

Antescofo

A not-so-short introduction to version 0.5x

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

November 27, 2013

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

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

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

Additional information on *Antescofo* can be found:

- on the project home page
<http://repmus.ircam.fr/antescfo>
- on the IrcamForum User Group
<http://forumnet.ircam.fr/user-groups/antescfo/>
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/antescfo/>
- on the web site of the MuTanT project
<http://repmus.ircam.fr/mutant>
where you can find the scientific and technical publications on *Antescofo*.

Table of Contents

How to use this document	3	7.3 Sequential iterations: Loop	41
Brief history of Antescofo	3	7.4 Parallel Iterations: ParFor	42
1 Understanding <i>Antescofo</i> scores	4	7.5 Sampling parameters: Curve	42
1.1 Structure of an <i>Antescofo</i> Score	4	7.6 Reacting to logical events: Whenever	45
1.2 Elements of an <i>Antescofo</i> Score	5	8 Synchronization and Error Handling Strategies	48
1.3 Identifiers	7	8.1 Synchronization Strategies	48
1.4 Comments and Indentation	8	8.2 Missed Event Errors Strategies	50
2 Events	10	9 Process	52
2.1 Event Specification	10	9.1 Macro <i>vs.</i> Processus	52
2.2 Event Parameters	10	9.2 Recursive Process	53
2.3 Events as Containers	11	9.3 Process as Values	53
2.4 Event Attributes	12	10 Macros	55
3 <i>Antescofo</i> Model of Time	13	10.1 Macro Definitions	55
3.1 Logical Instant	13	10.2 Expansion Sequence	55
3.2 Time Frame	14	10.3 Generating New Names	56
4 Actions in Brief	16	11 <i>Antescofo</i> Workflow	58
4.1 Delays	16	11.1 Editing the Score	58
4.2 Label	17	11.2 Tuning the <i>Antescofo</i> Listening Machine	58
5 Expressions	19	11.3 Debugging an <i>Antescofo</i> Score	58
5.1 Values	19	11.4 Dealing with Errors	58
5.2 Variables	25	11.5 Interacting with MAX	59
5.3 Operators and Predefined Functions	29	11.6 Interacting with PureData	60
5.4 Auto-Delimited Expressions in Actions	30	11.7 <i>Antescofo</i> Standalone Offline	60
6 Atomic Actions	31	11.8 Old Syntax	62
6.1 Assignments	31	12 Stay Tuned	63
6.2 Aborting and Cancelling an Action	32	A Changes in 0.51	65
6.3 Max Messages	33	B Detailed Table of Contents	66
6.4 OSC Messages	34		
6.5 I/O in a File	36		
6.6 Internal Commands	36		
6.7 Assertion assert	38		
7 Compound Actions	39		
7.1 Group	39		
7.2 Conditional Actions: If	40		

How to use this document

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

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

Brief history of *Antescofo*

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

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

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

1 Understanding *Antescofo* scores

1.1 Structure of an *Antescofo* Score

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

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

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

Actions in *Antescofo* extend the good-old *qlist* object elements in MAX and PD with additional *Models of Time* (chapter 3), *expression* (chapter 5), *Compound Actions* (chapter 7), *Synchronization Strategies*, *Processes* and more covered in separate chapters of this document.

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

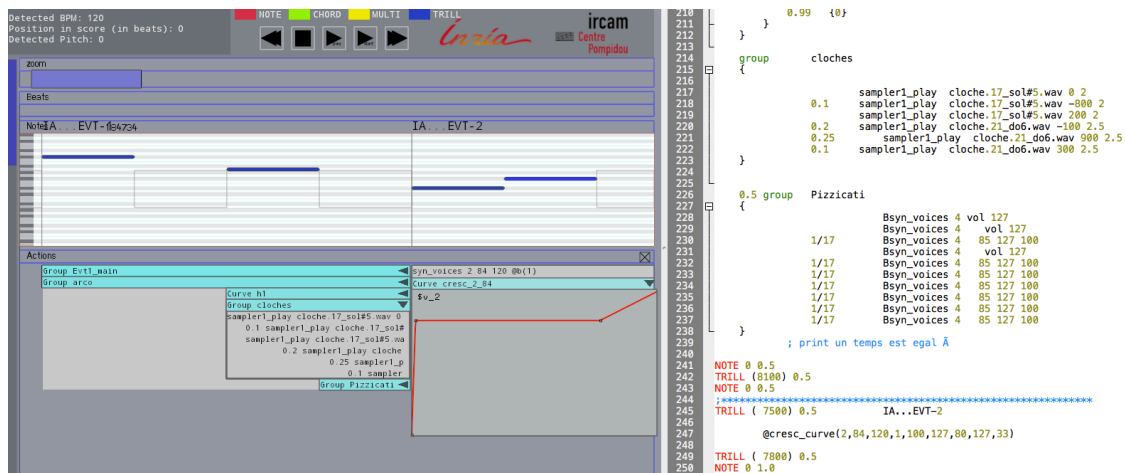


Figure 1: The beginning of *Tensio* (2010) by Philippe Manoury for String Quartet and Live electronics in *AscoGraph*

In Figure 1, 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) where as prior to that a compound **group** named `cloches` is supposed to

synchronize atomic actions (seen in text and collapsed group box in the visual screen) with the musician.

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**.

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

```
@insert macro.asco.txt
@insert "file_name_with_white_space_must_be_quoted"
```

The **@insert** keyword can be capitalized: **@INSERT**, as any other keyword beginning with a @ sign. An included file may includes (other) files. Macros can be defined to reuse program fragments, see section 10 (see also functions page 6 and processes section 9).

1.2 Elements of an *Antescofo* Score

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

As a consequence, an *Antescofo* program is a sequence of *statements, events* and *actions*. Events are recognized by the listening machine. They are described in section 2. Actions, outlined in in sections 4 and 6, are computations triggered by the occurrence of an event or of another action. The model of time of *Antescofo* is described in section 3 and the synchronization between events and actions is described in section 8. Actions are parameterized by *expressions* evaluated during the run of the program; they are described in section 5. Statements parameterize the behavior of the listening machine, of the reactive machine and the interactions between *Antescofo* and its environment.

Statements include:

- **bpm** specification: **bpm** 60 or **bpm** 50 **@modulate** (the pulsation can be given as an integer or a float);
- **transpose** *t* transposes the specification of the pitches in the following events by *t* (in midicents ; it can also be written **@transpose**);
- computation of the tempo: **tempo on** and **tempo off** (can also be written **@tempo**);
- external variable binding: **bind** \$mouse_position;
- variance specification: **variance** 3.7;
- additional connection with Max/MSP: **@inlet** x;
- function definition;
- process definition;

- macro definition.

