



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 8, 2015

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	<i>Understanding Antescofo scores</i>	7	7.4	Float Value	80
1.1	Structure of an <i>Antescofo</i> Score	7	7.5	String Value	80
1.2	Elements of an <i>Antescofo</i> Score	10	7.6	User-defined Functions	81
1.3	<i>Antescofo</i> keywords	11	7.7	Proc Value	82
1.4	@-identifiers: Functions, Macros, and Attributes	12	7.8	Exec Value	82
1.5	\$-identifiers: Variables	13			
1.6	::-identifiers: Processes	13			
1.7	Comments and Indentation	14			
2	Events	17	8	Data Structures	85
2.1	Event Specification	17	8.1	Map Value	85
2.2	Events as Containers	18	8.2	InterpolatedMap Value	88
2.3	Event Attributes	20	8.3	Tables	91
2.4	Importing Scores to <i>Antescofo</i>	20			
3	Actions in Brief	23	9	Synchronization and Error Handling Strategies	99
3.1	Delays	23	9.1	Synchronization Strategies	99
3.2	Label	25	9.2	Missed Event Errors Strategies	101
4	Atomic Actions	31	10	Macros	105
4.1	Message passing to Max/PD	31	10.1	Macro Definition and Usage	105
4.2	OSC Messages	33	10.2	Expansion Sequence	107
4.3	Assignments	34	10.3	Generating New Names	107
4.4	Aborting and Cancelling an Action	36			
4.5	I/O in a File	38	11	Process	111
4.6	Internal Commands	39	11.1	Calling a Process	111
4.7	Assertion Assert	41	11.2	Recursive Process	112
5	Compound Actions	43	11.3	Process as Values	112
5.1	Group	43	11.4	Aborting a Process	113
5.2	If, Switch : Conditional and Alternative Actions	46	11.5	Processes and Variables	113
5.3	Loop : Sequential iterations	48	11.6	Process, Tempo and Synchronization	115
5.4	Forall : Parallel Iterations	50	11.7	Macro <i>vs.</i> Processus	115
5.5	Curve : Continuous Actions	51			
5.6	Wheneve r: Reacting to logical events	61	12	Antescofo Workflow	117
6	Expressions	65	12.1	Editing the Score	117
6.1	Values	65	12.2	Tuning the Listening Machine	117
6.2	Variables	67	12.3	Debuging an <i>Antescofo</i> Score	118
6.3	Internal <i>Antescofo</i> Variables	72	12.4	Dealing with Errors	118
6.4	Temporal Variables	73	12.5	Interacting with MAX	119
6.5	Operators and Predefined Functions	74	12.6	Interacting with PureData	120
6.6	Action as Expressions	75	12.7	<i>Antescofo</i> Standalone Offline	120
6.7	Structuring Expressions	75	12.8	Old Syntax	122
6.8	Auto-Delimited Expressions in Actions	76	12.9	Stay Tuned	122
7	Scalar Values	79			
7.1	Undefined Value	79	A	Library of Predefined Functions	125
7.2	Boolean Value	79	B	Experimental Features	143
7.3	Integer Value	79	B.1	Reserved Experimental Keywords	143
			B.2	Tracks	143
			B.3	Abort Handler	144
			B.4	Patterns	146
			C	Index	155
			D	Detailed Table of Contents	165

Sidebars

Brief history of <i>Antescofo</i>	5
---------------------------------------------	---

Figures

1.1 Antescofo Score Excerpt showing basic events and actions	8
1.2 The beginning of <i>Tensio</i> (2010) by Philippe Manoury for String Quartet and Live electronics in <i>AscoGraph</i>	9
1.3 Example of score attribute affectation (top-down parsing) in <i>Antescofo</i> text scores.	11
1.4 Reserved keywords: Event keywords are in red whereas action keywords are in blue	12
1.5 Predefined functions	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 <i>Antescofo</i> 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 <i>AscoGraph</i>	52
5.2 Simplified Curve chain call	52
5.3 Full 2d Curve	54
5.4 Full 2d Curve embedded on Event Score in <i>AscoGraph</i>	55
5.5 Various interpolation type available in an <i>Antescofo</i> Curve and NIM	59
5.6 (cont.) Various interpolation type available in an <i>Antescofo</i> Curve and NIM	60
6.1 Example of simple expression and its value realisation in <i>AscoGraph</i>	66
8.1 The two forms a NIM definition	90
9.1 The effect of tempo-only synchronization for accompaniment phrases: illustration for different tempi.	100
9.2 Action behavior in case of a missed event for four synchronization and error handling strategies	102
10.1 Example of a Macro and its realisation upon score load	106
12.1 Help of the standalone offline command line.	121

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:

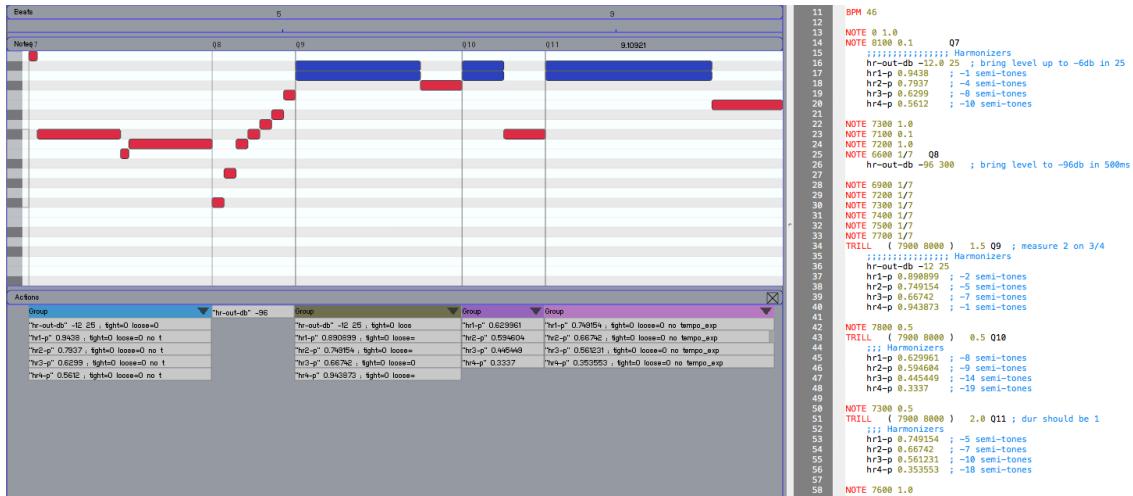
EVENTS are elements to be recognized by the score follower or machine listener, describing the dynamics of the outside environment. They consist of **NOTE**, **CHORD**, **TRILL** and other elements discussed in details in section 2.

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*

Figure 1.1 Antescofo Score Excerpt showing basic events and actions



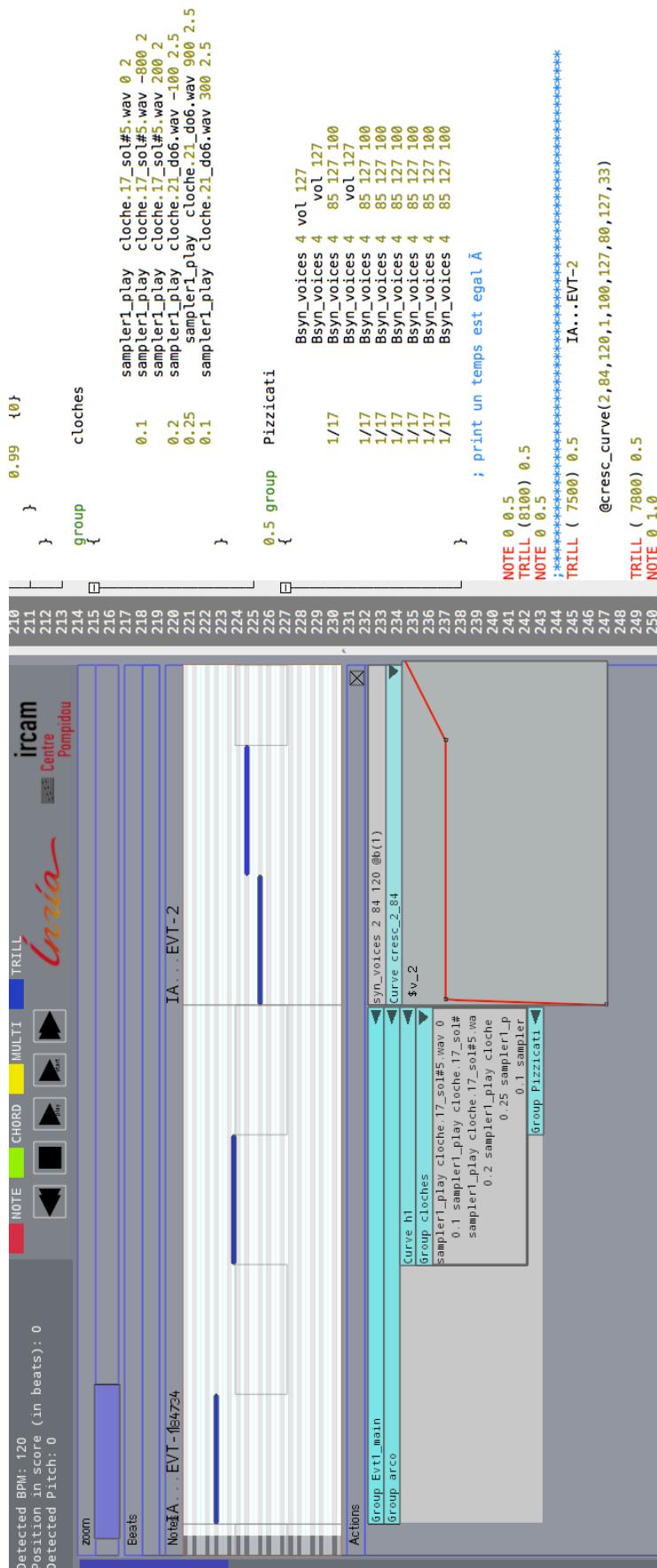
can use delays expressed in various time formats, and further include dynamic (i.e. real-time evaluated) expressions, data-structures and more. Details on action structures are discussed in section 3.

Figure 1.2 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*



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 whitespace must be quoted"
```

The `@insert` keyword can be capitalized: `@INSERT`, as any other keyword beginning with a `@` sign. An included file may include (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.

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

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` `@macro_def` `@pattern_def` `@proc_def` `@track_def`;
3. to introduce various attributes of an event or an action:

```
@abort @action @ante @coef @dsp_channel @dsp_inlet
@dsp_outlet @global @grain @guard @hook @immediate @inlet
@jump @kill @label @label @local @loose @modulate @name
@norec @post @refractory @staticscope @target @tempo @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*).

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 7.6.

Only ! ? . and _ are allowed as special characters after the @.

Figure 1.5 Predefined functions

@+	@-	@*	@/	@%	@insert	@iota	@is_bool	@permute	@pow
@<	@<=	@>	@>=		@is_defined	@is_fct	@push_back	@push_front	
@==	@!=				@is_float	@is_function			
@	@&&				@is_int		@rand	@rand_int	
@abs	@acos	@add_pair			@is_integer_indexed		@random	@range	@reduce
@approx	@asin	@at			@is_interpolatedmap		@remove		
@atan					@is_list	@is_map	@remove_duplicate		
@between					@is_nim	@is_numeric	@replace	@reshape	
@bounded_integrate					@is_prefix	@is_string	@resize	@reverse	
@bounded_integrate_inv					@is_subsequence		@rnd_bernoulli		
@car	@cdr	@ceil			@is_suffix	@is_symbol	@rnd_binomial		
@clear	@concat	@cons			@is_undef	@is_vector	@rnd_exponential		
@copy	@cos	@cosh			@lace	@listify	@rnd_gamma		
@count					@log		@rnd_geometric		
@dim	@domain				@log10	@log2	@rnd_normal		
@empty	@exp	@explode			@make_duration_map		@rnd_poisson		
@find	@flatten	@floor			@make_score_map	@map	@rnd_uniform_int		
@gnuplot	@gshift_map				@map_compose		@rnd_uniform_float		
					@map_concat		@rotate	@round	
					@map_reverse	@mapval			
					@max	@max_key	@scan	@scramble	
					@max_val		@select_map	@shape	
					@member	@merge	@shift_map	@sin	
					@min_key	@min_val	@size	@sort	
					@normalize		@sqrt	@stutter	
					@occurs			@tan	

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.*

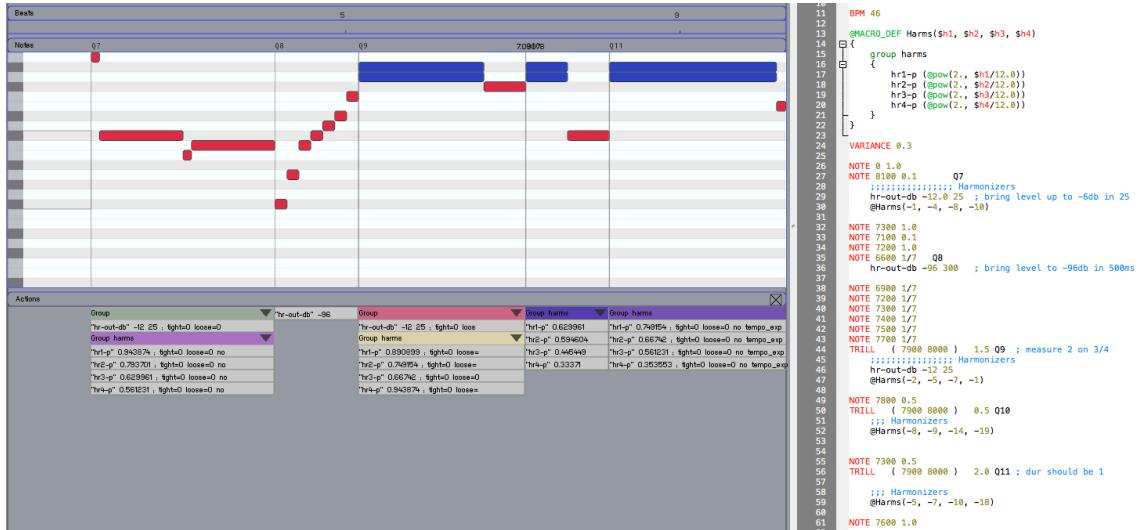
Figure 1.6 shows a rewrite of the score in Fig. 1.1 using a simple Macro and employing basic @ and \$ identifiers. The harmoniser command is here defined as a Macro (section 10) for convenience and since it is being repeated through the same pattern. The content of the “hr1-p” to “hr4-p” actions inside 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.

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 11).

Figure 1.6 Rewrite of Figure 1.6 using a Macro and expressions



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\u2014two" \
      "messages" \
      "are\u2014equivalent"
print "this\u2014two" "messages" "are\u2014equivalent"
```


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 polyphonic¹ 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 brain² 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]
```

Parameters for each syntax 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), MIDI cent number (e.g. 6900), or Standard Pitch Name (e.g. A4).

`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.



