      3 { 2 }
   }
}
```

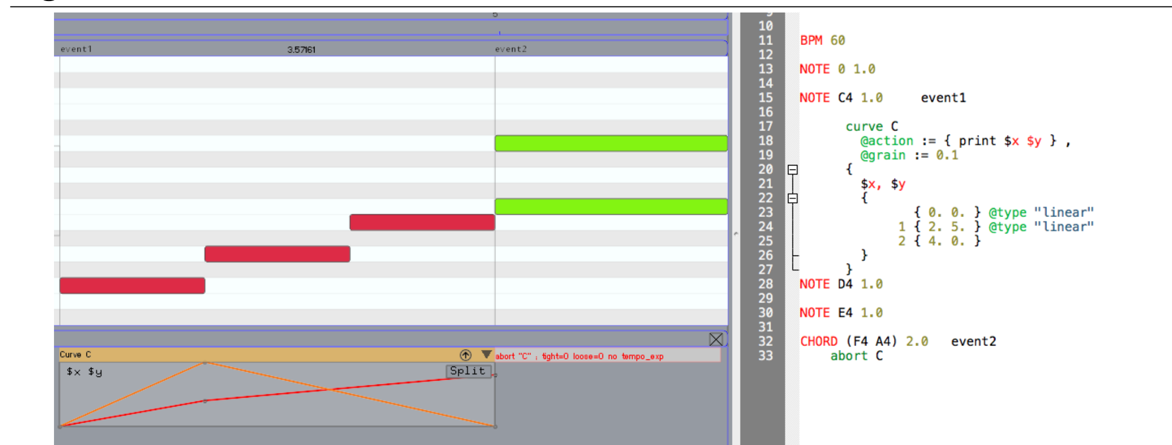


In the above example, curve parameters $x and $y have different breakpoints. The breakpoint definition on $x shows how to define a sudden change on step-function with a zero-delay value.

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.

Figure 5.3 shows a simple 2-dimensional vector curve on *Ascograph*. Here, two variables $x and $y are passed to the action. They share the same breakpoints but can be split within the same curve. The curve is also being aborted on "event2" by calling its name.

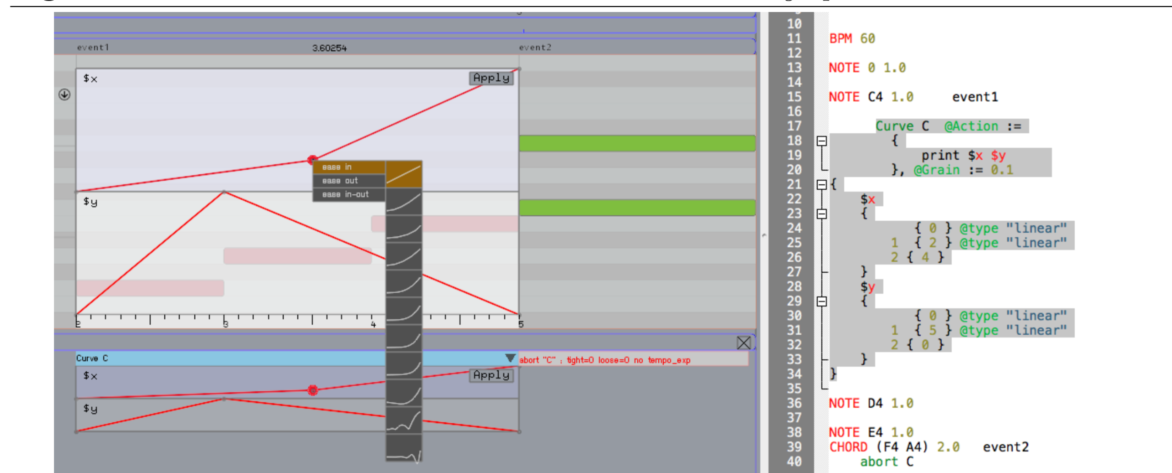
Figure 5.3 Full 2d Curve



Using *Ascograph*, you can graphically interact with curves: moving breakpoints vertically (changing their values) and horizontally (time position) by mouse, assigning new interpolation schemes graphically (control-click on break-point), splitting multi-dimensional curves and more. For many of these operations on multi-dimensional curves, each coordinate should be represented separately. This can be done by pressing the SPLIT button on the Curve box in *Ascograph* which will automatically generate the corresponding text in the score. Each time you make graphical modifications on a curve in *Ascograph*, you'd need to press *APPLY* to regenerate the corresponding text.

Figure 5.4 shows the curve of figure 5.3 embedded on the event score, split and in course of being modified by a user.

Figure 5.4 Full 2d Curve embedded on Event Score in *Ascograph*



In the following sections we will get into details of Curve attributes namely Actions, timing, and interpolation methods.

5.5.3 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 sent to receive identifier "print" and two beats later *action1* will be fired and one additional beat later *action2* will be fired.

If the attribute @action is missing, the curve simply assigns the variables specified in its body. This can be useful in conjunction with other parts of the code is the values is reused in expressions or other actions.

5.5.4 Step, Durations and Parameter Specifications

In the simple example above, the time step or grain size ((@grain) attribute) and breakpoints' delays are expressed in relative time. But they can be also expressed in absolute time and mixed arbitrarily (*e.g.* the time step in second and duration in beats, and there also is possible to mix duration in beats and in seconds).

Grain size, duration, as well as the parameters, can also be expressions. These expressions are evaluated when the curve is fired.

The sampling rate or grain size can be as small as needed to achieve perceptual continuity. However, in the MAX/PD environments, one cannot go below 1ms.

5.5.5 Interpolation Methods

The specification of the interpolation between two breakpoints is given by an optional string. The @type keyword is optional. By default, a linear interpolation is used. *Antescofo* offers a rich set of interpolation methods, mimicking the standard *tweeners* used in flash animation[3]. There are 10 different types:

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

[3]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

At the exception of the linear type, all the interpolations types comes in three "flavors" traditionally called *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 5.5. The corresponding interpolation keyword are listed below. Note that the interpolation can be different for each successive pair of breakpoints. The same interpolation methods are used for NIMs, cf. section 8.3.

"linear"

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

Programming an Interpolation Method. If your preferred interpolation mechanism is not included in the list above, 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.

5.5.6 Curve with a NIM

A NIM value (see section 8.3) can be used as an argument of Curve construction allowing to dynamically build breakpoints and their values as a result of computation. The syntax is the following:

```
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. Notice that, when a NIM is "played" by a curve, the first breakpoint of the NIM coincide with the start of the curve.

For example

```
$nim := NIM { ... }
; ...
Curve
@tempo := 30,
@grain := 0.1s,
@action := print $x
{ $x : $nim }
```

Any expression can be used which evaluates to a NIM. So, the following code plays a random NIM taken in a vector of 10 NIMs:

```
$nim1 := NIM { ...}
$nim2 := NIM { ...}
; ...
$nim10 := NIM { ...}

$tab := [ $nim1, $nim2, ..., $nim10 ]
; ...
Curve
@tempo := 30,
@grain := 0.1s,
@action := print $x
{ $x : $tab[@rand(11)] }
```

A typical situation is to play a NIM choosed in a NIM collection with a variable lenght duration. It can be done with the process ::NIMplayer.

```
$Nim1 := NIM { 0. 0.,0.05 1 "quad",
            0.1 0.2 "quad_out",
            0.85 0. "cubic" }

$Nim2:= NIM { 0. 0.,0.05 1.,
            0.9 1.,
            0.05 0. }

@proc_def ::NIMplayer($NIM,$dur)
    {
      curve readNIM @grain := 0.02s, @action := print ($NIM($x))
            {$x
                  {      { 0.}
                  $dur   { 1.}
                  }
            }
    }

NOTE 69 4
::NIMplayer($Nim1,4)
```

The play of the NIM is controled by a curve. We assume that the NIM is defined between 0 and 1, but see functions @min_key and @max_key if you have to adapt to other range.

Figure 5.5 *Various interpolation type available in an* Antescofo Curve *and* NIM. The label xxx[0] corresponds to the ease “in”, that is to the type "xxx_in" or equivalently "xxx"; the xxx[1] corresponds to the ease “out”, *i.e.* type "xxx_out"; and the label xxx[2] corresponds to the ease “in_out”, *i.e.* type "xxx_in_out";.

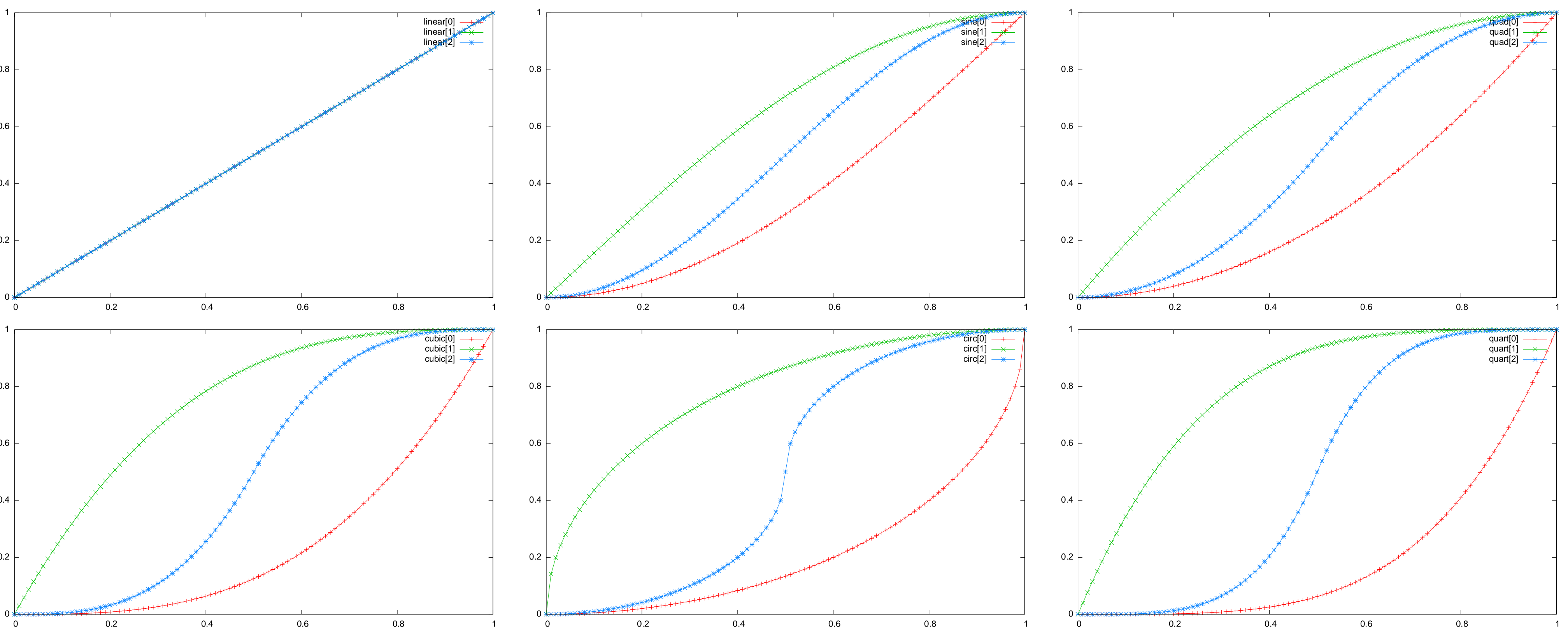
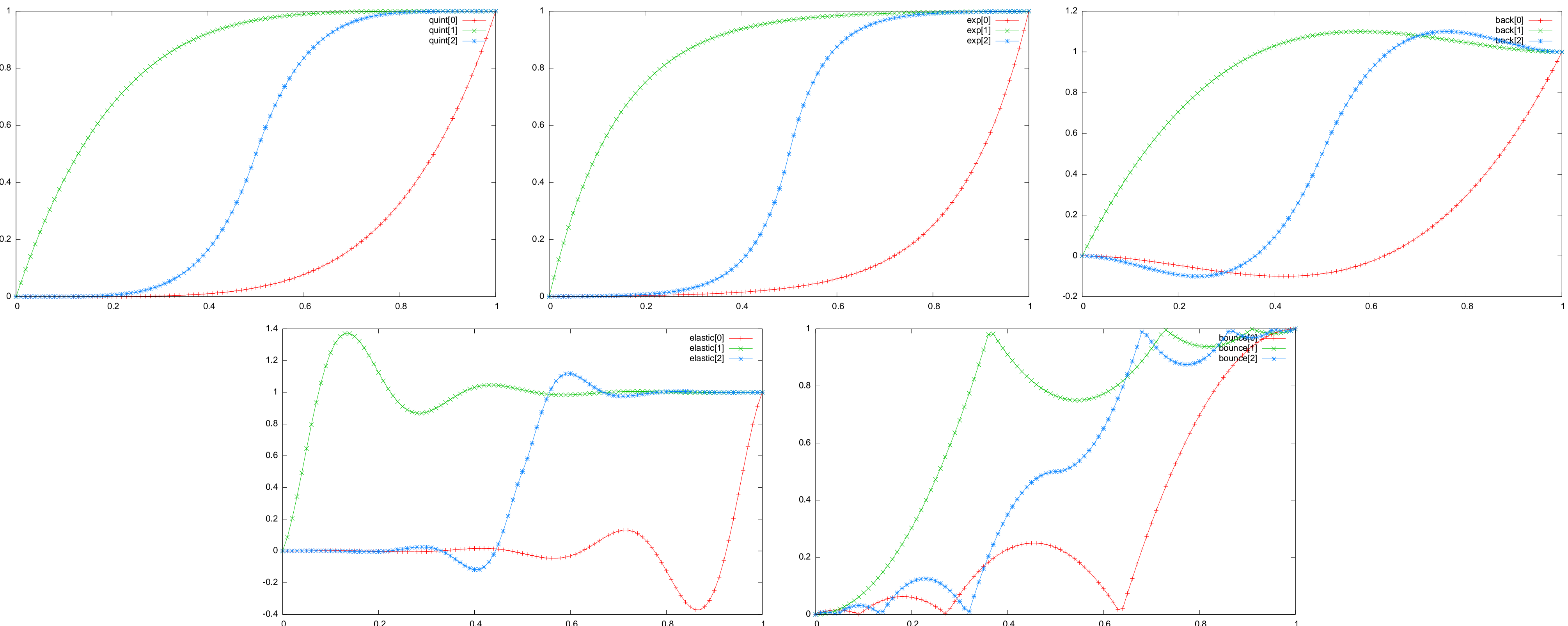


Figure 5.6 *(cont.) Various interpolation type available in an* Antescofo Curve *and* NIM. The label xxx[0] corresponds to the ease “in”, that is to the type "xxx_in" or equivalently "xxx"; the xxx[1] corresponds to the ease “out” and the label xxx[2] corresponds to the ease “in_out”.



5.6 Whenever: Reacting to logical events

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. An optionnal during clause can also appear.

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 5.6.1) or by its abort. After the firing of the whenever, and until its end, each time the variables of the `boolean_expression1` are updated, `boolean_expression1` is re-evaluated. We stress the fact that only the variables that appear explicitly in the boolean condition are tracked. We say that these variables are *watched* by the whenever. 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: that is, only when one of the variables that appears in the boolean expression is updated by an assignment elsewhere. To force the evaluation of the boolean expression when the whenever is fired, one can specify an @immediate attribute.

Notice also the difference with a conditional action (section 5.2): 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.

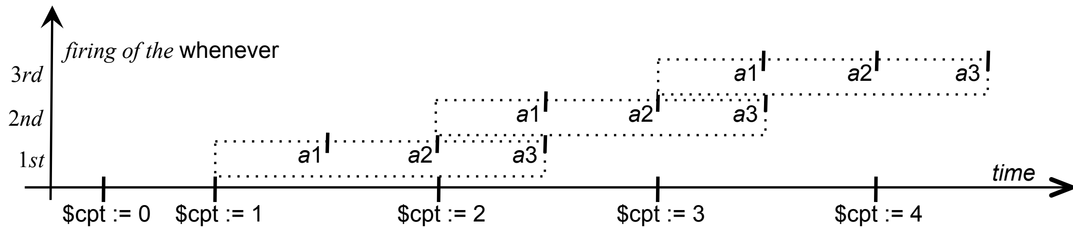
The whenever is a way to reduce and simplify the specification of the score particularly when actions have to be executed each time some condition is satisfied. It also escapes the sequential nature of traditional scores. Resulting actions of a whenever statement are not statically associated to an event 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:



Watching Restrictions. The whenever watchs variable, not values. It means that the whenever monitors the updates of the variables that appear in the logical condition. When a variable is updated, the logical condition is (re)evaluated to decide (if true) to lauch the whenever body.

So the update of an element in a tab cannot trigger a whenever, see 8.4 page 99. A whenever cannot watch a local variable referred through the dot notation. A whenever cannot watch a special variable, that is $NOW and $MYSELF and $THISOBJ. A whenever cannot watch the local variables nor the argument of a function. These restrictions ensure that *Antescofo* score remain causal and efficiently implementable.

Avoiding Overlapping Instances of a Whenever body. The activation of a whenever fire a new group and two such group may overlap in time. Sometimes it is necessary to avoid this behavior. It can be done using an explicit abort:

```
$last_activation := 0
whenever (...)
{
     abort $last_activation
     $last_activation := $MYSELF
     ...
}
```

which kill the previous instance of the body, if any. The same behavior can be obtained using the @exclusive attribute:

```
whenever (...) @exclusive
{
     ...
}
```

The @exclusive attribute may also be used on loops, see page. 52 and page 189 for the management of actions that takes place at the same date.

5.6.1 Stopping a whenever

A during and/or an until clause can be defined for a whenever (see sect. 4.4.1). 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" $X } 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" $X
}
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. 4.4.1.

5.6.2 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*:

$$\text{Start} \to \text{W1} \to \text{W2} \to \text{W1} \to \text{W2} \to \text{W1} \to \text{W2} \to \text{W1} \to \text{W2} \to \text{W1} \to \quad \ldots$$

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 := 10+1 → W1
→ 3: $y := 2+1 → W2
→ 4: $x := 11+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.

Chapter 6

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. Expressions can also be used inside the body of messages.

6.1 Values

Expression are evaluated into values at run-time (or live performance). They are two kind of values:

- *scalar* or *atomic values*, described in chapter 7, include *undefined value*, booleans, integers, floats (IEEE double), symbols, function definitions, process definitions and running processes (*exec*);
- *Data Structures* or *compound values* like strings (sequence of characters), tabs (tables, vectors), maps (dictionaries), and interpolated functions (NIM). Such data structures, described in chapter 8, can be arbitrarily nested, to obtain for example a dictionary of vector of interpolated functions.

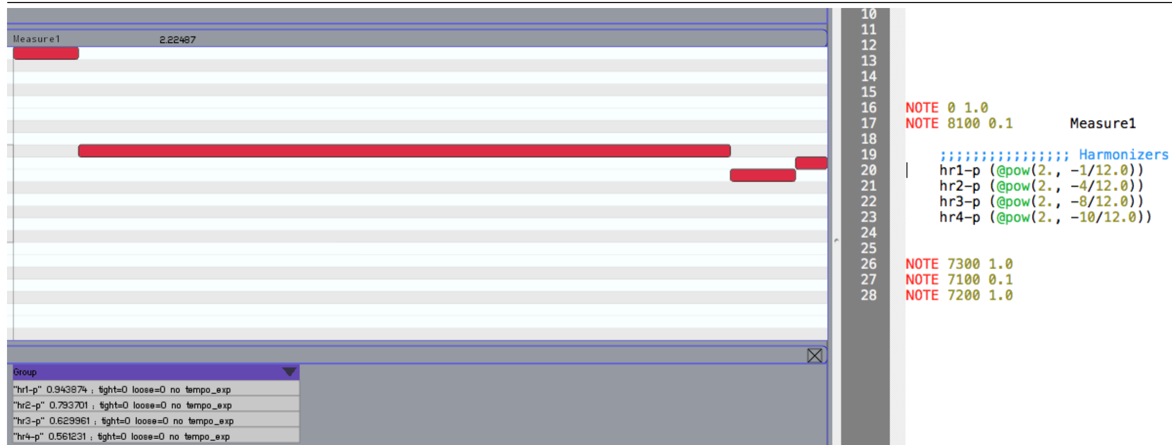
Figure 6.1 shows a simple score excerpt employing a simple expression and value. The text score on the right declares four expressions to be sent to receivers "hr1-p" to "hr4-p" (harmonisers) whose final value is being converted from semi-tones to pitch-scale factor. The *Ascograph* graphical representation shows their evaluation.

In this example we are able to the final values since the arguments of the expression are static. If a variable was to be used, the expression would stay intact in the *Ascograph* visual representation to be evaluated at run-time. Variables will be discussed in section 6.2.

A compound value v is *mutable* data structure: you can change an element in the data structure and this does not change the value itself. It means that the variables referring to the value v will refer to the changed data structure. On the contrary, atomic values are *immutable*: you cannot change an atomic value, you can only build a new atomic value.

Predefined functions can be used to combine values to build new values. The programmer can defines its own functions, see paragraph 11.

Figure 6.1 Example of simple expression and its value realisation in *Ascograph*



Dynamic Typing. From a user's perspective, value types in *Antescofo* do not need to be specified. They are checked during creation. They can be anything available to *Antescofo* and described in this chapter (int, float, symbol/string, tab, or map).

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 13.2 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.

6.2 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 stored in the box:

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

The two forms are equivalent. An assignment is an action (see sect. 4.3) and as other action, it can be done after a delay. We stress that variable assignments are actions, not expressions, so they cannot appear directly in the body of a function (they can appear indirectly, through an EXPR construct, see sect. 6.6).

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.

User variables are assigned within an augmented score using Assignment Actions (see section 4.3). However, they can also be assigned by the external environment, using a dedicated API:

- the reception of an OSC message, sect. 4.2.2;
- the setvar message, sect. 4.3;
- the function @loadvar, page 152.

see also the section 6.2.4.

6.2.1 Histories: Accessing the Past Values of a Variable

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 page 73. 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.

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.

6.2.2 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 5. 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 {
  @local $x
  $x := 2
  print "local var $x: " $x
}
print "global var $x: " $x
```

will print

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

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 by the children of G but no more.

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, beside documentation, 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 nth past value of a variable is accessed using the *dated access* construction (see 6.2):

```
[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.

The special form @history_length($x) returns the maximal length of the history of a variable.

6.2.3 History reflected in a Map or in a Tab.

The history of a variable may be accessed also through a map or a tab. Three special functions are used to build a map (resp. a tab) from the history of a variable:

- @map_history($x) returns a map where key n refers to the 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.

- @tab_history($x) is similar to the previous function but returns a tab where ith element refers to the the $n-1$ to the last value of $x.

- @map_history_date($x) returns a list where element n is the date (physical time) of $n-1$ to the last update of $x. The previous remark on the map size applies here too.
- @tab_history_date($x) builds a tab (instead of a map) of the dates in physical time of the of updates of the var $x.
- @map_history_rdate($x) returns a list where element n is the relative date of $n-1$ to the last update of $x. The previous remark on the map size applies here too.
- @tab_history_rdate($x) builds a tab (instead of a map) of the dates in relative time of the of updates of the var $x.

These six functions 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.

Plotting the history of a variable. The history of a variable can be plotted in absolute or in relative time using the command @plot and @rplot. These two functions are special forms accepting only a list of variables as arguments. They return true if the plot succeeded and false elsewhere.

If there is only one argument $x, the referred values can be a tab (of numeric value) and each element in the history of the tab is plotted as a time series on the same window. If they are more than one argument, each variable must refer to a numeric value and the time series of the variables values are plotted on the same window.

Note that only the values stored in the history are plotted : so usually one has to specify the length of the history to record, using a @global or @local declaration (see. page 73).

The @plot and @rplot special forms expand to a call to the function @gnuplot[1]. For example, the expression @plot($x, $y) expands into

```
@gnuplot( "$x", @history_tab_date($x), @history_tab($x),
          "$y", @history_tab_date($y), @history_tab($y) )
```

See description of @gnuplot in the annex page 148.

6.2.4 Accessing a Local Variable "from Outside its Scope of Definition"

A local variable can be accessed in its scope of definition, or from one of its child scopes, using its identifier. It is possible to access the variable from "outside its scope" using the :: notation or the dot notation through an *exec*. Here, "outside" means "not in the scope of definition nor in one of its children". Beware that accessing a local variable from outside its definition scope:

- is correct only within the lifetime of the variable,
- does not extend the lifetime of the variable which is still bound to the lifetime of its definition scope and its children.

[1] The gnuplot program is a portable command-line driven graphing utility for Linux, MS Windows, Mac OSX, and many other platforms. It must be installed in the system, Cf. http://www.gnuplot.info.

Beware that if the scope of definition of the variable is not currently running at the time of the access, an undefined value is returned and an error is signaled. Else, if there is no variable with this identifier locally defined in the scope, then the variable is looked up in the enclosing scope. The process is iterated until the top-level is reached. At this point, if there is no global variable with the specified identifier, an undefined value is returned and an error is signaled.

As preiously mentionned, a variable can be assigned from "outside *Antescofo*", see:

- the reception of an OSC message, sect. 4.2.2;
- the setvar message, sect. 4.3;
- the function @loadvar, page 152.

The setvar command can be used only for global variable. But local variable can be the target of the two other mechanisms.

The :: Notation. Here is an example of the :: notation:

```
Group G {
  @local $x
  $x := "G"
  loop 1 { print tic } during [10#]
}
Group H {
  @local $x
  $x := "H"
  loop 1 { print toc } during [10#]
}

1
print (G::$x) ; refers to $x in group G
print (H::$x) ; refers to $x in group H

20
print (G::$x)
// at this point in time, expression G::$x raises an error
// and returns undef  because group G does not exist anymore
```

The left hand side of the :: operator specifies the label of the group introducing the scope where the local variable is declared. The identifier of the local variable is specified in the right of :: operator.

Different groups may share the same label and may be active at the same time. In this case, the variable is looked-up in one of the group currently running (chosen at random, which is probably not very useful).

It is not possible to assign a local variable using the :: notation:

```
let G::$x := 33    // BAD
```

raises a syntax error.

The Dot Notation. To access the variable defined in one specific instance of a group or more generally of a compound action introducing a scope (whenever, loop, process call, etc.), one must use the dot notation, through the *exec* referring to this instance. *Exec* are introduced in section 7.7 and the dot notation is detailed in section 12.5.

It is possible to assign a local variable through the dot notation:

```
$p := ::P()
$p.$x := 33 // assign the local variable $x in the process ::P
```

See sect. 12.5. The expression at the left of the dot operator may be more complex than just a variable. In this case, the let keyword is mandatory:

```
$p := [ ::P() | (10) ]
let $p[2].$x := 33
```

The first line launch 10 instances of process ::P. The second line set the local variable $x of the third instance of ::P.