Statement cannot appears within an action. They must be at the top level in the file. However, most of these statements correspond also to an internal action, see section 6.6. Macros, processus and functions can be used only after their definition in the score. We suggest to put them at the beginning of the file or to put them in a separate file that will be included at the beginning of the score.

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

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

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

Predefined functions are referred through a predefined @-name (case-sensitive). They include logarithmic, exponential and trigonometric functions, simple string manipulations and so on. See section 5 where they are listed in the subsection related to the type of their principal argument.

A user-defined intentional function definition takes the form

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

(indentation and carriage-return do not matters). The name of a function is an @-identifier as the names of predefined functions. The body of the function is an expression between braces. The parameters are \$-identifiers. Such functions can be recursive. See section 5.1.8 for additional information.

A user-defined extensional function is a dictionary defined by giving a list of pairs (key, value), see section 5.1.9. When bot key and value are numeric, they corresponding function can be interpolated between the breakpoints, see section 5.1.10).

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

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

The name of a process is an ::-identifiers. The parameters are \$-identifiers. Process can be recursive and the name of a process can be used in expressions (*e.g.*, as the argument of

another process). The example shows a recursive process that turn on and off a filter `filter` until *Antescofo* stops.

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

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

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

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

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

1.3 Identifiers

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

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

1.3.1 Simple Identifiers

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

Simple identifiers are used for labels, external name (*e.g.*, `Max`, `PD` or file names) and some keywords. The current list of simple identifier that are reserved for keywords is:

```
abort action assert bind bpm chord closefile curve do else event
```

```

expr false g fwd group hook if imap in jump kill let lfwd loop
loose map ms multi napro_trace note of off on openoutfile oscoff
oscon oscrecv oscsend parfor port s symb tab transpose trill true
until variance when whenever while

```

These keyword are *case unsensitive*, that is

```

note NOTE Note NoTe notE

```

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

1.3.2 @-identifiers

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

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

They are also some reserved @-identifiers used for the definition of functions and macros:

```

@fun_def @insert @lid @macro_def @proc_def @uid

```

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

```

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

```

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

1.3.3 \$-identifiers

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

1.3.4 ::-identifiers

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

1.4 Comments and Indentation

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

```

/*  comment split
    on several lines
*/

```


Line-comment are also 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_after_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 indentation used for cue-lists.

2 Events

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

The listening machine specifically is in charge of real-time automatic alignment of an audio stream played by one or more musicians, into a symbolic musical score described by Events. The *Antescofo* listening machine is 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).

2.1 Event Specification

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

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

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

```
EVENT ...
```

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

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

2.2 Event Parameters

The parameters of an event are as follows:

pitch is given by a number representing the pitch in midi or midicent, or a note name.

A negative pitch means that its corresponding note is tie with the same note of the previous event. For instance

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 D4 3/2`

represents the occurrence of a D with a duration of 1.5 beat.

pitch_list is a sequence of *pitches*:

`D4b 1200 112 D5#`

is a list of 4 notes.

trill_list is a sequence of: 1) *pitches* and 2) sequences of *pitches* (between parenthesis):

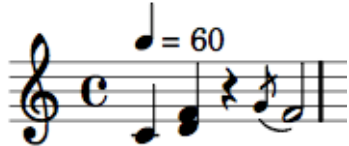
`D4b (E3 A3) D5`

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

`(E3 A3)'`

2.3 Events as Containers

Each event keyword in *Antescofo* in the above listing can be seen as *containers* with specific behavior and given nominal durations. A **NOTE** is a container of *one* pitch. A **chord** contains a vector of pitches. Figure 2 shows an example including simple notes and chords written in *Antescofo*:



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

Figure 2: Simple score with notes and chords.

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

Trill Similar to trills in classical music, a **Trill** is a container of events either as atomic pitches or chords, where the internal elements can happen in any specific order. Additionally, internal events in a **Trill** are not obliged to happen in the environment. This way, **Trill** can be additionally used to notate improvisation boxes where musicians are free to choose elements. A **Trill** is considered as a global event with a nominal relative duration. Figure 3 shows basic examples for Trill.

Multi Similar to **Trill**, a **Multi** is a compound event (that can contain notes, chords or event trills) but where the *order* of actions are to be respected and decoded accordingly in the listening machine. They can model continuous events such as *glissando*. Figure 4 shows an example of glissandi between chords written by **Multi**.




Figure 3 shows a musical staff with a treble clef and a common time signature (C). It contains three measures. The first measure has a quarter note on A4 with a trill ornament. The second measure has a quarter rest. The third measure has a quarter note on A#4 with a trill ornament. To the right of the staff, the following code is shown:

```

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

```

Figure 3: **TRILL** example on notes and chords



Figure 4 shows a musical staff with a treble clef and a common time signature (C). It contains two measures. The first measure has a half note on F4. The second measure has a half note on D4. To the right of the staff, the following code is shown:

```

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