A musical score for a single staff. The staff begins with a treble clef, a key signature of C major (no sharps or flats), and a common time signature. The tempo is set to 60 BPM. The score consists of the following events:

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

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



A musical score in G clef with a tempo of $\text{♩} = 60$. It shows a note followed by a trill symbol (a wavy line above the note), then a note and a chord. The score is followed by its definition:

```

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

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



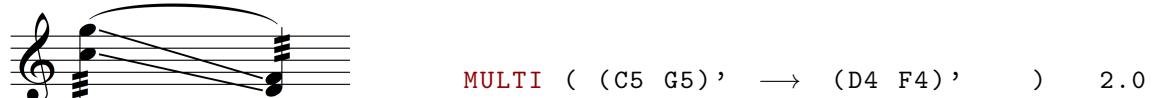
A musical score in G clef showing a sequence of chords: C major, F major, and G major. The score is followed by its definition:

```

MULTI ( (F4 C5) → (D4 A4) ) 4.0
  
```

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 inside the **MULTI** are **TRILLS** instead of regular notes or chords.

Figure 2.4 Glissandi with Tremolos



A musical score in G clef showing a glissando between chords C5 and G5, and D4 and F4. The score is followed by its definition:

```

MULTI ( (C5 G5)', → (D4 F4)', ) 2.0
  
```

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 **Trills** 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).
- A simple identifier or a string or an integer acts as a label for this event. They can be several such labels. If the label is a simple identifier, its \$-form can be used in a expression elsewhere in the score to denote the time in beat of the onset of the event.
- The keyword `jump` (or `@jump`) is followed by a comma separated list of simple identifiers referring to the label of an event in the score. This attribute specifies that this event can be followed by several continuations: the next event in the score, as well as the events listed by the `@jump`.

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

```
Note D4 1 here @fermata @jump 11, 12
```

defines an event labeled by `here` which is potentially followed by the next event (in the file) or the events labeled by 11 and 12 in the score. It has a *fermata*. Note that

```
Note D4 1 @jump 11, 12 here
```

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

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 d_1 after the recognition of the C note, and *action2* will be triggered d_2 after the launching of *action1*.

A delay can be any expression. This expression is evaluated when the preceding event is launched. That is, expression d_2 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 3.2.

Absolute and Relative Delay. A delay can be either absolute or relative. An absolute delay is expressed in seconds (respectively in milliseconds) and 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 score¹.

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

¹There 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.

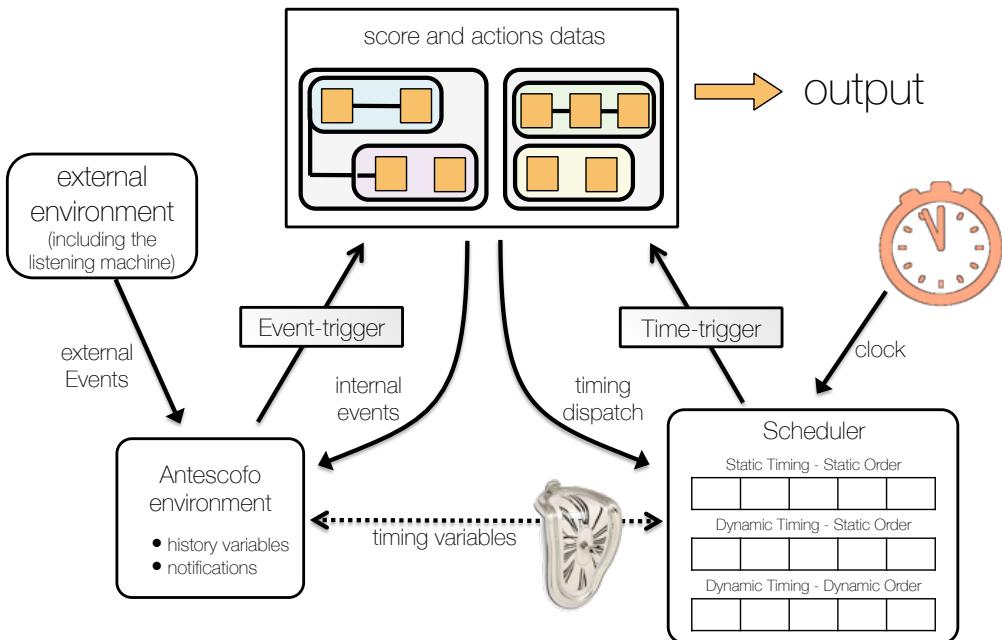
Experienced users should note that *Antescofo* is delivered with its own *Real-time Scheduler*. This is mainly to reduce utility costs of using internal Max and Pd *Timers* and to significantly reduce their interference with other actions in Max/Pd schedulers themselves. The *Antescofo* internal scheduler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



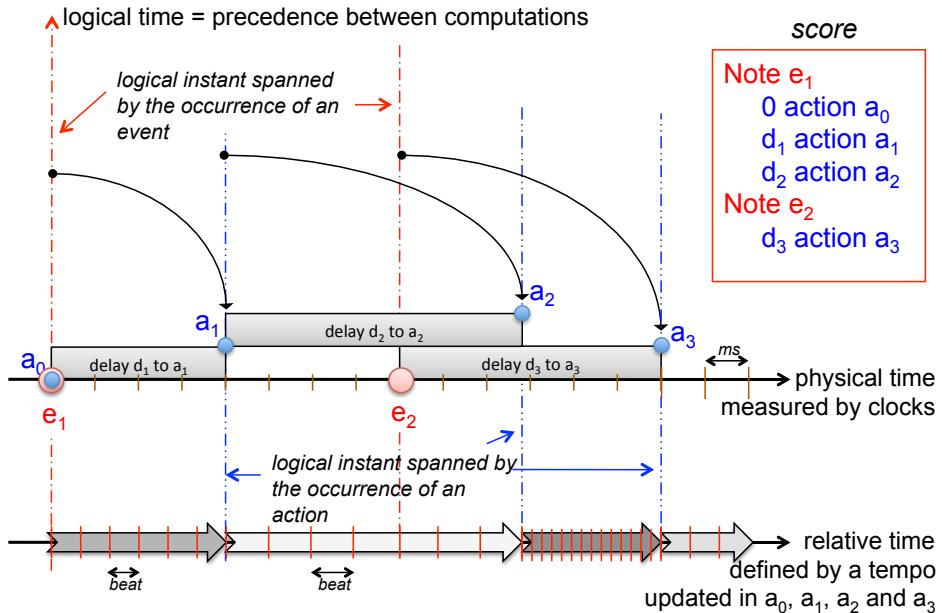
Logical Instant

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.

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.



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.

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 time². 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:

$$t_{\mathcal{T}} = \int_0^{t_{\mathcal{P}}} T_{\mathcal{T}} \quad (3.1)$$

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 referred as “*the tempo*” and has a tremendous importance because it is the time frame naturally associated with the musician part of the score³. This variable is extracted from the audio stream by the listening machine, relying on cognitive model of musician behavior⁴. 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 e1`). Then action a_2 can be launched either:

- $(d_1 + d_2)$ beats after the occurrence of the event `NOTE e1`,
- or $(d_1 + d_2 - 1.5)$ beats after the occurrence of the event `NOTE e2`

(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 e2` occurs *exactly* after 1.5 beat after event `NOTE e1`. But this is not the case in an actual performance.

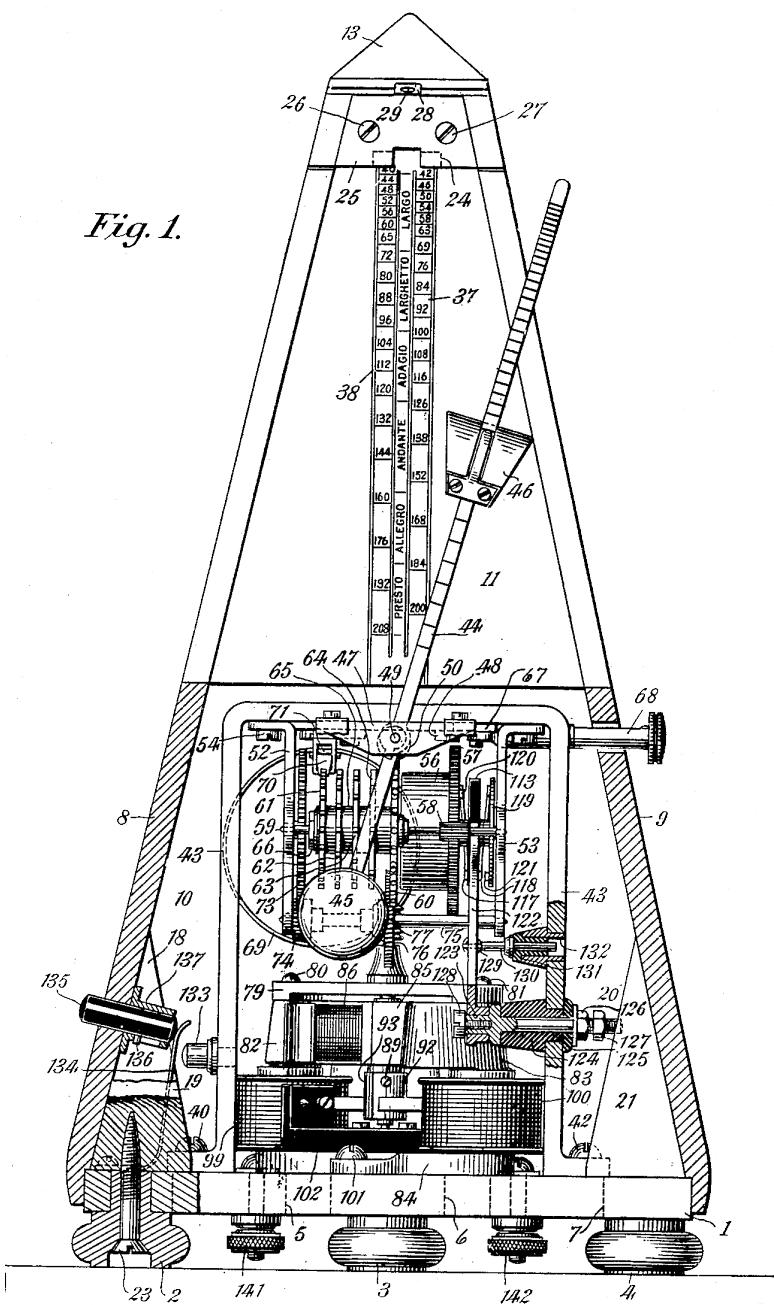
Antescofo allows a composer to choose the right way to compute the date of an action in a time frame, to best match the musical context. This is the purpose of the *synchronization strategy*. They are described in section 9.

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

³The `$RT_TEMPO` is computed by *Antescofo* to mimics the tracking of the tempo by a human, and implements an idea of smooth tempo fluctuation, rather than trying to satisfy exactly equation (3.1) at any moment. So, for the relative time frame, equation (3.1) is only an approximation. As a consequence, the current position in the score is explicitly given by the variable `$BEAT_POS` which is more accurate than the integration of `$RT_TEMPO`. See paragraph 6.3.1.

⁴A. 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 messages¹.

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

¹The 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 12.5. On the contrary, arbitrary messages can be sent from an *Antescofo* score to external MAX objects with appropriate receivers.

same identifier². 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:

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. 7.5.

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

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

prints the following to MAX/PD console

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

²As you go on, you will notice that everything in *Antescofo* can be dynamic. The receiver identifier can also be calculated using *string concatenation* (see section 7.5). In this case, use `@command()`. For example, if the value of \$num is 1, `@command("spat" + $num)` builds the “spat1” receiver.

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 protocol³ 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.

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 ... argn
```

The idea is that this construct and send the osc message

```
msg_prefix arg1 ... argn
```

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.

³<http://opensoundcontrol.org/>

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 $v1 ... $vn
```

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 \$v₁ ... \$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`.

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. Expressions e in the right hand side of `:=` are described in section 6. A variable as a value before its first assignment: the undefined value (sect. 7.1).

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 $e'[i_1, i_2, \dots] := e$` where e' is an expression that evaluates to a tab and i_1, i_2, \dots evaluate to integers (sect. 8.3 p. 93);
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. 6.2.3).

The `let` keyword it is mandatory when e' is more complex than a variable. Tab's element assignment are described in sect.8.3 p. 93). Assignment of local variable of an instance of a group accessed through its `exec` are described sect. 7.8 p. 82.

Activities Triggered by Assignments. The assignment of a value to a variable⁴ 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 variable⁵.

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.

⁴ and not to a tab element nor to a local variable, see paragraph 5.6 page 62

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.

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.

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 8.3) 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:

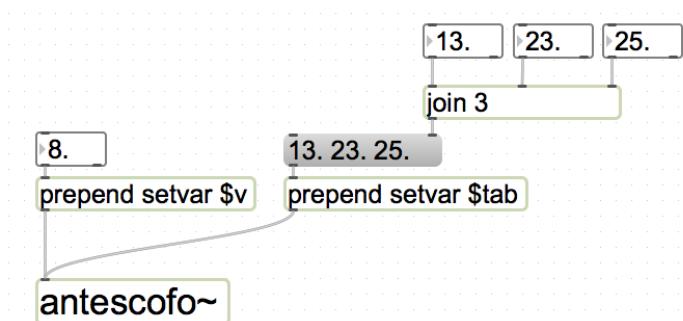
```
whenever ($tab)
{
    print "I just received the vector" $tab
}
```

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. There 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_suppress
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 (compare with the example in section ??).

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_World\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 negligible⁶.

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 varlue of some variables to be restored later (or in another program execution). See functions `@savevalue`, `@loadvalue`, `@dump`, `@dumpvar` and `@loadvar`.

4.6 Internal Commands

... This section is still to be written

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

Internal commands are named `antescofo::xxx` where the suffix `xxx` is the head of the corresponding MAX/PD message (cf. section 12.5):

- `antescofo::actions` *string* :
- `antescofo::analysis` *int* :
- `antescofo::tempo` *float* :
- `antescofo::before_nextlabel` (no argument) :
- `antescofo::bpmtolerance` *float* :
- `antescofo::calibrate` *int int int* :
- `antescofo::clear` (no argument) :
- `antescofo::gamma` *float* :
- `antescofo::getcues` (no argument) :
- `antescofo::getlabels` (no argument) :
- `antescofo::gotobeat` *float* :
- `antescofo::gotocue` *string* :
- `antescofo::gotolabel` *string* :
- `antescofo::harmlist` *float* ... (a list of floats corresponding to a vector) :

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).

- `antescofo::info` (no argument) :
- `antescofo::jumptocue` *string* :
- `antescofo::jumptolabel` *string* :
- `antescofo::killall` (no argument) :
- `antescofo::mode` *int* :
- `antescofo::mute` *string* :
- `antescofo::nextaction` (no argument) :
- `antescofo::nextevent` (no argument) :
- `antescofo::nextfwd` (no argument) :
- `antescofo::nextlabel` (no argument) :
- `antescofo::nofharm` *int* :
- `antescofo::normin` *float* :
- `antescofo::obsexp` *float* :
- `antescofo::pedalcoeff` *float* :
- `antescofo::pedaltime` *float* :
- `antescofo::pedal` *int* :
- `antescofo::piano` *int* :
- `antescofo::playfrombeat` *float* :
- `antescofo::playfrom` *string* :
- `antescofo::play` (no argument) :
- `antescofo::preload` *string* *string* : preloads a score and store it under a name (the second argument) for latter use;
- `antescofo::preventzigzag` *string* :
- `antescofo::previousevent` (no argument) :
- `antescofo::previouslabel` (no argument) :
- `antescofo::printfwd` (no argument) :
- `antescofo::printscore` (no argument) :
- `antescofo::read` *string* : loads the corresponding *Antescofo* score;
- `antescofo::report` (no argument) :

- `antescofo::setvar` *string numeric* : assign the value given by the second argument to the variable named by the first argument. Using this command, the environment may notify *Antescofo* some information. For instance, *Antescofo* may react because the variable is in the logical condition of a `whenever`). See section 4.3
- `antescofo::score` *string* : loads the corresponding *Antescofo* score;
- `antescofo::start` *string* :
- `antescofo::stop` (no argument) :
- `antescofo::suivi` *int* :
- `antescofo::tempoinit` *int* :
- `antescofo::temposmoothness` *float* :
- `antescofo::tune` *float* :
- `antescofo::unmute` *string* :
- `antescofo::variance` *float* :
- `antescofo::verbosity` *int* :
- `antescofo::verify` *int* :
- `antescofo::version` (no argument) : print the version on the MAX console;

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

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