6.3 Internal Antescofo Variables

Internal variables are those that belong to Antescofo and are updated automatically by the system. They are useful for interacting with, for example, machine listener during performances and creating interactive setups:

- $BEAT_POS is the position of the last detected event in the score. See also p. 109.
- $DURATION is the duration of the last detected event.
- $ENERGY is the current *normalised energy* of the audio signal from the listening machine. The returned value is always between 0.0 and 1.0 and is equivalent to the *Calibration Output* of the Antescofo object in Max and Pd. **NOTE:** The $ENERGY variable is updated with high frequency (equal to the analysis hop size). Use it with care inside processes and *Whenever* constructs.
- $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.
- $PITCH is the pitch (in MIDI Cents) of the current event. This value is well defined in the case of a Note and is not meaningful[2] for the other kinds of event.
- $LOCAL_TRANSPOSITION returns the transposition value (in MIDI CENTS) set by the *transpose* keyword inside the score. This value can change in the score depending on how many times *transpose* keyword is employed. Note that it also returns transposition value on Silences, ineffective but useful for reporting on a region.
- $GLOBAL_TRANSPOSITION returns the global transposition value (in MIDI CENTS) set by the *scoretranspose* methods sent to Antescofo objects. This value is global and unique to a score and reset to zero if a new score is loaded.

[2] in the current version of *Antescofo*

- $RCNOW is the date in relative time (in beats) of the "current instant". It can be interpreted as the current position in the score. This position is continuously updated between the occurence of two events as specified by the current tempo. Thus, if the next event occurs late w.r.t. its anticipated date, the value of $RCNOW will jumps backward.

- $RNOW is the date in relative time (in beats) of the "current instant". It can be interpreted as the current position in the score. This position is continuously updated between two events as specified by the current tempo. But, contrary to $RCNOW, the increase stops when the position in the score of the next waited event is reached and $RNOW is stuck until the occurrence of this event or the detection of a subsequent event (making this one missed). Thus, $RNOW cannot decrease. See also p. 109.

- $RT_TEMPO represents the tempo currently infered by the listening machine from the input audio stream.

- $SCORE_TEMPO returns the tempo constant in the score at the exact score position where it is called.

Note that when an event occurs, several system variables are susceptible to change simultaneously. Notice that, as for all variables, they are *case-sensitive*.

6.3.1 Special Variables

These variable are similar to system variables, but they cannot be watched by a whenever:

- $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.

- $MYSELF denotes the *exec* of the enclosing compound action.

- $THISOBJ may appears in method definitions where it refers to the object on which the method is applied, or in the clauses of an object definition where it refers to the current instance.

6.3.2 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 5.6.2. 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.

6.4 Temporal Variables

Starting version 0.8, *Antescofo* provides a specialised variable capable of tracking its own tempo and enabling synchronisation of processes with such variables. They are defined using the `@tempovar` keyword. For example:

```
@tempovar $v(60, 1/2), $w(45,1)
```

defines two variables `$v` and `$w`. `$v` has an initial tempo of 60 BPM and periodic updates of 1/2 beats, whereas $w has an initial tempo of 45 BPM and expected periodic updates of 1.0 beat.

In addition to regular variables, a *temporal variable* stores the following internal information that can be accessed at any time:

```
$v.tempo        // the internal tempo of $v
$v.position     // current beat position of $v
$v.frequency    // frequency (period) of $v
$v.rnow         // relative time of $v as tracked by its internal agent
```

Such internal attributes can be changed at any time like regular variables.

The "tempo of the variable" is computed using the same algorithm used by the listening machine to track the tempo of the musician. Using temporal variables, it is possible to define *synchronisation strategies* of processes (groups, loops, proc, etc.) based on their progression (instead of *Antescofo*'s musician tempo), using the `@sync` attribute as below:

```
group @target[5s] @sync $v
{
        ; actions...
}
```

Temporal variables can be set by the environment (cf. sect. 4.3), allowing the easy tracking any kind of processes and the synchronization on it. Temporal variables can also be used to set synchronization coordination schemes different than that of followed human musician (see Section 9.1.4).

6.5 Operators and Predefined Functions

The main operators and predefined functions are sketched in chapters 7 and 8 together with the main data type involved. We sketch here some operators or functions that are not linked to a specific type. The predefined functions are listed in annex A p. 143.

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

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

returns the value `exp1` if `bool_exp` evaluates to `true` and else `exp2`. 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 `exp1` is evaluated, and similarly for false and `exp2`.

In the body of a function, a conditional can be written using the usual syntax:

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

see page 124.

`@empty` and `@size`. The `@empty` predicate returns true if its argument is an empty `tab`, `map` or `string`, and false elsewhere. 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, for a `tab` the number of elements, for a `nim`, the dimension of the nim and for a `string` the number of characters in the string.
- 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. 6.1 p. 70).

6.6 Action as Expressions

An action can be considered as an expression: when evaluated, its value is an *exec*, cf. sect 7.7. To consider an action as an expression, the action must usually be enclosed in an `EXPR{ }` construct.

The action is fired when the expression is evaluated. The returned *exec* refers to the running instance of the action and can be used to kill this running instance or to access the local variables of the action[3]. An atomic action (with a 0-duration run) returns the special exec `'0`.

To simplify the writing:

- The surrounding `EXPR{ }` is optional in the case of a process call (sect. 12.1);
- The `EXPR` keyword is optional in the right hand side of an assignment. For example:

  ```
  $x := EXPR { whenever (...) { ... } }
  ```

  is equivalent to

  ```
  $x := { whenever (...) { ... } }
  ```

- And the surrounding `EXPR{ }` is optional in the body of a function, for messages and assignation (sect. 11.1.4).

[3] See sect. 12.5 for the access to the local variables of a process instance but the same mechanism can be used to access any local variable introduced by a compound action through its *exec*.

Example. In the following example, a tab of 5 elements is created. Each element refers to a running loop:

```
$tab := [ EXPR{ Loop 1 { @local $u ... } } | (5) ]
```

Thus, one can kill the second instance with

```
abort $tab[1]
```

and one can access the local variable `$u` of the third instance through the dot notation (see 12.5):

```
$uu := $x[2].$u
```

6.7 Structuring Expressions

Writing large expressions can be cumbersome and may involve the repetition of common sub-expressions. Functions can be used to avoid the repeated evaluations of common sub-expressions. In addition, the body of a function is an extended expression, which enable a more concise and more clear specification of expression, see chapter 11.

6.8 Auto-Delimited Expressions in Actions

Expressions appear everywhere to parameterize the actions (and actions may appear in expression, cf. 6.6). This may causes some syntax problems. For example when writing:

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

there is a possible 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 because there is no other possible interpretation (the code fragment `1 (2)` is not a valid expression). Variables are another example of auto-delimited expressions.

Being auto-delimited is a complicated property. *Antescofo* accepts a simple syntactic 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. This avoid any ambiguities. Notice that *every expression between parenthesis is an auto-delimited expression.*

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

```
 $x + 3  print BAD   // syntax error: $x + 3 is not auto-delimited
($x + 3) print OK    // parenthetized expressions are always auto-delimited
```

To disambiguate our first example, we use also parenthesis:

```
print (@f (1))   // is interpreted as the print of one argument: (@f(1))
print (@f) (1)   // is interpreted as the print of two arguments: @f and 1
```

Chapter 7

Scalar Values

Antescofo offer a rich set of value types described in this chapter and the next one. The value type examined in this chapter, undef (the undefined value), bool (booleans), int (integers), float (double floating point values), fct (intensional functions), proc (processes) and exec (threads of execution) are *indecomposable values*. The value types examined in the next chapter are data-structures that act as containers for other values.

7.1 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 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[1]: ! 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.

7.3 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

[1] The character "!" can be the first letter of a symbol. So it is wise to leave a blank space between the logical operator and its argument.

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`.

7.4 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.

7.5 User-defined Functions

An *Antescofo* intentional function is a value that can be applied to arguments to achieve a function call. As the others kinds of values, it can be assigned to a variable or passed as an argument in a function or a procedure call. Looking functions as values is customary in functional languages like Lisp or ML. But *Antescofo* functions cannot be defined in a nested way, only at top-level like in C.

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 11 page 119. In an *Antescofo* expression, the @-identifier of a function denotes a functional value that can be used for instance as an argument of higher-order functions (see examples of higher-order predefined function in section 8.2 for map building and map transformations).

Some intentional functions are predefined and available in the initial *Antescofo* environment like the IEEE mathematical functions. See annex A for a description of predefined 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`.

Functions are described more in details in chapter 11.

7.6 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 7.5). 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.

They are two main operations on proc values :

- "calling the corresponding process", see section 12 ;
- "killing" all instances of this process, see section 12.4.

Processes are described more in details in chapter 12.

7.7 Exec Value

An *exec value* refers to a specific run of a compound action. Such value are created when a process is instantiated, see section 12, but also when the body of a loop or of a whenever is spanned. This value can be used to abort the corresponding action. It is also used to access the values of the local variables of this action.

They are several ways to retrieve an *exec*:

- The *special variable* $MYSELF always refers to the *exec* of the enclosing compound action.
- The *special variable* $THISOBJ always refers to the object referred by a method (in a method definition) or in the clauses of an object definition.
- A process call returns the *exec* of the instance launched (see sect. 12.1).
- Through the "action as expression" construct, see sect. 6.6.

The action run referred by the *exec* may be elapsed or still running. In the former case we say that the exec is *dead* and *active* in the latter case. For example, the *exec* returned by evaluating an atomic action (with a 0-duration run) returns the special exec '0 which is always dead. A conditional construct can be used to check the status of the *exec*:

```
$p := ::proc(...)
...
if ($p)
{ /* performed if the instance of ::proc is still running */ }
else
{ /* performed if the exe is dead */ }
```

Exec values can be used as an argument of an abort command. An abort command on a dead *exec* does nothing (and does not rise an error). Notice that an *exec* refers to a specific instance of an action. So used in an abort command, it abort solely the referred instance while using the label of an action, will abort all the running instances of this action.

Exec can also be used to access the local variables of the referred compound action. This is mostly useful for processes (cf. sect. 12.5).

Accessing a Local Variable Through an *exec*. Accessing a local variable through an *exec* relies on the *dot notation*: the left hand side of the infix operator "." must be an expression referring to an active *exec* and the right hand side is a variable local to the referred *exec*.

Accessing a local variable through the dot notation is a dynamic mechanism and the local variable is looked first in the instance referred by the *exec*, but if not found in this context, the

variable is looked up in the context of the *exec* itself, *i.e.* in the enclosing compound action, and so on, until it is found. If the top-level context is reached without finding the variable, an undef value is returned and an error message is issued. See sect; 12.5 for an example.

The reference of a local variable using the dot notation can be used in an assignment, see sect. 12.5 for an example involving a process instance (but this feature works for any *exec*).

Chapter 8

Data Structures

Antescofo currently provides string, map, tab and nim data structures described in this section. They correspond to *sequences of characters*, *dictionaries* with arbitrary keys and values, *vectors* holding possibly heterogeneous values and *interpolated functions* defined by a sequence of breakpoints.

8.1 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.

Characters in a string can be accessed as if it was a tab (see tab on sect. 8.4). Characters in a string are numbered starting from 0, so:

```
"abc"[1] ->"b"
```

Note that the result is a string with only one character. There is no specific type dedicated to the representation of just one character. Notice also that strings are *immutable values*: contrary to tabs, it is not possible to change a character within a string. A new string must be constructed.

The + operator corresponds to string concatenation:

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

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

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

will output abc33.

Several predicates exists on strings: @count, @explode, @find, @is_prefix, @is_subsequence, @is_suffix, @member, @occurs, @reverse. See the description in the annex A.

8.2 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), ... }
```

The type of the keys and of the values is not necessarily homogeneous. So a map may include an entry which associate a string to a number and another entry which associate a map to a string, etc.:

```
map{ (1, "one"),
     ("dico", map{ ("pi", 3.14), ("e", 2.714), ("sqr2", 1.414) }),
     (true, [0, 1, 2, 3])
   }
```

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 7.5.

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 the same way you apply a predefined or a user-defined intentional 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 $\{1, \ldots, n\}$ 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.

The function @domain applied on a map returns the tab of its keys. The order of the keys in the returned tab is irrelevant. The function @range applied on a map returns the map of its values. The order in the returned tab is irrelevant. For example

```
@range({MAP{("zero", 0), ("0", 0), ("one", 1)}) ->[0, 0, 1]
```

The predicates @count, @find, @member, and @occurs work on maps: member(m, v) returns true if there is a key k such that m(k) == v and returns false elsewhere; count(m, v) returns the number of key k such that m(k) == v; occurs(m, v) returns the first key k (for the $<$ ordering) such that m(k) == v if such a key exists, else the undefined value; and find(m, f) return the first key k (for the $<$ ordering) such that f(k, v) returns true and the undef value if such entry does not exists.

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). Alternatively, the overloaded function @insert can be used (@insert can be used on tabs and maps; @add_pair is just the version of @insert specialized for maps).
- @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 $m'(x) = f(m(x))$.
- @merge combines two maps into a new one. The operator is asymmetric, that is, if $m = \text{merge(a, b)}$, then:

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

- @remove(m, k) removes the entry of key k in map m. If k is not present in m, the command has no effect (no new map is created). This function is overloaded and apply also on tabs.

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

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

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

  $$map\{ \ldots, (p_k, q_k), \ldots \}$$

  if it exists a k such that

  $$\$p(k) = p_k \text{ and } \$q(k) = q_k$$

- @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:

  $$map\{ (1, v_1), (2, v_2), \ldots, (p, v_p), \}$$

  the following list is build:

  $$map\{ (1, v_p), (2, v_{p-1}), \ldots, (p, v_1), \}$$

Score reflected in a Map. Several functions can be used to reflect the events of a score into a map[1]:

- @make_score_map 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).
- @make_duration_map 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).
- @make_label_pos this function, and the following, return a map whose key are the labels of the events and the value, the position (in beats) of the events.
- @make_label_bpm return a map associating the event labels to the BPM at this point in the score.
- @make_label_duration returns a map associating to the event of a label, the duration of this event.
- @make_label_pitches returns a map associating a vector of pitches to the label of an event. A NOTE corresponds to a tab of size 1, a CHORDS with n pitches to a tab of size n, *etc.*

These functions take two optional arguments (start, stop) to restrict the part on the score on which the map is built. The map contains the key corresponding to events that are in the interval [start,stop] (interval in relative time). Called with no arguments, the map are built for the entire score. With one argument start, the map are build for teh label or the position strictly greater than start.

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 6.2.1 pp. 73.

8.3 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, cf. section 5.5.5.

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 5.5.6.

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

[1] Besides these functions, 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.

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

which specifies a piecewise function f: between $\mathtt{x}_i$ and $\mathtt{x}_{i+1} = \mathtt{x}_i + \mathtt{d}_{i+1}$, function f is an interpolation of type $\mathtt{type}_{i+1}$ from y_i to y_{i+1}. See an illustration at the left of Fig. 8.1.

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

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

However, the functions @min_key and @max_key returns x_0 and x_n respectively (these functions also perform similarly on maps: a map is a function on a discrete range while nims are functions on a continuous range).

The type of the interpolation is either a constant string or an expression. But in this case, the expression must be enclosed in parenthesis. The names of the allowed interpolation types are the same as for curve and the interpolation types are illustrated on figures 5.5 and 5.6 pp 62.

Note that the previous definition specifies a continuous function because the value of f at the beginning of $[\mathtt{x}_i, \mathtt{x}_{i+1}]$ 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 $[\mathtt{x}_i, \mathtt{x}_{i+1}]$ the function is an interpolation between $\mathtt{y}_i$ and $\mathtt{Y}_i$. See illustration at the right of Fig. 8.1.

Vectorized NIM. the NIM construct admits tab as arguments: in this case, the result is a vectorial function. For example:

```
NIM{ [-1, 0] [0, 10], [2, 3] [1, 20] ["cubic", "linear"] }
```

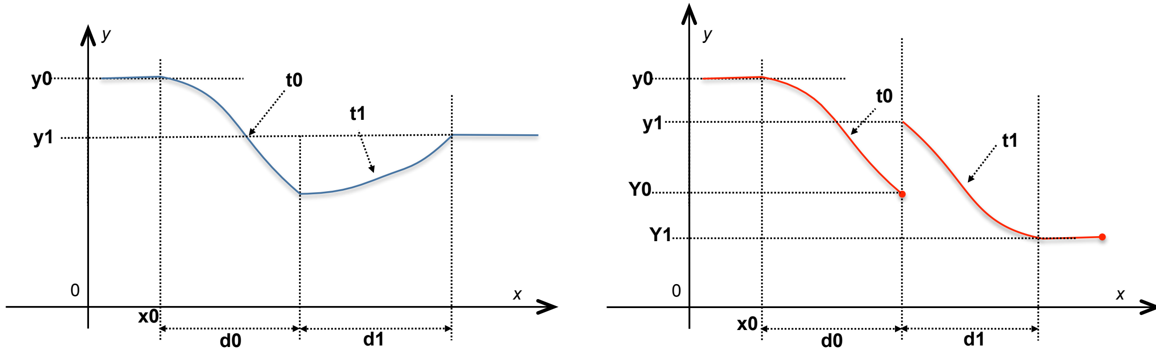
defines a vectorial NIM of two variables:

$$\vec{f}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} f_1(x_1) \\ f_2(x_2) \end{pmatrix}$$

where f_1 is a cubic interpolation between 0 and 1 for x_1 going from -1 to 1 and f_2 is a linear interpolation between 10 and 20 for x_2 going from 0 to 3.

The arguments of a vectorized NIM may includes scalar s: in this case, the scalar is extended implicitly into a vector of the correct size whose elements are all equal to s. This is the case even for the specification of the interpolation type. The specification of the interpolation type can be omitted: in this case, teh interpolation type is linear. For example:

Figure 8.1 The two forms a NIM definition. The diagram in the left illustrate the specification of a continuous NIM{x0 y0, d0 y1 t0, d1 y1 t1} while the diagram in the right illustrates the specification of a discontinuous NIM{x0, y0 d0 Y0 t0, y1 d1 Y1 t1}.




```
NIM{ 0, 0, [1, 2] 10 }
```

defines the function

$$\vec{f} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \quad \text{where } f_1(x) = \begin{cases} 0, & \text{if } x < 0 \\ 10, & \text{if } x > 1 \\ 10x, & \text{elsewhere} \end{cases} \quad \text{and } f_2(x) = \begin{cases} 0, & \text{if } x < 0 \\ 10, & \text{if } x > 2 \\ 5x, & \text{elsewhere} \end{cases} .$$

A vectorized NIM is *listable*: it can be applied to a scalar argument. In this case, the scalar argument x is implicitly extended to a vector of the correct dimension:

$$\vec{f}(x) = \vec{f} \begin{pmatrix} x \\ \dots \\ x \end{pmatrix}$$

The function @size returns the dimension of the image of a NIM, that is, 1 for a scalar NIM and n for a vectorized NIM, where n is the number of elements of the tab returned by the application of the NIM.

Extending a NIM. The function @push_back can be used to add a new breakpoints to an existing NIM (the NIM argument is modified):

```
@push_back(nim, d, y1)
@push_back(nim, d, y1, type)
@push_back(nim, y0, d, y1)
@push_back(nim, y0, d, y1, type)
@push_back(nim, nim)
```

The first two forms extends a NIM in a continuous fashion (the y0 value of the added breakpoint is the y1 value of the last breakpoint of the NIM). The next two forms specify explicitly the y0 value of the added breakpoint, enabling the specification of discontinuous function. The last form extends the nim in the first argument by the breakpoint of the nim in second argument. It effectively builds the function resulting in “concatenation” of the breakpoints.

Note that @push_back is an overloaded function: it also used to add (in place) an element at the end of a tab.

8.3.1 Nim transformation and smoothing

Inhomogeneous breakpoints in vectorial nim. A vectorial nim aggregates several scalar nims. The function @projection and @aggregate can be used to respectively extract a scalar nim from a vectorial nim and to aggregate several (scalar or vectorial) nims of dimension $d_1, \ldots, d_p$ into a nim of dimension $d_1 + \cdots + d_p$.

```
let $nim1 := NIM{ 0 0, 1 10, 1 0 "sine_in_out" }
let $nim2 := NIM{ −1 10, 4 0 "sine_in_out" }
let $nim3 := @aggregate($nim1, $nim2)
```

This process can be used to build nims whose breakpoints are different, as in the specification of multidimensional curves (cf. 56). There is no syntax to specify directly such nims and the @aggregate function must be used to build such nim value. When printing such nim, this function is used to list the several components of the nim:

```
print $nim3 −→ @aggregate(nim{0 0, 1 10 "linear", 1 0 "sine_in_out"},
                          nim{−1 10, 3 0 "sine_in_out"})
```

The @project function can be used to extract a component from a multidimensionnal dim:

```
@projection($dimC, 1) == $nimB −→ true
```

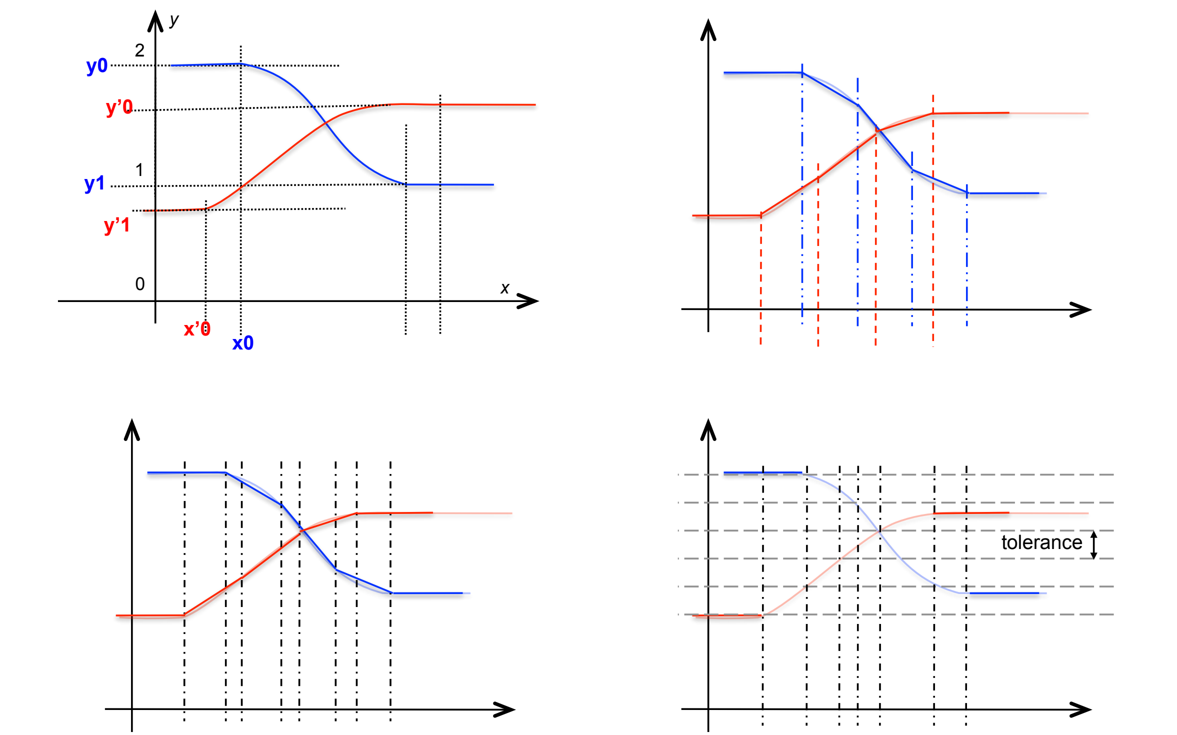
Sampling, homogeneization and linearization. It is possible to convert a nim with an arbitrary interpolation type to a nim with linear interpolation type. The first function @sample takes a nim nim and a number n and returns a nim with n breakpoints and linear interpolation type that approximates nim.

In the case of a vectorial nim with heterogeneous components, the approximation is done component by component. The function @align_breakpoints can be used on a nim with linear interpolation type to obtain an equivalent nim with homogeneous breakpoints:

```
let $nim4 := @sample($nim3, 5)
$nim4 −→ @aggregate( nim{0 0, 0.333333 3.33333 "linear",
                              0.333333 6.66667 "linear",
                              0.333333 10 "linear",
                              0.333333 7.5 "linear",
                              0.333333 2.5 "linear",
                              0.333333 0 "linear"},
                     nim{−1 10, 0.5 9.33013 "linear",
                                0.5 7.5 "linear",
                                0.5 5 linear, 0.5 2.5 "linear",
                                0.5 0.669873 "linear",
                                0.5 0 linear} )
@align_breakpoints($nim4) −→
   NIM { TAB[−1, −1],
         TAB[0, 10] TAB[0.5, 0.5] TAB[0, 10] TAB["linear", "linear"],
         TAB[0, 9.33013] TAB[0.5, 0.5] TAB[0, 9.33013] TAB["linear", "linear"],
         ; ...
         TAB[0, 0] TAB[0.333333, 0.333333] TAB[0, 0] TAB["linear", "linear"] }
```

The function @sample samples its nim argument homogeneously (component by component), see Fig. 8.2. This is not always satisfactory because the variation of the nim can differ greatly between two intervals. The function @linearize (nim, tol)) uses an adaptive sampling step to linearize nim, to align the breakpoints and achieve an approximation within a given tolerance tol.

Figure 8.2 The effect of @sample (top right), @align_breakpoints (bottom left), and @linearize (bottom right) on the nim pictured top left.