```

Figure 4: **MULTI** example on chords

2.4 Event Attributes

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

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

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

```

Note D4 1 here @jump 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. 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.

3 Antescofo Model of Time

Actions are computations triggered after a delay that elapses starting from the occurrence of an event or another action. In this way, *Antescofo* is both a reactive system, where computations are triggered by the occurrence of an event, and a temporized system, where computations are triggered at some date.

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.

3.1 Logical Instant

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

- the recognition of a musical event;
- the assignment of a variable by the external environment (*e.g.* through an OSC message or a MAX/MSP 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 5.2. Within the same logical instant, synchronous actions are performed sequentially in the same order as in the score.

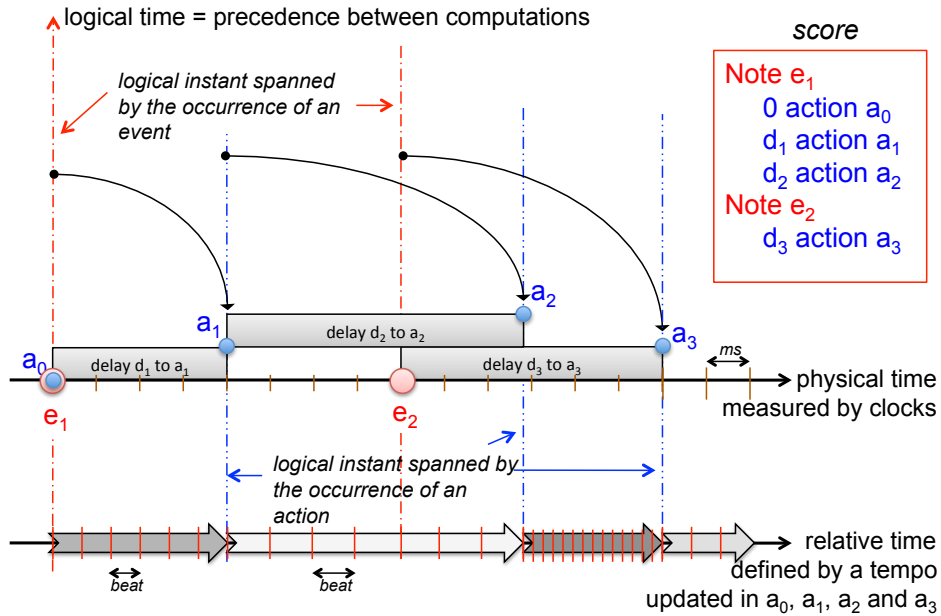


Figure 5: Logical instant, physical time frame and relative time frame corresponding to a computed tempo.

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 5, 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.

3.2 Time Frame

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

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

More generally, a frame of reference \mathcal{T} is defined by a *tempo* $T_{\mathcal{T}}$ which specifies the “passing of time in \mathcal{T} ” relatively to the physical time [?]. 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:

$$t_{\mathcal{T}} = \int_0^{t_{\mathcal{P}}} T_{\mathcal{T}} \quad (1)$$

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 7.1. This frame of reference is used for all relative delays and datation used in the actions within this group. The tempo expression is evaluated continuously in time for computing dynamically the relationships specified by equation (1).

Antescofo provides a predefined dynamic tempo variable through the system variable `$RT_TEMPO`. This tempo is 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

³The `$RT_TEMPO` is computed by *Antescofo* to mimics the tracking of the tempo by a human, and implements an idea of smooth tempo fluctuation, rather than trying to satisfy exactly equation (1) at any moment. So, for *the* relative time frame, equation (1) is only an approximation. As a consequence, the current position in the score is explicitly given by the variable `$BEATPOS` which is more accurate than the integration of `$RT_TEMPO`. See paragraph 5.2.3.

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

4 Actions in Brief

Actions are divided into *atomic actions* performing an elementary computation and *compound actions*. Compound actions group others actions. 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*, see the next paragraph. The old syntax for compound action, where the delay is after the keyword, is still recognized.

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

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

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

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

4.1 Delays

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

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

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

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

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

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

```

    a0
    1 s 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 7.1.1).

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 8.2.

4.2 Label

Labels are used to refer 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 score⁴.

Compound actions have an optional identifier (section 7). This identifier is a simple identifier and act as a label for the action.

⁴There is no useful notion of position of an action in the score because the same action may be fired several times (actions inside a **loop** or a **whenever** or associated to a **curve**).

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

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

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

5.1 Values

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

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

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

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

5.1.1 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 dependant.

5.1.2 Testing 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_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`.

5.1.3 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.

5.1.4 Boolean Value

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

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

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

5.1.5 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`.

5.1.6 Float Value

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

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

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

These functions correspond to the usual IEEE mathematical functions.

There is an additional function `@rand`, used to generate a random number between 0 and d : `@rand(d)`.

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

5.1.7 String Value

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

```
print "this_is_a_string_with_a\"_inside"
```

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

The `+` operator corresponds to string concatenation:

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

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

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

will output `abc33`.

5.1.8 Intentional Functions

Intentional functions f are defined by rules (*i.e.* by an expression) that specify how an image $f(x)$ is associated to an element x . Intentional functions can be defined and associated to an `@`-identifier using the `@fun_def` construct introduced in section 1.2 page 6. Some intentional functions are predefined and available in the initial *Antescofo* environment like the IEEE mathematical functions.

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

In an *Antescofo* expression, the `@`-identifier of a function denotes a functional value that can be used for instance as an argument of a higher-order functions (see examples of higher-order predefined function in section 5.1.9 for map building and map transformations).

5.1.9 Map Value

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

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

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

```
let $dico = map{ (1, "first"), (2, "second"), (3, "third") }
...
print ($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 and Intentional 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* defined in the next section 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 5.1.8.

In an expression, extensional function or intentional function can be used indifferently where a function is expected.

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

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

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

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

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

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

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

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

enriches the dictionary referred by `$dico` with a new entry (no new map is created).

- `@shift_map(m, n)` returns a new map m' such that $m'(x + n) = m(x)$

- `@gshift_map(m, f)` generalizes the previous operator using an arbitrary function f instead of an addition and returns a map m' such that $m'(f(x)) = m(x)$
- `@map_val(m, f)` compose f with the map m : the results m' is a new map such that $m'(x) = f(m(x))$.
- `@merge` combines two maps into a new one. The operator is asymmetric, that is, if $m = \text{merge}(a, b)$, then:

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

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

```
let $d1 := MAP{ (1, 10), (2, 20), (3, 30) }
let $d2 := MAP{ (2, 2), (3, 3), (4, 4) }
let $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 as Lists and Vectors. Vectors and lists can be emulated by map: a “map-emulated list” is a map satisfying the predicate `@is_list` and a “map-emulated vector” satisfies `@is_vector`. Several functions are defined on lists and vectors:

- Notice that `@map_val` is similar to the *map* function on list that exists in Lisp.
- `@concat(a, b)` returns the concatenation of two lists.
- Arithmetic operations on vectors are done pointwise.

However, using directly `tab`, cf. section 5.1.11, is probably simpler.

Maps transformations.

- `@listify` applied on a map m builds a new map where the key of m have been replaced by their order 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), ..., (1, vp), }
```

the following list is build:

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

- `@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

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

Score reflected in a Map. Two functions can be used to reflect the events of a score into a map⁵:

- `@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 5.2.1 pp. 26.

⁵Besides `@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.

5.1.10 InterpolatedMap Value

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

```
let $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.

5.1.11 Tab Value

Tab values are simple vectors.

Tab function `@tab_map`, `@concat`, `@min_val`, `@max_val`, `@size`, `@tab_reverse`, `@push_back`, `@resize`

5.1.12 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 5.1.8). Such value are qualified as *proc value*. Like intensionnal functions, proc value are first class value. They can be passed as argument to a function or a procedure call.

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

5.1.13 Exec Value

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

Warning: These features are still experimental.