```

action 1
Group {
  1 action 2
}
action 3

```

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 3.2.

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 killed¹.

¹It is possible to kill a specific instance using the `exec` that refers to this instance, see 7.8 and 11.4.

5.1.4 Aborting a group

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

- using an `abort` action, see 5.1.4,
- using a `until` (or a `while`) *logical clause*,
- using a `during` *temporal clause*.

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

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

There is a dual of the `until` keyword:

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

is equivalent to

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

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

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

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

The `[]` notation follows the notation used for the access to the history of a variable (cf. sect. 6.2.1 pp. 67). 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`.

5.2 `If, Switch`: Conditional and Alternative Actions

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 74 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.

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 starts 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).

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 progresses.

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 ...

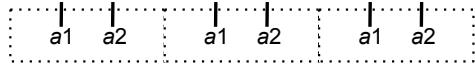
Note that the periods are taken cyclically in the vector. The `ms` specification after the period indicates that the period is given in millisecond.

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.

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 Curves² 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. 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 (2.0s in this case).

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:

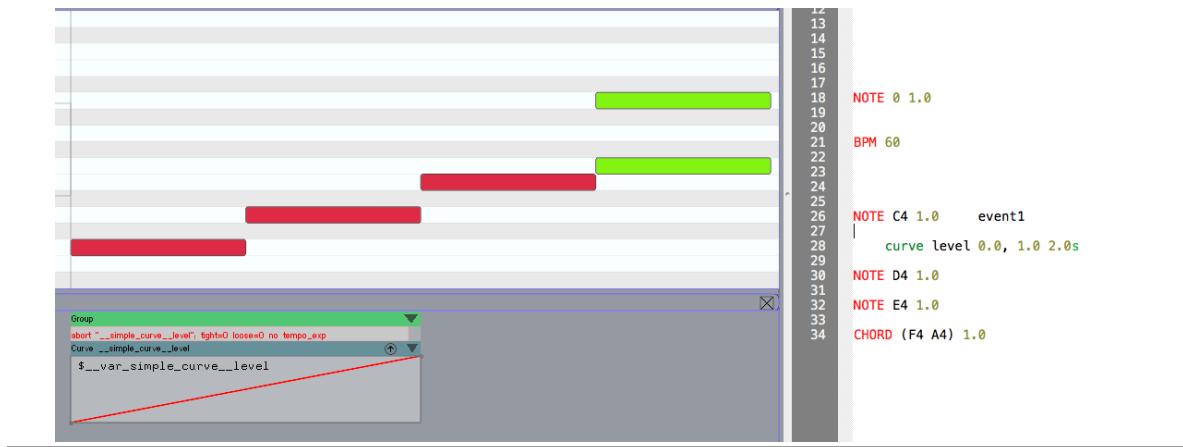
```
Curve level 0.5 1.0
```

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

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

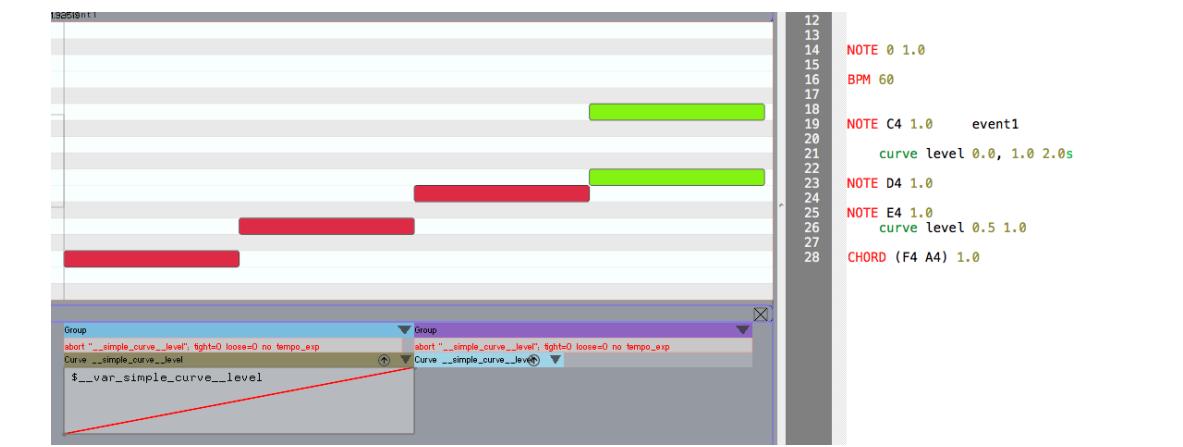
Figure 5.1 Simplified Curve syntax and its realisation in *Ascograph*



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,

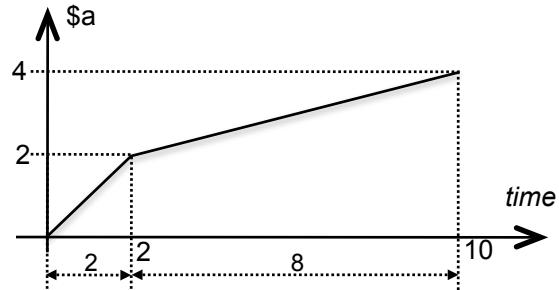
the time-step for interpolation in the above curve is 30 milli-seconds and hard-coded. A complete `curve` allows adjusting such parameters, having multi-dimensional interpolations, complex actions, and more. We detail the complete curve syntax hereafter.

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).

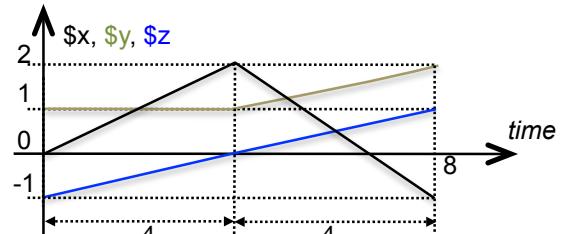
```
curve C
  @action := { level $y } ,
  @grain := 0.1
{
  $y
  {
    { 0 } @type "linear"
    2 { 2 } @type "linear"
    8 { 4 }
  }
}
```



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

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 }
  }
}
```

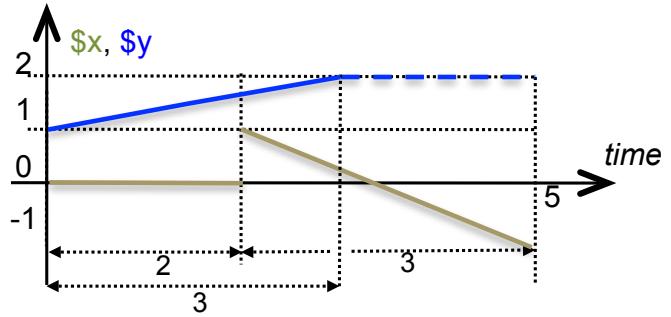


In the above example, all values in the three-dimensional vector share the same break-point and the same interpolation type. It is also possible to split the curve to multiple parameter clauses as below:

```

curve C
{
  $x
  {
    0 { 0 }
    2 { 0 }
    0 { 1 }
    3 { -1 }
  }
  $y
  {
    3 { 1 }
    3 { 2 }
  }
}

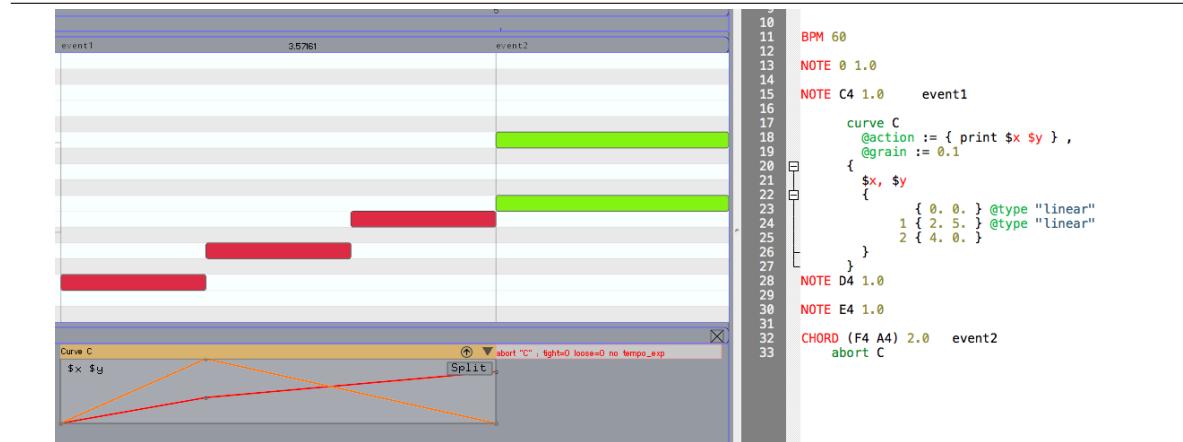
```



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.

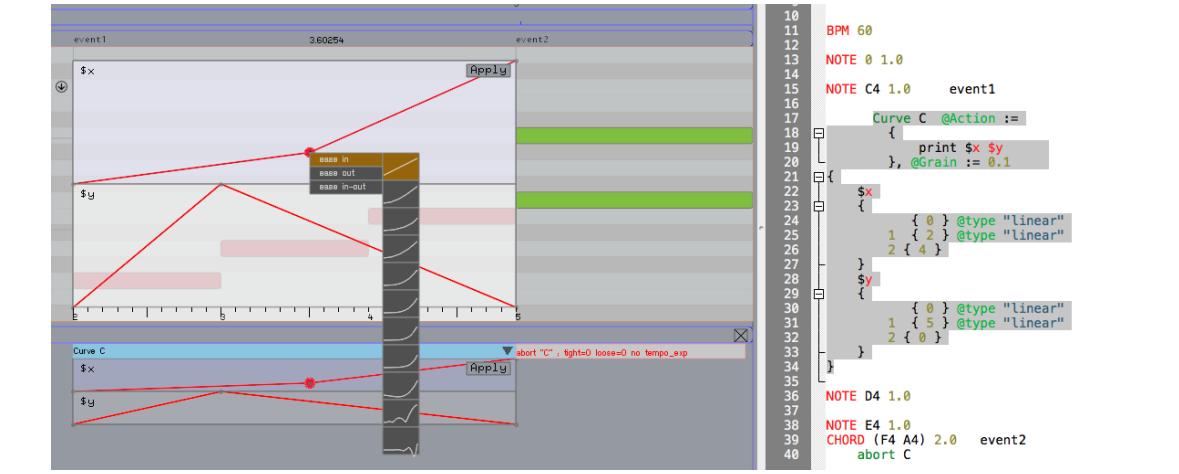
Figure 5.3 Full 2d Curve



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 if the values are 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 `tbreakpoints`’ 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 animation³. There are 10 different types:

- *linear, quad, cubic, quart, quint*: which correspond to polynomial of degree respectively one to five;
- *expo*: exponential, *i.e.* $\alpha e^{\beta t + \delta} + \gamma$
- *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 8.2.

³See <http://wiki.xbmc.org/?title=Tweeners>

```

"linear"
"back" or "back_in"      "exp" or "exp_in"      "elastic" or "elastic_in"
"back_out"                "exp_out"          "elastic_out"
"back_in_out"             "exp_in_out"        "elastic_in_out"
"bounce" or "bounce_in"   "quad" or "quad_in"   "sine" or "sine_in"
"bounce_out"              "quad_out"          "sine_out"
"bounce_in_out"           "quad_in_out"        "sine_in_out"
"cubic" or "cubic_in"    "quart" or "quart_in"
"cubic_out"               "quart_out"          "quart_in_out"
"cubic_in_out"            "quint" or "quint_in"
"circ" or "circ_in"       "quint_out"          "quint_in_out"
"circ_out"                "quint_in_out"
"circ_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 8.2) 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)] }
```

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`”;

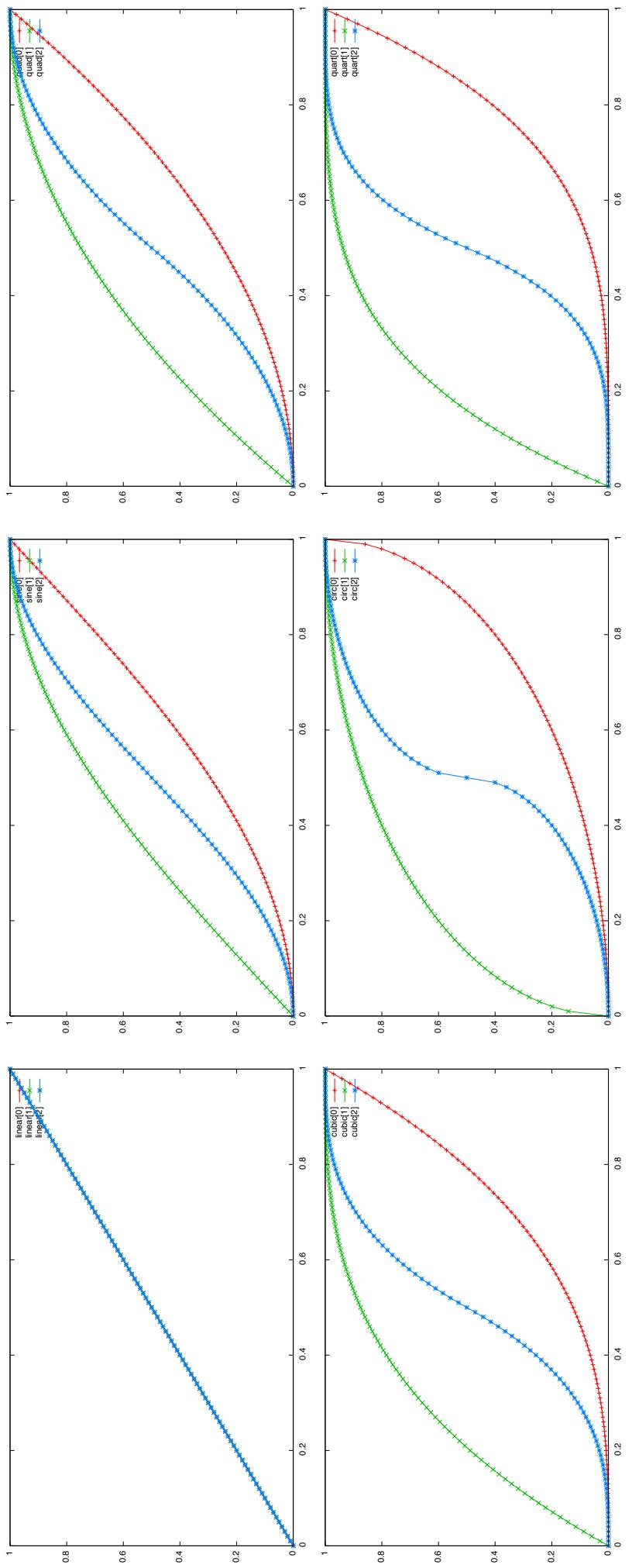
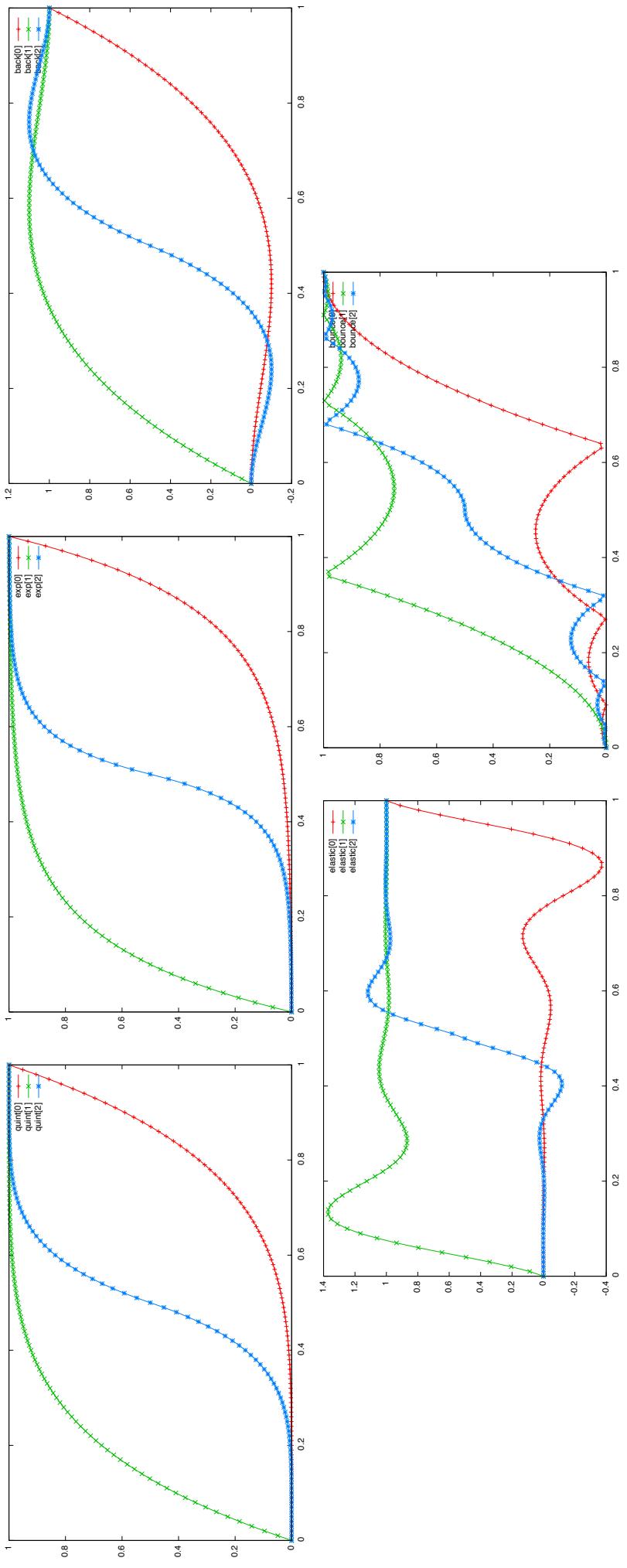


Figure 5.6 (cont.) Various interpolation type available in an Antescofo Curve and NIM. The label `xxx[0]` corresponds to the ease “in”, that is to the type “`xxx_in`” or equivalently “`xxx`”; the `xxx[1]` corresponds to the ease “out” and the label `xxx[2]` corresponds to the ease “`in_out`”.