```
@linearize($nim3, 1) ⟶  NIM { TAB[−1, −1],
  TAB[0, 10] TAB[0.609946, 0.609946] TAB[0, 9.01426] TAB["linear", "linear"],
  TAB[0, 9.01426] TAB[0.270743, 0.270743] TAB[0, 8.02012] TAB["linear", "linear"],
  ; ...
  TAB[0.563462, 0.0636838] TAB[0.152573, 0.152573] TAB[0, 0] TAB["linear", "linear"]
```

The application of the @linearize function can be time consuming and care must be taken to not perturb the real-time computations, *e.g.*, by precomputing a linearization (see @eval_when_load clause and function @loadvalue).

Nim simplification. Several functions can be used to reduce the number of breakpoints of a nim with linear interpolation. The underlying idea is that the nim approximate a curve known by a series of points, the breakpoints of the nim, and that this curve can be approximated by fewer points. The simplified nim consists of a subset of the breakpoints that defined the original nim.

All the simplification functions apply to scalar as well as vectorial nim, but consider only the "x-coordinate" given by the breakpoints of the first component. In case of a vectorial nim with heterogeneous breakpoints, this can be a drawback. In this case, an equivalent nim with homogeneous breakpoints can be first build using the @align_breakpoints functions.

The three following functions are inspired from polyline simplification functions developed in computer graphics:

`@simplify_radial_distance_t(nim, d)` simplifying each component of the nim in-

dependantly by reducing successive breakpoints that are clustered too closely to a single breakpoint. Because the simplification works on each component independantly, a breakpoint in this context is simply a point (x, y) in the plane. The simplification process starts from the first breakpoint and is iterated until it reaches the final breakpoints. The first and the last breakpoints are always part of the simplification. All consecutive breakpoints that fall within a distance `d` from a kept breakpoints are removed. The first encountered breakpoints that lies further away than `d` is kept.

`@simplify_radial_distance_v(nim, d)` This simplification function is similar to the previous one but instead of working on each component independantly, the vectorial nature of the nim is taken into account and the "x-coordinate" is ignored. The nim is seen as a sequence of points, the image of the nim at coordinates given by the x part of the breakpoint of the first component. It is this series of points that is simplified, to build the new nim. The distance is thus taken in a n-dimensional space, where n is the dimension of the nim.

`@simplify_lang_v(nim, tol, l)` This simplification function follow the same approach as the previous one and consider a sequence of points in a n-dimensional space, where n is the dimension of the nim and the points are given by the image of the breakpoints of the first component. Then, a *Lang* simplification algorithm is applied to reduce the number of points. The remaining points are used to build the simplified nim.

The Lang simplification algorithm defines a fixed size `l` search-interval. The first and last points of that interval form a segment. This segment is used to calculate the perpendicular distance to each intermediate point. If any calculated distance is larger than the specified tolerance `tol`, the interval will be shrunk by excluding its last point. This process will continue until all calculated distances fall below the specified tolerance, or when there are no more intermediate points. All intermediate points are removed and a new search interval is defined starting at the last point from the old interval.

The effect of these simplification function on a nim can be observed using the `@size` function which returns, for a nim, its number of breakpoints.

Smoothing and Transformation. The functions described here work independently on each component of a nim. They see each component as a sequence of points y that are smoothed in various way. The resulting points are used to build a new nim with linear interpolation type.

`@filter_median_t(nim, n)` smoother the y by replacing every image by the median in its range-`n` neighborhood. Notice that the median is taken in a sequence of `2n`+1 values. The first `n` points and the last `n` points are leaved untouched.

`@filter_min_t(nim, n)` filters the by replacing every image by the minimum value in a sequence of length `2n` +1 centered on y. The first `n` points and the last `n` points are leaved untouched.

`@filter_max_t(nim, n)` filters the by replacing every image by the maximum value in a sequence of length `2n` +1 centered on y. The first `n` points and the last `n` points are leaved untouched.

@window_filter_t(nim, coef, pos) replace every y by the scalar product of coef (a tab) with a sequence of n values, where n is the size of coef and pos the position in this window of the current y (numbering start with 0). The pos first values and the last $n - \text{pos}$ values are left untouched.

For example, @window_filter_t(nim, [2], 0) build a nim by scaling the image of nim by 2. @window_filter_t(nim, [0.1, 0.2, 0.5, 0.2, 0.1], 2) is a moving weighted average with symmetric weights around y.

8.4 Tables

Tab values (table) are used to define simple vectors and more. They can be defined by giving the list of their elements:

```
$t := tab [0, 1, 2, 3]      ; or
$t := [0, 1, 2, 3]          ; the 'tab' keyword is optional
```

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 ·[·] 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 that can be interpreted as 3 lines and 2 columns. For example, if $t is a 3×2 matrix, then

```
$t[1][0]
$t[1, 0]    ; equivalent form
```

access the first element of the second line.

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. 5.4. 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. Such 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:

```
$z := [ 0 | (100) ]           ; builds a vector of 100 elements, all zeros
$s := [ $i | $i in 40, $i % 2 == 0 ] ; lists the even numbers from 0 to 40
$t := [ $i | $i in 40 : 2]           ; same as previous
$u := [ 2*$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) ]

    $m := [ [1, 2, 3], [4, 5, 6] ]              ; builds a matrix of 3x2 dimension
```

```
    $m := [ [ @random() | (10) ] | (10) ] ; builds a random 10x10 matrix

; 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)

$v := [ @random() | (10) ] ; builds a vector of ten random numbers
; 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) ]
```

Changing an element in a Tab. A tab is a *mutable* data structure : one can changes an element within this data structure. Although a similar syntax is used, changing one element in a tab is an atomic action different from the assignment of a variable. For example

```
let $t[0] := 33
$t[0] := 33          ; the ''let'' is optionnal
```

changes the value of the first element of the tab referred by `$t` for the value `33`. But this is not a variable assignment: the variable `$t` has not been "touched": it is the value referred by the variable that has mutated. Consequently, a whenever watching the variable `$t` does not react to this operation.

The difference between variable assignment and mutating one element in a tab, is more sensible in the following example:

```
let [0, 1, 2][1] := 33
```

where it is apparent that no variable at all is involved. The previous expression is perfectly legal: it changes the second element of the tab `[1, 2, 3]`. This change will have no effect on the rest of the *Antescofo* program because the mutated tab is not refered elsewhere but this does not prevent the action to be performed.

In the previous example, the `let` keyword is mandatory: it is required when the expression in the left hand side of the assignment is more complex than a variable `$v`, a simple reference to an array element `$v[1, ...]` or a simple access to a local variable of an *exec* `$v.$w`. See sect. 4.3.

Because the array to mutate can be referred by an arbitrary expression, one may write thing like:

```
$t1 := [0, 0, 0]
$t2 := [1, 1, 1]
@fun_def @choose_a_tab() { (@rand(1.0) < 0.5 ? $t1 : $t2) }
let @choose_a_tab()[1] := 33
```

that will change the second element of a tab chosen randomly between `$t1` and `$t2`. Notice that:

```
let @choose_a_tab() := [2, 2, 2]   ; invalid statement
```

raises a syntax error: this is neither a variable assignment nor the update of a tab element (there is no indices to access such element).

Elements of nested tabs can be updated using the multi-index notation:

```
$t := [ [0, 0], [1, 1], [2, 2] ]
let $t[1,1] := 33
```

will change the tab referred by `$t` to `[[0, 0], [1, 33], [2, 2]]`. One can change an entire "column" using partial indices:

```
$t := [ [0, 0], [1, 1], [2, 2] ]
let $t[0] := [33, 33]
```

will produce `[[33, 33], [1, 1], [2, 2]]`. Nested tabs are not homogeneous, so the value in the r.h.s. can be anything.

We mention above that the mutation of the element of a tab does not trigger the whenever that are linked to the variables that refers to this tab. It is however very easy to trigger a `whenever` watching a variable `$t` referring to a tab, after the update of an element: it is enough to assign `$t` to itself:

```
$t := [1, 2, 3]
$q := $t
whenever ($t[0] == 0) { ... }
let $t[0] := 0 ; does not trigger the whenever
$t := $t       ; the whenever is triggered
```

Notice that variable `$q` refers also to the same tab as `$t`. So we can mutate the first element of `$t` through `$q` but an assignment to `$q` does not trigger the `whenever`:

```
let $q[0] := 0  ; does not trigger the whenever
$q := $q        ; does not trigger the whenever
```

the variable `$q` does not appear in the condition of the `whenever` and consequently is not watched by it. *Nota Bene* that the assignment to `$q` does not trigger the `whenever`: a whenever watches a set of variables, NOT the values referred by these variables.

`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. Works also on string and maps.

@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.

@drop(t, n) gives tab t with its first n elements dropped if n is a positive integer, and tab t with its last n elements dropped if n is a negative integer. If n is a tab x of integers, @drop(t, x) returns the tab formed by the elements of t whose indices are not in x.

@empty(t) returns true if there is no element in t, and false elsewhere. Works also on maps.

@find(t, f) returns the index of the first element of t that satisfies the predicate f(i, v). The first argument of the predicate f is the index i of the element and the second the element T[i] itself. Works also on strings and maps.

@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.

@gnuplot(t) plots the elements of the tab as a curve using the external command gnuplot. See the description page 148 in the annex for further informations and variations.

@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 function is overloaded and applies also on maps. The form @insert " file " is also used to include a file at parsing time.

@is_prefix(t1, t2) , @is_suffix and @is_subsequence operate on tabs as well as on strings. Cf. the description of these functions in A.

@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. Works also on string and map.

@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. Works also on string and map.

@permute(t, n) returns a new tab which contains the nth 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).

@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). This function is overloaded and apply also on maps.

@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 157.

@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 to v elements. If v is greater than the size of t, then additional elements will be *undefined*. This function returns a new tab.

@reverse(t) returns a new tab with the elements of t in 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])
```

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

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

@slice(t, n, m) gives the elements of t of indices between n included up to m excluded. If n > m the element are given in reverse order.

@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.

@take(t, n) gives the first n elements of t if n is a positive integer and the last n elements of t if n is a negative integer. If n is a tab x of indices, @take(t, x) gives the tab of elements whose indices are in x : [t[x[i]] | i in @size(x)].

Lists and Tabs. *Antescofo*'s tabs may be used to emulate lists

- @car, @cons, @cdr, @drop, @last, @map, @slice and @take are similar to well known functions that exists in Lisp.
- @concat(a, b) returns the concatenation (*append*) of two lists.
- Arithmetic operations on vectors are done pointwise, as in some Lisp variants.

In particular, the operators @cons, @car and @cdr can be used to destructure and build a tab and they can be used to define recursive functions on tabs in a manner similar to recursive functions on list. *However* @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 than a recursive definition.

Chapter 9

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. A *synchronization strategy* specifies the temporal relationships between a sequence of action and a sequence of event. They take into account discrete relations (like the onset of an event) as well as continuous one (*e.g.*, the tempo of the musician). An *error handling strategy* defines what to do with the action associated to an event, when this event if not recognized (the origin of this "non-recognition" does not matter).

The musical context that determines the correct synchronization and error handling strategies is at the composer or arranger's discretion[1].

9.1 Synchronization Strategies

The *Antescofo* language allows precise expression how a sequence of electronic actions is synchronized in real-time with musician events. In *Antescofo*, an electronic phrase is written specifying delays between each actions into a block (group, loop, whenever, curve, *etc.*). Through a specific attribute, a particular synchronization strategy can be applied that will manage the temporal evolution of this phrase depending on the musician performance.

The *Antescofo* view of musician's performance. From the synchronization perspective, the musician performance can be summarized by two parameters: the musician's position (in the score) and the musician's tempo. These two parameters are computed by the listening machine from the detection in the audio stream of the events specified in the *Antescofo* score. The estimation of the tempo is used to compute an estimated position of the musician between two events and also to anticipate the arrival of future events.

[1]The interested reader will find on the *Antescofo* forum a the address http://forumnet.ircam.fr/user-groups/antescofo/forum/topic/synchronization-strategies-examples/ a patch that can be used to compare the effect of the various synchronization strategies, including the synchronization on a variable.

Coordination of actions with musician's performance. The *Antescofo* system maintain a local position and local tempo for each groups (sequence of actions), loops (repeated groups), curve (continuous actions), *etc.* These local position and tempo can be completely independent to that of the musician when the tempo is explicitly specified :

```
group @tempo := 70  { ... }
```

But when the position and the tempo of the group have to be coordinated with that of the musician, there are different synchronisation strategies to define how the position and tempo in the sequence of actions depends of the musician's position and tempo:

- @loose: which rely only on tempo to synchronize the actions;
- @tight: which primarily use the position information to synchronize the actions;
- @target: which are an intermediate between tight and loose strategies, aimed to dynamically and locally adjust the tempo of a sequence for a smooth synchronization with anticipated events.

When both information of information position of tempo information are used, they can be contradictory (*e.g.*, an event occurs early or latter than anticipated from the tempo information). Two approaches are possible following the priority given to one parameter or the other. They are specified using the @conservative and @progressive attributes.

Finally, the synchronization mechanisms can be generalized to refer to the updates of an arbitrary variable instead to the musical events. The @sync attribute and the @tempovar declaration are used in this case.

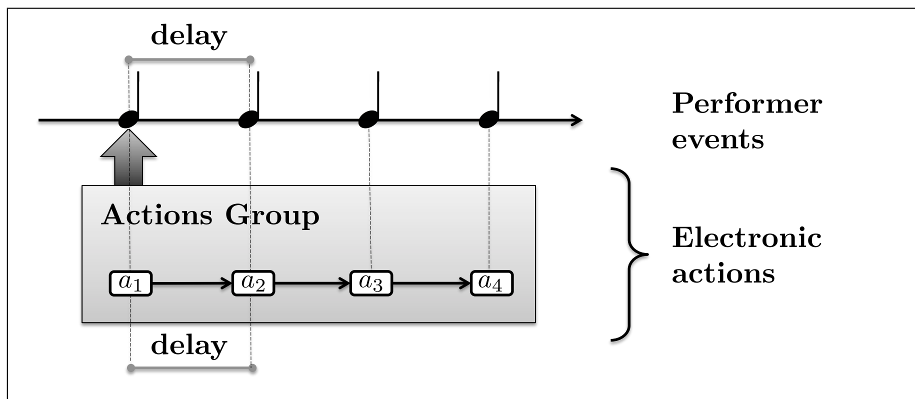
9.1.1 Loose Synchronization

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.

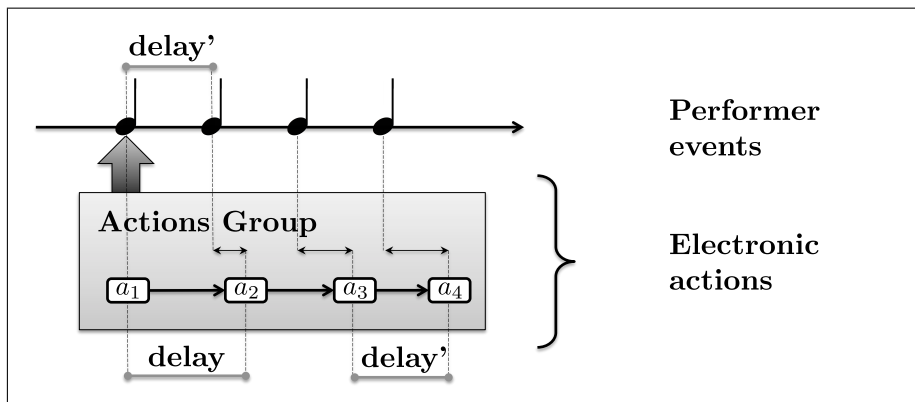
Figure 9.1 attempts to illustrate this within a simple example: Figure 9.1(a) shows the *ideal performance* or how actions and instrumental score is given to the system. In this example, an accompaniment phrase is launched at the beginning of the first event from the human performer. The accompaniment 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 9.1(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 9.1(b) and 9.1(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.

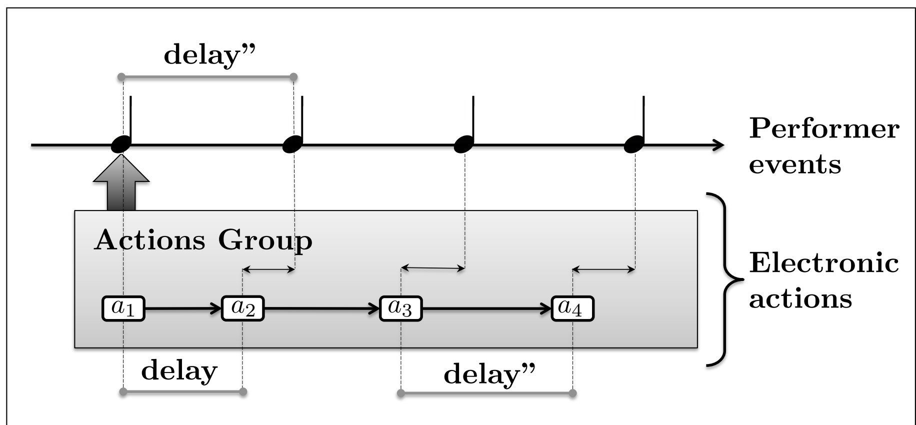
Figure 9.1 *The effect of tempo-only synchronization for accompaniment phrases: illustration for different tempi.* In the score, the actions are written to occur simultaneously with the notes, cf. fig. (a). Figure (b) and (c) illustrate the effect of a faster or a slower performance. In these cases, 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.



(a) Ideal performance



(b) Faster performance (delay' < delay)



(c) Slower performance (delay" > delay)

9.1.2 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 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.

9.1.3 Target Synchronization

In many interactive scenarios, the required synchronization strategy lies "in between" the @loose and @tight strategies. *Antescofo* provides two mechanisms through the @target attribute to dynamically and locally adjust the tempo of a sequence for a smooth synchronization.

- *static* targets rely on the specification of a subset of events to take into account in the tempo adjustment, while
- *dynamic* targets rely on a resynchronization window.

Static Targets

In some case, a smooth time evolution is needed, but some specific events are temporally meaningful and must be taken into account. For example this is the case when two musicians plays two phrases at the same time: they try most often to be perfectly synchronous (tight) on some specific events while others events are less relevant. These tight events can correspond to the beginning, or the end of a phrase or to other significant events commonly referred to as *pivot event*. The language let the composer to list pivot events for a given block. During the performance, the local tempo of the block is dynamically adjusted with respect to the actual occurrence theses pivots. (cf. Fig. 9.2). In the following example:

```
NOTE 60 2.0
    group @target:={e5, e10}
    {
        actions...
    }
    events...
NOTE 45 1.2 e5
    events...
NOTE 55 1.2 e10
```

the local tempo of the group will be computed depending on the successive arrival estimations of events $e5$ and $e10$. Notice that the pivots are referred through their label and listed between braces.

The computed tempo aims to converge position and tempo of the block to position and tempo of the musician at the anticipated date of the next pivot. The tempo adjustment is continuous: it follows a quadratic function of the position and the prediction is based on the last position and tempo values notified by the listening module, cf. Fig. 9.2. This strategy is smooth and preserve the continuity of continuous curves.

Dynamic Target

Instead of declaring *a priori* pivots, synchronizing positions can be dynamically looked as a temporal horizon: the idea is that position and tempo of the block must coincide with position and tempo of the musician at some date in the futur. This date depends of a parameter of the dynamic target called the *temporal horizon* of the target. This horizon can be expressed in beats, seconds or in number of events into the future. It corresponds to the necessary time to converge if the difference between the musician and electronic positions is equal to 1 beats. In the following example:

```
NOTE 60 2.0 e1
     group @target := [2s]
     {
          actions...
     }
     events...
```

the tempo and the position of the actions converges to the tempo of the musician but the convergence date is fixed by the following property: a difference of 1 between the position of the actions and the position of the musician is "absorbed" in 2 seconds. The syntax @target := [2] is used to specify an horizon in beats and @target := [2#] to define an horizon in number of events.

A small time horizon means that the difference between the position of action and the position of the musician must be reduced in a short time. A bigger time horizon allows more time to shrink the difference. Notice that the relationship between the difference in position and the time needed to bring it to zero is not linear. As in the static target synchronization, when a new event is detected, durations and delays are computed according to a quadratic function of the position. The convergence date (position, tempo) only depends on the difference between the musician and electronic positions.

This strategy is smoother than *static targets* since the occurrence of events are used only to compute the anticipated synchronization in the future.

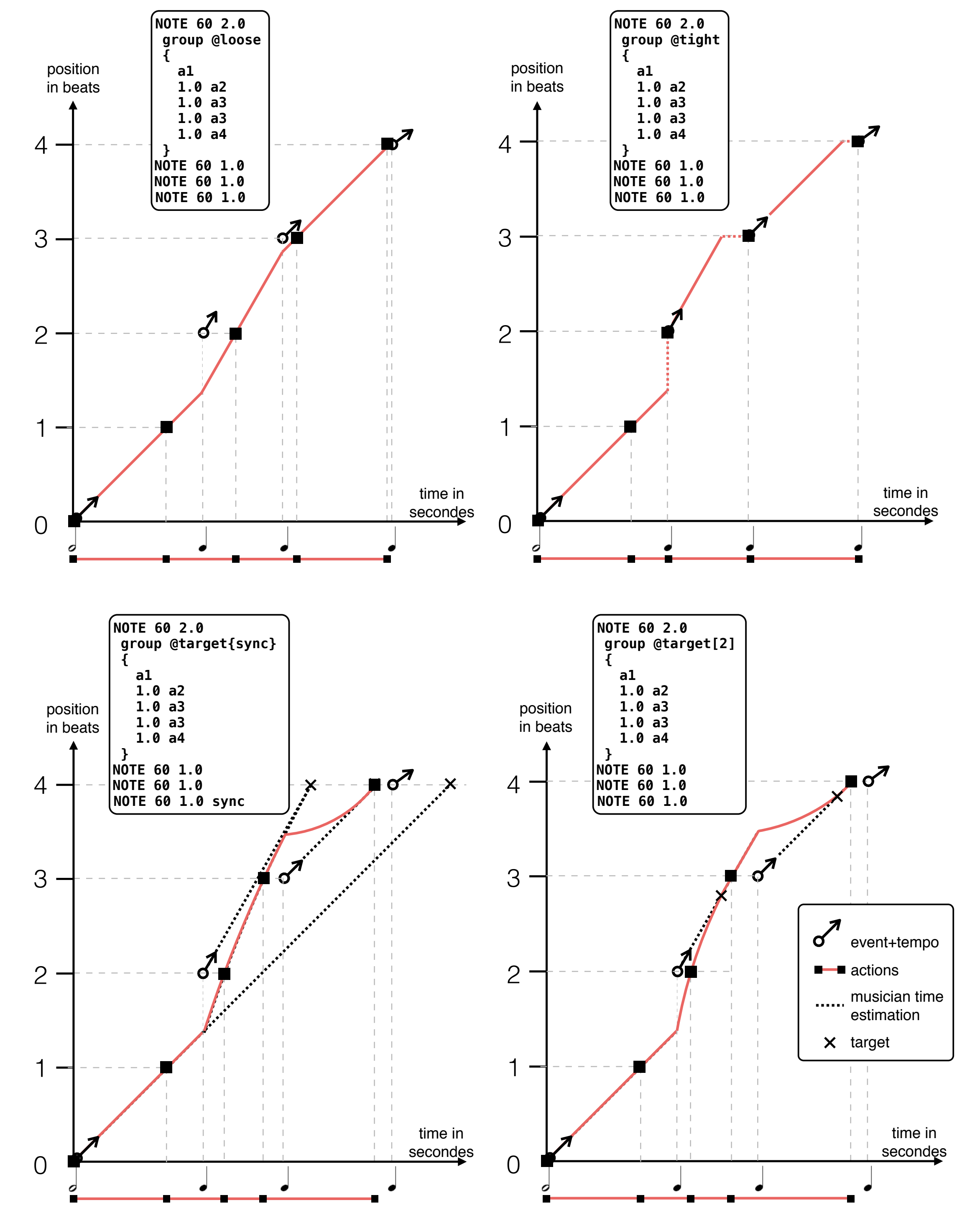
9.1.4 Adjusting the Coordination Reference

How to Compute the Position in case of Conflicting Informations

The @conservative and @progressive attributes parameterize the computation of the position of the musician in the synchronization strategy. They are relevant only for the @tight and @target strategies where both events and tempo are used to estimate the musician's position.

With @conservative attribute, the occurrence of events is more trusted than the tempo estimation to compute the musician's position. So, when the anticipated date of an event is reached, the computed position is stuck until the occurrence of this event.

Figure 9.2 These figures represent temporal evolution of an electronic phrase with several synchronization strategies. The graph shows the relationship between the relative time in beats with respect to absolute time in seconds. The musician events are represented by vectors where the slope correspond to the tempo estimation. The actions are represented by squares and the solid line represents the flow of time in the group enclosing these actions. From left to right and top to bottom, then strategies represented are : @loose, @tight, @target{sync}, @target [2] .



With the @progressive attribute, the default one, the estimation of the position will continue to advance even if the forecasted event is not detected.

Several system variables are updated by the system during real-time performance to records these various point of view in the position progression. They are used internally but the user can access their values. Variable $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. Variables $RNOW and $RCNOW are estimation of the current instant of the musician in the score expressed in beats. At the beginning of a performance,

$$\$\mathrm{NOW}_0 = \$\mathrm{RNOW}_0 = \$\mathrm{RCNOW}_0 = 0$$

At other instants of the performance, let e_n be the last decoded event by the listening machine (with $\$\mathrm{NOW} = \$\mathrm{NOW}_{e_n}$), e_{n+1} the following event, p_n et p_{n+1} their relative position in beats in the score, and *del* the delay in beats since the detection of e_n. These variable are linked by the following equations:

$$del = (\$\mathrm{NOW} - \$\mathrm{NOW}_{e_n}) + \$\mathrm{RT_TEMPO}/60$$

where $RT_TEMPO is the last decoded tempo (in BPM) by the listening machine. Then

$$\begin{aligned} \$\mathrm{RNOW} &= min(p_n + del, p_{n+1}) \\ \$\mathrm{RCNOW} &= p_n + del \end{aligned}$$

$RCNOW and $RNOW values differ when the estimated date of the next event is exceeded: $RNOW correspond to the conservative notion of time progression and remains at the same value until an event is detected, whereas the variable $RCNOW corresponds to the *progressive* notion of time progression and continues to grow following the tempo.

From a musical point of view, the position estimation with $RCNOW variable is more reliable when an event is missed (the musician does not play the note or the listening module does not detect it) but sometimes the values has to "go back" when the prediction is ahead unlike $RNOW.

Specifying Alternative Coordination Reference

The synchronization mechanisms of a sequence of actions with the stream of musical events (specified in the score) has been generalized to make possible the synchronization of a sequence of actions with the updates of an ordinary variable. The variable can be updated in the score or from the external environment (for example with message `setvar $v 3` or with OSC messages).

The updates of the variable act as the events in a score but the expected updates must be specified using the @tempovar declaration:

```
@tempovar $v(60, 2)
```

means that the "position" of the variable `$v` is incremented by 2 each time `$v` is updated. A "tempo" corresponding to the pace of the updates is also estimated using the same algorithm used to estimate the tempo of the musical events. The first attribute of the declaration @tempovar defines the initial value of this "tempo" (in the example, the initial value is 60). The position and tempo of the variable `$v` can be used in *Antescofo* expression using the dot syntax: `$v.position` and `$v.tempo`.

The @sync attribute is used to specify the synchronization of a sequence of action with the update of a variable. For instance:

```
Curve C
@sync $v,
@target[10],
@Action := ...
{
      $pos { {0} 5 {1} }
}
```

specifies that the curve C must go from 0 to 1 in 5 beats, but these beats are measured in the time-frame of the variable $v. In addition, the relation between the current position in the curve and the position of $v is specified using a dynamic target strategy.

9.2 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 initialized 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 9.3 attempts to showcase these four situations for a simple hypothetical performance setup similar to Figure 9.1.

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).