5.2 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 in the box:

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

An assignment is an action, and as other action, it can be done after a delay. See sect. 6.1.

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.

5.2.1 Historicized Variables

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 the `let` construct. However, they can also be assigned by the external environment, using a dedicated API.

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

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

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

5.2.2 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 7. 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

```
let $x := 1
group {
  loc $x
  let $x := 2
  print "local_var_␣$x:␣" $x
}
print "global_var_␣$x:␣" $x
```

will print

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

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

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

```
 $n : $v$ 
```

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

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

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

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

The 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 5.2):

```
[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 but no more.

5.2.3 System Variables

There are several variables which are updated by the system in addition to `$RT_TEMPO`. Composers have read-only access to these variables.

Variable `$NOW` corresponds to the absolute date of the “current instant” in seconds. The “current instant” is the instant at which the value of `$NOW` is required. Variable `$RNOW` is the date in relative time (in beats) of the “current instant”. Variables `$PITCH`, `$BEATPOS` and `$DUR` are respectively the pitch, the position in the score and the duration of the last detected event.

Note that when the listening machine is waiting an event, `$BEATPOS` increases until reaching the position in the score of the waited event. The `$BEATPOS` is stuck until the occurrence of this event or the detection of a subsequent event (making this one missed).

5.2.4 Variables and Notifications

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

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

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

Temporal Shortcuts. The notification of a variable change may trigger a computation that may end, directly or indirectly, in the assignment of the same variable. This is known as a “temporal shortcut” or a “non causal” computation. The Event Manager takes care of stopping the propagation when a cycle is detected. See section 7.6.1. 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.

5.2.5 Dates functions

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

5.3 Operators and Predefined Functions

Most of the operators and predefined functions have been sketched in the section 5.1. We sketch here operators or function that are not linked to a specific type.

Conditionnal Expression. An important operator is the conditional:

```
if (bool_exp , exp1 , exp2)
```

returns the value `exp1` if `bool_exp` evaluates to `true` and else `exp2`. The parenthesis are mandatory.

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

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

The “size” of an undefined value is -1, and this can be used to test if a variable refers to an undefined value or not.

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

5.4 Auto-Delimited Expressions in Actions

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

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

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

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

```
print 1 2
```

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

Antescofo accepts only auto-delimited expressions to avoid possible ambiguities in the places where this is needed, *i.e.*:

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

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

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

6 Atomic Actions

An atomic action corresponds to

- an assignment,
- the abort of another action;
- an external action, that is: a MAX/PD message or an OSC message,
- an internal command,
- an assertion.

6.1 Assignments

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

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

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

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

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

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

Expressions in the right hand side of `:=` are described in section 5. A variable as a value before its first assignment: its value is the undefined value (sect. 5.1.3).

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

- the evaluation of a `whenever` that depends on this variable (see section 7.6);
- the reevaluation of the delays that depends on a relative tempo that depends on this variable⁶;

System variables cannot be assigned: `$RT_TEMPO`, `$PITCH`, `$BEATPOS`, `$DUR`, `$NOW`, `$RNOW`. These variables are *read-only* for the composer: they are assigned by the system during the performance. However, like usual variables, their assignment may trigger some activities. For instance delays expressed in the relative time are updated on `$RT_TEMPO` changes. Tight actions waiting on a specific event (cf. section 8.1.2) are notified on `$BEATPOS` changes. Etc. Refer to section 5.2.4 for additional information.

⁶As mentioned in section 4.1, the expression specifying a delay is evaluated only once, when the delay is started. It is not re-evaluated after that, even if the variable in the expression are assigned to new values. However, if the delay is expressed in a relative time, its conversion in physical time must be adjusted when the corresponding tempo changes.

6.2 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 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 action 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.

6.2.1 Abort of an Action

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

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

`kill abort` 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 7.6). All occurrences of an action labeled by *name* are aborted.

Abort and the hierarchical structure of compound actions. By default, the abort command applies recursively on the whole hierarchical structure. 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*.

6.2.2 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 ??).

6.3 Max Messages

The computations that can be performed within an *Antescofo* score are somewhat limited. Indeed, the role of *Antescofo* is to act as a coordinator between multiple tasks and not to be a general purpose language. So, *Antescofo* includes powerful features to interact with external processes. Such actions are called *external actions*. They are currently two main mechanisms to interact with external tasks: OSC messages described in the next section and MAX/PD messages⁷.

A MAX/PD message starts by an optional delay followed by a simple identifier referring to a MAX or PD receiver . This identifier must be different from the simple identifiers listed in section 1.3 page 7.

It is then followed by 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)

For instance,

```
let $a := 1
```

⁷The 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 6.6 and 11.5. On the contrary, arbitrary messages can be sent from an *Antescofo* score to external MAX objects.

```
print "the_value_of_the_variable_a_is" $a
print and here is a second message (2 * $a)
```

will print

```
the value of the variable a is 1
and here is a second message 2
```

Indeed, *Antescofo* expressions are evaluated to give the argument of the message. For the first print, there is 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 string, *Antescofo* integer into MAX/PD integer, etc.). In the second print message they are 7 arguments: the first six are simple identifiers converted into the corresponding symbol and the seventh argument is evaluated into an integer.

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 on the console

```
"this_string_will_appear_quoted"
```

6.4 OSC Messages

The OSC protocol⁸ 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.

6.4.1 OSCSEND

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

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

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

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

The idea is that this construct and send the osc message

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

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

⁸<http://opensoundcontrol.org/>

```

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

```

can be used to invoke later

```

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

```

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

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

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

6.4.2 OSCRECV

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