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 optional `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: 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.

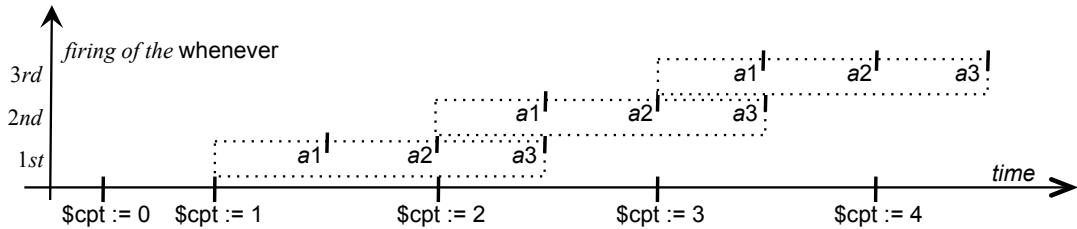
The `whenever` is a way to reduce and simplify the specification of the score particularly when actions have to be executed each time some condition is satisfied. It also escapes the sequential nature of traditional scores. Resulting actions of a whenever statement are not statically associated to an 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` watches variable, not values. So the update of an element in a tab cannot trigger a `whenever`, see 8.3 page 93. A `whenever` cannot watch a local variable referred through the dot notation. A `whenever` cannot watch a special variable, that is `$NOW` and `$MYSELF`. These restrictions ensure that *Antescofo* score remain causal and efficiently implementable.

5.6.1 Stopping a `whenever`

A `during` and/or an `until` clause can be defined for a `whenever` (see sect. 5.1.4). These clauses are evaluated each time the logical condition of the `whenever` must be evaluated, irrespectively of its `false` or `true` value. For example,

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

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

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

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

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

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

Another option is to give the `whenever` a label and to abort it, see sect. 5.1.4.

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*:

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

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

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

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

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

```

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

```

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

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

Chapter 6

Expressions

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

6.1 Values

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

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

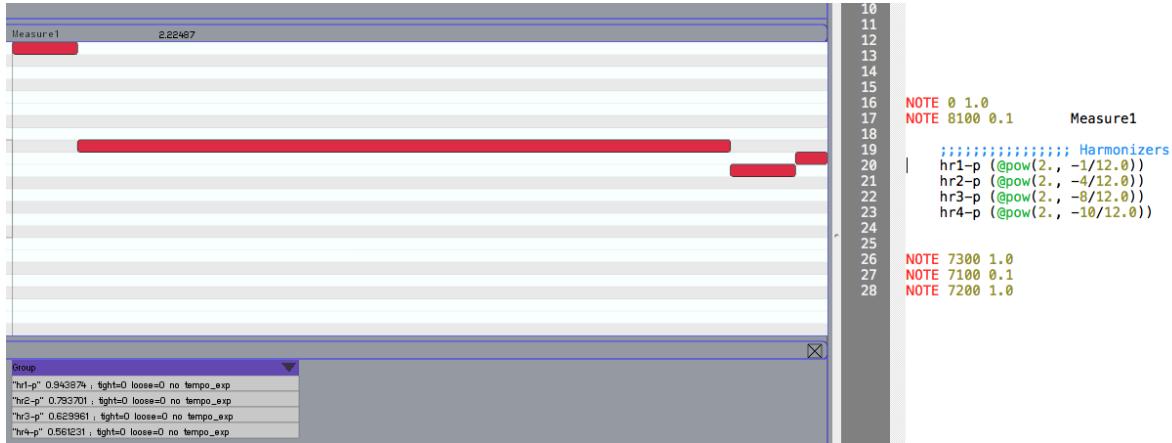
Figure 6.1 shows a simple score excerpt employing a simple expression and value. The text score on the right declares four expressions to be sent to receivers “hr1-p” to “hr4-p” (harmonisers) whose final value is being converted from semi-tones to pitch-scale factor. The *Ascograph* graphical representation 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 7.6.

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 12.3 for useful hints on how to debug an *Antescofo* score.

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

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

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

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

< <= = != => >

or the `@min` and `@max` operators. The comparison of two values of the same type is as expected: arithmetic comparison for integers and floats, lexicographic comparison for strings,

etc. When an integer is compared against a float, the integer is first converted into the corresponding float. Otherwise, comparing two values of two different types is well defined but implementation dependent.

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, and as other action, it can be done after a delay. See sect. 4.3.

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

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

6.2.1 Accessing Variable Histories

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

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

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

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

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

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

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:

- `@history_map($x)` returns a map where key n refers to the $n - 1$ to the last value of $\$x$. In other word, the element associated to 1 in the map is the current value, the previous value is associated to element 2, etc. The size of this list is the size of the variable history, see the paragraph *History Length of a Variable* below. However, if the number of update of the variable is less than the history length, the corresponding undefined values are not recorded in the map.
- `@history_tab($x)` is similar to the previous function but returns a tab where i th element refers to the $n - 1$ to the last value of $\$x$.
- `@history_map_date($x)` returns a list where element n is the date (physical time) of $n - 1$ to the last update of $\$x$. The previous remark on the map size applies here too.
- `@history_tab_date($x)` builds a tab (instead of a map) of the dates in physical time of the updates of the var $\$x$.
- `@history_map_rdate($x)` returns a list where element n is the relative date of $n - 1$ to the last update of $\$x$. The previous remark on the map size applies here too.
- `@history_tab_rdate($x)` builds a tab (instead of a map) of the dates in relative time of the 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 69).

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) )
```

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>.

See description of `@gnuplot` in the annex page 129.

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
  $x := 2
  print "local\uvar\u$x:\u" $x
}
print "global\uvar\u$x:\u" $x
```

will print

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

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

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

```
n : $v
```

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

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

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

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

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

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.

6.2.3 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.

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.

A variable can be assigned from “outside *Antescofo*”, cf sect. 4.3.
 sec:OSCreceive

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 $x in group G
print (G::$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 7.8 and the dot notation is detailed in section 11.5.

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. 11.5. 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 current position in the score. This position is updated between two events, as the time progress: after the occurrence of an event, **\$BEAT_POS** increases until reaching the position in the score of the next waited event. The **\$BEAT_POS** is stuck until the occurrence of this event or the detection of a subsequent event (making this one missed).
- **\$DURATION** is the duration of the last detected event.
- **\$LAST_EVENT_LABEL** is the label of the last event seen. This variable is updated on the occurrence 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 meaningful² for the other kinds of 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.
- **\$RNOW** is the date in relative time (in beats) of the “current instant”.
- **\$RT_TEMPO** represents the tempo currently inferred by the listening machine from the input audio stream.

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.

²in the current version of *Antescofo*

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 `tempo` 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.

6.3.3 Dates functions

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

6.4 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.

6.5 Operators and Predefined Functions

The main operators and predefined functions have been sketched in the section 7 and 8. We sketch here some operators or functions that are not linked to a specific type. The predefined functions are listed in annex A p. 125.

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

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

or equivalently

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

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

They are also conditional actions, cf. sect. 5.2. Conditional actions have a syntax similar to the `if...else...` form, but beware not to confuse them: a conditional expression appears only where an expression is expected. Another difference is that the true and the false branch of a conditional expression are both mandatory (because an expression must always have a value, irrespectively of the value of the condition).

`@empty` and `@size`. The `@empty` predicate returns true if its argument is an empty `tab` or an empty `map`, 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 `atab` the number of elements and for a `nim`, the dimension of the `nim`;
- 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. 66).

Notice that the length of a string is not returned by `@size`, nor check by `@empty`, because strings are scalar values.

6.6 Action as Expressions

An action can be considered as an expression: when evaluated, its value is an `exec`, cf. sect 7.8. To consider an action as an expression, the action must 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 action³. An atomic action (with a 0-duration run) returns the special `exec '0'`.

The surrounding `EXPR{ }` is optional in the case of a process call (sect. 11.1). The `EXPR` keyword is optional in the right hand side of an assignment. For example:

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

is equivalent to

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

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 11.5):

```
$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 but spread the understanding of the expression in several places. The `with...in` construct may help:

³See sect. 11.5 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`.

```
with $x1 = e1
and $x2 = e2
and ...
in e
```

evaluates the expression e where the identifier $\$x_i$ denote the value of the expression e_i . These expression are evaluated in turn, that is, e_i may refers to the $\$x_j$ for $j < i$. The identifiers $\$x_i$ are not actual variable: they are just references to the value of the associated expressions (hence the use of the symbol “ $=$ ” and not “ $:=$ ”). But these references hide the local or the global variables with the same identifiers in the scope of the `with`. For example

```
$x := 666
$a := with $x = 1
      and $y = $x + 1
      and $z = $x + $y
      in $x + $y + $z
```

assigns 6 to $\$a$ because $1 + (1 + 1) + (1 + (1 + 1))$. Note that the right hand sides of $=$ are evaluated only once.

This construct is similar to the *let** in Lisp, nested *let* in ML or the *block* construct in Mathematica.

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    // parenthesized 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

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 form¹: `! 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 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`.

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.

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 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. 8.3). 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.

7.6 User-defined Functions

Intentional functions f are defined by rules (*i.e.* by an expression) that specify how an image $f(x)$ is associated to an element x . Intentional functions can be defined and associated to an @-identifier using the `@fun_def` construct introduced in section 7.6 page 81. Some intentional functions are predefined and available in the initial *Antescofo* environment like the IEEE mathematical functions. See annex A for a description of available internal 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`.

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 (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 }
```

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 8.1 for map building and map transformations).

Curryfied Functions. In *Antescofo*, intentional functions are implicitly *Curryfied*. Such notion was introduced 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`.

7.7 Proc Value

The `::-`name of a processus can be used in an expression to denote the corresponding process definition, in a manner similar of the `@`-identifier used for intensionnal functions (see 7.6). Such value are qualified as *proc value*. Like intensionnal functions, proc value are first class value. They can be passed as argument to a function or a procedure call.

The main operation on proc value is “calling the corresponding process”, see section 11.

7.8 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 11, 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.
- A process call returns the *exec* of the instance launched (see sect. 11.1).
- 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. 11.5).

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. 11.5 for an example.

The reference of a local variable using the dot notation can be used in an assignment, see sect. 11.5 for an example involving a process instance (but this feature works for any *exec*).

Chapter 8

Data Structures

Antescofo currently provides experimental `map` and `tab` data structures described in this section.

8.1 Map Value

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

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

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 7.6.

In an expression, extensional function or intentional function can be used indifferently where a function is expected. In other words, you can apply an extensional function to get a value, in 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 $\{1, \dots, n\}$ for some n . The predicate `@is_vector` returns true if the predicate `@is_list` is satisfied and if every element in the range satisfies `@is_numeric`.

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

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

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

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:

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

- `@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{ (k1, p1), (k2, p2), ..., (kn, pn), ... }
$q := map{ (k'1, q1), (k'2, q2), ..., (k'm, qm), ... }
```

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

```
map{ ..., (pk, qk), ... }
```

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 $3 < 4 < \text{"abc"}$.

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

```
map{ (1, v1), (2, v2), ..., (p, vp), }
```

the following list is build:

```
map{ (1, vp), (2, vp-1), ..., (p, v1), }
```

Score reflected in a Map. Two functions can be used to reflect the events of a score into a map¹:

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

The corresponding maps are *vectors*.

History reflected in a map. The sequence of the values of a variable is keep in an history. This history can be converted into a map, see section 6.2.1 pp. 68.

8.2 InterpolatedMap Value

Interpolated map are values representing a piecewise interpolated function. They have been initially 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.

¹Besides `@make_score_map` and `@make_duration_map`, recall that the label of an event in `$-form`, can be used in expressions as the position of this event in the score in relative time.

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:

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

which specifies a piecewise function f : between x_i and $x_{i+1} = x_i + d_{i+1}$, function f is an interpolation of type $type_{i+1}$ from y_i to y_{i+1} . See an illustration at the left of Fig. 8.1.

The function f is extended outside $[x_0, x_n]$ such that

$$f(x) = \begin{cases} y_0 & \text{for } x \leq x_0 \\ y_n & \text{for } x \geq x_n = x_0 + \sum_{i=0}^n d_i \end{cases}$$

The type of the interpolation is either a constant string or an expression. But in this case, the expression must be enclosed in 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 59.

Note that the previous definition specifies a continuous function because the value of f at the beginning of $[x_i, x_{i+1}]$ 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 $[x_i, x_{i+1}]$ the function is an interpolation between y_i and Y_i . See illustration at the right of Fig. 8.1.

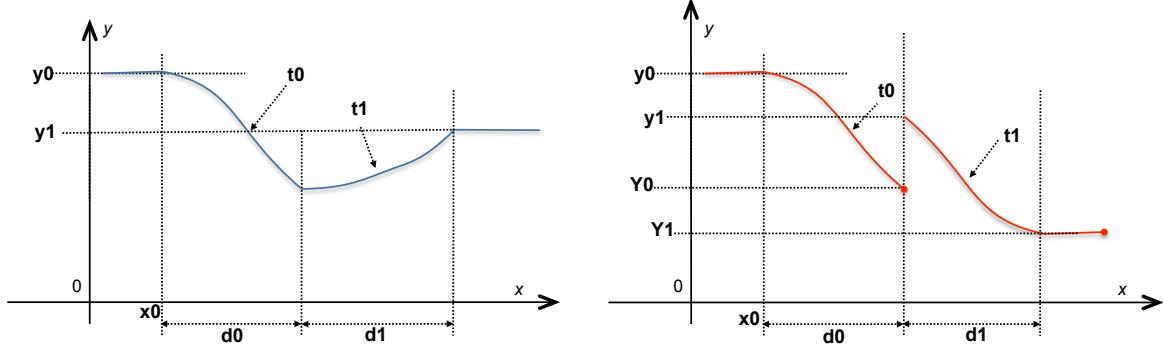
Vectorized NIM. the **NIM** construct admits tab as arguments: in this case, the result is a vectorial function. For example:

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

defines a vectorial **NIM** of two variables:

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

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}`.



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, the interpolation type is linear. For example:

`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 y_0 value of the added breakpoint is the y_1 value of the last breakpoint of the [NIM](#). The next two forms specify explicitly the y_0 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.

Interpolated Map. Interpolated Map are deprecated: this type of value is now superseded by [NIM](#).

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

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

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

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

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

8.3 Tables

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

```
$t := tab [0, 1, 2, 3]      ; or
$t := [0, 1, 2, 3]
$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 $\cdot[\cdot]$ notation: $\$t[n]$ refers to the $(n + 1)$ th element of tab $\$t$ (elements indexing starts at 0).