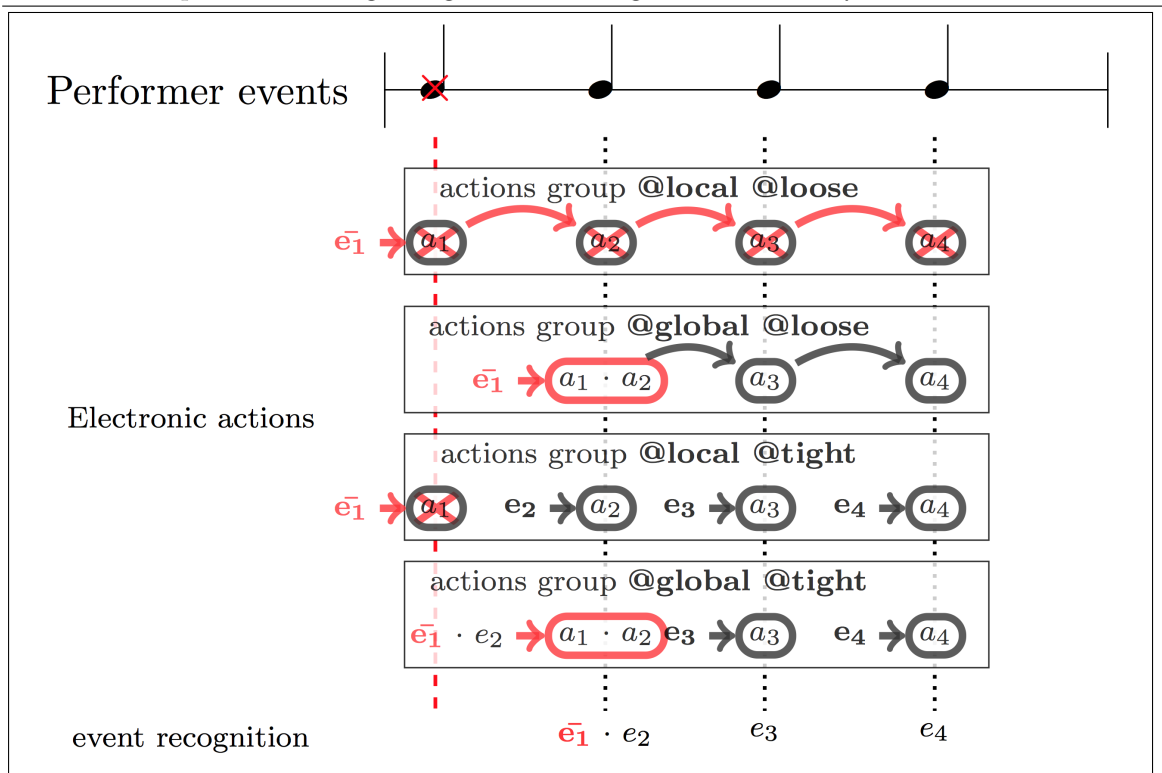
- local and tight: Strict synchrony of inside actions whenever there's a spatial correspondence between events and actions in the score. However actions within the strict vicinity of a missing event are dismissed. This case corresponds to an ideal concerto-like accompaniment system.

- 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 9.3. 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.

Figure 9.3 *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 $\bar{e_1}$.



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).

Remarks:

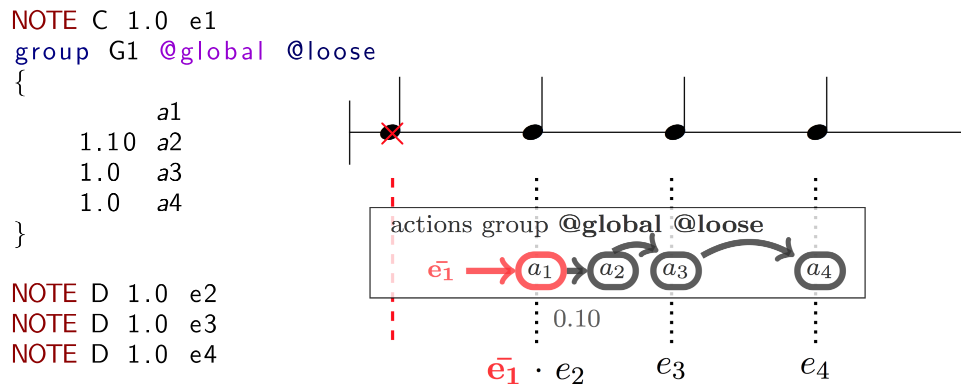
- A sequence of actions following an event in an *Antescofo* score correspond to a phantom group with attributes **@global** and **@loose**. In other words, the two following scores are similar.

```
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) :



Chapter 10

Macros

If computerised actions in your score observe repetitive conceptual patterns similar to electronic leitmotivs, then you might want to simplify your score by defining *Macros*, *Functions* or *Process* and create those pattern by simply calling these user-defined objects with arguments. *Macros* are described in this chapter, *functions* in the next one and *process* in chapter 12.They latter two can be often used in place of a macro with several advantages. See paragraph 12.7 for a comparison of the three constructs.

10.1 Macro Definition and Usage

Macro are defined using the @macro_def construct. Macros are called by their @-name followed by their arguments between parenthesis. A call to a macro is simply replaced by its definitions and given argument in the core of a score: this process is called *macro-expansion*. Macros are thus evaluated at score load and are NOT dynamic. They are purely textual. Macros can not be recursive. The macro-expansion is thus a syntactic replacement that occurs during the parsing and before any evaluation. The body of a macro can call (other) macros.

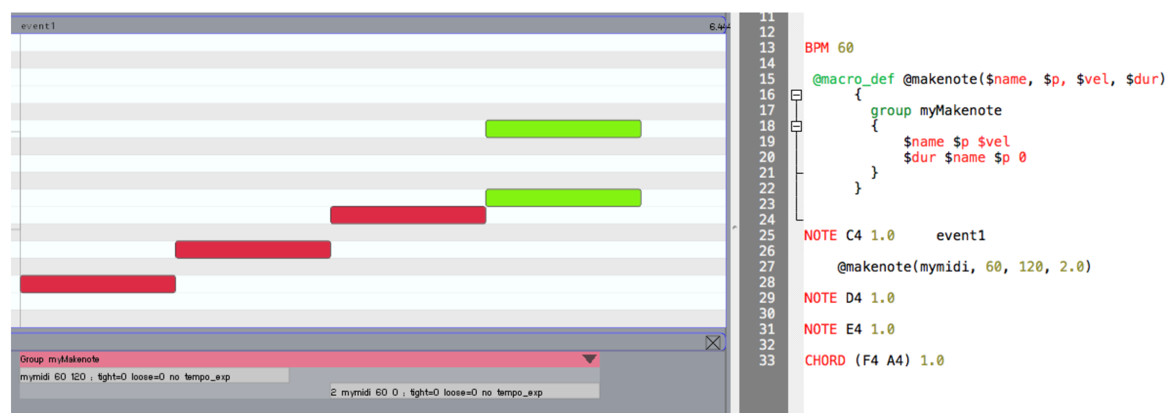
Macro name are @-identifier (preceded by @ symbol). For backward compatibility reason, a simple identifier can be used but the @-form must be used to call it. Macro arguments are considered as variables and thus use $-identifiers (preceded by $ symbol). The body of the macro is between braces. The white spaces and tabulation and carriage-returns immediately after the open brace and immediately before the closing brace are not part of the macro body.

The following code shows a convenient macro called makenote that simulates the *Makenote* objects in Max/Pd. It creates a group that contains a note on with pitch $p, velocity $vel sent to a receive object on $name, and triggers the note-off after duration $dur. The two lines inside the group are Max/PD messages and the group puts them in a single unit and enables polyphony or concurrency (see chapter 5).

```
@macro_def @makenote($name, $p, $vel, $dur)
{
	group myMakenote
	{
		$name $p $vel
		$dur $name $p 0
	}
}
```

Figure 10.1 shows the above definition with its realisation in a score as shown in AscoGraph. The call to macro can be seen in the text window on the right, and its realisation on the graphical representation on the left. Since Macros are expanded upon score load, you can only see the expansion results in the graphical end of AscoGraph and not the call.

Figure 10.1 Example of a Macro and its realisation upon score load



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
)
```

results to the following code after score is loaded:

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

Macros can 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.

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).

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

```
loop 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):

```
loop 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, $action)
{
  group @name := Gengroup$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

```
group @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
```

Chapter 11

Functions

Antescofo offers several kind of functions. Nim (section 8.3) and maps (section 8.2) are two examples of data values that can act as functions: they can be applied to an argument to provide a value. Maps are *extensional functions*: they are qualified as extensional because they enumerate explicitly the image of each possible argument in the form of a (key, value) list. Nim are also a kind of extensional functions: if the image of each argument x is not explicitly given, the association between the argument and its image is taken in a limited set of possibilities constrained by the breakpoints.

In this chapter, we will focus on *intentional functions*. Intentional functions f are defined by arbitrary 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.

For example, the following code shows a convenient user-defined function in *Antescofo* that converts MIDI pitch values to Herz. Any call to (for example) @midi2hz(69) anywhere in the action language where an expression is allowed (inside messages etc.) will be replaced by its value 440.0 at run-time.

```
@fun_def midi2hz($midi)
{
        440.0 * exp(($midi-69) * log(2) / 12 )
}
```

The example above is rather dubious since we do not use any of *Antescofo*'s interactive facilities! The following example is another classical *Antescofo* user-defined function that employs the internal variable $RT_TEMPO (musicians's real-time recognised tempo in BPM) with the goal of converting beat-time to milli-seconds using the latest tempo from musician. This function has been used in various pieces to simulate trajectories of effects based on score time (instead of absolute time). Note that indentation and carriage-returns do not matter:

```
@fun_def beat2ms($beats) {    1000.*$beats*60.0/$RT_TEMPO    }
```

Functions lives in the domain of expressions and values:

(*i*) they are defined by an expression that specifies how the values provided as arguments in a function call are transformed into a(nother) value;

(*ii*) they are themselves values (see section 11.2);

(*iii*) a function call can appear only inside an expression (as a sub-expression) or where an expression is expected (for example in the attributes of an action)

(*iv*) and the call of a function takes "no time" (see the so-called synchronous hypothesis sect. B.13).

This is not the case for macros nor processes (see page 133 for a comparison of the three constructs).

Some intentional functions are predefined and available in the initial *Antescofo* environment like the IEEE mathematical functions. See annex A for a description of more than 190 predefined 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` .

11.1 Function definition

The two examples above are example of simple functions whose bodies are just one expression

```
@fun_def name(arg1 , arg2 , ...) { expression }
```

Writing a large function can become cumbersome and may involve the repetition of common sub-expressions. To face these problems, since version 0.8, the body of a function can also be an *extended expression*.

Extended expressions. An extended expression is an arbitrary sequence of

1. simple expressions
2. local variables declarations introduced by `@local`
3. local variables assignments using := (right hand side is a simple expression)
4. global variables assignments using := (right hand side is a simple expression)
5. extended conditional expressions `if` ... `else` and `switch` ... `case` whose sub-expression are extended expressions
6. iterations expressions `Loop` and `Forall` whose sub-expression are extended expressions
7. Max/PD messages
8. abort actions,
9. one `return` statement followed by an expression.

An extended expression is allowed only in the body of a function. This is not because they have something special: they are no more than "ordinary" expressions. The only motivation behind this constraint is to avoid syntactic ambiguities when the score is parsed. With extended expressions, *Antescofo* function definitions are similar to `C` function definitions that mix expression and statement. As a matter of fact, *Antescofo* function mix expression and

(some kind of) actions. However, only a limited set of actions are allowed in functions: some of the actions that have zero-duration. The rational is the following: a function call must have no extent in time and the evaluation must be more efficient than a process call.

After some simple introductory examples, we details these 9 constructions.

First examples. The @polynome function definition

```
@fun_def polynome($x $a, $b, $c, $d)    // compute ax^3 + bx^2 + cx + d
{
    @local $x2, $x3
    $x2 := $x * $x
    $x3 := $x2 * $x
    return $a*$x3 + $b*$x2 + $c*$x + $d
}
```

must be clear: the extended expression specifying the function body introduces two local variables used to factorize some computations. The result to be computed is specified by the expression after the return statement.

In function

```
@fun_def fact($x)
{
    if ($x <= 0) { return 1 }
    else { return $x * @fact($x - 1) }
}
```

the if extended conditional is equivalent but more readable than the conditional expression (cf. page 78):

```
($x <= 0 ? 1 : $x * @fact($x - 1))
```

Notice however that, despite the syntax, this if is definitively NOT the action described page 48: the branches of this if are an extended expression, not a group of actions.

Because Max/PD messages are included in extended expressions, they can be used to trace (and debug) functions:

```
@fun_def fact($x)
{
    print "call fact(" $x ")"
    if ($x <= 0)
    {
        print "return 1"
        return 1
    }
    else
    {
        @local $ret
        $ret := $x * @fact($x - 1)
        print "return " $ret
        return $ret
    }
}
```

(but see the predefined functions @Tracing and @UnTracing for a more easy tracing of function calls).

A loop construct can be used to compute the factorial in an iterative manner, instead of a recursive one:

```
@fun_def fact_iterative($x)
{
    @local $i, $ret
    $ret := 1
    $i := 1
    Loop {
        $ret := $ret * $i
        $i := $i + 1
    } until ($i == $x + 1)
    return $ret
}
```

which can also be written

```
@fun_def fact_iterative_bis($x)
{
    @local $i, $ret
    $ret := 1
    $i := 1
    Loop {
        $ret := $ret * $i
        $i := $i + 1
    } during [$x #]
    return $ret
}
```

Notice that the during clause specifies the number of iterations of the loop, it cannot specifies a duration in seconds or in relative time.

11.1.1 The return Statement

The value of an extended expression is the value computed as the argument of a return. This is not necessarily the last statement of the sequence. If there is no return in the extended expression, the returned value is the value of the last expression in the sequence. If they are multiple return, a warning is issued and only the last one is taken into account.

For example:

```
@fun_def @print($x)
{
    print $x
}
```

When applied to a value, this function will send the print message that would eventually output the argument $x on the Max or PD console. We will see below that the value returned by sending a Max message is the exec '0.

A common pitfall. A confusing point is that, contrary to some programming language, return *is NOT* a control structure: it indicates the value returned by the nearest enclosing extended expression, not the value returned by the function. Thus:

```
@fun_def pitfall($x)
```

```
{
    if ($x) { return 0 }
    return 1
}
```

is a function that always returns 1. As a matter of fact, the return 0 if the indication of the value returned by the branch of the if, not the value returned by the body of the function. However, function

```
@fun_def work_as_expected($x)
{
    if ($x) { return 0 }
    else { return 1 }
}
```

returns as expected 0 or 1 following the value of the argument $x, simply because the value of the function body is the value returned by the if which is the value returned by the function (the if is the last (and only) statement of the function body.

11.1.2 Function's Local Variables and Assignations

To factorize common sub-expressions, and then to avoid re-computation of the same expressions, extended expressions may introduce local variables using the @local keyword. The syntax mimics the syntax used for local variables in compound actions (group, whenever, *etc.*), but local variables in function are distinct from the local variables in actions:

- their lifetime is limited to one instant, the instant of the function call,
- so there is no needs nor the possibility to refer to these variable outside of their definition scope (an extended expression).

As a consequence, their implementation is optimized (for example, we know that these variables cannot appear in the clause of a whenever so the run-time do not need to monitor their assignments). The cost of accessing a function local variable is the same as accessing a function argument. This cost is only a little bit less than accessing a global variable (the gain is usually negligible) but the gain in the memory footprint and in housekeeping the environment is noticeable.

Local variables are introduced using the @local keyword in the first statement of an extended expression. Every variable that appears in the left-hand-side of an assignment and whose name does not appear in a @local clause, is supposed to be a global variable.

The initial value of a local variable is undef. Then, the value referred by a local variable is the last value assigned to this variable during the evaluation process. With the definition

```
@fun_def f($x)
{
    @local $y
    $y := $x * $x
    $y := $y * $y
    return $y + 1
}
```

the expression @f(x) will compute $x^4 + 1$. Notice that the value of a local variable assignment, is, as for any assignment, the the exe '0. So:

```
@fun_def g($x)
{
    @local $y
    $y := 2*$x
    $y := $y + 1
}
```

will return '0 when called with x. See section 7.7 for a description of exec values.

Notice, the right hand side of a function local variable assignment is an expression, not an extended expression.

11.1.3 Extended Conditional Expressions and Iteration Expressions

Extended expressions enrich expressions with four constructs that mimic some action constructions: if, switch, loop and forall. The keywords used are the same used to specify the corresponding actions. But the constructions described here are expressions not actions:

- their sub-expressions involve extended expressions and not sequences of actions,
- their evaluation takes "no time" (they have zero-duration which is usually not the case of the corresponding actions),
- they have no label,
- they have no synchronization attributes,
- they have no delays.

These expressions are qualified as *pseudo-actions*.

The if Extended Expression. The if expression mimics the action if but its branches are extended expressions and it is not possible to define a label or the other attributes of an action. This construction is similar to the conditional expression

```
(cond ? exp_if_true : exp_if_false)
```

However, the else branch is optional. If cond evaluates to false and the else branch is missing, the returned value is undef.

The switch Extended Expression. The syntax of the switch expression construct follows, *mutatis mutandis*, the syntax of the action switch presented page 49. For instance, the Fibonacci recursive function can be defined by:

```
@fun_def sup1($x) { return $x > 1 }
@fun_def fibonacci($x)
{
    switch ($x)
    {
      case 0: return 1
      case 1: return 1
      case @sup1:
         @local $x1, $x2
```

```
            $x1 := $x − 1
            $x2 := $x1 − 1
            return @fibonacci($x1) + @fibonacci($x2)
        }
    }
```

Recall that there are two form of the switch construct. In this example, we use the form that compare the selector $x to the values 0, 1. The third value is a predicate which is applied to the selector, and if true, the attached expression (here an extended expression) is evaluated.

The other form of switch do not rely on a selector: the expression after the case is evaluated and if true, the corresponding expression (or extended expression) is evaluated.

The value of the construct if the value of the expression attached to the selected case. If there is no matching case, the value returned by the switch is undef.

The Loop Extended Expression. The loop expression mimics the loop action construction, but, as for the other pseudo-actions, there is no delay and no attributes. Furthermore, a loop action has no period (because it is supposed to have zero-duration), and it does not accept the during clause with a relative or an absolute time (a logical time, corresponding to an iteration count, is accepted).

The value of a loop is undef. Thus, a loop expression is used for its side effects. For example, the computation of the square root of a strictly positive number p can be computed iteratively using the Newton's formula[1]:

$$x_0 = p, \quad x_{n+1} = \frac{1}{2}(x_n + \frac{p}{x_n})$$

by the following function:

```
    @fun_def @square_root($p, $error)
    {
        @local $xn, $x, $cpt
        $cpt := 0
        $x := $p
        $xn := 0.5 * ($x + 1)
        Loop
        {
            $x := $xn
            $cpt := $cpt + 1
            $xn := 0.5 * ($x + $p/$x)
        } until (($cpt > 1000) || (@abs($xn −$x) < $error))

        if ($cpt > 1000)
        { print "Warning: square root max iteration exceeded" }

        return $xn
    }
```

We stress the fact that a return inside the loop is useless. As explained page 122, a return is not the indication of a non-local exit, but the specification of the value returned by the nearest enclosing extended expression. A return in the loop body, will specify the value of the body, which is throw away by the loop construct that always returns undef. This is why the

[1] Note that there is a more efficient predefined @sqrt function.

exit of the loop is controled here by an until clause and a return at the end of the function body is used to return the right value.

The Forall Extended Expression. The forall expression mimics the forall action. As loop expression, it is used for its side-effect. The forall expression make possible the iteration over the elements of a tab, a map, or over a range of integers.

11.1.4 Atomic Actions in Expressions

Some atomic actions, that is, actions with zero-duration, are directly allowed in an extended expression: messages, abort and assignation to a global variable. Such action may not have a label nor other action's attributes. The value of these actions in an extended expression is '0, that is, the value returned by the action-as-expression, see section 6.6.

Note that one can launch an arbitrary action within a function body, using the EXPR construct, see page 79. This make possible to access a function local variable that no longer exists:

```
@fun_def pitfall2()
{
     @local $x
     $x := 1
     _ := EXPR { 1 print $x }
     return $x
}
$res := @pitfall2()
```

will set the global variable $res to 1 but an error is signaled:

```
Error: Vanished local variable or function arguments
bad access at line ...
Do you try to access an instantaneous variable from an action spanned in a function ?
```

Indeed, the action spanned by the EXPR construct happens one beat after the evaluation of the function call, which is instantaneous. So when the action is performed, the local variable $x does not exists anymore.

11.2 Functions as Values

In an *Antescofo* expression, the @-identifier of a function denotes a functional value that can be used for instance as an argument of higher-order functions (see examples of higher-order predefined function in section 8.2 for map building and map transformations).

11.3 Curryfied Functions

In *Antescofo*, intentional functions are implicitly *Curryfied*. Such notion was introduced and developed by the mathematician Haskell Curry. The idea is to see a function that takes n argument as equivalent to a function that takes only p arguments, with $0 < p < n$, and that returns a function that takes $n - p$ arguments.

Consider for instance

```
@fun_def @f($x, $y, $z) { $x + 2*$y + 3*$z }
```

This function takes 3 arguments, so

```
@f(1, 2, 3)  returns 14  computed as: 1 + 2*2 + 3*3
```

The idea of a curryfied function is that one can provide less than three arguments to the function @f. For example

```
@f(11)
```

is a function still awaiting 2 arguments y and z to compute finally $11 + 2*y + 3*z$. And function

```
@f(11, 22)
```

is a function still awaiting one argument z to compute finally $55 + 3*z$.

Curryfied functions are extremely useful as argument of higher-order function (*i.e.*, function taking other functions as argument). Consider the function @find(t, f) that returns the first index i such that f(i, t[i]) is true. Suppose that we are looking for the first index whose associated value is greater than a. The value a will change during the program execution. Without relying on curryfication, one may write

```
@global $a
@fun_def @my_predicate($i, $v) { $v > $a }
...
$t := ... ; somme tab computation
$a := 3
$i := @find($t, @my_predicate)
```

But this approach is cumbersome: one has to introduce a new global variable and must remember that the predicate @my_predicate work with a side effect and that variable $a must be set before calling @my_predicate. Using curryfication, the corresponding program is much simpler and does not make use of an additional global variable:

```
@fun_def @my_pred($a, $i, $v) { $v > $a }
...
$t := ... ; somme tab computation
$i := @find($t, @my_pred(3))
```

The expression @my_pred(3) denotes a function awaiting two arguments i and v to compute @my_pred(3,i,v), which is exactly what @find expects.

All user defined functions are implicitly curryfied and almost all *Antescofo* predefined functions are curryfied. The exception are the special forms and overloaded predefined functions that take a variable number of arguments, namely: @dump, @dumpvar, @flatten, @gnuplot, @is_prefix, @is_subsequence, @is_suffix, @normalize, @plot, @rplot, and @sort.

Chapter 12

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 7.6, and it is used for calling the process.

12.1 Calling a 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 are `@`-identifiers and the name of processes are `::`-identifiers.

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

```
NOTE C4 1.3
    group _trace_body1 {
          print begin "C4"
      1.3 print end "C4"
    }
    action1
NOTE D3 0.5
```

```
group _trace_body2 {
      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 7.7. This value refers to the group lauched by the process instanciation and is eventually used in the computation of the surrounding expression. This value is accessible within the process body itself through a *special variable* $MYSELF. This variable is read-only is managed by the *Antescofo* run-time.

12.2 Recursive Process

A process may call other processes and can be recursive, calling itself directly or indirectly. For instance, an 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.

12.3 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)
  :: $q(2)
  2 ::Clock($p, $q)
}
```

A call to Clock(::Tic, ::Toc) will print TIC one beat after the call, then TOC one beat after the TIC latter, and then TIC again at date 3, TOC at date 4, etc.

In the previous code a "::" is used in the first two lines of the ::Clock process to tell *Antescofo* that the sentence is an action (a process call) and not an expression (a function call). This indication is mandatory because at the syntactic level, there is no way to know for sure that $p(1) alone is a function call or a process call.

12.4 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, through its *exec* value:

```
$p1 := ::P()
$p2 := ::P()
$p3 := ::P()
...
abort $p2   ; abort only the second instance
abort ::P   ; abort all remaining instances
```

Using the special variable $MYSELF, it is possible to implement a self suicide on a specific condition *e*:

```
@proc_def ::Q()
{
  ...
  whenever (e) { abort $MYSELF }
  ...
}
```

12.5 Processes and Variables

Processes are defined at top-level. So, the body of the process can refers only to global variables and to local variables introduced in the body.

Variables that are defined @local to a process are defined per process instance: they are *not* shared with the other calls. One can access to a local variable of an instance of a process through its *exec* value, using the *dot notation*:

```
@proc_def DrunkenClock()
{
  @local $tic
  $tic := 0
  Loop (1. + @random(0.5) - 0.25)
  { $tic := $tic + 1 }
}
$dclock := ::DrunkenClock()
...
if ($dclock.$tic > 10)
{ print "Its 10 passed at DrunkenClock time" }
```

The left hand side of the infix operator "." must be an expression whose value is an active *exec*. The right hand side is a variable local to the referred *exec*. If the left hand side refers to a dead *exec* (cf. 7.7), the evaluation of the dot expression raises an error. In the previous example an instance of ::DrunkenClock is recorded in variable $dclock. This variable is then used to access to the variable $tic which is local to the process. This variable is incremented with a random period varying between 0.75 and 1.25.

Accessing a local variable through the dot notation is a dynamic mechanism and the local variable is looked first in the instance referred by the *exec*, but if not found in this group, the variable is looked up in the context of the *exec*, *i.e.* in the group containing the process call, and so on (*e.g.* in case of recursive process) until it is found. If the top-level context is reached without finding the variable, an undef value is returned and an error message is issued.

This mechanism is useful to access dynamically a variable defined in the scope of the call. For example:

```
@proc_def ::Q($which)
{ print $which ":␣" ($MYSELF.$x) }

$x := "x␣at␣top␣level"

Group G1 {
  @local $x
  $x := "x␣local␣at␣G1"
  ::Q("Q␣in␣G1")
}

Group G2 {
  @local $x
  $x := "x␣local␣at␣G2"
  ::Q("Q␣in␣G2")
}

::Q("Q␣at␣top␣level")
```

will print:

```
Q in G1: x local at G1
Q in G2: x local at G2
Q at top level: x at top level
```

The reference of an instance of the process can be used to assign a variable local to a process "from the outside", as for example in:

```
1  @proc_def ::P()
2  {
3    @local $x
4    whenever ($x} { print "$x␣has␣changed␣for␣the␣value␣" $x }
5    ...
6  }
7  $p := ::P()
8  ...
9  $p.$x := 33
```

the last statement will change the value of the variable $x only for the instance of P launched at line 7 and this will trigger the whenever at line 4.

Notice that the features described here specifically for a process instance, works in fact for any *exec* (see sections 6.6, 7.7 and 6.2.4). A local variable may also be accessed through the name of its definition scope, refer to paragraph 6.2.4 using the :: operator.