```

oscrecv name port msg_prefix $v_1 ... $v_n

```

where:

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

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

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

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

6.4.3 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.

6.5 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 open and bind 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"
...
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 file is automatically closed at *Antescofo* exit. If not explicitly closed, it 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.

6.6 Internal Commands

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

An internal command start with a predefined simple name following the pattern `antescofo::xxx` where the suffix `xxx` is the head of the corresponding MAX/PD message recognized by *Antescofo* (cf. section 11.5) and is one of the following:

- `antescofo::actions` (one argument of type string)
- `antescofo::analysis` (one argument of type int)
- `antescofo::tempo` (one argument of type float)
- `antescofo::before_nextlabel` (no argument)
- `antescofo::bpmtolerance` (one argument of type float)
- `antescofo::calibrate` (three arguments of type int)
- `antescofo::clear` (no argument)
- `antescofo::gamma` (one argument of type float)

- `antescofo::getcues` (no argument)
- `antescofo::getlabels` (no argument)
- `antescofo::gotobeat` (one argument of type float)
- `antescofo::gotocue` (one argument of type string)
- `antescofo::gotolabel` (one argument of type string)
- `antescofo::harmlist` (multiple arguments of type float corresponding to a vector)
- `antescofo::info` (no argument)/WHYinfo a completer
- `antescofo::jumptocue` (one argument of type string)
- `antescofo::jumptolabel` (one argument of type string)
- `antescofo::killall` (no argument)
- `antescofo::mode` (one argument of type int)
- `antescofo::nextaction` (no argument)
- `antescofo::nextevent` (no argument)
- `antescofo::nextfwd` (no argument)
- `antescofo::nextlabel` (no argument)
- `antescofo::nofharm` (one argument of type int)
- `antescofo::normin` (one argument of type float)
- `antescofo::obsexp` (one argument of type float)
- `antescofo::pedalcoeff` (one argument of type float)
- `antescofo::pedaltime` (one argument of type float)
- `antescofo::pedal` (one argument of type int)
- `antescofo::piano` (one argument of type int)
- `antescofo::playfrombeat` (one argument of type float)
- `antescofo::playfrom` (one argument of type string)
- `antescofo::play` (no argument)
- `antescofo::preload` (two argumentst of type string) preloads a score and store it under a name (the second argument) for latter use;
- `antescofo::preventzigzag` (one argument of type string)
- `antescofo::previousevent` (no argument)

- `antescofo::previouslabel` (no argument)
- `antescofo::printfwd` (no argument)
- `antescofo::printscore` (no argument)
- `antescofo::read` (one argument of type string) loads the corresponding *Antescofo* score;
- `antescofo::report` (no argument)
- `antescofo::score` (one argument of type string) loads the corresponding *Antescofo* score;
- `antescofo::start` (one argument of type string)
- `antescofo::stop` (no argument)
- `antescofo::suivi` (one argument of type int)
- `antescofo::tempoint` (one argument of type int)
- `antescofo::temposmoothness` (one argument of type float)
- `antescofo::tune` (one argument of type float)
- `antescofo::variance` (one argument of type float)
- `antescofo::verbosity` (one argument of type int)
- `antescofo::verify` (one argument of type 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..

6.7 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).

7 Compound Actions

Compound actions act as containers for others actions. The actions “inside” a container inherits some of the attribute of the container. The cations in this 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 is its immediately enclosing container, if it exists.

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

7.1 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.

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

7.1.1 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.

7.1.2 Attributes of Group and Compound Actions

Synchronization (cf. section 8.1)

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

and error strategies (cf. section 8.2)

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

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

7.1.3 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 `parfor`, reaction 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 killed⁹.

7.1.4 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 6.2.1,
- using a `until` (or a `while`) *logical clause*,
- using a during *temporal clause*.

The `until` Clause. There is a dual of the `until` keyword:

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

is equivalent to

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

The during Clause.

7.2 Conditional Actions: `If`

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

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

⁹It is possible to kill a specific instance using the `exec` that refers to this instance. This feature will be implemented soon.

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.

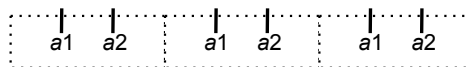
The actions of the “true” (resp. of the “else”) parts of a condition are members of an implicit group named *xxx_true_body* (resp. *xxx_false_body*) where *xxx* is the label of the conditional itself.

7.3 Sequential iterations: Loop

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

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

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



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

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

will print only 3 OK.

7.4 Parallel Iterations: `ParFor`

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

```
$t := tab [1 2 3]
parfor $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 successively

```
OK 3
OK 2
OK 1
```

The general form of a parallel iteration is:

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

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

Parallel iterations accepts also `maps`¹⁰ using two variables to refers to the keys and values in the map:

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

will print:

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

7.5 Sampling parameters: `Curve`

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

¹⁰Interpolated maps are planned.

We introduce the syntax¹¹ starting with the simple case of one curve. Then we detail the complete features of this construction.

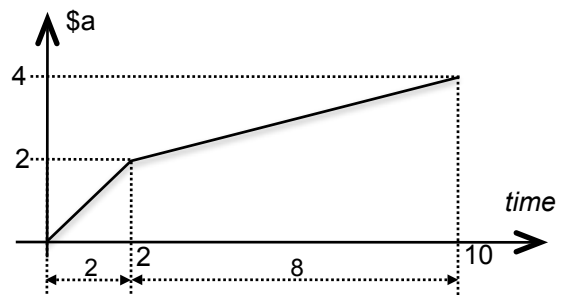
7.5.1 A Simple Curve

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

```

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

```



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

7.5.2 Actions Fired by a Curve

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

```

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

```

at each sampling point the value of `$y` is immediately printed and two beats later `action1` will be fired and one additional beat later `action2` will be fired.

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

¹¹Curve can be edited graphically using the *Ascograph* editor.

7.5.3 Step, Durations and Parameter Specifications

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

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

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

7.5.4 Interpolation Methods

The specification of the interpolation between two breakpoint is optional. By default, a linear interpolation is used. *Antescofo* offers a rich set of interpolation methods:

- : piecewise constant function
- : linear interpolation
- to be completed

Note that the interpolation can be different for each successive pair of breakpoints.

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

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

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

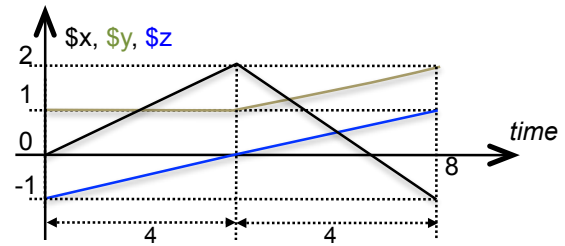
7.5.5 Managing Multiple Curves Simultaneously

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

```

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

```

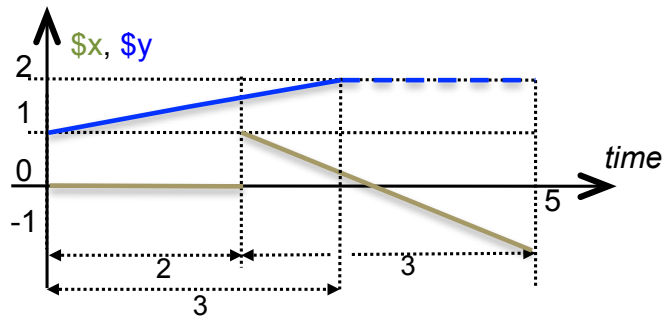


Note that with the previous syntax it is not possible to define simultaneous curves with breakpoint at different time. This is possible using multiple parameter clauses, as in:

```

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

```



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

7.6 Reacting to logical events: `Whenever`

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

```

whenever (boolean_expression1)
{
  actions_list
} until (boolean_expression2)

```

The behavior of this construction is the following: The `whenever` is active from its firing until `boolean_expression2` evaluates to false.. After the firing of the `whenever`, 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. If the condition evaluates to true, the body of the `whenever` is launched.