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

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

is a 3×2 matrix which 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 i th 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, its 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:

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

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

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

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

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

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

Changing an element in a Tab. A tab is a *mutable* data structure : one can change 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 referred 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(0.1) < 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 accept `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 exist 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.

`@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 129 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 ≤ @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 `n`th permutations of the elements of `t`. They are factorial `s` permutations, where `s` is the size of `t`. The first permutation is numbered 0 and corresponds to the permutation which rearranges the elements of `t` in an array `t0` such that they are sorted increasingly. The tab `t0` 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 `t1` such that `t0 < t1` for the lexicographic order and such that any other permutation gives an array `tk` lexicographically greater than `t0` and `t1`. Etc. The last permutation (factorial `s - 1`) returns a tab where all elements of `t` are in decreasing order.

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

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

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

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

`@remove(t, n)` removes the element at index `n` in `t` (`t` is modified in place). 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 [138](#).

`@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])
```

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

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

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

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

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

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

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

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

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

9.1 Synchronization Strategies

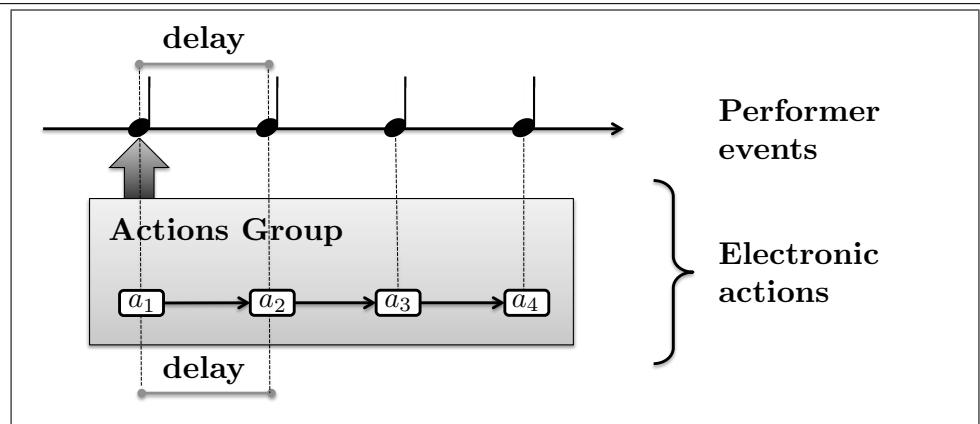
9.1.1 Loose Synchronization

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`.

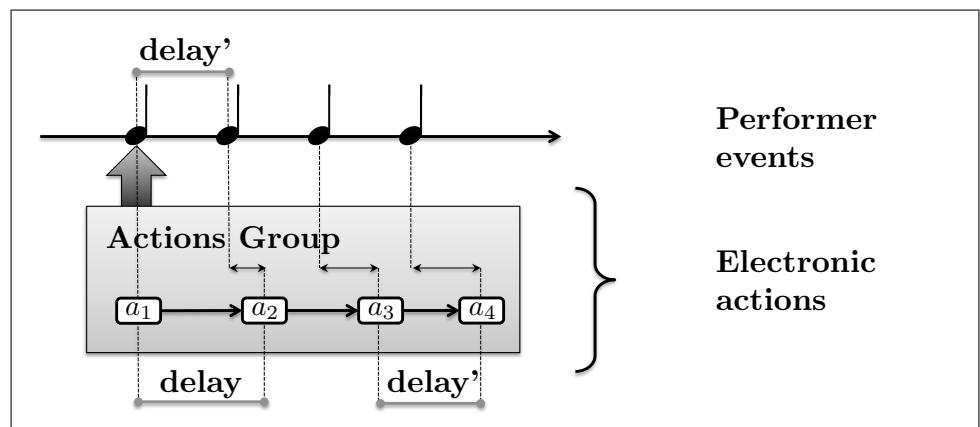
Figure 9.1 attempts to illustrate this within a simple example: Figure 9.1(a) shows the *ideal performance* or how actions and instrumental score is given to the system. In this example, an accompaniment phrase is launched at the beginning of the first event from the human performer. The accompaniment 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 alternative synchronization strategy where the electronic actions will be launched as close as possible to the events detection.

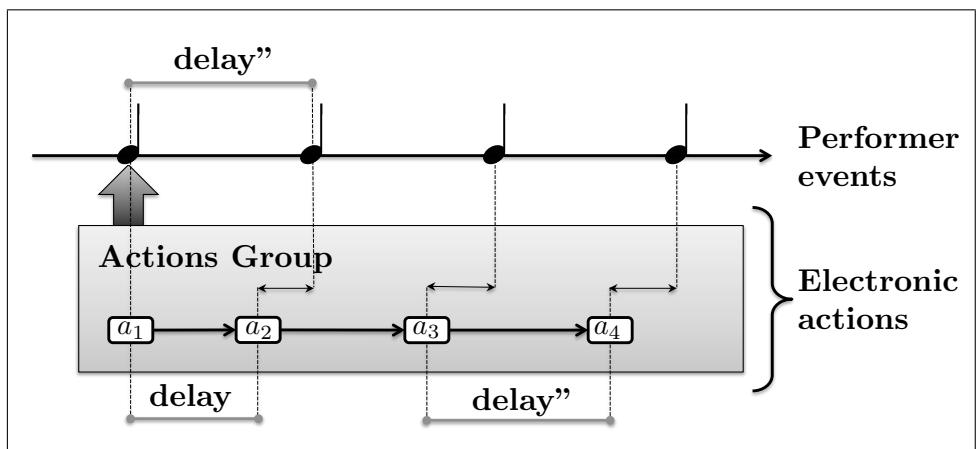
Figure 9.1 The effect of tempo-only synchronization for accompaniment phrases: illustration for different tempi. In the score, the actions are written to occur simultaneously with the notes, cf. fig. (a). Figure (b) and (c) illustrate the effect of a faster or a slower performance. In these cases, the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

9.2 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 responsibility 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 9.2 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:

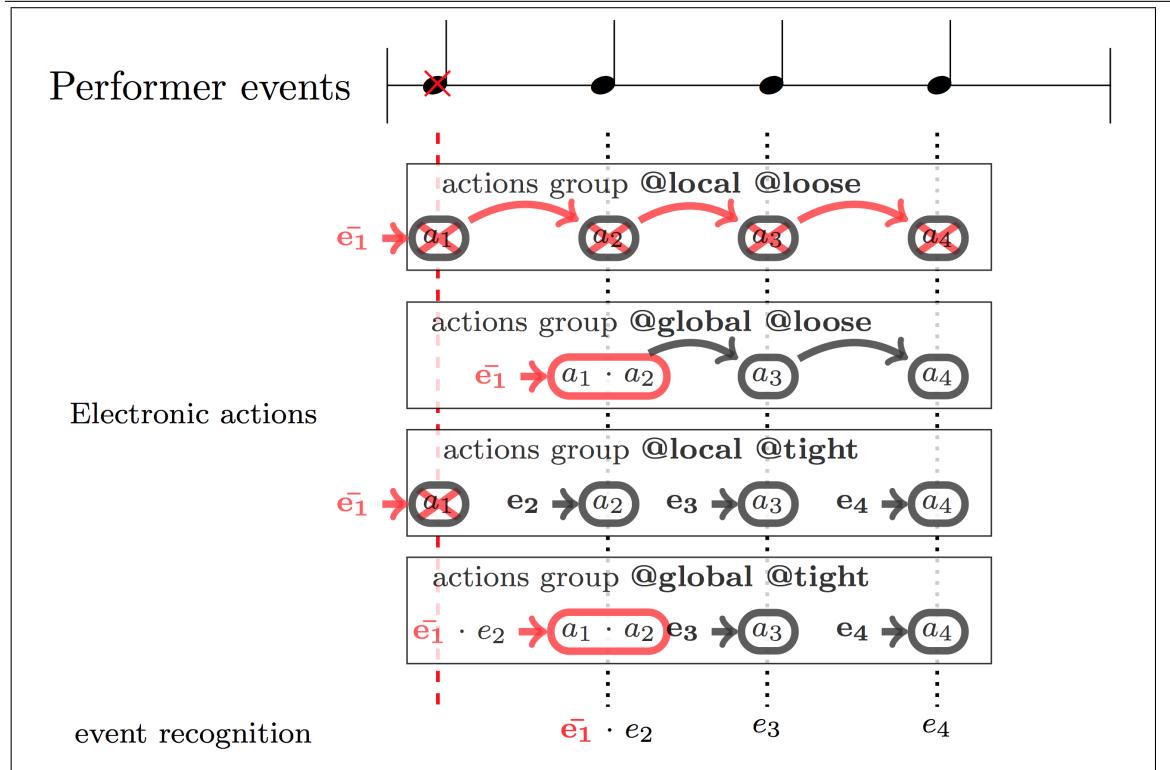
- `local` and `loose`: A block that is both local and loose correspond to a musical entity with some sense of rhythmic independence with regards to synchrony to its counterpart

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 9.2. To have a good

Figure 9.2 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_1 is reported. The signaling of the missing e_1 is denoted by \bar{e}_1 .



understanding of the picture note that:

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

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.

```

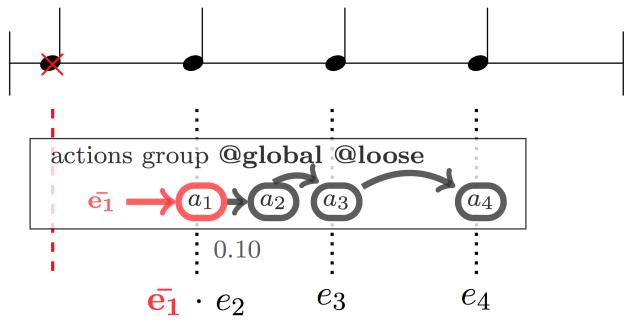
NOTE C 2.0
Group @global @loose
{
  d1 action1
  d2 group G1
  {
    action2
  }
  NOTE D 1.0
}
NOTE C 2.0
Group @global @loose
{
  d1 action1
  d2 group G1
  {
    action2
  }
}
NOTE D 1.0

```

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

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

```
NOTE C 1.0 e1
group G1 @global @loose
{
    a1
    1.10 a2
    1.0  a3
    1.0  a4
}
NOTE D 1.0 e2
NOTE D 1.0 e3
NOTE D 1.0 e4
```



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* and create those pattern by simply calling user-defined macros with arguments.

Macro can be defined using the `@macro_def` construct. Macros are called by the `@` operator and their arguments and a call to a Macro is simply replaced by its definitions and given argument in the core of a score. 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. When a syntax error occurs in the expansion of a macro, the location given refers to the text of the macro and is completed by the location of the macro-call site (which can be a file or the site of another macro-expansion).

A *process* (section 11) can be often used in place of a macro with several advantages. See paragraph 11.7 for a comparison of macros and processes.

10.1 Macro Definition and Usage

Macro name are `@`-identifier (preceded by `@` symbol). For 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`.

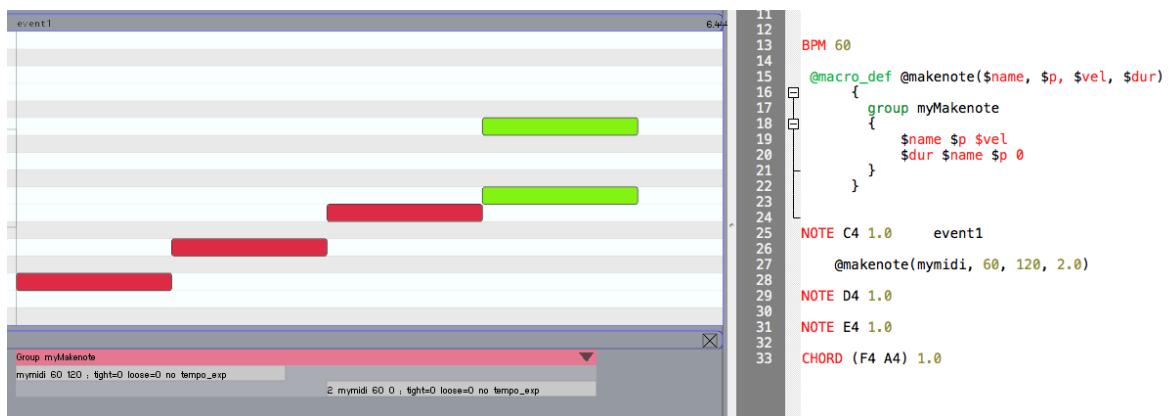
```
@macro_def @makenote($name, $p, $vel, $dur)
{
    group myMakenote
    {
        $name $p $vel
        $dur $name $p 0
    }
}
```

```
}
```

The two lines inside the `group` are *atomic actions* (similar to cues in a `qlist` object in Max/Pd with additional capabilities – see chapter 3) and the `group` puts them in a single unit and enables polyphony or concurrency (see chapter 5).

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.

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 `idxxx` 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 `idxxx` 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, there 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 0k)
```

will expand into

```
group @name = Gengroupone
{
  5 print 0k
  5 print 0k
}
```

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 definitions,

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

is replaced by

`yysuffix`
3.14159265

Chapter 11

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 “instantiation of the process”. They can be several instantiations 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 [7.7](#), and it is used for calling the process.

11.1 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 introduce a new scope for variables). However, the result of the call is evaluated as a group that takes place 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 instantiation 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 7.8. This value refers to the group launched by the process instantiation 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.

11.2 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
{
    ...
    actioni ...
}

```

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

```

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

```

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.

11.3 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($q, $p)
}

```

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.

11.4 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 }
  ...
}

```

11.5 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\u1d31passed\u1d31at\u1d31DrunkenClock\u1d31time" }

```

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. 7.8), 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 $which ":_\u1d31" ($MYSELF.$x) }

$x := "x\u1d31at\u1d31top\u1d31level"

Group G1 {
  @local $x
  $x := "x\u1d31local\u1d31at\u1d31G1"
  ::Q("Q\u1d31in\u1d31G1")
}

Group G2 {
  @local $x
  $x := "x\u1d31local\u1d31at\u1d31G2"
  ::Q("Q\u1d31in\u1d31G2")
}

::Q("Q\u1d31at\u1d31top\u1d31level")

```

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, 7.8 and 6.2.3). A local variable may also be accessed through the name of its definition scope, refer to paragraph 6.2.3 using the `::` operator.

11.6 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.

11.7 Macro *vs.* Processus

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

- the expression that are arguments of a call are computed only once at call time, and call time is when the call is fired (not when the program file is parsed);
- the actions that are launched consequently to the firing of a process application are computed when the process is applied;
- process can be recursive;
- the instance of a process is an autonomous entity that can be referred through its `exec`, for example to access its local variables or to abort it;

- process are higher-order values: a process $::P$ can be used as the value of the argument of a function or a process call. This enable the parameterization of process, an expressive and powerful construction to describe complex compositional schemes.

Chapter 12

Antescofo Workflow

This chapter is still to be written...

12.1 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.

...

12.2 Tuning the Listening Machine

...

12.3 Debuging an *Antescofo* Score

12.4 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 12.8 for additional information on the old syntax.

12.4.1 Monotoring with Notability

...

12.4.2 Monotoring with *Ascograph*

...

12.4.3 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 is interpreted as `false` (which will prohibit the triggering of the `whenever` body).

12.5 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.

12.5.1 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 fiddle or yin objects).
- 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.

12.5.2 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

12.5.3 Predefined Messages

The *Antescofo* object accepts predefined message sent to `antescofo-mess1`. These messages corresponds to the internal commands described in section 4.6.

12.6 Interacting with PureData

...

12.7 *Antescofo* Standalone Offline

...

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

The help of the command line is given in Fig. 12.1.

Figure 12.1 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.
--recognition : Audio recognition-only mode (no action is triggered) (-r)
An audio file is needed.

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

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

Reactive module options:
--messagename : 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)
```

12.8 Old Syntax

...