12.6 Process, Tempo and Synchronization

The tempo of a process can be fixed at its specification using a tempo attribute:

```
@proc_def ::P() @tempo := ...
```

In this case, every instance of ::P follows the specified tempo. If the tempo is not specified at the process definition, then the tempo of an instance is implicitly *inherited* from the call site (as if the body of the process was inserted as a group at the call site).

For example:

```
Group G1 @tempo :=  60 { Clock(::Tic, ::Tic) }
Group G2 @tempo := 120 { Clock(::Toc, ::Toc) }
```

will launch two clocks, one ticking every second, the other one tocking two times per second.

12.7 Macro *vs.* Function *vs.* Processus

How to chose between macros, functions and processes? These three mechanisms can be used to abstract and reuse a code fragment. However they have not the same benefits, flexibilities nor shortcomings. The table 12.1 compare the three mechanisms from several point of views.

Figure 12.1 Comparaisons between the mechanisms of macro, function and process.

	MACRO	**FUNCTION**	**PROCESS**
"*lives in*"	text	expression	action
parameterized by (kind of argument)	arbitrary text	values (through expression)	values (through expression)
body is	an arbitrary text	an extended expression	a group of actions
returns	none text expanded *in place*	a value the result of the evaluation	a value the process *exec*
lifespan of the execution	not applicable the expanded text may have one	0 evaluation is instantaneous	*d* running actions may takes time
where a call may appears	anywhere	where an expression is expected	where an action is expected
when the arguments are computed	when the score is loaded, for each occurrence of the argument in the macro body	when the function call is reached by the evaluation flow, one time per argument	when the process call is reached by the execution flow, one time per argument
definition can be recursive	no	yes	yes
partial application	no	yes (the result is a function)	no
is a denotable entity	no	yes a function is a value referred by its name	yes a process is a value referred by its name, as well as the result of the evaluation referred by its exec
possibility to create local variable	no (and yes) the macro may expand into a group which contains new local variable, but the possibility is not linked to the macro mechanism itself.	yes but the variable exists only during the evaluation (which takes zero time), and they cannot be referred from outside the function body	yes and the local variables of a process can be referred from outside, used in a whenever, etc.
may launch actions	yes	only messages and assignation (but you can use the `EXPR` construct)	yes

Chapter 13

Antescofo Workflow

This chapter is still to be written...

13.1 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.

...

13.2 Tuning the Listening Machine

...

13.3 Debuging an *Antescofo* Score

13.4 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 13.8 for additional information on the old syntax.

13.4.1 Monitoring with Notability

...

13.4.2 Monitoring with *Ascograph*

.

...

13.4.3 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).

Tracing the Evaluation of Functions. Function calls can be traced, see the description of the functions @Tracing and @UnTracing, page 171.

13.5 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.

13.5.1 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.

13.5.2 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

13.5.3 Predefined Messages

The *Antescofo* object accepts predefined message. These messages corresponds to the internal commands described in section 4.6.

13.6 Interacting with PureData

...

13.7 *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. 13.1.

Figure 13.1 Help of the standalone offline command line.

```
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.
  --print : Print the result of the parsing in the output specified by --message or stdout
  --recognition : Audio recognition-only mode (no action is triggered) (-r)
                  An audio file is needed.
  --ideal : Produce an ideal trace from the given score (-i).
            The trace is written in an output file (option --output)
            or standard output by default. A score file (--score) is needed.
  --test : Test mode, simulate an Antescofo execution from an input trace  (-T).
            An output trace is written in a trace output file (tout) if given or
            in standard output by default.
            A score file (--score) and an input trace (--tin) are needed

Input 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)
  --channel int: select input channel in case of multi-channel audio file
                 (0: mix all channels / 1: left / default: 1)
  --normalization (0|1) : if on, the input audio is divided by its peak maximum value (default: 1)
  --duration float : crop audio file at the given duration (in seconds)
  --tin filename : input event trace file.
  --rel : Positions are given in beats in input trace file (default, alternative --phy)
  --phy : Positions are given in physical time (seconds) in input trace file and
          it contains no tempo values (alternative --rel)
  --notempo : tempo values are not read in the input trace file

Output files:
  --lab filename : output file in the LAB format in recognition mode (default: standard output)
  --mirex filename : output file in the MIREX format in recognition mode (default: standard output)
  --trace filename : trace all events and actions (use 'stdout' for standard output) (-t)
  --tout filename : Idem but with the output trace format (used in test)
  --evttrace filename : trace all events (the output is a tin).
                        Used to produce an input trace from an audio

Listening module options:
  --fftlen samples : fft window length (default: 4096) (-F)
  --hopsize samples : antescofo resolution in samples (default: 512) (-S)
  --gamma float : Energy coefficient (default: -1) (-G)
  --pedal (0|1) : pedal on=1/off=0 (default: 0)
  --pedaltime float : pedaltime in milliseconds (default: 600) (-P)
  --nofharm n : number of harmonics used for recognition (default: 10) (-H)
  --pianomode (0|1) : Piano-based harmonic templates on=1/off=0 (default: 0)

Listening module advanced options:
  --preventzigzag (0|1) : if 0, the decoded states can go backwards (defaut : 1)
  --dummysilences (0|1) : if 1, use dummysilences (one after each note) (default: 0)

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)
```

13.8 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.

13.9 Stay Tuned

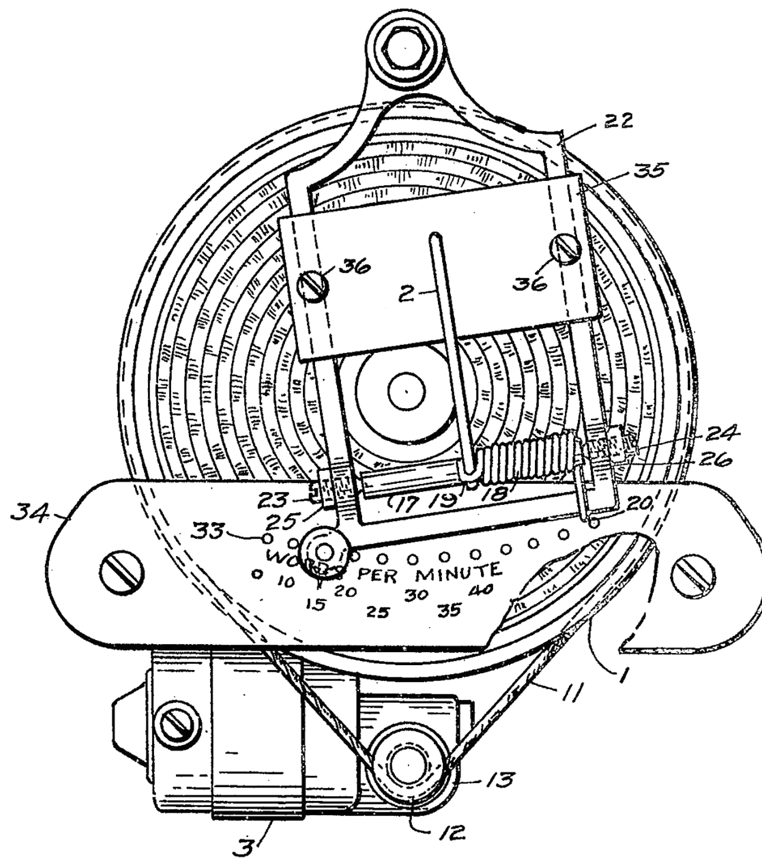
Antescofo is in constant improvement and evolution. Several directions are envisioned; to name a few:

- 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



22
35
36
36
2
24
26
34
23
25
17
19
18
20
33
WORDS PER MINUTE
10
15
20
25
30
35
40
11
1
13
12
3

Appendix A

Library of Predefined Functions

Antescofo includes a set of predefined functions. They are described mostly in the section 6. 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*. We also qualify as *impure* functions that do not produce a side-effect but that may returns different values when called with the same arguments (for example, the functions that return a random number).

@+ (*value*, *value*), *listable*: prefix form of the infix + binary operator: @+(x, y) ≡ x+y. The functional form of the operator is useful as an argument of 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 operator, the functional form is *not lazy*, cf. sect. 7.2 p. 83: 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.

@aggregate (n1:nim, n2:nim, . . .) aggregates the arguments into a vectorial nim. The number of components of the result is the sum of the number of components of each arguments. This function admits a variable number of argument and cannot be curryfied. See @projection.

@align_breakpoints (nim) returns a nim with linear interpolation type whose (homogeneous) breakpoints are the breakpoints of all components of the arguments. See p. 94 and function @sample and @linearize.

@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 following calls to @approx. Notice that using this function to check if a number is near zero is a bad idea: (x ~ 0) returns always 1 if x is null.

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).

@arch_darwin (): this predicate returns true if the underlying host is Mac OSX and false elsewhere.

@arch_linux (): this predicate returns true if the underlying host is a Linux system and false elsewhere.

@arch_windows (): this predicate returns true if the underlying host is Windows and false elsewhere.

@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 *or* map *or* string, *value*): computes the number of times the second argument appears in the first argument. For a map, the second argument refers to a value stored in the dictionary. See also @find, @member and @occurs.

@dim (t:tab *or* n:nim): if the argument is a tab t, it returns the dimension t, i.e. the maximal number of nested tabs. If the argument is a nim n, it returns the number of elements in the tab returned by the application of the nim. In either case, the returned value is an integer strictly greater than 0. If the argument is not a tab nor a nim, the dimension is 0.

@domain (m:map) returns a tab containing all the keys present in the map m.

@drop (t:tab, n:numeric) gives tab t with its first n elements dropped if n is positive, and tab t with its last n elements dropped if n is negative. See also functions @cdr, @slice and @take.

@drop (t:tab, x:tab) returns the tab formed by the elements of t whose indices are not in x. See also functions @cdr, @slice and @take.

@dump (*file*:string, $variable_1$, ... $variable_p$) is a special form (the $variable_i$ arguments are restricted to be variables). Calling this special form store the values of the variables in the file whose path is *file*.

The stored value can be restored using the function @loadvar.

The dump file can be produced during one program execution and can be read in another program execution. This mechanism can be used to manage *presets*. See also functions @savevalue and @loadvalue.

Note that this special form expands into the ordinary function @dumpvar. The same comments apply. This function is a special form, so it cannot be curryfied.

The dump file produced by @dump is in a human readable format and corresponds to a fragment of the *Antescofo* grammar.

@dumpvar (*file*:string, id_1:string, v_1, ..., id_p:string, v_p) save in *file* the value v_i) under the name id_i). Then *file* can be used by the function @loadvar to define and set or to reset the variable id_i. The format used in *file* is a text format corresponding to the *Antescofo* grammar. See also functions @savevalue and @loadvalue.

The number of argument is variable, so this function cannot be curryfied.

Dumping the values of the variables is done in a separate thread, so the "main" computation is not perturbed. However, it means that the file is created asynchronously with the function call and when the function returns, the file may not be completed.

@empty (*value*): returns true for an empty tab or an empty dictionary and false elsewhere.

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

@explode (s:string): returns a tab containing the characters of the s (represented as string with only one element). For example:

```
@explode("")   ->[]
@explode("abc")  ->["a", "b", "c"]
@reduce(@+, @explode("abc")) ->"abc"
@scan(@+, @explode("abc")) ->["a", "ab", "abc"]
```

@find (t:tab *or* m:map *or* s:string, f:function) returns the index of the first element of t, m or s that satisfies the predicate f. The predicate f is a binary function taking the index or the key as the first argument and the associated value for the second argument. See also @count, @member and @occurs. The undef value (for maps) or the integer −1 (for tab and string) is returned if there is no pair satisfying the predicate.

@filter_max_t (nim, n:numeric) replaces the image y of each breakpoint by the max taken on a sequence made of the n previous and the n next images in addition to y. See p. 96.

@filter_median_t (nim, n:numeric) replaces the image y of each breakpoint by the median taken on a sequence made of the n previous and the n next images in addition to y. See p. 96.

@filter_min_t (nim, n:numeric) replaces the image y of each breakpoint by the min taken on a sequence made of the n previous and the n next images in addition to y. See p. 96.

@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]
```

@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.

@gnuplot (data:tab): The function @gnuplot plots the elements in the data tab as a time series. If data is a tab of numeric values, a simple curve is plotted: an element e of index i gives a point of coordinate (i, e). If data is a tab of tab (of p numeric values), p curves are plotted on the same window. Each @gnuplot invocation lead to a new window. Function @gnuplot returns true if the plot succeeded, and false elsewhere.

To work, the gnuplot program must be installed on the system Cf. http://www.gnuplot.info and must be visible from the *Antescofo* object. They are three ways to make this command visible:

1. set the global variable $gnuplot_path to the absolute path of the gnuplot executable (in the form of a string);
2. alternatively, set the environment variable GNUPLOT of the shell used to launch the *Antescofo* standalone or the Max/PD host of the *Antescofo* object, to the absolute path of the gnuplot executable;
3. alternatively make visible the gnuplot executable visible from the shell used by the user shell to launch the *Antescofo* standalone or the Max/PD host of the *Antescofo* object (*e.g.* through the PATH variable).

The search of a gnuplot executable is done in this order. The command is launched on a shell with the option -persistent and the absolute path of the gnuplot command file. The data are tabulated in a file /tmp/tmp.antescofo.data.n (where n is an integer) in a format suitable for gnuplot. The gnuplot commands used to plot the data are in the file /tmp/tmp.antescofo.gnuplot.n . These two files persists between two *Antescofo* session and can then be used to plot with other option.

The @gnuplot function is overloaded and accepts a variety of arguments described below. The @gnuplot function is used internally by the special forms @plot and @rplot described page 74.

@gnuplot (title:string, data:tab): same as the previous form, but the first argument is used as the label of the plotted curve. If data is a tab of tab, (*e.g.* the history of a tab valued variable), then the label of each curve takes the form title $[i]$.

@gnuplot (time:tab, data:tab): plots the points (time[i], data[i]). As for the previous form, data can be a tab of tab (of numeric values). The time tab corresponds to the x coordinate of the plot and must be a tab of numeric values.

@gnuplot (title:string, time:tab, data:tab): Same as the previous entry but the first argument is used as the label of the curve(s).

@gnuplot (title1:string, time1:tab, data1:tab, title2: string, time2:tab, data2:tab, ...): In this variant, several curves are plotted in the same window. One curve is specified by 2 or 3 consecutive arguments. Three arguments are used if the first considered argument is a string: in this case, this argument is the label of the curve. The following argument is used as the x coordinates and the next one as the y coordinates of the plotted point.

In this variant, the data$i$ arguments must be tab of numeric values (they cannot be tab of tab).

@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.

@history_length (*variable*) : This is a special form. It returns the maximal length of the history of a variable, *i.e.* the number of update that are recorded. See. sect. 6.2.3 page 73 and the @map_history_XXX @tab_history_XXX functions.

@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.

@insert (m:map, k:val, v:val), *impure*: inserts "in place" a new entry k with value v in the map m. See @add_pair. Notice that the form @insert " file " is also used to include a file at parsing time.

@integrate *not yet documented*

@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 (map, *value*): the predicate returns true if the second argument is a key present in the first argument. Do not mismatch with the negation of the predicate @is_undef.

@is_exec (*value*): the predicate returns true if the argument represents an *exec*, that is, the instance of a compound action (especially the result of a process call). The predicate returns true even if the compound action has finished its computation. However, the *exec* itself used in a boolean condition evaluates to true if the compound action is still running and false elsewhere.

@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_obj (*value*): the predicate returns true if its argument is an object (implemented by a running process).

@is_obj__xxx(*value*) where xxx is the name (without the prefix obj::) of an object defined through @obj_def. This predicate is automatically generated with an object definition and check that a value represents an instance of obj:: xxx.

@is_prefix (s1:string, s2:string): the predicate returns true if s1 is a prefix of s2. See the overloaded versions below and also the functions @is_suffix and @is_subsequence.

@is_prefix (s1:string, s2:string, cmp:fct): the predicate returns true if s1 is a prefix of s2 where the characters are compared with the function cmp. The characters are passed to the function cmp as strings of length one.

@is_prefix (t1:tab, t2:tab): the predicate returns true if t1 is a prefix of t2, that is, if the elements of t1 are the final elements of t2.

@is_prefix (t1:tab, t2:tab, cmp:fct): same as the previous version but the function cmp is used to test the equality between the elements, instead of the usual comparison between values. For example:

```
@fun_def cmp($x, $y) { $x > $y }
@is_prefix([11, 22], [5, 6, 7], @cmp) ->true
```

true is returned because @cmp(11, 6) and @cmp(22, 7) hold.

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

@is_subsequence (s1:string, s2:string): the function returns the index of the first occurrence of string s1 in string s2. A negative value is returned if s1 does not occurs in s2. See the overloaded versions below and also the functions @is_prefix and @is_suffix .

@is_subsequence (s1:string, s2:string, cmp:fct): same as above but the argument cmp is used to compare the characters of the strings (represented as strings of only one element).

@is_subsequence (t1:tab, t2:tab): the predicate returns the index of the first occurrence of the elements of t1 as a sub-sequence of the elements of t2. A negative value is returned if t2 does not appear as a subsequence of tab t2. For example

```
@is_subsequence([], [1, 2, 3])          ->0
@is_subsequence([1, 2, 3], [1, 2, 3])   ->0
@is_subsequence([1], [1, 2, 3])         ->0
@is_subsequence([2], [1, 2, 3])         ->1
@is_subsequence([3], [1, 2, 3])         ->2
@is_subsequence([1, 2], [0, 1, 2, 3])   ->1
```

@is_subsequence (t1:tab, t2:tab, cmp:fct): same as the version above but the function cmp is used to compare the elements of the tabs.

@is_suffix (s1:string, s2:string): the predicate returns true if s1 is a suffix of s2. See the overloaded versions below and also the functions @is_prefix and @is_subsequence.

@is_suffix (s1:string, s2:string, cmp:fct): the predicate returns true if s1 is a suffix of s2 where the characters are compared with the function cmp. The characters are passed to the function cmp as strings of length one.

@is_suffix (t1:tab, t2:tab): the predicate returns true if t1 is a suffix of t2, that is, if the elements of t1 are the final elements of t2.

@is_suffix (t1:tab, t2:tab, cmp:fct): same as the previous version but the function cmp is used to test the equality between the elements, instead of the usual comparison between values. For example:

```
@fun_def cmp($x, $y) { $x < $y }
@is_suffix([1, 2], [5, 6, 7], @cmp) ->true
```

true is returned because @cmp(1, 6) and @cmp(2, 7) hold.

@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. Do not mismatch with the negation of the predicate @is_defined.

@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, "
```

@last (t:tab) returns the last element of a tab, or undef.

@linearize (nim, tol:numeric) build a new nim that approximates the argument.The new nim uses only linear interpolation and homogeneous breakpoints. An adaptive sampling step achieve an approximation within tol (*i.e.* for any point in the domain, the images of the nim argument and the nim result are withing tol. See p. 94 and function @sample and @align_breakpoints.

@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.

@loadvalue (*file*:string) : read a file produced by a call to the function @savevalue and returns the value that was saved. If something goes wrong, an undefined value is returned. See also function @dump, @dumpvar and @loadvar.

Because the variables are supposed to have their saved values right after the call to @loadvar, we deliberately use a synchronous scheme where the function returns after having completed all the initializations. A call to this function may take a noticeable time depending on the size of the values to store in the dump file. While this time is usually negligible, loading a tab of 10 000 integers represents a file of size about 60 Kb and takes between 2 ms and 3 ms. This computational cost may have a negative impact on the audio processing in heavy cases. However, the intended use of @loadvalue and @loadvar functions is to restore a "preset" at isolated places like the beginning of the score or between musical sequences, a usage where this cost should have no impact. Notice that saving a value or variables is done *asynchronously* and does not disturb the "main" computation. See the remarks of function @dump.

@loadvar (*file*:string) : read a file produced by a call to @dump (or @dumpvar) and set the value of the corresponding variables.

The basic use of @loadvar is to recover values of global variables that have been previously saved with a @dump command. If the loaded variable is not defined at the calling point, a new global variable is implicitly defined by @loadvar. If the variable with the same name exist at the calling point, either global or local, this variable will be set with the saved value.

To be more precise, when @dump is called, the name of the variables in the arguments list are stored as well as the corresponding values in *file*. The variables in the argument list can be global or local variable.

When @loadvar is called, it is called in some scope *sc*. Each identifier in the dump file is searched in the current scope *sc*. If not found, the englobing scope is looked up, and the process is iterated until a variable is found or until reaching the global scope. If no variable with the same identifier is found, a global variable with this identifier is created. The value associated to the identifier is used to set the selected variable.

Beware that @loadvar does not trigger the whenevers.

Nota Bene: because @loadvar can be called in a context which differs from the context of the call of @dump, there is no reason that the 'same' variables will be set.

Here is an example:

```
@global $a, $b, $c
$a := 1
$b := 2
$c := 3
$ret := @dump("/tmp/dump1", $a, $b, $c)
$a := 0
$b := 0
$c := 0
Group G1
{
  @local $b
  $b := 22
  Group G2
  {
    @local $c
    $c := 33
    $ret := @loadvar("/tmp/dump1")
    print $a $b $c  ; print 1 2 3
  }
  print $a $b $c
  ; print 1 2 0
  ; because the variable $c set by @loadvar is in G1
}
print $a $b $c ; print 1 0 0

$ret := @loadvar("/tmp/dump1")
print $a $b $c ; print 1 2 3
```

@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 () *or* (start:numeric) *or* (start:numeric, stop:numeric):
@make_duration_map(a, b) returns a map where the duration (in beat) of the ith event of the score, is associated to i (the keys of the map are the ranks of the events). Called with no arguments, the events considered are all the events in the score. With start, only the events whose position is greater than start are considered. If a stop is specified, all events must have a position between start and stop.

@make_label_pos () *or* (start:numeric) *or* (start:numeric, stop:numeric): this function returns a map whose keys are the labels of the events and the value, the position (in beats) of the events. Events with no label do not appear in the map. Called with no arguments, the events considered are all the events in the score. With start, only the events whose position is greater than start are considered. If a stop is specified, all events must have a position between start and stop.

@make_label_bpm () *or* (start:numeric) *or* (start:numeric, stop:numeric): returns a map associating the event labels to the BPM at this point in the score. Events with no label do not appear in the map. Called with no arguments, the events considered are all the

events in the score. With start, only the events whose position is greater than start are considered. If a stop is specified, all events must have a position between start and stop.

@make_label_duration () *or* (start:numeric) *or* (start:numeric, stop:numeric): returns a map associating to the event of a label, the duration of this event. Events with no label do not appear in the map. Called with no arguments, the events considered are all the events in the score. With start, only the events whose position is greater than start are considered. If a stop is specified, all events must have a position between start and stop.

@make_label_pitches () *or* (start:numeric) *or* (start:numeric, stop:numeric): returns a map associating a vector of pitches to the label of an event. A NOTE corresponds to a tab of size 1, a CHORDS with n pitches to a tab of size n, *etc.*. Events with no label do not appear in the map. Called with no arguments, the events considered are all the events in the score. With start, only the events whose position is greater than start are considered. If a stop is specified, all events must have a position between start and stop.

@make_score_map () *or* (start:numeric) *or* (start:numeric, stop:numeric): @make_score_map(a, b) returns a map where the keys are the rank i of the musical events and thevalue, the position (in beat) of the ith event. Called with no arguments, the events considered are all the events in the score. With start, only the events whose position is greater than start are considered. If a stop is specified, all events must have a position between start and stop.

@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_history (*variable*) : This is a special form. It returns a map of the value of the variable in argument. See. sect. 6.2.3 page 73 and the @tab_historyXXX functions.

@map_history_date (*variable*) : This is a special form. It returns a map of the date in physical time of the updates of the variable in argument. See. sect. 6.2.3 page 73 and the @tab_historyXXX functions.

@map_history_rdate (*variable*) : This is a special form. It returns a map of the date in relative time of the updates of the variable in argument. See. sect. 6.2.3 page 73 and the @tab_historyXXX functions.

@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_key (nim): return the coordinate x_n of the last breakpoint o the nim. This coordinate is the sum of x_0 (the coordinate of the first breakpoint of the nim, and of all intervals d_i of the breakpoints i. If the nim is vectorial, x_n is a tab. See sect. 8.3.

@max_val (tab *or* map *or* nim): 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 or a nim.

@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 *or* map *or* string, *value*): returns true if the second argument is an element of the first. For a map, the second arguments refers to a value stored in the dictionary. See also @count, @find and @occurs.

@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_key (nim): return the coordinate x_0 of the initial breakpoint. If the nim is vectorial, x_0 is a tab. See sect. 8.3.

@min_val (tab *or* map *or* nim): returns the minimal element present in the tab or the minimal element in the domain (if the argument is a map or a nim)

@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 *or* map *or* string, *value*): returns the first index or the first key whose value equals the second argument. For example

```
@occurs(["a", "b", "c", "a", "b"], "b") ->1
@occurs("xyz", "z")  ->2
@occurs(map{ ("zero", 0), ("null", 0), ("void", 0) }, 0)  ->"null"
```

In the last example, the answer "null" is returned because "null" < "void" < "zero". See also @count, @find and @member.