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

Notice also the difference with a conditional action (section 7.2): 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.

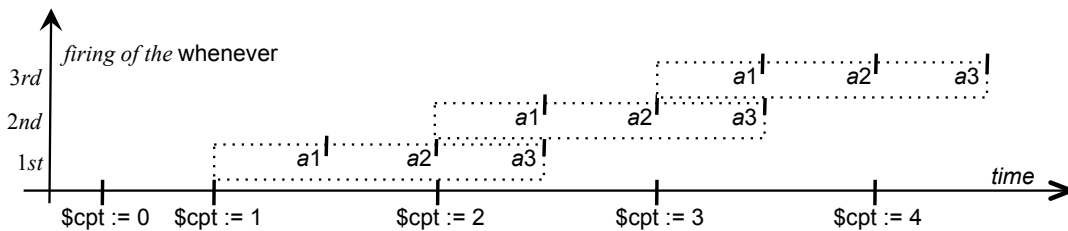
Nota Bene that multiple occurrence of the body of the same **whenever** may be active simultaneously, as shown by the following example:

```

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:



7.6.1 Causal Score and Temporal Shortcuts

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

```

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

```

```

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

```

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

$$\text{Start} \rightarrow W1 \rightarrow W2 \rightarrow W1 \rightarrow W2 \rightarrow W1 \rightarrow W2 \rightarrow W1 \rightarrow W2 \rightarrow W1 \rightarrow \dots$$

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

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

```

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

```

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