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

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

where:

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

12.9 Stay Tuned

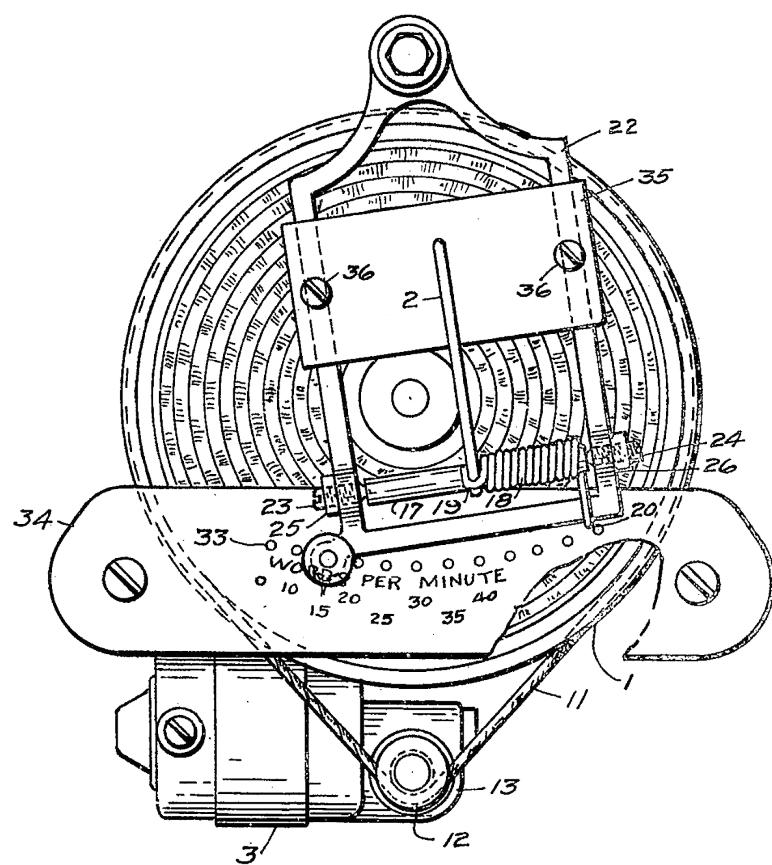
Antescofo is in constant improvement and evolution. Several directions are envisioned; to name a few:

- 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>



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. 79: so `@&&(a, b)` evaluates `b` irrespectively of the value of `a`

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

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

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

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

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

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

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

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

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

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

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

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

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

`@bounded_integrate_inv`

`@bounded_integrate`

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

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

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

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

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

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

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

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

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

`@count` (tab 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.

`@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.

`@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. There 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 68.

`@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 `datai` 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_map` (`variable`) : This is a special form. It returns a map of the value of the variable in argument. See. sect. 6.2.1 page 68

`@history_tab` (`variable`) : This is a special form. It returns a `tab` of the value of the variable in argument. See. sect. 6.2.1 page 68

`@history_map_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. 6.2.1 page 68

`@history_tab_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. 6.2.1 page 68

`@history_map_date (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. 6.2.1 page 68

`@history_tab_date (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. 6.2.1 page 68

`@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`

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

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

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

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

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

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

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

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

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

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

`@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,
```

`@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.

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` (numeric, numeric): `@make_duration_map(a, b)` returns a map where the integer i is associated to the duration (in beat) of the i th event of the score, for $a \leq i \leq b$.

`@make_score_map` (numeric, numeric): `@make_score_map(a, b)` returns a map where the integer i is associated to the position (in beat) of the i th event of the score, for $a \leq i \leq b$.

`@map` (f :function, t :tab) returns a tab such that element i is the result of applying f to element $t[i]$. Note that

```
@map(f, t) ≡ [f($x) | $x in t]
```

`@map_compose` (a :map, b :map): returns a map c such that $c(x) = b(a(x))$.

`@map_concat` (a :map, b :map): returns a map c which contains the (key, value) pairs of a and a pair (n, e) for each pair (k, v) in b with n ranging from `@size(a)` to `@size(a) + @size(b)`. If a and b are “vectors” (*i.e.* the range is an interval $[0, p]$ for some p), then `@map_compose` is the vector concatenation.

`@map_normalize` (map)

`@map_reverse` (map): `@map_reverse(m)` returns a new map p such that $p(i) = m(sz - i)$ with $sz = @size(m)$.

`@mapval` (map, function): `mapval(m, f)` return a map p such that $p(x) = f(m(x))$.

`@max_key` (map): returns the maximal element in the domain of the argument.

`@max_val` (tab *or* map): returns the maximal element in the tab if it is a map, the maximal element in the domain of the map if the argument is a map.

`@max` (*value, value*); return the maximum of its arguments. Values in *Antescofo* are totally ordered. The order between two elements of different type is implementation dependent. However, the order on numeric is as expected (numeric ordering, the integers are embedded in the decimals). For two argument of the same type, the ordering is as expected (lexicographic ordering for string, etc.).

`@member` (tab *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): asymmetric merge of the two argument maps. The result of `@merge(a, b)` is a map c such that $c(x) = a(x)$ if $a(x)$ is defined, and $b(x)$ elsewhere.

`@min_key`(map): return the minimal element in the domain of its argument.

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

`@min`(value, value): returns the minimal elements in its arguments.

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

`@occurs`(tab 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 lexicographically 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(variable1, ... variablep)` 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 68 and also the library function `@rplot`.

`@pow`(numeric, numeric), listable: `@pow`(x, y) compute x raised to the power y.

`@push_back`(tab, value), impure: add the second argument at the end of the first argument. The first argument, modified by side-effect, is the returned value.

`@push_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_bernouilli(p:float)`, *impure*. The members of the `@rnd_distribution` family return a random generator in the form of an impure function `f` taking no argument. Each time `f` is called, the value of a random variable following the `distribution` distribution is returned. The arguments of the `@rnd_distribution` are the parameters of the distribution. Two successive calls to `@rnd_distribution` returns two different random generators for the same distribution, that is, generators with unrelated seeds.

`@rnd_bernouilli(p)` returns a boolean random generator with a probability `p` to have a `true` value. For example

```
$bernoilli := @rnd_bernoilli(0.6)
$t := [ $bernoilli() | (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(variable1, ... variablep)` 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 68 and also the library function `@plot`.

`@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`

`@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 dimension of the nim.

`@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, 3, 3, 3, 4, 5, 5, 5 ]
```

`@sqrt` (*x*:numeric), *listable*: computes the non-negative square root of *x*.

`@stutter` (*t*:tab, *n*:numeric), *impure*: returns a new tab whose elements are repeated *n* times. The receiver is unchanged. The argument is unchanged.

```
@stutter([1, 2, 3, 4, 5, 6], 2)
→ [ 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 ]
```

`@system` (*cmd*:string), *impure*: This function hands the argument *cmd* 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 occurred (in this case an error message is issued).

`@tan` (*x*:numeric), *listable*: computes the tangent of *x* (measured in radians).

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 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 expressions¹ (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"
```

Tracks 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: their arguments are evaluated as usual but the final shipping to Max or PD is 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 define the track:

```
@track_def track::all { ".*" }
```

and mute it:

```
antescofo::mute track::all
```

B.3 Abort Handler

An `abort` command can be used to stop a compound action before its natural end, cf. section 5.1.4. It is then often convenient to execute some dedicated actions at the premature end, actions that are not needed when the compound action reaches its natural end².

¹The 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

²This 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

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 actions³. 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
```

`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.

³with the current exception of `whenever`

```

print: Curve Aborted at 1.5
print: handler curve: 1.5
print: handler curve: 1.2
print: handler curve: 0.9
print: handler curve: 0.6
print: handler curve: 0.3
print: handler curve: 0.

```

B.4 Patterns

Patterns are a simple way to define complex logical conditions to be used in a `whenever`. A pattern is a sequence of *atomic patterns*. They are three kinds of atomic patterns: `Note`, `Event` and `State`.

Such a sequence can be used as the condition of a `whenever` to trigger some actions every time the pattern matches. It can represent a *neume*, that is a melodic schema defining a general shape but not necessarily the exact notes or rhythms involved. It can also be used in broader contexts involving not only the pitch detected by the *Antescofo* listening machine, but also arbitrary variables.

Warning: The notion of pattern used here is very specific and the recognition algorithm departs from the recognition achieved by the listening machine. Patterns defines an exact variation in time of variables while the listening machine recognizes the most probable variation from a given dictionary of musical events. The latter relies on probabilistic methods. The former relies on algorithms like those used for recognizing regular expressions. So the pattern matching available here is not relevant for the audio signal, even if it can have some applications⁴.

B.4.1 `Note`: Patterns on Score

The basic idea is to react to the recognition of a musical phrase defined in a manner similar to event's specification. For example, the statement:

```

@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"
}

```

⁴ Note also that the pattern matching is running asynchronously on variables supposed to be updated at most at the rate of control.

The `Note` pattern is an *atomic pattern* and the `@pattern_def` defines and gives a name to a sequence of atomic patterns.

In the current version, the only event recognized are `Note`: `Trill`, `Chord`, etc., cannot be used (see however the other kinds of atomic patterns below). Contrary to the notes in the score, the duration may be omitted to specify that any duration is acceptable.

Pattern Variables. To be more flexible, patterns can be specified using local variables that act as wildcards:

```
@pattern_def pattern::Q
{
  @Local $h

  Note $h
  Note $h
}
```

This pattern defines a repetition of two notes of the same pitch (and their respective duration do not matter). The wildcard, or *pattern variable* `$h`, is specified in the `@Local` clause at the beginning of the pattern definition. Every occurrence of a pattern variable must refer to the same value. Here, this value is the pitch of the detected note (given in midicents).

Pattern variables are really local variables and their scope extends to the body of the `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 (both corresponding to midicents); a symbolic midi pitch; a pattern variable or a variable. The duration of a pattern `Note` can be an integer, a pattern variable or a variable. For example,

```
@pattern_def pattern::R
{
  Note $X
  Note C4 $Y
}
```

specifies a sequence of two notes, the first one must have a pitch equal to the value of the variable `$X` (at the time where the pattern is checked) and the pitch of the second one is `C4`, and the duration of the first is irrelevant while the duration of the second must be equal to

the value of $\$Y$ (as for $\$X$, this variable is updated elsewhere and the value considered is its value at the time where the pattern is checked).

An additional `where` clause can be used to give finer constraints:

```
@pattern_def pattern::R
{
  @Local $h, $dur1, $dur2

  Note $h $dur1 where $h > 6300
  Note $h $dur2 where $dur2 < $dur1
}
```

`pattern::R` specifies a sequence of two successive notes such that:

- their pitch is equal and this value in midicents is the value of the local variable $\$h$;
- $\$h$ is higher than 6300 midicents;
- and the duration of the second note must be lower than the duration of the first note.

Pattern Causality. In a `where` clause, all variables used must have been set before. For example, it is not possible to refer to $\$dur2$ in the `where` clause of the first note: the pattern recognition is *causal* which means that the sequence of pattern is recognized “on-line” in time from the first to the last without guessing the future.

A Complete Exemple. It is possible to refer in the various clause of a pattern to variables (or expression for the `where` clause) computed elsewhere. For example

```
@pattern_def pattern::M
{
  @Local $h, $dur

  Note $X $dur
  Note $h $dur where $dur > $Y
  Note C4
}
```

defines a sequence of 3 notes. The first note has a pitch equal to $\$X$ (at the moment where the pattern is checked); the second note as an unknown pitch referred by $\$h$ and its duration $\$dur$, which is the same as the duration of the first note, must be greater than the current value of $\$Y$; and finally, the third note as a pitch equal to $C4$.

B.4.2 `Event` on Arbitrary Variables

From the listening machine perspective, a `Note` is a complex event to detect in the input audio stream. But from the pattern matching perspective, a `Note` is an atomic event that can be detected looking only on the system variables `$PITCH` and `$DURATION` managed by the listening machine.

It is then natural to extend the pattern-matching mechanism to look after any *Antescofo* variable. This generalization from `$PITCH` to any variable is achieved using the `Event` pattern:

```
@pattern_def pattern::Gong
{
  @Local $x, $y, $s, $z

  Event $S value $x
  Event $S value $y at $s where $s > 110
  Before [4]
  Event $S value $z where [$x < $z < $y]
}
```

The keyword `Event` is used here to specify that the event we are looking for is an *update* in the value of the variable `$S`⁵. We say that `$S` is the *watched variable* of the pattern.

An `Event` pattern is another kind of atomic pattern. `Note` and `Event` patterns can be freely mixed in a `@pattern_def` definition.

Four optional clauses can be used to constraint an `event` pattern:

1. The `before` clause is used to specify a temporal scope for looking the pattern.
2. The `value` clause is used to give a name or to constraint the value of the variable specified in the `Event` at matching time.
3. The `at` clause can be used to refer elsewhere to the time at which the pattern occurs.
4. The `where` clause can be used to specify additional logical constraint.

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 than the `value` clause in a `note` pattern: it accepts a pattern variable or an arbitrary expression. An arbitrary 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.

⁵A variable may be updated while keeping the same value, as for instance when evaluating `$S := $S`. Why `$S` is updated or what it represents does not matter here. For example, `$S` can be the result of some computation in *Antescofo* to record a rhythmic structure. Or `$S` is computed in the environment using a pitch detector or a gesture follower and its value is notified to *Antescofo* using a `set_var` message.

The `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 unlucky 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 $p = \text{Event } \$x\dots$ 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 constraints⁶. However the *Antescofo* pattern matching stops looking for further occurrences in the same temporal scope, after having found the first one. This behavior is called the *single match* property.

For instance, if the variable `$V` takes the same value three times within 3 seconds, say at the dates $t_1 < t_2 < t_3$, then `pattern::twice` occurs three times as (t_1, t_2) , (t_1, t_3) , and (t_2, t_3) . Because *Antescofo* stops to look for further occurrences when a match starting at a given date is found, only the two matches (t_1, t_2) and (t_2, t_3) are reported.

Finally, notice that the temporal scope defined on an event starts with the preceding event. So a `before` clause on the first `Event` of a pattern sequence is meaningless and actually forbidden by the syntax.

Watching Multiple Variables Simultaneously. It is possible to watch several variables simultaneously: the event occurs when one of the watched variable is updated (and if the constraints are fulfilled). For instance:

```
@pattern_def pattern::T
{
  @Local $s1, $s2

  Event $X, $Y at $s1
  Event $X, $Y at $s2 where ($s2 - $s1) < 1
}
```

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.

⁶If there is no `before` clause, the temporal scope is “the first value change” which implies that there is *at most* one match.

B.4.3 **State** Patterns

The **Event** pattern corresponds to a logic of *signal*: each variable update is meaningful and a property is checked on *a given point in time*. This contrasts with a logic of *state* where a property is looked *on an interval of time*. The **State** pattern can be used to face such case.

A Motivating Example. Suppose we want to trigger an action when a variable $\$X$ takes a given value v for at least 2 beats. The following pattern:

```
Event $X value v
```

does not work because the constraint “at least 2 beats” is not taken into account. The pattern matches every times $\$X$ takes the value v .

The pattern sequence

```
@Local $start, $stop
Event $X value v at $start
Event $X value v at $stop where ($stop - $start) >= 2
```

is not better: it matches two successive updates of $\$X$ that span over 2 seconds. Converting the absolute duration in relative time is difficult because it would imply to track all the tempo changes in the interval. More importantly, it would not match three consecutive updates of $\$X$ for the same value v , one at each beat, a configuration that should be recognized.

This example shows that is not an easy task to translate the specification of a state that lasts over an interval into a sequence of instantaneous events. This is why, a new kind of atomic pattern to match states has been introduced. Using a **State** pattern, the specification of the previous problem is easy:

```
State $X where $X == v during 2
```

matches an interval of 2 beats where the variable $\$X$ constantly has the value v (irrespectively of the variable updates).

Four optional clauses can be used to constraint a **state** pattern:

1. The **before** clause is used to specify a temporal scope for looking the pattern.
2. The **start** clause can be used to refer elsewhere to the time at which the matching of the pattern has started.
3. The **stop** clause can be used to refer elsewhere to the time at which the matching of the pattern stops.
4. The **where** clause can be used to specify additional logical constraint.
5. The **during** clause can be used to specify the duration of the state.

The **before** clause must be given before the **state** keyword. The others can be given in any order after the specification of the watched variable. There is no **value** clause because the value of the watched variable may change during the matching of the pattern, for instance when the state is defined as “being above some threshold”.

The first three clauses are similar to those described for an `event` pattern, except that the `at` is split into the `start` and the `stop` clauses because here the pattern is not “point wise” but spans an interval of time.

The initiation of a `state` Pattern. Contrary to `note` and `event`, the `state` pattern is not driven solely by the updates of the watched variables. So the matching of a `state` is initiated immediately after the end of the previous matching.

The `during` Clause. The optional `during` clause is used to specify the time interval on which the various constraints of the pattern must hold. If this clause is not provided, the `state` finishes to match as soon as the constraint becomes false. Figure B.1 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 .

There is no specification of a duration, so the second pattern finishes its matching as soon as `$X` becomes smaller than b .

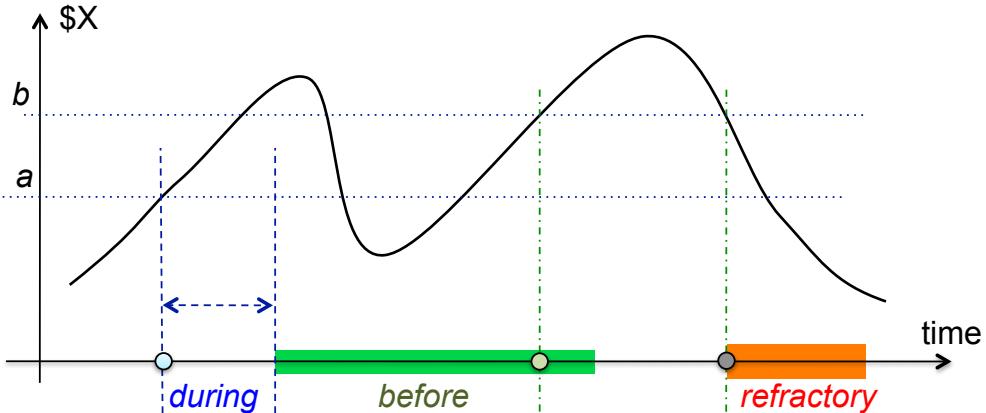
With the sketched curve, there are many other possible matches corresponding to delaying in time the start of the first `state` while still maintaining `$X > b`. Because the start time of these matches are all different, they are not ruled out by the *single match* property. A `@refractory` period is used to restrict the number of successful (reported) matches.

B.4.4 Limiting the Number of Matches of a Pattern

A `@Refractory` clause specify the period after a successful match during which no other matches may occur. This period is counted starting from the end of the successful match. The refractory period is represented in red in Figure B.1. 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.

Figure B.1 State patterns with `during`, `before` and `@refractory` clauses.



B.4.5 Pattern Compilation

Patterns are not a core feature of the *Antescofo* language: internally they are compiled in a nest of `whenever`, conditionals and local variables. If verbosity is greater than zero, the `printfwd` command 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 reported⁷.

⁷ Alternative behaviors may be considered in the future.

Appendix C

Index

@&&, 13, 126
 @*, 13, 126
 *, 79
 @+, 13, 125
 +, 79
 @-, 13, 126
 -, 79
 . (dot operator), 83, 113
 @/, 13, 126
 /, 79
 : (conditional expression), 74
 :=, *see* assignment
 @<, 13, 126
 <, 66
 @<=, 13, 126
 <=, 66
 =, 66
 @==, 13, 126
 @>, 13
 >, 66
 @>=, 13, 126
 >=, 66
 ? (conditional expression), 74
 @%, 13, 126
 %, 79
 @&&, 126
Antescofo
 bug report, 5
 discussion group, 123
 scientific publications, 5
 web page, 5
Antescofo
 bug report, 1
 forum, 1
 scientific publications, 1
 web page, 1
 \$-identifiers, 13
 ::-identifiers, 13
 @-identifiers, 12

 @abort, 12
 abort, 37
 flat, 37
 handler, 144
 processes, 37
 recursive, 37
 abort, 12
 @abs, 13, 80, 126
 @acos, 13, 80, 126
 action, 7, 10, 23, 31, 43
 abort, 36
 active (compound action), 36
 as expression, 75
 cancel, 38
 case, 46
 child, 43
 compound, 43
 conditional, 46
 external, 31
 father, 43
 fired, 23
 triggered, 23
 @action, 12
 action, 12
 actions command, 39
 @add_pair, 13, 86, 126, 131
 analysis command, 39
 and, 12
 @ante, 12
 ANTEIOI, 120
 antescofo-mess1, 120
 antescofo::actions, 39
 antescofo::analysis, 39
 antescofo::before_nextlabel, 39
 antescofo::bpmtolerance, 39
 antescofo::calibrate, 39
 antescofo::clear, 39
 antescofo::gamma, 39
 antescofo::getcues, 39
 antescofo::getlabels, 39
 antescofo::gotobeat, 39
 antescofo::gotocue, 39
 antescofo::gotolabel, 39
 antescofo::harmlist, 39
 antescofo::info, 40
 antescofo::jumptocue, 40
 antescofo::jumptolabel, 40
 antescofo::killall, 40
 antescofo::mode, 40
 antescofo::mute, 144
 antescofo::mute, 40
 antescofo::nextaction, 40
 antescofo::nextevent, 40
 antescofo::nextfwd, 40
 antescofo::nextlabel, 40
 antescofo::nofharm, 40
 antescofo::normin, 40
 antescofo::obsexp, 40
 antescofo::pedal, 40
 antescofo::pedalcoeff, 40
 antescofo::pedaltime, 40
 antescofo::piano, 40
 antescofo::play, 40
 antescofo::playfrom, 40
 antescofo::playfrombeat, 40
 antescofo::preload, 40
 antescofo::preventzigzag, 40
 antescofo::previousevent, 40
 antescofo::previouslabel, 40
 antescofo::printfwd, 40
 antescofo::printscore, 40
 antescofo::read, 40
 antescofo::report, 40
 antescofo::score, 41
 antescofo::setvar, 41
 antescofo::start, 41

antescofo::stop, 41
antescofo::suivi, 41
antescofo::tempo, 39
antescofo::tempoinit, 41
antescofo::temposmoothness, 41
antescofo::tune, 41
antescofo::unmute, 144
antescofo::unmute, 41
antescofo::variance, 41
antescofo::verbosity, 41
antescofo::verify, 41
antescofo::version, 41
@approx, 13, 127
arithmetic operators, 79
array, 91
Ascograph
 Curve editing, 54
Ascograph
 monitoring, 118
 score conversion, 117
 score edition, 117
Ascograph
 curve edition, 51
@asin, 13, 80, 127
assignation, 114
assignment, 34
 external, 36
 local variable (from outside its scope), 35
 of a tab element, 35, 93
@at, 13
at, 150
at, 12
@atan, 13, 80, 127
atomic values, 65
auto-delimited expression, 76

back interpolation, 57
back_in interpolation, 57
back_in_out interpolation, 57
back_out interpolation, 57
backslash, 32, 80
\, 32
BEATNUM, 120
\$BEAT_POS, 72
before, 150
before, 12
before_nxtlabel command, 39
@between, 13, 127
bind, 12
boolean value, 79
bounce interpolation, 57
bounce_in interpolation, 57
bounce_in_out interpolation, 57
bounce_out interpolation, 57
@bounded_integrate, 13, 91, 127
@bounded_integrate_inv, 13, 127
BPM, 11
bpm, 10, 12

bpmtolerance command, 39
calibrate command, 39
@car, 13, 95, 127
@car, 98
carriage-return, 80
case, 12, 46
causal score, 63, 73
causality, 63
@cdr, 13, 95, 127
@cdr, 98
@ceil, 13, 80, 127
CERTAINTY, 120
CFWD, 122
child of a compound action, 43
CHORD, 7, 11, 17, 18
Chord, 11
chord, 12, 18, 19
circ interpolation, 57
circ_in interpolation, 57
circ_in_out interpolation, 57
circ_out interpolation, 57
@clear, 13, 87, 95, 127
clear command, 39
closefile, 12
@coef, 12
column, 14
@command, 32
comments, 14
comparison, 66
@compose_map, 87
compound action, 43
 abort, 45
 instance, 44
 premature end, 45
compound values, 65
comprehension, 92
 predicate, 93
@concat, 13, 95, 98, 127
conditional
 action, 46
 expression, 74
conjunction, 79
@cons, 13, 95, 127
@cons, 98
containers, 43
@copy, 13, 127
@cos, 13, 80, 128
@cosh, 13, 80, 128
@count, 13, 80, 86, 95, 128, 129, 135, 136
cubic interpolation, 57
cubic_in interpolation, 57
cubic_in_out interpolation, 57
cubic_out interpolation, 57
curve
 full syntax, 53
 simplified syntax, 51
curve, 51

curve, 12
graphical edition with *Ascograph*, 51
curve
and NIM, 57

Data Structures, 65
dated access, 67, 70
@defined, 86
delay, 23
absolute, 24
beat, 24
relative, 24, 44
second, 24
zero, 24

dictionary, 85
@dim, 13, 95, 128
@dim, 92, 140
dimension, 92, 140
disjunction, 79
do, 12
domain, 85
@domain, 13, 86, 128
dot notation, 83, 113
@dsp_channel, 12
@dsp_inlet, 12
@dsp_outlet, 12
@dump, 39, 82, 128
@dumpvar, 39, 82, 128
duration, 18
\$DURATION, 72
during, 45, 153
during, 12

elastic interpolation, 57
elastic_in interpolation, 57
elastic_in_out interpolation, 57
elastic_out interpolation, 57
if, 74
else, 12, 46
@empty, 13, 74, 95, 128
@empty, 74
end of line, 14, 32, 41, 80
ENDBANG, 120
\$ENERGY, 72
error (in predefined functions), 66
error handling strategy, 101
EVENT, 18
event, 7, 10, 17
attributes, 20
fermata, 20
jump, 20
MIDI, 20
musicXML, 20
pizz, 20

event, 12
exec
abort, 83
active, 83

dead, 83
dot access, 83, 113
exec value, 82
@exp, 13, 80, 128
exp interpolation, 57
exp_in interpolation, 57
exp_in_out interpolation, 57
exp_out interpolation, 57
@explode, 13, 80, 128
EXPR, 75
expr, 12
expression
action as ~, 75
auto-delimited, 76
conditional, 74
external actions, 31

false, 12
father, 43
@fermata, 20
fermata, 20
file, 38
@find, 13, 80, 82, 86, 96, 128, 129, 135, 136
@flatten, 13, 82, 96, 129
float value, 80
@floor, 13, 80, 129
forall, 50
forall, 12
forum, 123
frames of reference, 27
@fun_def, 12
function, 81, 85, 88
curried, 81
domain, 85
extensional definition, 85
intentional definition, 81, 85
interpolated map, 88
listable, 125
map, 85
predefined, 13, 81
pure, 125
side-effect, 125

gamma command, 39
getcues command, 39
getlabels command, 39
GFWD, 122
gfwd, 12
@global, 12, 44
@gnuplot, 13, 82, 96, 129, 130
gnuplot, 68
gnuplot, 68
gnuplot, 69
gotobeat command, 39
gotocue command, 39
gotolabel command, 39
@grain, 12
group, 8, 11, 12

@gshift_map, 13, 87, 130
@guard, 12
handler, 144
harmlist command, 39
hierarchy
 of actions, 37, 43
 OSC names, 33
history
 as a map, 88
@history_map, 130
@history_map(\$x), 68
@history_map_date, 130, 131
@history_map_date(\$x), 68
@history_map_rdate(\$x), 68
@history_tab, 130
@history_tab(\$x), 68
@history_tab_date, 131
@history_tab_date(\$x), 68
@history_tab_rdate(\$x), 68
@hook, 12, 20
hook, 12, 20
HZ, 119
HZ, 119
identifier
 clash with keywords, 11
if, 74
if, 12, 46
imap, 12
@immediate, 12
@immediate, 61
immutable, 65
import
 midi file, 117
 MusicXML file, 117
in, 12
indentation, 14
info command, 40
inlet
 HZ, 119
 KL, 119
 MIDI, 119
@inlet, 12
@insert, 12, 13, 86, 96, 131
instances of a group, 44
integer value, 79
@integrate, 91, 131
internal command, *see antescofo::xxx*
interpolated map value, 88
interpolation
 linear, 56
 see also curve, 56
 step function, 56
 type, 57
 user defined, 57
IO, 38
@iota, 13, 131
@is_prefix, 82, 96
@is_subsequence, 82
@is_suffix, 82
@is_bool, 13, 66, 131
@is_defined, 13, 131
@is_exec, 66
@is_fct, 13, 66, 131
@is_float, 13, 66, 131
@is_function, 13, 66, 131
@is_int, 13, 66, 131
@is_integer_indexed, 13, 86, 131
@is_interpolatedmap, 13, 66, 131
@is_list, 13, 86, 131
@is_map, 13, 66, 132
@is_nim, 13, 66
@is_numeric, 13, 66, 132
@is_prefix, 13, 80, 132, 133
@is_proc, 66
@is_string, 13, 66, 132
@is_subsequence, 13, 80, 96, 132, 133
@is_suffix, 13, 80, 96, 132, 133
@is_symbol, 13, 66, 133
@is_tab, 66
@is_undef, 13, 66, 133
@is_vector, 13, 86, 133
iteration, *see loop*, *see forall*

@jump, 12, 20
jump, 12, 20
jmpctocue command, 40
jmpctolabel command, 40

KILL, 122
@kill, 12
kill, 37
kill, 12
KILL OF, 122
killall command, 40
KL, 119
KL, 119

@label, 12
@lace, 13, 96, 133
\$LAST_EVENT_LABEL, 72
let, 12, 34
LFWD, 122
lfwd, 12
@lid, 12
linear interpolation, 57
list
 as **tab**, 98
listable, 95, 125
@listify, 13, 88, 133
@loadvalue, 39, 133
@loadvar, 39, 134
@local, 12, 44, 69, 113
@Local, 147
@log, 13, 80, 135

@log10, 13, 80, 134
@log2, 13, 80, 134
 logical and, 79
 logical instant, 27
 logical operator, lazy, 79
 logical or, 79
Loop, 11
loop, 48

- halting, 49
- period, 49

loop, 12
@loose, 12, 44

@macro_def, 12
Main outlet, 119, 120
@make_duration_map, 13, 135
@make_duration_map(start, stop), 88
@make_score_map, 13, 135
@make_score_map(start, stop), 88
@map, 13, 96, 135

- map
 - arithmetic extension, 87
 - as an extensional function, 85, 86
 - construction, 85, 86
 - domain, 86
 - history representation, 88
 - iteration, 50
 - range, 86
 - score representation, 88
 - value, 85
- map, 12

@map_compose, 13, 135
@map_concat, 13, 135
@map_normalize, 135
@map_reverse, 13, 88, 135
@map_val, 87
@mapval, 13, 135
 matrix, 91
 matrix computations, 93
MAX, 31

- message, 31
- symbol, 32

@max, 13, 66, 80, 135
@max_val, 96
@max_key, 13, 86, 135
@max_val, 13, 86, 135
@member, 13, 80, 86, 96, 128, 129, 135, 136
@merge, 13, 87, 135
 MIDI, 119
 MIDI, 119
 midi file (import), 117
MIDIOUT, 120
@min, 13, 66, 80, 136
@min_val, 96
@min_key, 13, 86, 136
@min_val, 13, 86, 136