@permute (t:tab, n:numeric) returns a new tab which contains the nth 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]
```

@plot (*variable*) or @plot(*variable*$_1$, ... *variable*$_p$) is a special form (the arguments are restricted to be variables). Calling this special form plots the values stored in the history of the variables as time series in absolute time using the gnuplot function. See page 74 and also the library function @rplot.

@pow (numeric, numeric), *listable*: @pow(x, y) computes x raised to the power y.

@priority () returns the priority of the action where this function is called. The results is a tab with two elements: the first is the static priority and the second is the exec of the current action.

@projection (nim, p:numeric) extracts the pth component of a vectorial nim. See @aggregate.

@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_back (nim:NIM, d:numeric, y1:numeric, it:string), *impure*: add a breakpoint as the last breakpoint of nim. The first argument, modified by side-effect, is the nim, which is also the returned value. The argument d specifies the length of the interpolation since the previous breakpoint, y1 is the final value attained at the end of the breakpoint, and it is the interpolation type. The interpolation type can be omitted: in this case, the interpolation is linear. The initial value y0 of the breakpoint is the y1 value of the previous breakpoint.

@push_back (nim:NIM, y0:numeric, d:numeric, y1:numeric, it:string),*impure*: similar to the previous function but y0 is explicitly given , making possible to specify discontinuous NIM.

@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.

@range (m:map) returns the tab of the values present in the map. The order in the tab is irrelevant.

@remove (t:tab, n:numeric), *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 (m:map, k:val), *impure*: removes the entry k in map m (m is modified in place). Does nothing if the entry k is not present in map m.

@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): returns a new tab where the element are given in the reverse order.

@reverse (string): returns a new string where the character are given in the reverse order.

@rnd_bernoulli (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.

@rplot (*variable*) or @rplot(*variable*$_1$, ... *variable*$_p$) is a special form (the arguments are restricted to be variables). Calling this special form plots the values stored in the history of the variables as time series in relative time using the `gnuplot` function. See page 74 and also the library function @plot.

@sample nim, p:numeric build a new nim with linear interpolation type by sampling each component at `p` points equally spaced between the first and the last breakpoint. See p. 94 and function @linearize and @align_breakpoints.

@savevalue (s:string, *value*), *impure*: argument is interpreted as the path of a file where the value of the second argument is saved. The format of the file is textual and corresponds to the *Antescofo* grammar. This value can then be read using the function @loadvalue. See also functions @dump, @dumpvar and @loadvar.

@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

@simplify_lang_v (nim, tol:numeric, n:numeric) simplifies a nim by aggregating breakpoints using a Lang polyline simplification algorithm. See p. 96.

@simplify_radial_distance_t (nim, tol:numeric) simplifies a nim by aggregating consecutive breakpoints whose images are wihin a distance of tol. See p. 95.

@simplify_radial_distance_v (nim, tol:numeric) simplifies a nim independantly for each component, by aggregating consecutive breakpoints that are wihin a distance of tol. See p. 96.

@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 or a tab, it returns the number of element in its argument (which is a positive integer). If it is a nim, it returns the number of breakpoints of the nim (which is not the dimension of the nim). Note that a nim with zero breakpoints is the result of a wrong definition.

@slice (t:tab, n:numeric, m:numeric) gives the elements of t of indices between n included up to m excluded. If $n > m$ the element are given in reverse order. So

```
@slide(t, @size(t), 0)
```

is equivalent to

```
@reverse(t)
```

See also functions @cdr, @drop and @take.

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

@sort (t:tab, cmp:fct) *impure*: 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`.

@string2fun (s:string): returns the function whose name is given in argument. The initial `@` in the function name can be omitted. Useful to convert directly a string received through OSC or through the `setvar` message into a function that can be applied.

@string2proc (s:string): returns the proc whose name is given in argument. The initial :: in the proc identifier can be omitted. Useful to convert directly a string received through OSC or through the `setvar` message into a function that can be applied.

For example, assuming that `$channel` is set in the Max environment to a tab of two element, the first being a process identifier and the second an integer, then the code

```
whenever($channel)
{
      :: (@string2proc($channel[0])) ($channel[1])
}
```

will react to the assignment to `$channel` by calling the corresponding processes with the specified integer.

@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 ]
```

@system (cmd:string), *impure*: This function hands the argument command to the command interpreter sh. The calling process waits for the shell to finish executing the command, ignoring SIGINT and SIGQUIT, and blocking SIGCHLD. A false boolean value is returned if an error occured (in this case an error message is issued).

@tab_history (*variable*) : This is a special form. It returns a tab of the value of the variable in argument. See. sect. 6.2.3 page 73 and the @map_historyXXX functions.

@tab_history_date (*variable*) : This is a special form. It returns a tab of the date in physical time of the updates of the variable in argument. See. sect. 6.2.3 page 73 and the @map_historyXXX functions.

@tab_history_rdate (*variable*) : This is a special form. It returns a tab of the date in relative time of the updates of the variable in argument. See. sect. 6.2.3 page 73 and the @map_historyXXX functions.

@take (t:tab, n:numeric) gives the first n elements of t if n is a positive integer and the last n elements of t if n is a negative integer.

@take (t:tab, x:tab) gives the tab of elements whose indices are in tab x. Equivalent to

```
[t[x[$i]] | $i in @size(x)]
```

See also functions @cdr, @drop and @slice.

@tan (x:numeric), *listable*: computes the tangent of x (measured in radians).

@Tracing () : start to trace the calls to all user-defined functions.

@Tracing (x:function *or* string *or* tab, ...) : start to trace the calls to all functions specified by the arguments. Functions to trace are given by their name (as a string) or by their value (through their identifier) or by a tab containing such values.

@UnTracing () : stop to trace the calls to all user-defined functions.

@UnTracing (x:function *or* string *or* tab, ...) : stop to trace the calls to all functions specified by the arguments. Functions to trace are given by their name (as a string) or by their value (through their identifier).

@window_filter_t (nim, coef:tab, pos:numeric) computes a new nim, where the image y_0 of each breakpoint is replaced by the dot product of coef with a sequence of y's of the same length as coef where position pos corresponds to y_0. See p. 97.

Appendix 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 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.

B.2 Constant BPM expresssion

It is possible to write *constant expression* in a BPM specification.

```
BPM (1.1*120)
```

Constant expression must have a well-defined value when loading the score. Variables cannot appears in a constant expression but macro-definition combines well

```
@macro_def @MyTempo { 120 }
; ...
BPM @MyTempo
; ...
BPM (@MyTempo + 10)
```

B.3 @eval_when_load Clause

A @eval_when_load clause specifies a list of actions that must be performed when a file is loaded. The action are evalueted only when the file is loaded. Several such clauses may exists in a file: they are performed in the order of appearance right after having completed the parsing of the full score.

Such clause can be used for instance to read some parameter saved in a file or to precompute some values. For example

```
@fun_def fib($x)
{
  if ($x < 2) { return 1 }
  else { return @fib($x-1) +@fib($x-2) }
}

@eval_when_load {
  $fib36 := @fib(36)
}

; ...

NOTE C4
  print $fib36
```

When this file is loaded, the @eval_when_load clause is evaluated to compute @fib(36) (which takes a noticeable amount of time). This value is then used when the program is strated and the event C4 occurs, without requiring a costly computation.

By using @insert, @insert_once, @eval_when_load and preload commands, and @dumpvar, @loadvar, @loadvalue and @savevalue functions, one is able to manage easely a library of reusable functions and *reusable setups* mutualized between pieces.

B.4 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[1] (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
```

B.5 Abort Handler

An abort command can be used to stop a compound action before its natural end, cf. section 4.4.1. 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[2].

[1]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

[2] 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.

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[3]. 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.

When 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 } }
}
```

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.
```

[3] with the current exception of whenever

B.6 Continuations

Performing an action at the end of a group may be difficult if the delays of the group's actions are expressions or if some conditional constructs are involved. Even with constant delay and no control structure, computing the duration of a group, a loop/whenever/forall body, can be cumbersome.

This observation advocates for the introduction of two additional sequencing operators that are used to launch an action at the end of the preceding one:

a b	a ==>b	a +=>b
(no special name)	*followed-by*	*ended-by*
b is launched together with a	b is launched at the end of a	b is launched at the end of a *and* its eventual children.

The juxtaposition (launch together), the followed-by operator ==> (launch at the end) and the ended-by operator +=> (launch at child ends) are called *continuation combinators*[4]. They compose freely between actions and are right associative, see table B.6. To integrate the delays in this picture, it is convenient to look at the delays as actions that do nothing during the specified amount of time.

a ==>b ==>c	equivalent to a ==>{ b ==>c} starts b ==>c at the end of a
{a ==>b} ==>c	starts c at the end of a ==>b, that is, with the end of b
{a ==>b} c	starts c with the start of a ==>b, that is, with the start of a
a ==>b c	equivalent to a ==>{b c} starts {b c} with the start of a
a b ==>c	equivalent to a {b ==>c} starts {b ==>c} with the end of a (which means that b starts with a and c with the end of b)
a +=>b ==>c	equivalent to a +=>{b ==>c} starts {b ==>c} at the end of a and its children
etc.	*etc.*

Table B.1: Some examples of continuation expressions.

For example, suppose we want to make an action after the end of a loop:

```
$cpt := 0
Loop 1
   {
```

[4]Do not to confuse the *continuation* defined here with the notion of *continuation* in functional language, used sometimes to implement concurrency.

```
        print "tic" $cpt
        4 print "tac" $cpt
        $cpt := $cpt + 1
} during [3#]
+=> print "loop␣ended"
```

Here there will be 3 iterations of the loop. So, if the loop starts at date 0, the first iteration starts at 0 and ends at 3, the second one starts at 1 and ends at 4 and the last one starts at 2 and finishes at 5.

Instead of computing explicitly these numbers to launch an action at the right time, we have used the continuation +=> which waits the end of the loop and all the loop bodies, to trigger the print message: the message "loop␣ended" will appear at date 5.

As you can see, the end of a loop is distinct from the ends of the loop bodies: the loop in itself terminates when the last iteration is launched. As a matter of fact, the computation associated to a compound action a can be seen as a tree, with the sub-computations rooted at a. Thus, there is no need to maintain a after having launched the last sub-computation. So the end of a is usually not the same as the end of the last sub-computation spanned by a and this is why the +=> operator is usually more handy than ==>.

Nevertheless, the end of an action is always precisely defined altough it can be only dynamically known:

atomic action the start and the end of the action coincide.

coumpounds action without duration are actions that launch other action but they do not have a duration by itself because they do not need to persist in time. Examples of such actions are the Forall, the Switch and the If constructs.

From this point of view, they behave like an atomic action. So the start and the end of these actions coincide. They have no children (the actions launched by these constructs are children of the enclosing group).

compound action with a duration and childrens the start and the end of these actions usually differ:

- Group G { a ... b }: the start of G coincides with the start of a.
  The end of G coincides with the start of b (last action in the group).
  The children of G are all actions launched directly (they appear explicitly in the group body) or indirectly (they are launched by a child of G).
- Loop L { a ...} the start of L coincides with the start of the first iteration of a.
  The end of L coincides with the last iteration of a.
  The children of L are the actions launched in the loop bodies.
- Whenever W { ... }: there is no relationships between the start of W and the actions in the body.
  Usually, there is no end to a whenever except if there is a during or an until clause.
  In this case, the whenever terminates when the clause becomes true.
  The children of W are the actions launched by the whenever body.
- ::P(): the end of a process call corresponds to the end of the called process.
  The children of ::P() are the actions launched by the process body.
  So the expression ::P() ==> b will launch b at the end of P's body.

Continuation and abort. An abort handler, defined by the @abort attribute, is considered as a child of the associated action. So, when an abort handler exists, and the associated actions is aborted, the abort handler is launched with the followed-by continuation (if it exists). Because the abort handler is necessarily defined before, it happens before the followed-by continuation. The ended-by handler is launched after the end of the abort handler. Continuations are not considered as childs of the continued actions. So in a ==>b, b do not has access to the local variable of a, contrary to the @abort clause of a.

For example (note the bracketing of the process call):

```
@proc_def ::P()
@abort { print abort P $NOW }
{
  print start P $NOW
  10 print BAD END P $NOW
}

{ ::P() ==> print continuation P $NOW }

5
print "launch␣abort" $NOW
abort ::P
```

will give the following trace:

```
start P 0.0
launch abort 5.0
abort P 5.0
continuation P 5.0
```

If the abort handler is replaced by:

```
@abort { 11 print abort P $NOW }
```

the corresponding trace is:

```
start P 0.0
launch abort 5.0
continuation P 5.0
abort P 16.0
```

because the followed-by continuation does not wait the end of the abort handler. So, if we replace the followed-by continuation by an ended-by continuation

```
{ ::P() +=> print continuation P $NOW }
```

the trace becomes:

```
start P 0.0
launch abort 5.0
abort P 16.0
continuation P 16.0
```

B.7 Open Scores and Dynamic Jumps

The jump attribute of an event (cf. 2.3) is used to specify the possible “continuations” of the score (as a list of labels). This feature makes possible to escape the customary linearity of

a score to specify *open score* where the musician may choose between several alternative to proceed.

It is possible to use a variable instead of a fixed list of labels. This variable must refer to an event position or an event label (through a string) or a tab of them. This variable may change value in the course of the performance. In this way, it is possible to achieve "*dynamic* open score" where the graph of the possibilities is updated following the choices made by the musician, external events, internal computations, etc.

Dynamic changes in the score graph (through the variables appearing in the jump attribute) affect the listening machine. The listening machine maintains a set of hypothesis about the potential events to recognize in the audio stream. If the changes in the score graph are anticipated enough with respect to the actual jumps, the listening machine will automatically accommodate the changes.

However, if the computation of the jumps are not far enough in time from the actual jumps, the listening machine must be explicitly warned. This is done by setting to true the *system variable* $JUMP_UPDATED when the modifications are done. It is not easy to define what it means "far enough" because the temporal horizon used by the listening machine is adaptive.

Here is a toy example:

```
$jumps := [ "begin_part2", "begin_part3", "next_part" ]
$part2_done := false
$part3_done := false

// INTRO
NOTE G3 1 INTRO
; ...
NOTE G4 1 @jump $jumps

// PART2
NOTE D2 1 begin_part_2
    $part2_done := true
    ; ...
NOTE D3 1 end_part_2 @jump INTRO
    $jumps := if ($part3_done) { "next_part" }
              else { ["begin_part3", "next_part"] }
    $JUMP_UPDATED := true

// PART3
NOTE E3 1 begin_part_3
    $part3_done := true
    ; ...
NOTE E5 1 end_part_3 @jump INTRO
    $jumps := if ($part2_done) { "next_part" }
              else { ["begin_part2", "next_part"] }
    $JUMP_UPDATED := true

// NEXT_PART
; ...
```

In this example, the musician may choose to perform one of the following five scenarios:

```
INTRO → NEXT_PART
INTRO → PART2 → INTRO → NEXT_PART
INTRO → PART2 → INTRO → PART3 → INTRO → NEXT_PART
```

INTRO → PART3 → INTRO → NEXT_PART
INTRO → PART3 → INTRO → PART2 → INTRO → NEXT_PART

B.8 Tracing Function Calls

It is possible to (un)trace the calls to a function during the program run with the two predefined functions: @Tracing and @UnTracing. The trace is emitted on Max or PD console (or on the output specified by the -message option for the standalone).

The two predefiend functions admit a variety of arguments:

- no argument: all user-defined function are traced/untraced.
- the functions to trace/untrace: as in@Trace(@in_between, "@fib"), will trace/untrace the call and the returns to the listed functions. Notice that the function to (un)trace can specified with their name or via a string.
- a tab that contains the functions to (untrace through their name or through strings.

Here is an example:

```
@fun_def @fact($x) { if ($x < 1) { 1 } else { $x * @fact($x-1) } }
_ := @Tracing(@fact)
_ := @fact(4)
```

which generates the following trace:

```
+--> @fact($x=4)
|     +--> @fact($x=3)
|     |     +--> @fact($x=2)
|     |     |     +--> @fact($x=1)
|     |     |     |     +--> @fact($x=0)
|     |     |     |     +<-- 1
|     |     |     +<-- 1
|     |     +<-- 2
|     +<-- 6
+<-- 24
```

B.9 Infix notation for function calls

A function call is usually written in prefix form:

```
@drop($t, 1)
@scramble($t)
```

It is possible to write function calls in *infix* form, as follows:

```
$t.@drop(1)
$t.@scramble()
```

The `@` character is optional in the naming of a function in infix call, so we can also write:

```
$t.drop(1)
$t.scramble()
```

This syntax is reminiscent of the function/method call in *SuperCollider*. The general form is:

```
arg1 . @fct(arg2, arg3, ...)       ; or more simply
arg1 . fct(arg2, arg3, ...)
```

The arg_i are expressions. Notice that the infix call, with or without the `@` in the function name, is not ambiguous with the notation used to refer to a variable local `$x` in a compound action from the exe of this action, *exe*.`$x`, because `$x` cannot be the name of a function.

The infix notation is less general than the prefix notation, because in the prefix notation, the function can be given by an expression. For example, functions can be stored into an array and then called following the result of an expression:

```
$t := [@f, @g]
; ...
($t[exp])()
```

will call `@f` or `@g` following the value returned by the evaluation of *exp*. Only function name (with or without `@`) are accepted in the infix notation. The interest of this notation will become apparent with the notion of *method* presented in the next section.

B.10 Methods (*or* Running Processes as Concurrent Objects)

A process instance can be used as a kind of autonomous entity. In fact, a running process can be seen as an object or as an actor[5]:

- a process instance is similar to the instance of a class: the process is the class and calling a process corresponds to class instantiation;
- the *exe* correspond to the reference to an object;
- the state of the object (running process) corresponds to the values of its local variables;
- interactions with the object can be achieved by assigning its local variable[6].

Local variable assignments act as messages and in response to a message that it receives, a running process can make local decisions, create more processes, send more messages, and determine how to respond to the next message received. For example:

```
@proc_def ::channel($ch, $init, $period)
{
      @local $velocity, $tab, $i
      $velocity := $init[0]
      $i := 0
```

[5]The actor model of programming has been developed in the begining of the '70, with the work of Carl Hewitt and languages like `Act`. Later Actor programming languages includes the Ptolemy programming language, and languages offering "parallel object" like Scala or Erlang.

[6] see sect. 12.5, relying on the *exe* of the runing process and the dot notation.

```
        whenever ($velocity == 0) @immediate
        { $tab := init }

        whenever ($velocity)
        { _ := @push_back($tab, $velocity) }

        Loop $period
        {
             $i := ($i + 1) % @size($tab)
             @command{ "harmo" ++ $ch} ($velocity[$i])
        }
}
; ...

$p0 := ::channel(0, [12, 15], 1)
$p1 := ::channel(1, [10, 8, 9, 11], 1.5)
; ...

$p0.$velocity := 15
; ...
```

An object (process'instance) of class (process) ::channel is supposed to control some audio channel. The object iterates periodically over a list of parameters to be sent to an harmonizer. By assigning the local variable $velocity, a new parameter is added to the list. By assigning it to 0, this list is set to its initial value.

The dot notation is efficient but does not make apparent the interactions with the object (running process). Functions can be used to make these interaction more explicit. Suppose we want simultaneously change the period and reset the parameter list to its initial value. We can write a function:

```
@fun_def reset($pid, $per)
{
    $pid.$period := $per
    $pid.$velocity := 0
}
```

then we can call the function @reset:

```
@reset($pid, 1.5)
```

and using the infix notation for function call introduced in section B.9:

```
$pid.reset(1.5)
```

This last form is in line with the usual notation used to call an object's method: we ask the object (specified through its *exe*) to perform the *method* reset with parameter 1.5.

However, the definition of the "method" @reset makes apparent the "coding" of an object style in *Antescofo*: the programmer has to specify an additional argument $pid and to prefix every access to an object slot $x (a process local variable) by $pid.. Furthermore, nothing prevents to apply the function reset to an instance of a process ::Q instead of ::P, as long as ::P and ::Q defiens the local variable accessed in @reset[7].

[7]Despite these shortcomings, this translation of an object method into a function to be applied on the object, is the basic implementation scheme used for instance for ordinary methods in C++.

To simplify the writing of methods, and to make the link between the method and the process more apparent, on can define a *method* instead of a function. Method definitions are introduced using the keyword @method_def followed by the name of the process referred by the method :

```
@method_def<::channel> reset($per)      ; the process ::channel can
{                                       ; be specified without the '::'
    $period := $per
    $velocity := 0
}
```

The rest of the method definition is similar to a function definition, except that one can refer to the local variables of the process specified between < > in the header, without the dot notation. The body of a method is an extended expression (see page 120). The variable that are not local and do not appear as local variables of the process, are assumed to be global.

A method is implemented internally by a function (with one additional argument) so its an efficient mechanism. But methods have some distinguishing advantage:

- A method can be overloaded, that is, the same method name can be used for different processes.
- When called, a method checks that is is called on a live instance of the specified process.
- If a method `m` is called on a dead process, the function `@m` is called instead if it exists. If function `@m` does not exists, an error is signaled.

These benefits come at some cost:

- A method can be called only through the infix call notation `obj . method(...)` .
- Methods are not first class values (*e.g.* you cannot pass them as argument) but you can apply them partially (and use the partial application which is a first class value).
- There is only one possible ambiguity in infix function/method call: when the function `@f` is called through the infix notation without the `@`, and the first argument of the function is an exe of a live process on which a method f is defined. In this case, the rule is to call the method. If you want to call the function, use the `@`-identifier in the call: `exp.@f(exe, ...)` .

B.11 Objects

The previous idea — relying on processes to achieve a kind of concurrent object oriented programming — is pushed further with the @obj_def construction. An @obj_def definition is internally expanded into a process definition and into methods and functions definitions. So, there is no real new mechanism involved. However, the dedicated syntax makes the programming much readable and reusable.

An *obj* definition is introduced by the @obj_def keyword and consists in a sequence of clauses:

@local introduces the declaration of the *fields* (also known as the *attribute*) of the object;

@init defines a sequence of actions that will be launched when the object is instantiated;

@method_def or **@fun_def** specifies a new method, *i.e.* a function that can be run on a specific obj; such method are also named *instance method* or *object method* because they involve a specific instance of an object; the body of a method is an extended expression;

@proc_def specifies a new method, which is similar to the previous construction, except that the body of a routine is a sequence of actions (not an extended expression); this methods are sometimes called *routines*;

@broadcast declares a function that performs simultaneously on all instance of an object[8];

@whenever (respectively @react) introduces a *daemon* which triggers a sequence of actions (respectively an extended expression) when some logical expression becomes true;

@abort defines an abort handler that will be triggered when the obj is killed.

A clause of a given type may appears several times in an object definition.

B.11.1 A basic example

An example of object definition is given in fig. B.1. The object is called obj::Metro and correspond to a *type*. This type can be instantiated by giving the expected argument for the object creation:

```
$metro1 := obj::Metro(2/3, "left_channel")
$metro2 := obj::Metro(1, "right_channel")
```

An object of type obj::Metro is created with an initial period $p and send a message top to the receiver $receiver each period. The loop implementing the periodic emission of the top is triggered by a whenever controlled by a field (a local variable) $trigger. The exec of this loop is saved in field $body and used to abort the loop when the reset is broadcasted.

Two methods are provided: set_period is used to change the value of the period (the change is taken into account at the end of the current period) and current_period is used to query the period actually used by a Metro obj. The signal reset can be used to reset the period of all running instances of a obj::Metro to their initial value (the value given at creation time). Here are some examples:

```
_ := $metro1.set_period(2 * $metro2.current_period())
```

sets the period of the first Metro object to twice the period of the second Metro object. All period are reset calling the broadcast:

```
_ := @reset()
```

[8]Broadcast are reminiscent of *static method* because they do not involve a particular object instance, but concern all instances. Notice however that contrary to the static method in Java or C++, broadcast executes a code for each instance, not once for the class.

Figure B.1 Example of an object definition.

```
@obj_def  Metro($p, $receiver)
{
     @local $period, $trigger, $body

     @init {
       $trigger := false
       $body := 0
     }

     @whenever ($trigger)
     {
         $body := { Loop $period { @command($receiver) top } }
     }

     @init {
       $period := $p
       $trigger := true
     }

     @broadcast reset()
     {
         abort $body
         $period := $p
         $trigger := true
     }

     @method_def current_period() { return $period }
     @method_def set_period($x)   { $period := $x }

     @abort { print "object␣" $THISOBJECT "is␣killed" }
}
```

Note that a broadcast corresponds to an ordinary function. This function launch simultaneously, for all active Metro instances, the code associated to the broadcast.

Notice that the obj:: Metro definition defines two @init clauses: the first one takes place before the @whenever and initialize the field of the object. The second @init is used to launch the loop when the object is created and after the initialization of the whenever.

An object lives “forever”. It can be killed and the @abort clause is used to execute a code at object termination. In the example, the abort handler uses the system variable $THISOBJ that refers, in the scope of an object clause, to the current instance of the object.

B.11.2 Object definition

An object definition plays a role similar to a *class* in object-oriented programming, except that there is no notion of class inheritance in the current *Antescofo* version. Another difference is that objects run "in parallel" and their actions are subject to synchronization with the musician or on a variable, they can be performed on a given tempo,*etc*. As a matter of fact, as previously mentioned, objects are processes with some syntactic sugar.

Field definition: @local

The @local clause has the same syntax as the @local declaration used to introduce local variables in a compound action. Here each "local variable" is used as a field of the object and corresponds to a local variable in the process that implements the object. The values of the fields/local variables represents the state of the object. *Antescofo* is a dynamically typed programming language, so the fields of an object have no specified type and can hold any kind of values in the course of time.

Several @local clauses can be defined and their order and placement is meaningless. Object fields are present from the start and initialized with the undef value.

Note that the argument of an object corresponds to implicitly defined fields. So, in the example given in Fig B.1, the state of the object is given by 5 variables: the initial period $p, the receiver $receiver , the current period $period, a control variable $trigger and a reference to the actual loop that implements the object behavior $body.

A reference to a field may appear anywhere in a clause and always refers to the corresponding local variable. A variable identifier that is not declared as a local variable, refers to a global variable.

Performing action at the object construction: @init

Fields are initialized in @init clauses. Init clauses are interleaved with @whenever clauses and this order is preserved in the implementation, which makes possible to control the order of evaluation and the triggering of the whenever clauses.

Specifying an object method: @method_def and @proc_def

Method definitions[9] are introduced using the keyword @method_def or @fun_def followed by the name of the method and its argument:

```
@method_def reset($per)
{
    $period := $per
    $velocity := 0
}
```

[9]We already encountered the notion of methods in section B.10 p. 172. Such methods are called *external* because they are defined outside the scope of a @obj_def construct. So they can be defined after a process definition or an object definition.

Methods defined through the @method_def clauses within an object definition are called *internal*. There is no need to specify the referred process between < ... > because internal methods re methods defined for the object at hand.

The rest of the method definition is similar to a function definition, except that one can refer to the fields of the object without the dot notation. The body of a method is an extended expression (see page 120). The variable that are not local and do not appear as object's field, are assumed to be global.

A method is implemented internally by a function (with one additional argument) so its an efficient mechanism. But methods have some distinguishing advantage:

- A method can be overloaded, that is, the same method name can be used for different objects (but methods name are unique within an object).
- When called, a method checks that is is called on a live instance of the specified process.
- If a method `m` is called on a dead process, the function `@m` is called instead if it exists. If function `@m` does not exists, an error is signaled.

These benefits come at some cost:

- Outside an object, method can be called only through the infix call notation `obj . method(...)` .
- Methods are not first class values (*e.g.* you cannot pass them as argument) but you can apply them partially (and use the partial application which is a first class value).
- There is only one possible ambiguity in infix function/method call: when the function `@f` is called through the infix notation without the `@`, and the first argument of the function is an exe of a live process on which a method f is defined. In this case, the rule is to call the method. If you want to call the function, use the `@`-identifier in the call: `exp.@f(exe, ...)` .

All method calls in the object definition which refers to the current object instance, can be written in an abbreviated infix form that omit the receiver:

. *methodname* (. . .)

instead of

$THISOBJ . *methodname* (. . .)

(special variable $THISOBJ refers to the current instance, see below). However, the full syntax to call a method must be used if the receiver is not the current instance.

Routines. Internal methods are not restricted to evaluate an extended expression. They can also execute a sequence of (durative) actions. In this case, they are introduced by the @proc_def keyword because this construct is very similar to a process definition. We call such method *routine* when we want to stress the difference with the previous methods. Actions in the routine body may have duration. So several instances of the same routine may be active at the same moment.

A routine call is similar to a method call. And as for methods, the name of a routine is a simple identifier. Routines can only be defined in the scope of an object definition.

As in ordinary methods, the fields of an obj can be accessed in a routine. Launching another routine must use the full form of method call $THISOBJ . methodname(...). Routines may have arguments that are "local" to the routine instance (they cannot be accessed by others methods nor other routines).

Specifying an object broadcast: @broadcast

Each broadcast clause defines a function with the broadcast name. The syntax is similar to a function definition, except that it is introduced by the keyword @broadcast. Calling this function will execute the body of the function for each active instance of the object.

Specifying a reaction: @whenever and @react

@Whenever clauses can be used to define the triggering of some actions *or* some expressions when some logical conditions occurs. They are similar to whenever (and implemented by a whenever).

This construct makes possible to defines *daemons* that responds automatically to some events. They are two possibilities. To launch actions, the syntax is:

```
@whenever(expression) { actions }
```

and to launch an extended expression, the syntax is

```
@react(expression) { extended expression }
```

The second version is appropriate if the reaction consists only in state update and instantaneous computations. The first form can be used to launch child processes and other durative actions. Note that because some actions are allowed in extended expressions, often it is possible to use one or the other form indifferently.

In either cases, it is possible to use termination guards as in

```
@whenever($x == $x) { $cpt := $cpt + 1 } until ($cpt > 3)
```

Specifying an abort handler: @abort

The @abort clauses are gathered together and are launched when an object instance is killed. Objects instance "live forever" and they must explicitly be killed by an abort action or by a antescofo::stop command.

Referring to the object : $THISOBJ

The special variable $THISOBJ may appears in method definitions where it refers to the object on which the method is applied, or in the clauses of an object definition where it refers to the current instance.

This variable is special: it has a meaning only in the scope of a method or in the clauses of an object. It cannot be watched by a whenever. And assigning this variable leads to unpredictable result.

Checking the type of an object : @is_obj and @is_obj_xxx

An instance of an object is implemented by a process, so it is of the *exe* type and the predicate @is_exec returns true on an object. The predicate @is_obj can be used to distinguish between instances of compound action (and in particular process instances) and object instances.

In addition, each time an object obj::xxx is defined (using @obj_def, a predicate @is_obj_xxx is defined. This predicate returns true if its argument is an object instance of obj::xxx.

Object instantiation

An instance of an object is created using a syntax similar to a process call (and is actually implemented by a process call):

```
$metro1 := obj::Metro(2/3, "left_channel")
```

creates an object of type obj::Metro with parameter $p sets to 2/3 and parameter $receiver sets to "left_channel".

When an object is created, the @init ad the @whenever clauses are performed in the order of definition. Then, the object is alive and ready to interact. Interactions can be done through

- methods calls,
- broadcast,
- direct assignment of the object fields (with the dot notation $o.$var, see p. 131)

and by killing the object.

B.11.3 Object expansion into process, functions and methods

The previous construction is internally expanded in a process definition and in several functions and methods definitions. So there is no new evaluation mechanisms involved. However such construct help to structure the score.

Keep in mind that an object is a process that can be synchronized, like any other process. In particular, object inherit their synchronization from the synchronization strategy defined at their creation.

B.12 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[10].

B.12.1 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:

```
@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
```

[10] Note also that the pattern matching is running asynchronously on variables supposed to be updated at most at the rate of control.