```

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

```

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

8 Synchronization and Error Handling Strategies

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

8.1 Synchronization Strategies

8.1.1 Loose Synchronization

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

Figure 6 attempts to illustrate this within a simple example: Figure 6(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 6(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 6(b) and 6(c) where the performer hypothetically accelerates or decelerates the consequent events in her score. In these two cases, the delays between the actions will grow or decrease until converge to the performer tempo.

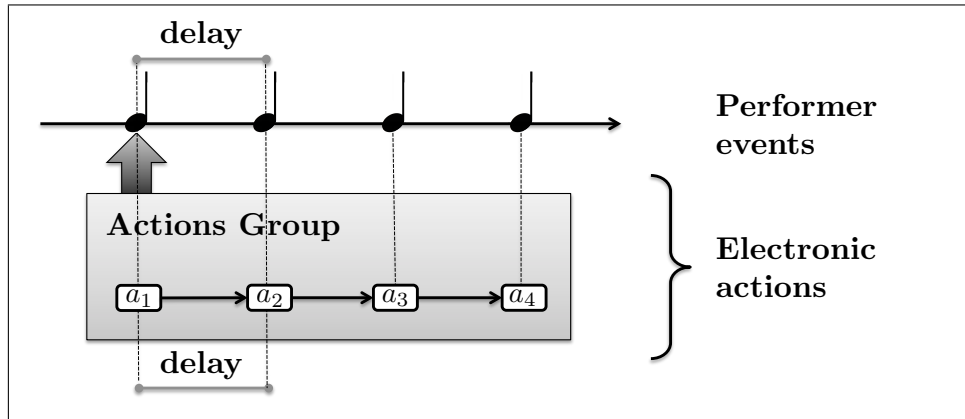
The loose synchronization strategy ensures a fluid evolution of the actions launching but it does not guarantee a precise synchronization with the events played by the musician. Although this fluid behavior is desired in certain musical configurations, there is an alternative synchronization strategy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

8.1.2 Tight Synchronization

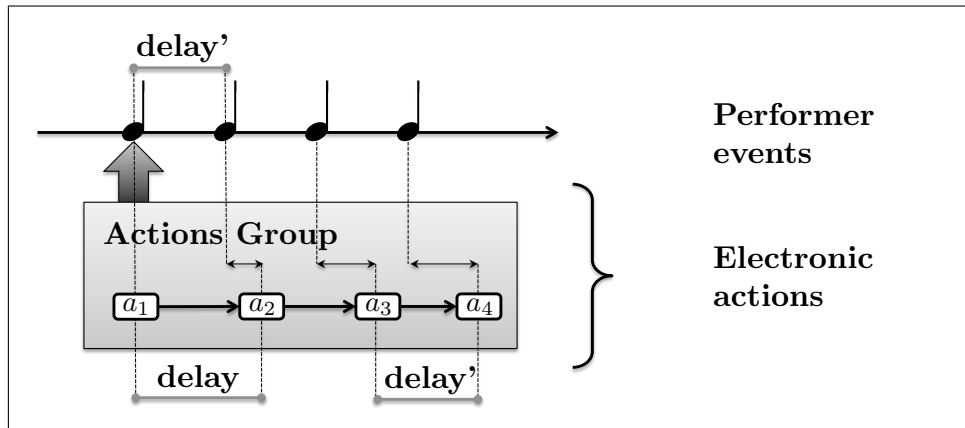
If a group is tight, its actions will be dynamically analyzed to be triggered not only using relative timing but also relative to the nearest event in the past. Here, the nearest event is computed in the ideal timing of the score.

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

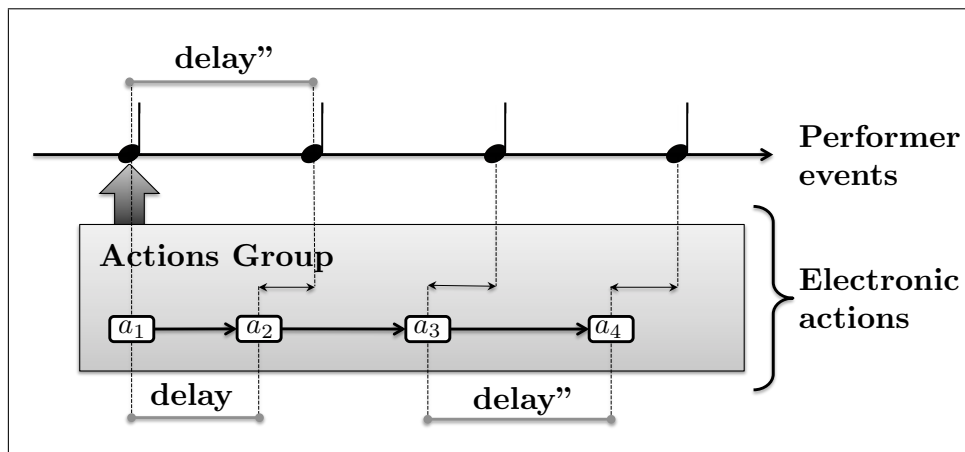
Note that the arbitrary nesting of groups with arbitrary synchronization strategies do not always make sense: a group tight nested in a group loose has no well defined triggering event (because the start of each action in the loose group are supposed to be synchronized



(a) Ideal performance



(b) Faster performance ($\text{delay}' < \text{delay}$)



(c) Slower performance ($\text{delay}'' > \text{delay}$)

Figure 6: *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.

dynamically with the tempo). All other combinations are meaningful. To acknowledge that, groups nested in a loose group, are loose even if it is not enforced by the syntax.

8.2 Missed Event Errors Strategies

Parts but not all of the errors during the performance are handled directly by the listening modules (such as false-alarms and missed events by the performer). The critical safety of the accompaniment part is reduced to handling of missed events (whether missed by the listening module or human performer). In some automatic accompaniment situations, one might want to dismiss associated actions to a missed event if the scope of those actions does not bypass that of the current event at stake. On the contrary, in many live electronic situations such actions might be initializations for future actions to come. It is the 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 7 attempts to showcase these four situations for a simple hypothetical performance setup similar to Figure 6.

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).

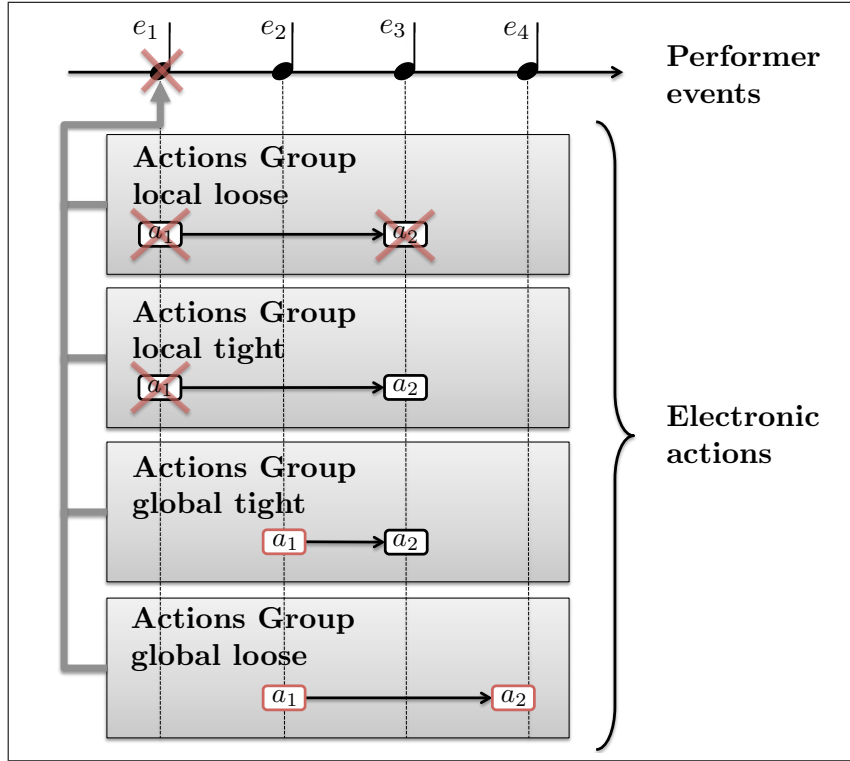


Figure 7: *Accompaniment behavior in case of missed event for four synchronization and error handling strategies.* In this example, the score is assumed to demand for four distinct performer events (e_1 to e_4) with grouped actions whose two actions are initially aligned on e_1 and e_3 . Figure 6 illustrates the system behavior in case e_1 is missed but the rest detected without tempo change during live performance for the four configurations discussed above. Note that e_1 is detected as missed (in real-time) once of course e_2 is reported.

9 Process

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

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

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

The name of the process denotes a proc value, see 5.1.12, and it is used for calling the process. A process call, is similar to a function call: arguments are between parenthesis:

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

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

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

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

A process can also be called in an expression and the instanciacion 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 5.1.13. This value refers to the group lauched by the process instanciacion and is eventually used in the computation of the surrounding expression.

9.1 Macro vs. Processus

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

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

Let give some examples of higher order recursive processes.

9.2 Recursive Process

A infinite loop

```
Loop L 10
{
  ... 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.

9.3 Process as Values

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

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

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

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

10 Macros

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

10.1 Macro Definitions

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

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

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

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

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

is expanded as:

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

Macro have been extended to accept zero argument:

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

10.2 Expansion Sequence

The body of a macro @m can contain calls to others macro, but they will be expanded the expansion of @m.

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

```

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

```

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

```

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

```



```

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

```

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

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

```

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

```

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

```
@Gen(one, 5, print 0k)
```

will expand into

```

GFWD @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 definition,