MISSED, 120
mode command, 40

model of time, 26
@modulate, 12
ms, 12, 24
MULTI, 17, 19
Multi, 11, 18, 19
multi, 12
multi_list, 17
 MusicXML file (import), 117
mutable, 65, 93
mute, 144
mute command, 40
\$MYSELF, 72
\$MYSELF, 83, 112

@name, 12
napro_trace, 12
 negation, 79
 nesting groups, 101
nextaction command, 40
nextevent, 18
nextevent command, 40
nextfwd command, 40
nextlabel command, 40
NIM, 88

- continuous, 89
- discontinuous, 89
- extension, 90
- vectorized, 89

nofharm command, 40
@norec, 12
@normalize, 13, 82, 96, 136
normin command, 40
NOTE, 7, 11, 17, 18
Note, 11
note, 146
note, 12, 19
NOTENUM, 120
\$NOW, 72

obsexp command, 40
@occurs, 13, 80, 86, 96, 128, 129, 135, 136
of, 12
off, 12
offline, 120
on, 12
open, 38
openoutfile, 38
openoutfile, 12
 operator

- listable, 95

OSC, 33
oscoff, 12
oscon, 12
oscrecv, 12
OSCSEND, 33
oscsend, 12
outlet, 119

- ANTEIOI, 120

BEATNUM, 120
 CERTAINTY, 120
 ENDBANG, 120
 score label, 120
 Main outlet, 120
 MIDIOUT, 120
 MISSED, 120
 NOTENUM, 120
 SCORETEMPO, 120
 TDIST, 120
tempo, 120
 TRACE, 120
 VELOCITY, 120

 parenthesizing expression, 76
parfor, 12
patch, 12
pattern, 146

- @Local**, 147
- @refractory**, 153
- at**, 150
- atomic, 147
- before**, 150
- causality, 148, 154
- duration, 147
- during**, 153
- event, 148
- on arbitrary variables, 148
- score, 146
- sequence, 147
- single match property, 151, 153, 154
- start**, 152
- state**, 152
- stop**, 152
- temporal scope, 150
- value**, 149
- variables, 147
- watched variables, 149, 151
- where**, 147, 150

@pattern_def, 12
 PD, 31

- message, 31
- symbol, 32

pedal command, 40
 pedalcoeff command, 40
 pedaltime command, 40
@permute, 13, 96, 136
 physical time, 28
 piano command, 40
 pitch, 17
\$PITCH, 72
pitch_list, 17
@pizz, 20
pizz, 20
 play command, 40
 playfrom command, 40
 playfrombeat command, 40
@plot, 82, 136

- @plot**, 68
- port, 12
- @post**, 12
- @pow**, 13, 80, 136
- preload command, 40
- preset, 128
- preventzigzag command, 40
- previousevent command, 40
- previouslabel command, 40
- printfwd command, 40
- printscore command, 40
- proc value, 82
- @proc_def**, 12
- process, 111
 - abort**, 37
 - abort**, 113
 - as value, 112
 - iteration, 50
 - recursive, 112
 - tempo, 115
 - variable, 113
- processus, 82
- Pure Data, *see* PD
- @push_back**, 97
- @push_front**, 97
- @push_back**, 13, 136, 137
- @push_back**, 90
- @push_front**, 13, 137

quad interpolation, 57
 quad_in interpolation, 57
 quad_in_out interpolation, 57
 quad_out interpolation, 57
 quart interpolation, 57
 quart_in interpolation, 57
 quart_in_out interpolation, 57
 quart_out interpolation, 57
 quint interpolation, 57
 quint_in interpolation, 57
 quint_in_out interpolation, 57
 quint_out interpolation, 57

- @rand**, 13, 80, 137
- @rand_int**, 13, 137
- @random**, 13, 137
- random number, 80
- @range**, 13, 86, 137
- @rank**, 97
- reactive system, 26
- read command, 40
- @reduce**, 13, 97, 137
- @refractory**, 12
- @refractory**, 153
- relational operators, 66, 79
- relative time, 28
- @remove**, 13, 87, 97, 137
- @remove_duplicate**, 97, 138
- @remove_duplicate**, 13

@replace, 13, 97, 138
report command, 40
@reshape, 13, 97, 138
@resize, 13, 97, 138
@reverse, 13, 80, 97, 138
@rnd_bernouilli, 138
@rnd_bernoulli, 13
@rnd_binomial, 13, 139
@rnd_exponential, 13, 139
@rnd_gamma, 13, 139
@rnd_geometric, 13, 139
@rnd_normal, 13, 139
@rnd_poisson, 13
@rnd_uniform_float, 13, 139
@rnd_uniform_int, 13, 139
\$RNOW, 72
@rotate, 13, 97, 140
@round, 13, 140
@rplot, 82, 140
@rplot, 68
\$RT_TEMPO, 72

s, 12, 24
@savevalue, 39, 140
scalar, 65
scalar product, 93
@scan, 13, 97, 140
score
 as a map, 88
 pattern, 146
score command, 41
score label, 120
SCORETEMPO, 120
@scramble, 13, 97, 140
@select_map, 13, 86, 140
set_var, 149
setvar, 36
setvar command, 41
shape, 140
@shape, 13, 140
@shape, 92
@shift_map, 13, 87, 140
@sin, 13, 80, 140
sine interpolation, 57
sine_in interpolation, 57
sine_in_out interpolation, 57
sine_out interpolation, 57
@sinh, 13, 80, 140
size
 non atomic value, 75
 scalar value, 75
 undefined value, 75
@size, 13, 74, 90, 97, 140
@sort, 13, 82, 97, 140, 141
special variables
 \$NOW, 72
@sputter, 13, 98, 141
@sqrt, 13, 80, 141

[], 91
standalone, 120
start, 152
start, 12
start command, 41
state, 152
state, 12
statement, 10
@staticscope, 12
stop, 152
stop, 12
stop command, 41
strategy
 error handling, 101
 synchronization, 99
string
 accessing a char, 80
 immutable, 80
 tab access, 80
string value, 80
@stutter, 13, 98, 141
suivi command, 41
switch, 12, 46
symb, 12
synchronization
 on a variable, 73
synchronization strategy, 24, 99
@system, 13, 141
system variables
 assignment, 35
 \$BEAT_POS, 72
 \$DURATION, 72
 \$ENERGY, 72
 \$LAST_EVENT_LABEL, 72
 \$MYSELF, 72
 \$PITCH, 72
 \$RNOW, 72
 \$RT_TEMPO, 72

tab, 91
 assignment, 93
 comprehension, 92
 extension, 92
tab
 and list, 98
 iteration, 50
tab, 12
tab value, 91
tabulation, 80
@tan, 13, 80, 141
@target, 12
TDIST, 120
tempo, 28
 the, 28
 inherited, 115
 local, 28
 tracking, 28
@tempo, 12, 44

tempo, 120
@tempo, 115
tempo command, 39
tempoinit command, 41
temporal clause, 45
temporal coordinate systems, 26
temporal scope, 45
temporal shortcuts, 63, 73
temporized system, 26
tempsmoothness command, 41
@tempovar, 163
@tempovar, 73
the tempo, 28
@tight, 12, 44
time
 absolute, 28
 logical instant, 27
 model of, 26
 physical, 28
 relative, 28, 44
 shortcuts, 63, 73
 wall clock, 28
time frame, 26
time frames, 27
TRACE, 120
track, 143
 mute, 144
 unmute, 144
@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, 41
@type, 12
type of a value, 66

@uid, 12
undefined value, 79
unmute, 144
unmute command, 41
until, 45
until, 12

value
 atomic, 65
 boolean, 79
 exec, 82
 float, 80
 immutable, 65
 integer, 79
 interpolated map, 88
 map, 85
 mutable, 65
 non atomic, 65
 ordering, 135
 proc, 82
 scalar, 65
 string, 80
 tab, 91
 types of, 66
 undefined, 35, 79
value, 149
value, 12
variable
 :: access, 70
 whenever restriction, 62
 access, 83, 113
 access outside its definition scope, 70, 113
 accessing through a label, 70
 assignment, 67
 assignment through OSC messages, 34
 dated access, 67
 declaration, 69
 dot access, 70, 83, 113
 external assignment, 36
 global, 69
 history, 67, 69
 lifetime, 70
 local, 69
 local to a pattern, 147
 scope, 69
 special, 35, 72, 83, 112
 stream of values, 67
 system, 35, 72
 tempovar, 73
 watched, 149
variance, 10, 12
variance command, 41
VELOCITY, 120
verbosity command, 41
verify command, 41
version command, 41

 wall clock time, 28
 watched variables, 61
whenever, 61
 and assigning a tab's element, 93
 during, 62
 pattern, 146
 temporal scope, 62
 temporal shortcut, 73
 temporal shortcuts, 63
 until, 62
 variable notification, 73
whenever, 12
where, 147, 150
where, 12
while, 45
while, 12
with, 75
 writing in a file, 38

Appendix D

Detailed Table of Contents

1	Understanding <i>Antescofo</i> scores	7
1.1	Structure of an <i>Antescofo</i> Score	7
1.2	Elements of an <i>Antescofo</i> Score	10
1.3	<i>Antescofo</i> keywords	11
1.4	@-identifiers: Functions, Macros, and Attributes	12
1.5	\$-identifiers: Variables	13
1.6	::-identifiers: Processes	13
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 <i>Antescofo</i>	20
2.4.1	Importing MIDI scores to <i>Antescofo</i>	21
2.4.2	Importing MusicXML scores to <i>Antescofo</i>	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
4	Atomic Actions	31
4.1	Message passing to Max/PD	31
4.2	OSC Messages	33
4.2.1	OSCSEND	33
4.2.2	OSCRECV	34

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.	36
4.4 Aborting and Cancelling an Action	36
4.4.1 Abort of an Action	37
Abort and the hierarchical structure of compound actions.	37
4.4.2 Cancelling an Action	38
4.5 I/O in a File	38
4.6 Internal Commands	39
4.7 Assertion Assert	41
5 Compound Actions	43
5.1 Group	43
5.1.1 Local Tempo.	44
5.1.2 Attributes of Group and Compound Actions	44
5.1.3 Instances of a Group	44
5.1.4 Aborting a group	45
The until Clause.	45
The during Clause.	45
5.2 If, Switch : Conditional and Alternative Actions	46
5.2.1 If : Conditional Actions	46
5.2.2 Switch : Alternative Actions	46
Alternative Action without Selector.	47
Alternative Action with a Selector.	47
5.3 Loop : Sequential iterations	48
Loop Period.	49
Stopping a Loop	49
5.4 Forall : Parallel Iterations	50
5.5 Curve : Continuous Actions	51
5.5.1 Simplified Curve Syntax	51
5.5.2 Full Curve Syntax	53
5.5.3 Actions Fired by a Curve	55
5.5.4 Step, Durations and Parameter Specifications	55
5.5.5 Interpolation Methods	56
Programming an Interpolation Method.	57
5.5.6 Curve with a NIM	57
5.6 Wheneve r: Reacting to logical events	61
Watching Restrictions.	62
5.6.1 Stopping a whenever	62
5.6.2 Causal Score and Temporal Shortcuts	63
Automatic Temporal Shortcut Detection.	64

6 Expressions	65
6.1 Values	65
Dynamic Typing.	66
Checking the Type of a Value.	66
Value Comparison.	66
6.2 Variables	67
6.2.1 Accessing Variable Histories	67
History reflected in a Map or in a Tab.	68
Plotting the history of a variable.	68
6.2.2 Variables Declaration	69
Local Variables.	69
History Length of a Variable.	69
Lifetime of a Variable.	70
6.2.3 Accessing a Local Variable “from Outside its Scope of Definition”	70
The :: Notation.	71
The Dot Notation.	71
6.3 Internal Antescofo Variables	72
6.3.1 Special Variables	72
6.3.2 Variables and Notifications	73
Temporal Shortcuts.	73
6.3.3 Dates functions	73
6.4 Temporal Variables	73
6.5 Operators and Predefined Functions	74
Conditional Expression.	74
@empty and @size.	74
6.6 Action as Expressions	75
Example.	75
6.7 Structuring Expressions	75
6.8 Auto-Delimited Expressions in Actions	76
7 Scalar Values	79
7.1 Undefined Value	79
7.2 Boolean Value	79
7.3 Integer Value	79

7.4	Float Value	80
7.5	String Value	80
7.6	User-defined Functions	81
	Curryfied Functions.	81
7.7	Proc Value	82
7.8	Exec Value	82
	Accessing a Local Variable Through an <i>exec</i>	83
8	Data Structures	85
8.1	Map Value	85
	Extensional Functions.	85
	Domain, Range and Predicates.	86
	Constructing Maps.	86
	Extension of Arithmetic Operators.	87
	Maps Transformations.	87
	Score reflected in a Map.	88
	History reflected in a map.	88
8.2	InterpolatedMap Value	88
	NIM interpolated Map.	89
	Vectorized NIM.	89
	Extending a NIM.	90
	Interpolated Map.	91
8.3	Tables	91
	Multidimensional tab.	91
	Tab Comprehension.	92
	Changing an element in a Tab	93
	Tab operators.	95
	Tab manipulation.	95
	Lists and Tabs.	98
9	Synchronization and Error Handling Strategies	99
9.1	Synchronization Strategies	99
	9.1.1 Loose Synchronization	99
	9.1.2 Tight Synchronization	101
9.2	Missed Event Errors Strategies	101
	Combining Synchronization and Error Handling.	101
10	Macros	105
10.1	Macro Definition and Usage	105
10.2	Expansion Sequence	107
10.3	Generating New Names	107
11	Process	111
11.1	Calling a Process	111

11.2 Recursive Process	112
11.3 Process as Values	112
11.4 Aborting a Process	113
11.5 Processes and Variables	113
11.6 Process, Tempo and Synchronization	115
11.7 Macro <i>vs.</i> Processus	115
12 Antescofo Workflow	117
12.1 Editing the Score	117
12.2 Tuning the Listening Machine	117
12.3 Debuging an <i>Antescofo</i> Score	118
12.4 Dealing with Errors	118
12.4.1 Monotoring with Notability	118
12.4.2 Monotoring with <i>Ascograph</i>	118
12.4.3 Tracing an <i>Antescofo</i> Score	118
Printing the Parsed File.	118
Verbosity.	118
The TRACE Outlet.	119
Tracing the Updates of a Variable.	119
12.5 Interacting with MAX	119
12.5.1 Inlets	119
12.5.2 Outlets	119
12.5.3 Predefined Messages	120
12.6 Interacting with PureData	120
12.7 <i>Antescofo</i> Standalone Offline	120
12.8 Old Syntax	122
12.9 Stay Tuned	122
A Library of Predefined Functions	125
B Experimental Features	143
B.1 Reserved Experimental Keywords	143
B.2 Tracks	143
B.3 Abort Handler	144
Example.	145
B.4 Patterns	146
B.4.1 Note: Patterns on Score	146
Pattern Variables.	147
Specifying Duration.	147
Specifying Additional Constraints.	147
Pattern Causality.	148
A Complete Exemple.	148
B.4.2 Event on Arbitrary Variables	148
The value Clause.	149

The <code>at</code> Clause.	150
The <code>where</code> Clause.	150
The <code>before</code> Clause.	150
Watching Multiple Variables Simultaneously.	151
A Complex Example.	151
B.4.3 State Patterns	152
A Motivating Example.	152
The initiation of a <code>state</code> Pattern.	153
The <code>during</code> Clause.	153
B.4.4 Limiting the Number of Matches of a Pattern	153
B.4.5 Pattern Compilation	154
C Index	155
D Detailed Table of Contents	165