```
  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 whenevers 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.12.2 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[11]. 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.

[11]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.

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.

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 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[12]. *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
```

[12]If there is no before clause, the temporal scope is "the first value change" which implies that there is *at most* one match.

```
  Event $X, $Y at $s1
  Event $X, $Y at $s2 where ($s2 − $s1) < 1
}
```

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.12.3 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 B.2 illustrates the behavior of the pattern

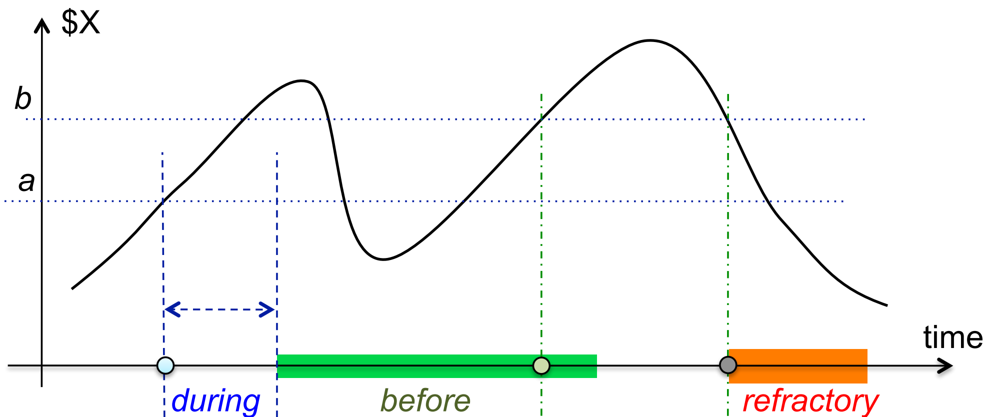
```
@Refractory r

State $X during ℓ where $X > a
Before [d]
  State $X where $X > b
```

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.

Figure B.2 *State patterns with* during, before *and* @refractory *clauses.*



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.12.4 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 B.2. 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.12.5 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[13].

[13] Alternatives behaviors may be considered in the future.

B.13 Scheduling Priorities

Each action performed by *Antescofo* is performed at some date and it may happen that several actions must be performed at the same date “in parallel”. This is a problem. For example consider the fragment:

```
Group G1 { 1 $x := 0 }
Group G2 { 1 $x := 1 }
2 print $x
```

The two groups are launched in parallel and they schedule two contradictory assignments to be performed at the same date, after the expiration of a delay of one beat. The problem is to knwo what will be printed when we print the value of $x? If we assume a “true” parallel execution, the outcome is not defined and in the best case, the result is either 0 or 1 but not deterministically: it varies from one execution to the other.

The *synchronous hypothesis* used in the development of real-time embedded systems assumes that *the actions that occur at the same date are performed in a specific order.* This hypothesis may seem odd at first sight: one has to postulate instantaneous action, *i.e.* actions that take no time to be performed, to make possible a sequence of actions all occurring at the same date. But the success of the synchronous hypothesis in the field of real-time systems demonstrate the usefulness of this hypothesis: at a certain abstraction level, we may assume that an action takes no time to be performed (*i.e.* the execution time is negligible at this abstraction level) and relying on a sequential execution model (the sequence of actions is performed in a specific and well determined order) leads to a deterministic and predictable behavior.

Antescofo fulfills the synchronous hypothesis and the purpose of this section is to explain the execution order used to schedule actions at the same date:

> *Two action instances that occurs are the same date are ordered by their order of appearance in the score and if they are instances of the same action, they are ordered by seniority,* except *for the action inside the body of a* whenever *that are performed following their causal activation order.*

Same Execution Date. A first remark: in this is section, when we speak about actions scheduled at the same date, we have in mind two actions that must be performed at the same physical date, irrespectively of their specification in the score.

Two actions that are specified at the same date in the score, may be well lead to two distinct execution dates. For example, in

```
NOTE C4 1
     Group H1 @loose
     {
        1 $x := 0
        ...
     }
     Group H2 @tight
     {
        1 $x := 1
        ...
     }
NOTE D3 .5
```

the two assignments, which occur at the same date in the ideal time of the score, may not happen at the same date during the performance because the groups H1 and H2 have not the same synchronization strategy:

- the assignment to 1 is performed when note D3 occurs,
- while the assignment to 0 is performed 1 beat after the occurrence of C4 (and the conversion from beat to physical time rely on the tempo estimated on C4.

These two "logical instants" are not necessarily the same: event D3 may occur earlier or later than the specification given in the score. Thus, the value of $x depends of "external" events (the musical events produced on stage) which are not deterministic but do not depend on *Antescofo* itself.

Conversely, two unrelated actions may, by chance, occur at the same date. For example:

```
NOTE E4 0.3
2 Group I1 { 3 $x := 0 }
3 Group I2 { 2 $x := 1 }
```

The two assignment to $x occur at the same date because the sum of the delays occurring from the initial event (the occurrence of the musical event which triggers the actions) are the same. If they are really unrelated, their execution order probably does not matter. But they are other cases when two actions are clearly related in the score, are scheduled for the same date, and indeed are executed at the same date. The group G1 and G2 given in the introduction, or the I1 and I2 groups, are such examples. In this case, order matters and the behavior of *Antescofo* must be easy to understand, deterministic and relevant.

The Syntactic Ordering of Actions. The presentation in this paragraph and the next, does not apply fully to the actions spanned by a whenever. The handling of the ordering of the actions spanned by a whenever are discussed below.

At the exception of whenever, the execution order followed by *Antescofo* is simple: when two actions are scheduled at the same date, the *syntactic order* $\prec$ of appearance in the program is used to determine which one is scheduled first. The syntactic order is roughly the order of appearance in the linear score but takes into account the nesting structure of compound actions.

More precisely, a vector of integers $w(a)$, called the *location* of a, is associated to each action a. This vector locates uniquely the action a in the syntactic structure of the score. Two actions a and a' scheduled at the same date are performed following the syntactic order of $w(a)$ and $w(a')$[14]. In the sequel, we write vectors by listing their element separated by a dot: 1.2.3 is the vector with the three elements 1, 2 and 3. The vector $w(a)$ associated to an action a is build as follows:

- Actions at top-level (appearing before the firs musical event or associate to a musical event) are identified by their rank i of apparition in the score: $w(a) = i$.
- The ith action of a compound action G, like a group, a loop, a whenever, a forall or the action of a curve, is located at $w(\mathsf{G}).i$.

[14]The syntactic order is a generalization of the way the alphabetical order of words is based on the alphabetical order of their component letters. Here instead of letters and their alphabetical order, we use integers and their numerical order. If $w(a) = w_1 w_2 \dots w_n$ and $w(a') = w'_1 w'_2 \dots w'_p$ then $w(a) \prec w(b)$ if and only if the first i where w_i and w'_i differs, we have $w_i < w'_i$.

The lexicographic order is best explained on an example. The localization of each action is given on the left and the actual trace of the program is given at the right:

```
1       $i := 0
2       Loop L1 1
        {
2.1         print loop L1 iteration $i at $RNOW
2.2         $i := $i + 1
        }

3       $j := 0
4       Loop L2 1
        {
4.1         print loop L2 iteration $j at $RNOW
4.2         $j := $j + 1
        }
```

```
loop L1 iteration 0 at 0.0
loop L2 iteration 0 at 0.0
loop L1 iteration 1 at 1.0
loop L2 iteration 1 at 1.0
loop L1 iteration 2 at 2.0
loop L2 iteration 2 at 2.0
loop L1 iteration 3 at 3.0
loop L2 iteration 3 at 3.0
loop L1 iteration 4 at 4.0
loop L2 iteration 4 at 4.0
...
```

Nota Bene:

- The loop has a location which is distinct from the location of its body.
- The syntactic order does not take into account the fact that an action may have several occurrence (this will be handled in the next paragraph).

This program exhibit several actions that occurs at the same date:

- The assignment of `$i` to `0` and the start of the loop `L1` appears at the same date. The assignment is performed first because $1 \prec 2$.
- For the same reason, the assignment of `$i` to `0` is performed before the assignment of `$j` to `0` and before the start of the loop `L2`. The start of loop `L1` is executed before the assignment to `$j` and the start to `L2`, *etc.*
- At time n, two prints occurs together but the `print` message in `L1` is issued before the `print` message in `L2` because $2.1 \prec 4.1$.

A Full Temporal Address with 3-Component. The syntactic order is based solely on the syntactic structure of the score and neglects the difference between an action and the (multiple) realizations of this action (called *instance*): for example, an action a in a loop is performed at each iteration. All these instances are associated to the same location $w(a)$. To compare these actions, that share the same location, we use their instance number.

This way, the temporal address of the execution of an action has 3 components:

date(a)	$w(a)$	instance number(a)

Temporal addresses are *lexicographicaly ordered*:

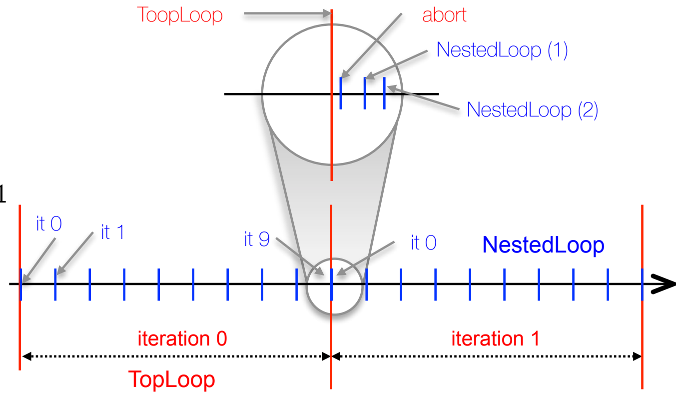
- if two actions have the same date, then their locations are used,
- and if two actions have the same date and the same location, they are compared using their instance number.

In words: two action instances that occurs are the same date are ordered by their order of appearance in the score and if they are instances of the same action, they are ordered by seniority.

Relevance. The resulting order $\ll$ is total: two different actions a and a' are always comparable and $a \ll b$ or $b \ll a$. Thus, this order entails a deterministic execution. The order $\ll$ is not necessarily the order which is needed and there is no way to alter it in *Antescofo*. However, the corresponding scheduling seems relevant on several paradigmatic examples.

For instance, a classical problem is given by two nested loops:

```
$lab := 0
loop TopLoop 1
{
     abort $lab
     $lab := {
          Loop NestedLoop 0.1
          {
               $X := $X + 1
          }
     }
}
```



The loop TopLevel iterates a nested loop NestedLoop which assigns variable $X. The abort command launched at iteration n of the TopLevel loop is supposed to kill the NestedLoop spanned at the previous iteration to avoid two assignments of $X at the same date. The situation is pictured at the right of the program. Remark that the expected behavior can be achieved without an explicit abort using the @exclusive attribute (see p. 52).

Several actions share the same date:

- The 10th assignment of $X in the ith instance of NestedLoop. The temporal address of this assignment is $\boxed{i \,|\, 2.2.1.1 \,|\, 10i}$.
- The first assignment of $X in the $i+1$th instance of NestedLoop. The temporal address of this assignment is $\boxed{i \,|\, 2.2.1.1 \,|\, 10i+1}$.
- the abort command issued by the i iteration of TopLoop. The temporal address of this action is $\boxed{i \,|\, 2.1 \,|\, 10i}$.

The final value of $X depends on the order of executions of these three instances. For example, this result differs if the abort command is issued after the two assignments or before. Because we have

$$\boxed{i \,|\, 2.1 \,|\, 10i} \ll \boxed{i \,|\, 2.2.1.1 \,|\, 10i} \ll \boxed{i \,|\, 2.2.1.1 \,|\, 10i+1}$$

the abort command is issued first and cancel the 10th assignment. So, when TopLoop is reiterated, there is only one assignment that corresponds to the first iteration of the new NestedLoop.

Scheduling of whenevers. Whenevers span the execution of their body when activated by variable's assignments. Thus, if an activation occur at the same date as the firing of another action a, the order of the two depends of relative order between the assignment and a: *the whenever body is activated as soon as the variable is assigned* and if two whenevers are activated by the same variable, they are activated following their syntactic order. This order cannot be solely deduced from the syntactic structure of the score. It is however deterministic.

Here are several examples. In the following fragment:

```
whenever W1 ($x > 0) { print A }
whenever W2 ($x > 2) { print B }          ⟶ A B
let $x := 3
```

the trace produced shows that W1 is activated before W2. Indeed, the two whenever are activated by the same cause: the assignment to $x. In this case, the whenevers are activated following the syntactic order explained above.

In this example

```
whenever W1 ($x) { print A }
whenever W2 ($y)
{
    print B
    let $x := 1                            ⟶ B A C
}
whenever W3 ($x) { print C }
let $y := 1
```

the activation order is W2, W1, W3 because the activation of W1 and W3 is caused by the assignment in the body of W1: so they cannot appear before the activation of W1. The, the activation of W1 and W3 is done in this order, following their syntactic order.

The following example shows that the order of activation is dynamic, *i.e.* it may depends of the values of the variables

```
whenever W1 ($x) { print A }
whenever W2 ($y) { print B }
if ($x > $y)
{
  $x := $x + 1
  $y := $y − 1                             ⟶ A B    if $x > y
}                                          ⟶ B A    if $x <= y
else
{
  $y := $y + 1
  $x := $x − 1
}
```

In addition, do not forget that a whenever is activated at most once in a logical instant (p. 66). So in the trace of the following fragment:

```
whenever W1 ($x || $y || $z) { print A }
whenever W2 ($x || $y)
{
  print B
  $z := true                               ⟶ A B
}
$x := true
$y := true
```

A and B appears only once.

Appendix C

Index

@&&, 13, 130
@*, 13, 130
*, 81
@+, 13, 129
+, 81
@−, 13, 130
−, 81
. (dot operator), 85, 117
@/, 13, 130
/, 81
: (conditional expression), 76
:=, *see* assignment
@<, 13, 130
<, 68
@<=, 13, 130
<=, 68
=, 68
@==, 13, 130
@>, 13
>, 68
@>=, 13, 130
>=, 68
? (conditional expression), 76
@%, 13, 130
%, 81
@&&, 130
Antescofo
    bug report, 5
    discussion group, 127
    scientific publications, 5
    web page, 5
_, 34
Antescofo
    bug report, 1
    forum, 1
    scientific publications, 1
    web page, 1
$-identifiers, 13
::-identifiers, 14
@−identifiers, 12

@abort, 12
abort, 37
    flat, 37
    handler, 150
    processes, 37
    recursive, 37
abort, 12
@abs, 13, 82, 130
@acos, 13, 82, 130
action, 7, 10, 23, 31, 45
    abort, 36
    active (compound action), 36
    as expression, 77
    cancel, 37
    case, 48
    child, 45
    compound, 45
    conditional, 48
    external, 31
    father, 45
    fired, 23
    location, 153
    triggered, 23
@action, 12
action, 12
actions command, 39
add_completion_string command, 39
@add_pair, 13, 89, 130, 135
analysis command, 39
and, 12
@ante, 12
ANTEIOI, 124
antescofo::actions , 39
antescofo :: add_completion_string, 39
antescofo::analysis , 39
antescofo :: asco_trace, 39
antescofo :: ascographheight_set, 39
antescofo :: ascographwidth_set, 39
antescofo :: ascographxy_set, 39
antescofo :: before_nextlabel, 39
antescofo::bpmtolerance, 39
antescofo::calibrate , 39
antescofo::clear , 39
antescofo::decodewindow, 39
antescofo::filewatchset , 39
antescofo::gamma, 39
antescofo :: get_current_score, 39
antescofo :: get_patch_receivers, 40
antescofo::getlabels , 40
antescofo::gotobeat, 40
antescofo::gotolabel , 40
antescofo::harmlist , 40
antescofo::info , 40
antescofo::killall , 40
antescofo::mode, 40
antescofo::mute, 150
antescofo::mute, 40
antescofo::nextaction , 40
antescofo::nextevent , 40
antescofo::nextlabel , 40
antescofo::nextlabel tempo, 40
antescofo::nofharm, 40
antescofo::normin, 40
antescofo::obsexp, 40
antescofo::pedal , 40
antescofo::pedal coeff, 40
antescofo::pedal time, 40
antescofo::piano , 40
antescofo::play , 41
antescofo::play frombeat, 41
antescofo::play fromlabel, 41
antescofo::play string , 41
antescofo::play string _append, 41
antescofo::play tobeat, 41
antescofo::play tolabel , 41
antescofo::preload , 41
antescofo::preventzigzag , 41
antescofo::previousevent , 41
antescofo::previouslabel , 41
antescofo::printfwd , 41

antescofo::printscore , 41
antescofo::read , 41
antescofo::report , 41
antescofo::score , 42
antescofo::scrubtobeat , 42
antescofo::scrubtolabel , 42
antescofo::setvar , 42
antescofo::start , 42
antescofo::start frombeat, 42
antescofo::start fromlabel, 42
antescofo :: static _analysis, 42
antescofo::stop , 42
antescofo::suivi , 42
antescofo::tempo, 42
antescofo::tempo init , 42
antescofo::temposmoothness, 42
antescofo::tune , 42
antescofo::unmute, 150
antescofo::unmute, 42
antescofo::variance , 42
antescofo::verbosity , 42
antescofo::version , 42
@approx, 13, 131
@arch_darwin, 13, 131
@arch_linux, 13, 131
@arch_windows, 13, 131
arithmetic operators, 81
array, 94
asco_trace command, 39
Ascograph
    Curve editing, 56
Ascograph
    monitoring, 122
    score conversion, 121
    score edition, 121
Ascograph
    curve edition, 53
ascographheight_set command, 39
ascographwidth_set command, 39
ascographxy_set command, 39
@asin, 13, 82, 131
assignation, 118
assignment, 34
    external, 35
    local variable (from outside its scope), 35
    of a tab element, 35, 96
at, 161
at, 12
@atan, 13, 82, 131
atomic values, 67
auto-delimited expression, 79
automation, 148

@back, 12
back interpolation, 59
@back_in, 12
back_in interpolation, 59
@back_in_out, 12
back_in_out interpolation, 59
@back_out, 12
back_out interpolation, 59
backslash, 32, 87
\, 32
BEATNUM, 124
$BEAT_POS, 74
before, 162
before, 12
before_nextlabel command, 39
@between, 13, 131
bind, 12
boolean value, 81
@bounce, 12
bounce interpolation, 59
@bounce_in, 12
bounce_in interpolation, 59
@bounce_in_out, 12
bounce_in_out interpolation, 59
@bounce_out, 12
bounce_out interpolation, 59
@bounded_integrate, 13, 94, 131
@bounded_integrate_inv, 13, 131
BPM, 20
BPM, 147
BPM, 11
bpm, 10, 12
bpmtolerance command, 39

calibrate command, 39
@car, 13, 98, 131
@car, 100
carriage-return, 87
case, 12, 48
causal score, 65, 75
causality, 65
@cdr, 13, 98, 131
@cdr, 100
@ceil, 13, 82, 131
CERTAINTY, 124
CFWD, 126
childs of a compound action, 45
CHORD, 7, 11, 17, 18
Chord, 11
chord, 12, 18, 19
@circ, 12
circ interpolation, 59
@circ_in, 12
circ_in interpolation, 59
@circ_in_out, 12
circ_in_out interpolation, 59
@circ_out, 12
circ_out interpolation, 59
@clear, 13, 90, 98, 131
clear command, 39
closefile , 12
@coef, 12
column, 14
@command, 12
@command, 31
comments, 14
comparison, 68
@compose_map, 90
compound action, 45

abort, 47
instance, 46
premature end, 47
compound values, 67
comprehension, 95
predicate, 95
@concat, 13, 98, 100, 132
conditional
action, 48
expression, 76
conjunction, 81
@cons, 13, 98, 132
@cons, 100
@conservative, 12, 102, 105
containers, 45
@copy, 13, 132
@cos, 13, 82, 132
@cosh, 13, 82, 132
@count, 13, 87, 89, 98, 132, 133, 140
@cubic, 12
cubic interpolation, 59
@cubic_in, 12
cubic_in interpolation, 59
@cubic_in_out, 12
cubic_in_out interpolation, 59
@cubic_out, 12
cubic_out interpolation, 59
curve
full syntax, 55
simplified syntax, 53
curve, 53
curve, 12
graphical edition with *Ascograph*, 53
curve
and NIM, 59

Data Structures, 67
@date, 12, 70
dated access, 69, 71
decodewindow command, 39
@defined, 88
delay, 23
absolute, 24
beat, 24
relative, 24, 46
second, 24
zero, 24
dictionary, 88
@dim, 13, 98, 132
@dim, 94, 145
dimension, 94, 145
disjunction, 81
do, 12
domain, 88
@domain, 13, 89, 132
dot notation, 85, 117
@dsp_channel, 12
@dsp_cvar, 12
@dsp_inlet, 12
@dsp_link, 12
@dsp_outlet, 12
@dump, 12, 13, 38, 84, 132
@dumpvar, 13, 38, 84, 132
duration, 18
$DURATION, 74
during, 47, 164
during, 12
dynamic target, 105

@elastic, 12
elastic interpolation, 59
@elastic_in, 12
elastic_in interpolation, 59
@elastic_in_out, 12
elastic_in_out interpolation, 59
@elastic_out, 12
elastic_out interpolation, 59
if, 76
else, 12, 48
@empty, 13, 77, 98, 132
@empty, 77
end of line, 14, 32, 42, 87
ENDBANG, 124
$ENERGY, 74
$JUMP_UPDATED, 157
error (in predefined functions), 68
error handling strategy, 108
@eval_when_load, 12
@eval_when_load, 148
EVENT, 18
event, 7, 10, 17
attributes, 20
fermata, 20
jump, 20
MIDI, 21
musicXML, 21
pivot, 104
pizz, 20
tight, 104
event, 12
@exclusive, 12, 51, 52, 64, 155
exec
abort, 85
active, 85
dead, 85
dot access, 85, 117
exec value, 84
@exp, 12, 13, 82, 133
exp interpolation, 59
@exp_in, 12
exp_in interpolation, 59
@exp_in_out, 12
exp_in_out interpolation, 59
@exp_out, 12
exp_out interpolation, 59
@explode, 13, 87, 133
EXPR, 77
expr, 12
expression
action as ~, 77
auto-delimited, 79
conditional, 76

external actions, 31

false , 12
father, 45
@fermata, 12, 20
fermata, 20
file, 38
filewatchset command, 39
@find, 13, 84, 87, 89, 98, 132, 133, 140
@flatten , 13, 84, 98, 133
float value, 82
@floor, 13, 82, 133
forall , 52
forall , 12
forum, 127
frames of reference, 28
@fun_def, 12
function, 82, 88, 91
    curryfied, 83
    domain, 88
    extensional definition, 88
    intentional definition, 82, 88
    interpolated map, 91
    listable, 129
    map, 88
    predefined, 13, 82
    pure, 129
    side-effect, 129

gamma command, 39
get_current_score command, 39
get_patch_receivers command, 40
getlabels command, 40
GFWD, 126
gfwd, 12
@global, 12, 46
@gnuplot, 13, 84, 98, 133, 134
gnuplot, 72
gnuplot, 72
gnuplot, 72
goto
    in score, 43
gotobeat command, 40
gotolabel command, 40
@grain, 12, 55
group, 8, 11, 12
@gshift_map, 13, 89, 134
@guard, 12

handler, 150
harmlist command, 40
hierarchy
    of actions, 37, 45
    OSC names, 33
history
    as a map, 71, 91
    as a tab, 71
    length, 71
@history_length, 13, 71, 134
@hook, 12, 20
hook, 12, 20

HZ, 123
HZ, 123

identifier
    clash with keywords, 11
if , 76
if , 12, 48
imap, 12
@immediate, 12
@immediate, 63
immutable, 67
implicit with, 76, 78, 83
import
    midi file, 121
    MusicXML file, 121
in, 12
indentation, 14
info command, 40
inlet
    HZ, 123
    KL, 123
    MIDI, 123
@inlet , 12
@insert , 12, 13, 89, 98, 134, 135
@insert , 10
@insert_once, 10
instances of a group, 46
integer value, 81
@integrate, 13, 94, 135
internal command, *see* antescofo::*xxx*
interpolated map value, 91
interpolation
    linear, 58
    see also curve, 58
    step function, 58
    type, 59
    user defined, 59
IO, 38
@iota, 13, 135
@is_prefix , 84, 98
@is_subsequence, 84
@is_suffix , 84
@is_bool, 13, 68, 135
@is_defined, 13, 135
@is_exec, 68
@is_fct, 13, 68, 135
@is_float , 13, 68, 135
@is_function, 13, 68, 135
@is_int, 13, 68, 135
@is_integer_indexed, 13, 88, 135
@is_interpolatedmap, 13, 68, 135
@is_list , 13, 88, 135
@is_map, 13, 68, 135
@is_nim, 68
@is_numeric, 13, 68, 135
@is_prefix , 13, 87, 135, 136
@is_proc, 68
@is_string , 13, 68, 136
@is_subsequence, 13, 87, 98, 135, 136
@is_suffix , 13, 87, 98, 135, 136
@is_symbol, 13, 68, 136

@is_tab, 68
@is_undef, 12, 13, 68, 136
@is_vector, 13, 88, 137
iteration, *see* loop, *see* forall

jump
    in score, 43
@jump, 12, 20
jump
    dynamic, 156
jump, 12, 20

KILL, 126
@kill , 12
kill , 37
kill , 12
KILL OF, 126
killall command, 40
KL, 123
KL, 123

@label, 12
@lace, 13, 98, 137
@last, 13, 137
$LAST_EVENT_LABEL, 74
@latency, 12
let , 12, 34
lexicographic order, 153
LFWD, 126
lfwd, 12
libraries, 148
@lid, 12
linear interpolation, 59
@linear_in, 12
@linear_in_out, 12
@linear_out, 12
list
    as tab, 100
listable, 97, 129
@listify , 13, 90, 137
@loadvalue, 13, 38, 137
@loadvar, 13, 38, 69, 73, 137
@local, 12, 46, 70, 117
@Local, 158
location, 153
@log, 13, 82, 138
@log10, 13, 82, 138
@log2, 13, 82, 138
logical and, 81
logical instant, 27
logical operator, lazy, 81
logical or, 81
Loop, 11
loop, 50
    @exclusive, 51
    halting, 51
    overlapping iteration, 51
    period, 51
loop, 12
@loose, 12, 46, 102

@macro_def, 12
Main outlet, 124
@make_duration_map, 13, 91, 138
@make_label_bpm, 13, 91, 138
@make_label_duration, 13, 91, 139
@make_label_pitches, 13, 91, 139
@make_label_pos, 13, 91, 138
@make_score_map, 13, 91, 139
@map, 13, 99, 139
map
    arithmetic extension, 90
    as an extensional function, 88
    construction, 88, 89
    domain, 88
    history representation, 71, 91
    iteration, 52
    range, 88
    score representation, 91
    value, 88
map, 12
@map_compose, 13, 139
@map_concat, 13, 139
@map_history, 13, 71, 139
@map_history_date, 13, 72, 139
@map_history_rdate, 13, 72, 139
@map_normalize, 13, 139
@map_reverse, 13, 90, 139
@map_val, 89
@mapval, 13, 140
matrix, 94
matrix computations, 95
MAX, 31
    message, 31
    symbol, 32
@max, 13, 68, 82, 140
@max_val, 99
@max_key, 13, 60, 88, 92, 140
@max_val, 13, 89, 140
@member, 13, 87, 89, 99, 132, 133, 140
@merge, 13, 89, 140
MIDI, 123
MIDI, 123
midi file (import), 121
MIDIOUT, 124
@min, 13, 68, 82, 140
@min_val, 99
@min_key, 13, 60, 88, 92, 140
@min_val, 13, 89, 140
MISSED, 124
mode command, 40
model of time, 26
@modulate, 12, 20
ms, 12, 24
MULTI, 17, 19
Multi, 11, 18, 19
multi, 12
multi_list, 17
MusicXML file (import), 121
mutable, 67, 96
mute, 150

mute command, 40
$MYSELF, 75
$MYSELF, 84, 116

@name, 12
napro_trace, 12
negation, 81
nesting groups, 104
nextaction command, 40
nextevent, 18
nextevent command, 40
nextlabel command, 40
nextlabeltempo command, 40
NIM, 91
  continuous, 91
  discontinuous, 92
  extension, 93
  vectorized, 92
nofharm command, 40
@norec, 12
@normalize, 13, 84, 99, 140
normin command, 40
NOTE, 7, 11, 17, 18
Note, 11
note, 158
note, 12, 19
NOTENUM, 124
$NOW, 75

obsexp command, 40
@occurs, 13, 87, 89, 99, 132, 133, 140
of, 12
off, 12
offline, 124
on, 12
open, 38
open score, 156
openoutfile, 38
openoutfile, 12
operator
  listable, 97
OSC, 32
oscoff, 12
oscon, 12
oscrecv, 12
OSCSEND, 33
oscsend, 12
outlet, 123
  ANTEIOI, 124
  BEATNUM, 124
  CERTAINTY, 124
  ENDBANG, 124
  score label, 124
  Main outlet, 124
  MIDIOUT, 124
  MISSED, 124
  NOTENUM, 124
  SCORETEMPO, 124
  TDIST, 124
  tempo, 124
  TRACE, 124
  VELOCITY, 124

parenthesizing expression, 78
parfor, 12
patch, 12
pattern, 158
  @Local, 158
  @refractory, 165
  at, 161
  atomic, 158
  before, 162
  causality, 160, 165
  duration, 159
  during, 164
  event, 160
  on arbitrary variables, 160
  score, 158
  sequence, 158
  single match property, 162, 165
  start, 164
  state, 163
  stop, 164
  temporal scope, 162
  value, 161
  variables, 158
  watched variables, 160, 162
  where, 159, 161
@pattern_def, 12
PD, 31
  message, 31
  symbol, 32
pedal command, 40
pedalcoeff command, 40
pedaltime command, 40
@permute, 13, 99, 141
physical time, 28
piano command, 40
pitch, 17
$PITCH, 74
pitch_list, 17
pivot, 104
@pizz, 12, 20
pizz, 20
play
  in score, 43
play command, 41
playfrombeat command, 41
playfromlabel command, 41
playstring command, 41
playstring_append command, 41
playtobeat command, 41
playtolabel command, 41
@plot, 12, 13, 84, 141
@plot, 72
port, 12
position
  of a variable, 107
@post, 12
@pow, 13, 82, 141
preload command, 41
preset, 132

preventzigzag command, 41
previousevent command, 41
previouslabel command, 41
printfwd command, 41
printscore command, 41
priority, 151
proc value, 84
@proc_def, 12
process, 115
  abort, 37
  abort, 117
  as value, 116
  iteration, 52
  recursive, 116
  tempo, 119
  variable, 117
processus, 84
@progressive, 12, 102, 105, 107
Pure Data, *see* PD
@push_back, 99
@push_front, 99
@push_back, 13, 141
@push_back, 93
@push_front, 13, 141

@quad, 12
quad interpolation, 59
@quad_in, 12
quad_in interpolation, 59
@quad_in_out, 12
quad_in_out interpolation, 59
@quad_out, 12
quad_out interpolation, 59
@quart, 12
quart interpolation, 59
@quart_in, 12
quart_in interpolation, 59
@quart_in_out, 12
quart_in_out interpolation, 59
@quart_out, 12
quart_out interpolation, 59
@quint, 12
quint interpolation, 59
@quint_in, 12
quint_in interpolation, 59
@quint_in_out, 12
quint_in_out interpolation, 59
@quint_out, 12
quint_out interpolation, 59

@rand, 13, 82, 142
@rand_int, 13, 141
@random, 13, 142
random number, 82
@range, 13, 89, 142
@rdate, 12, 70
reactive system, 26
read command, 41
@reduce, 13, 99, 142
@refractory, 12
@refractory, 165
relational operators, 68, 82
relative time, 28
@remove, 13, 89, 99, 142
@remove_duplicate, 13, 99, 142
@replace, 13, 99, 142
report command, 41
@reshape, 13, 99, 143
@resize, 13, 99, 143
return, 76, 83
@reverse, 13, 87, 99, 143
@rnd_bernoulli, 13, 143
@rnd_binomial, 13, 143
@rnd_exponential, 13, 143
@rnd_gamma, 13, 143
@rnd_geometric, 13, 143
@rnd_normal, 13, 144
@rnd_uniform_float, 13, 144
@rnd_uniform_int, 13, 144
$RNOW, 74
@rotate, 13, 100, 144
@round, 13, 144
@rplot, 12, 13, 84, 144
@rplot, 72
$RT_TEMPO, 75

s, 12, 24
@savevalue, 13, 38, 144
scalar, 67
scalar product, 95
@scan, 13, 100, 144
scheduling, 151
score
  as a map, 91
  graph, 156
  moving in, 43
  open, 156
  pattern, 158
score command, 42
score label, 124
SCORETEMPO, 124
$SCORE_TEMPO, 75
@scramble, 13, 100, 144
scrub
  in score, 43
scrubtobeat command, 42
scrubtolabel command, 42
@select_map, 13, 89, 144
set_var, 160
setup, 148
setvar, 35
setvar command, 42
shape, 145
@shape, 13, 145
@shape, 94
@shift_map, 13, 89, 145
simultaneity, 151
@sin, 13, 82, 145
@sine, 12
sine interpolation, 59
@sine_in, 12
sine_in interpolation, 59

@sine_in_out, 12
sine_in_out interpolation, 59
@sine_out, 12
sine_out interpolation, 59
@sinh, 13, 82, 145
size
    non atomic value, 77
    scalar value, 77
    undefined value, 77
@size, 13, 77, 93, 100, 145
@sort, 13, 84, 100, 145
special variables
    $NOW, 75
@sputter, 13, 100, 145
@sqrt, 13, 82, 145
[], 94
@staccato, 12
standalone, 124
start, 164
start, 12
start command, 42
startfrombeat command, 42
startfromlabel command, 42
state, 163
state, 12
statement, 10
static target, 104
static_analysis command, 42
@staticscope, 12
stop, 164
stop, 12
stop command, 42
strategy
    error handling, 108
    synchronization, 101
string
    accessing a char, 87
    immutable, 87
    tab access, 87
string value, 87
@string2fun, 13, 145
@string2proc, 13, 145
@stutter, 13, 100, 146
suivi command, 42
switch, 12, 48
symb, 12
@sync, 12, 102, 108
synchronization
    on a variable, 75
    on a variable, 108
synchronization strategy, 24, 101
synchronous hypothesis, 151
@system, 13, 146
system variables
    assignment, 35
    $BEAT_POS, 74
    $DURATION, 74
    $ENERGY, 74
    $JUMP_UPDATED, 157
    $LAST_EVENT_LABEL, 74
    $MYSELF, 75
    $PITCH, 74
    $RNOW, 74
    $RT_TEMPO, 75
    $SCORE_TEMPO, 75

tab, 94
    assignment, 96
    comprehension, 95
    extension, 95
tab
    and list, 100
    history representation, 71
    iteration, 52
tab, 12
tab value, 94
@tab_history, 13, 71, 146
@tab_history_date, 13, 72, 146
@tab_history_rdate, 13, 72, 146
tabulation, 87
@tan, 13, 82, 146
@target, 12, 102, 104, 105
@target
    dynamic, 105
    horizon, 105
    static, 104
TDIST, 124
tempo, 28
    the, 28
    inherited, 119
    local, 28
    of a variable, 107
    tracking, 28
@tempo, 12, 46
tempo, 124
@tempo, 119
tempo command, 42
tempoinit command, 42
temporal address, 154
temporal clause, 47
temporal coordinate systems, 26
temporal horizon, 105
temporal scope, 47
temporal shortcuts, 65, 75
temporized system, 26
temposmoothness command, 42
@tempovar, 12, 102, 107, 174
@tempovar, 75
the tempo, 28
@tight, 12, 46, 102, 104, 105
time
    absolute, 28
    logical instant, 27
    model of, 26
    physical, 28
    relative, 28, 46
    shortcuts, 65, 75
    wall clock, 28
time frame, 26
time frames, 28
TRACE, 124

trace of function calls, 148
@Tracing, 13, 123, 146, 148
Tracing, 148
track, 149
  mute, 150
  unmute, 150
@track_def, 12
@transpose, 12
transpose, 12
TRILL, 7, 8, 17, 19
Trill , 11, 18, 19
trill , 12
trill_list, 17
true, 12
tune command, 42
@type, 12
type of a value, 67

@uid, 12
undefined value, 81
underscore, 34
unmute, 150
unmute command, 42
until , 47
until , 12
@UnTracing, 13, 123, 146, 148
UnTracing, 148

value
  atomic, 67
  boolean, 81
  exec, 84
  float, 82
  immutable, 67
  integer, 81
  interpolated map, 91
  map, 88
  mutable, 67
  non atomic, 67
  ordering, 140
  proc, 84
  scalar, 67
  string, 87
  tab, 94
  types of, 68
  undefined, 34, 81
value, 161
value, 12
variable
  :: access, 72
  whenever restriction, 64
  access, 85, 117
  access outside its definition scope, 72, 117
  accessing through a label, 72
  assignment, 69
  assignment through OSC messages, 33
  dated access, 69
  declaration, 70
  dot access, 72, 85, 117
  external assignment, 35
  global, 71
  history, 69, 71
  lifetime, 70
  local, 70
  local to a pattern, 159
  position, 107
  scope, 70
  special, 35, 75, 84, 116
  stream of values, 69
  system, 35, 74
  tempo, 107
  tempovar, 75
  watched, 160
variance, 10, 12
variance command, 42
VELOCITY, 124
verbosity command, 42
version command, 42

wall clock time, 28
watched variables, 63
whenever
  ordering, 155
whenever, 63
  @exclusive, 64
  and assigning a tab's element, 96
  during, 64
  overlapping instances, 64
  pattern, 158
  temporal scope, 64
  temporal shortcut, 75
  temporal shortcuts, 65
  until, 64
  variable notification, 75
whenever, 12
where, 159, 161
where, 12
while, 47
while, 12
with, 78
  implicit, 76, 78, 83
writing in a file, 38

Appendix D

Detailed Table of Contents

1 Understanding *Antescofo* scores **7**

1.1 Structure of an *Antescofo* Score 7

1.2 Elements of an *Antescofo* Score 10

1.3 *Antescofo* keywords 11

1.4 @-identifiers: Functions, Macros, and Attributes 12

1.5 $-identifiers: Variables 13

1.6 ::-identifiers: Processes 14

1.7 Comments and Indentation 14

2 Events **17**

2.1 Event Specification 17

2.2 Events as Containers 18

2.3 Event Attributes 20

Event Label. 20

The @modulate Attribute. 20

2.4 Importing Scores to Antescofo 21

2.4.1 Importing MIDI scores to Antescofo 21

2.4.2 Importing MusicXML scores to Antescofo 21

3 Actions in Brief **23**

Action Attributes. 23

3.1 Delays 23

Zero Delay. 24

Absolute and Relative Delay. 24

Evaluation of a Delay. 24

Synchronization Strategies. 24

3.2 Label 25

3.3 Action Execution 25

4 Atomic Actions **31**

4.1 Message passing to Max/PD 31

4.2 OSC Messages 32

4.2.1 OSCSEND . . . 33
4.2.2 OSCRECV . . . 33
4.2.3 OSCON and OSCOFF . . . 34
4.3 Assignments . . . 34
Assignment to Vector Elements and to Local variables. . . . 35
Activities Triggered by Assignments. . . . 35
External Assignments. . . . 35
4.4 Aborting and Cancelling an Action . . . 36
4.4.1 Abort of an Action . . . 36
Abort and the hierarchical structure of compound actions. . . . 37
4.4.2 Cancelling an Action . . . 37
4.5 I/O in a File . . . 38
4.6 Internal Commands . . . 39
4.7 Assertion @assert . . . 43

5 Compound Actions **45**
5.1 Group . . . 45
5.1.1 Local Tempo. . . . 46
5.1.2 Attributes of Group and Compound Actions . . . 46
5.1.3 Instances of a Group . . . 47
5.1.4 Aborting a group . . . 47
The until Clause. . . . 47
The during Clause. . . . 47
5.2 If, Switch: Conditional and Alternative . . . 48
5.2.1 If: Conditional Actions . . . 48
5.2.2 Switch: Alternative Actions . . . 49
Alternative Action without Selector. . . . 49
Alternative Action with a Selector. . . . 50
5.3 Loop: Sequential iterations . . . 50
Loop Period. . . . 51
Stopping a Loop. . . . 52
Avoiding Overlapping Iterations of a Loop body. . . . 52
5.4 Forall: Parallel Iterations . . . 53
5.5 Curve: Continuous Actions . . . 54
5.5.1 Simplified Curve Syntax . . . 54
5.5.2 Full Curve Syntax . . . 55
5.5.3 Actions Fired by a Curve . . . 58
5.5.4 Step, Durations and Parameter Specifications . . . 58
5.5.5 Interpolation Methods . . . 58
Programming an Interpolation Method. . . . 59
5.5.6 Curve with a NIM . . . 59
5.6 Whenever: Reacting to logical events . . . 64
Watching Restrictions. . . . 65
Avoiding Overlapping Instances of a Whenever body. . . . 65
5.6.1 Stopping a whenever . . . 65

5.6.2 Causal Score and Temporal Shortcuts . . . 66
Automatic Temporal Shortcut Detection. . . . 67

6 Expressions **69**
6.1 Values . . . 69
Dynamic Typing. . . . 70
Checking the Type of a Value. . . . 70
Value Comparison. . . . 70
6.2 Variables . . . 71
6.2.1 Histories: Accessing the Past Values of a Variable . . . 71
Dates functions. . . . 72
6.2.2 Variables Declaration . . . 72
Local Variables. . . . 72
Lifetime of a Variable. . . . 72
History Length of a Variable. . . . 73
6.2.3 History reflected in a Map or in a Tab. . . . 73
Plotting the history of a variable. . . . 74
6.2.4 Accessing a Local Variable “from Outside its Scope of Definition” . . . 74
The :: Notation. . . . 75
The Dot Notation. . . . 76
6.3 Internal Antescofo Variables . . . 76
6.3.1 Special Variables . . . 77
6.3.2 Variables and Notifications . . . 77
Temporal Shortcuts. . . . 78
6.4 Temporal Variables . . . 78
6.5 Operators and Predefined Functions . . . 78
Conditional Expression. . . . 78
@empty and @size. . . . 79
6.6 Action as Expressions . . . 79
Example. . . . 80
6.7 Structuring Expressions . . . 80
6.8 Auto-Delimited Expressions in Actions . . . 80

7 Scalar Values **83**
7.1 Undefined Value . . . 83
7.2 Boolean Value . . . 83
7.3 Integer Value . . . 83
7.4 Float Value . . . 84
7.5 User-defined Functions . . . 84
7.6 Proc Value . . . 84
7.7 Exec Value . . . 85
Accessing a Local Variable Through an *exec*. . . . 85

8 Data Structures **87**
8.1 String Value . . . 87

8.2 Map Value . . . 88
Extensional Functions. . . . 88
Domain, Range and Predicates. . . . 88
Constructing Maps. . . . 89
Extension of Arithmetic Operators. . . . 90
Maps Transformations. . . . 90
Score reflected in a Map. . . . 91
History reflected in a map. . . . 91
8.3 InterpolatedMap Value . . . 91
NIM interpolated Map. . . . 91
Vectorized NIM. . . . 92
Extending a NIM. . . . 93
8.3.1 Nim transformation and smoothing . . . 94
Inhomogeneous breakpoints in vectorial nim. . . . 94
Sampling, homogeneization and linearization. . . . 94
Nim simplification. . . . 95
Smoothing and Transformation. . . . 96
8.4 Tables . . . 97
Multidimensional tab. . . . 97
Tab Comprehension. . . . 98
Changing an element in a Tab. . . . 99
Tab operators. . . . 100
Tab manipulation. . . . 101
Lists and Tabs. . . . 103

9 Synchronization and Error Handling Strategies 105
9.1 Synchronization Strategies . . . 105
The *Antescofo* view of musician's performance. . . . 105
Coordination of actions with musician's performance. . . . 106
9.1.1 Loose Synchronization . . . 106
9.1.2 Tight Synchronization . . . 108
9.1.3 Target Synchronization . . . 108
Static Targets . . . 108
Dynamic Target . . . 109
9.1.4 Adjusting the Coordination Reference . . . 109
How to Compute the Position in case of Conflicting Informations . . . 109
Specifying Alternative Coordination Reference . . . 111
9.2 Missed Event Errors Strategies . . . 112
Combining Synchronization and Error Handling. . . . 112

10 Macros 115
10.1 Macro Definition and Usage . . . 115
10.2 Expansion Sequence . . . 116
10.3 Generating New Names . . . 117

11 Functions **119**
11.1 Function definition 120
Extended expressions. 120
First examples. 121
11.1.1 The return Statement 122
A common pitfall. 122
11.1.2 Function's Local Variables and Assignations 123
11.1.3 Extended Conditional Expressions and Iteration Expressions 124
The if Extended Expression. 124
The switch Extended Expression. 124
The Loop Extended Expression. 125
The Forall Extended Expression. 126
11.1.4 Atomic Actions in Expressions 126
11.2 Functions as Values 126
11.3 Curryfied Functions 126

12 Process **129**
12.1 Calling a Process 129
12.2 Recursive Process 130
12.3 Process as Values 130
12.4 Aborting a Process 131
12.5 Processes and Variables 131
12.6 Process, Tempo and Synchronization 133
12.7 Macro *vs.* Function *vs.* Processus 133

13 *Antescofo* Workflow **135**
13.1 Editing the Score 135
13.2 Tuning the Listening Machine 135
13.3 Debuging an *Antescofo* Score 136
13.4 Dealing with Errors 136
13.4.1 Monitoring with Notability 136
13.4.2 Monitoring with *Ascograph* 136
13.4.3 Tracing an *Antescofo* Score 136
Printing the Parsed File. 136
Verbosity. 136
The TRACE Outlet. 137
Tracing the Updates of a Variable. 137
Tracing the Evaluation of Functions. 137
13.5 Interacting with MAX 137
13.5.1 Inlets 137
13.5.2 Outlets 137
13.5.3 Predefined Messages 138
13.6 Interacting with PureData 138
13.7 *Antescofo* Standalone Offline 138
13.8 Old Syntax 140

13.9 Stay Tuned . . . 140

A Library of Predefined Functions **143**

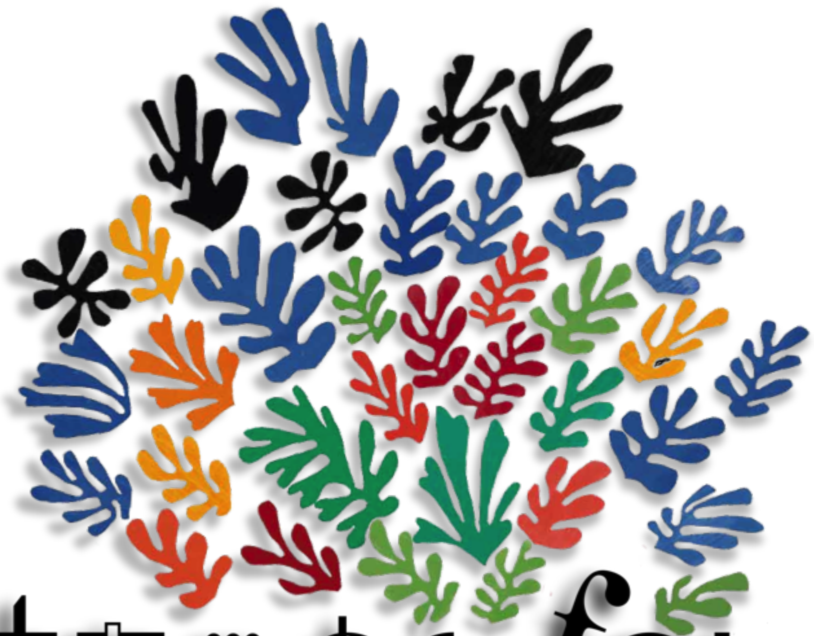
B Experimental Features **163**

B.1 Reserved Experimental Keywords . . . 163
B.2 Constant BPM expresssion . . . 163
B.3 @eval_when_load Clause . . . 164
B.4 Tracks . . . 164
B.5 Abort Handler . . . 165
Example. . . . 166
B.6 Continuations . . . 167
Continuation and abort. . . . 169
B.7 Open Scores and Dynamic Jumps . . . 169
B.8 Tracing Function Calls . . . 171
B.9 Infix notation for function calls . . . 171
B.10 Methods . . . 172
B.11 Objects . . . 174
B.11.1 A basic example . . . 175
B.11.2 Object definition . . . 177
Field definition: @local . . . 177
Performing action at the object construction: @init . . . 177
Specifying an object method: @method_def and @proc_def . . . 177
Routines. . . . 178
Specifying an object broadcast: @broadcast . . . 179
Specifying a reaction: @whenever and @react . . . 179
Specifying an abort handler: @abort . . . 179
Referring to the object : $THISOBJ . . . 179
Checking the type of an object : @is_obj and @is_obj_xxx . . . 179
Object instantiation . . . 180
B.11.3 Object expansion into process, functions and methods . . . 180
B.12 Patterns . . . 181
B.12.1 Note: Patterns on Score . . . 181
Pattern Variables. . . . 181
Specifying Duration. . . . 182
Specifying Additional Constraints. . . . 182
Pattern Causality. . . . 183
A Complete Exemple. . . . 183
B.12.2 Event on Arbitrary Variables . . . 183
The value Clause. . . . 184
The at Clause. . . . 184
The where Clause. . . . 184
The before Clause. . . . 185
Watching Multiple Variables Simultaneously. . . . 185
A Complex Example. . . . 186

B.12.3 State Patterns . . . 186
A Motivating Example. . . . 186
The initiation of a state Pattern. . . . 187
The during Clause. . . . 187
B.12.4 Limiting the Number of Matches of a Pattern . . . 188
B.12.5 Pattern Compilation . . . 188
B.13 Scheduling Priorities . . . 189
Same Execution Date. . . . 189
The Syntactic Ordering of Actions. . . . 190
A Full Temporal Address with 3-Component. . . . 191
Relevance. . . . 192
Scheduling of whenevers. . . . 192

C Index **195**

D Detailed Table of Contents **205**


Antescofo