```

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

```

is replaced by

```

$yyysuffix
3.14159265

```

11 *Antescofo* Workflow

11.1 Editing the Score

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

11.2 Tuning the *Antescofo* Listening Machine

11.3 Debugging an *Antescofo* Score

11.4 Dealing with Errors

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

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

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

11.4.1 Monitoring with Notability

11.4.2 Monitoring with *Ascograph*

11.4.3 Tracing an *Antescofo* Score

There 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 visualized using the `printfwd` internal command. This command opens a text editor.

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 or 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 sent 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 `$v`, it is enough to add a corresponding `whenever` at the beginning of the scope that defines `$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).

11.5 Interacting with MAX

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

11.5.1 Inlets

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

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

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

11.5.2 Outlets

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

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

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

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

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

11.5.3 Predefined Messages

The *Antescofo* object accepts predefined message sent to antescofo-mess1. These messages corresponds to the internal commands described in section 6.6.

11.6 Interacting with PureData

11.7 *Antescofo* Standalone Offline

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

The help of the command line is given in Fig. 8.

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:

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

Reactive module options:

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

Others options

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

Figure 8: Help of the standalone offline command line.

11.8 Old Syntax

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

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

where:

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

12 Stay Tuned

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

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

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

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

A Changes in 0.51

With respect to the previous official release:

- the listening machine has been improved in many ways;
- new syntax for groups and loops;
- delays and tempi can be defined by an expression;
- expressions have been extended with conditionals, global and local variables, histories, recursive functions, etc.;
- the types of values managed by *Antescofo* has been greatly extended to include: boolean, integer, float, string, symbol, function, process, vector, dictionary (map) and interpolated function;
- extended and new atomic actions (assert, abort, erase of an action in a group, process call, assignment);
- new compound actions have been introduced:
 - curve,
 - whenever,
 - conditionals,
 - processes,
 - parallel iterations;
- the new synchronization strategies (loose and tight) and the new error handling strategies (local or global) can be used uniformly on each compound actions;
- improved error messages including the location in the score file;
- management of OSC communications;
- output to a file;
- trace, communication with Notability, integration with Ascograph;
- internalization of Max or PD messages as *Antescofo* atomic actions;
- standalone (offline) and 64 bit version (Mac, Linux).

B Detailed Table of Contents

How to use this document	3
Brief history of Antescofo	3
1 Understanding <i>Antescofo</i> scores	4
1.1 Structure of an <i>Antescofo</i> Score	4
1.2 Elements of an <i>Antescofo</i> Score	5
Function Definition.	6
Process Definition.	6
Macro Definition.	7
Function, Process and Macro Application.	7
1.3 Identifiers	7
1.3.1 Simple Identifiers	7
1.3.2 @-identifiers	8
1.3.3 \$-identifiers	8
1.3.4 ::-identifiers	8
1.4 Comments and Indentation	8
2 Events	10
2.1 Event Specification	10
2.2 Event Parameters	10
2.3 Events as Containers	11
2.4 Event Attributes	12
3 <i>Antescofo</i> Model of Time	13
3.1 Logical Instant	13
3.2 Time Frame	14
4 Actions in Brief	16
Action Attributes.	16
4.1 Delays	16
Zero Delay.	16
Absolute and Relative Delay.	17
Evaluation of a Delay.	17
Synchronization Strategies.	17
4.2 Label	17
5 Expressions	19
5.1 Values	19
5.1.1 Value Comparison	19
5.1.2 Testing a Value	20

5.1.3	Undefined Value	20
5.1.4	Boolean Value	20
5.1.5	Integer Value	20
5.1.6	Float Value	20
5.1.7	String Value	21
5.1.8	Intentional Functions	21
5.1.9	Map Value	21
	Extensional and Intentional Functions.	22
	Domain, Range and Predicates.	22
	Constructing Maps.	22
	Extension of Arithmetic Operators.	23
	Maps as Lists and Vectors.	23
	Maps transformations.	23
	Score reflected in a Map.	24
	History reflected in a map.	24
5.1.10	InterpolatedMap Value	25
5.1.11	Tab Value	25
	Tab function	25
5.1.12	Proc Value	25
5.1.13	Exec Value	25
5.2	Variables	25
5.2.1	Historicized Variables	26
	History reflected in a map.	26
5.2.2	Variables Declaration	27
	Local Variables.	27
	History Length of a Variable.	27
	Lifetime of a Variable.	28
5.2.3	System Variables	28
5.2.4	Variables and Notifications	28
	Temporal Shortcuts.	29
5.2.5	Dates functions	29
5.3	Operators and Predefined Functions	29
	Conditionnal Expression.	29
	@size.	29
5.4	Auto-Delimited Expressions in Actions	30
6	Atomic Actions	31
6.1	Assignments	31
6.2	Aborting and Cancelling an Action	32
6.2.1	Abort of an Action	32
	Abort and the hierarchical structure of compound actions.	32
6.2.2	Cancelling an Action	33

6.3	Max Messages	33
6.4	OSC Messages	34
6.4.1	<code>OSCSEND</code>	34
6.4.2	<code>OSCRECV</code>	35
6.4.3	<code>OSCON</code> and <code>OSCOFF</code>	36
6.5	I/O in a File	36
6.6	Internal Commands	36
6.7	Assertion <code>assert</code>	38
7	Compound Actions	39
7.1	<code>Group</code>	39
7.1.1	Local Tempo.	39
7.1.2	Attributes of <code>Group</code> and Compound Actions	39
7.1.3	Instances of a <code>Group</code>	40
7.1.4	Aborting a <code>group</code>	40
	The <code>until</code> Clause.	40
	The <code>during</code> Clause.	40
7.2	Conditional Actions: <code>If</code>	40
7.3	Sequential iterations: <code>Loop</code>	41
	Stopping a <code>Loop</code>	41
7.4	Parallel Iterations: <code>ParFor</code>	42
7.5	Sampling parameters: <code>Curve</code>	42
7.5.1	A Simple Curve	43
7.5.2	Actions Fired by a <code>Curve</code>	43
7.5.3	Step, Durations and Parameter Specifications	44
7.5.4	Interpolation Methods	44
7.5.5	Managing Multiple Curves Simultaneously	44
7.6	Reacting to logical events: <code>Whenever</code>	45
7.6.1	Causal Score and Temporal Shortcuts	46
	Automatic Temporal Shortcut Detection.	47
8	Synchronization and Error Handling Strategies	48
8.1	Synchronization Strategies	48
8.1.1	Loose Synchronization	48
8.1.2	Tight Synchronization	48
8.2	Missed Event Errors Strategies	50
	Combining Synchronization and Error Handling.	50
9	Process	52
9.1	Macro <i>vs.</i> Processus	52
9.2	Recursive Process	53
9.3	Process as Values	53
10	Macros	55

10.1	Macro Definitions	55
10.2	Expansion Sequence	55
10.3	Generating New Names	56
11	<i>Antescofo</i> Workflow	58
11.1	Editing the Score	58
11.2	Tuning the <i>Antescofo</i> Listening Machine	58
11.3	Debugging an <i>Antescofo</i> Score	58
11.4	Dealing with Errors	58
11.4.1	Monotoring with Notability	58
11.4.2	Monotoring with <i>Ascograph</i>	58
11.4.3	Tracing an <i>Antescofo</i> Score	58
	Printing the Parsed File.	58
	Verbosity.	58
	The TRACE Outlet.	59
	Tracing the Updates of a Variable.	59
11.5	Interacting with MAX	59
11.5.1	Inlets	59
11.5.2	Outlets	60
11.5.3	Predefined Messages	60
11.6	Interacting with PureData	60
11.7	<i>Antescofo</i> Standalone Offline	60
11.8	Old Syntax	62
12	Stay Tuned	63
A	Changes in 0.51	65
B	Detailed Table of Contents	66