

Antescofo

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

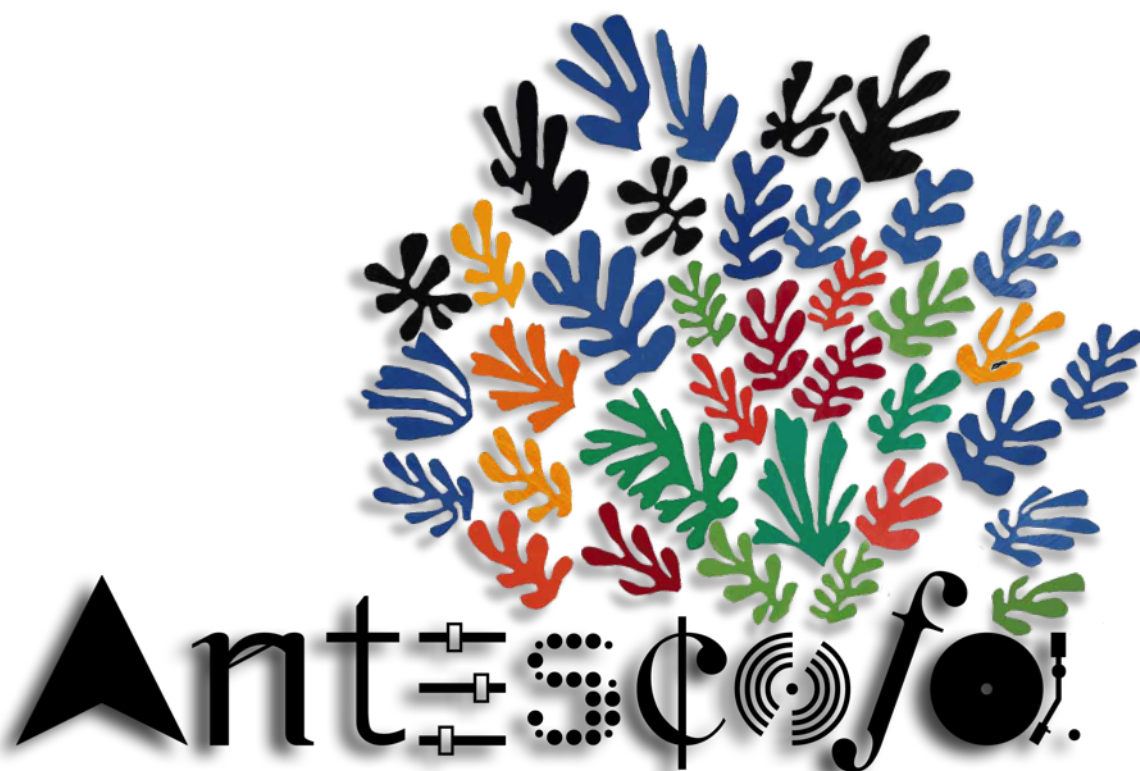
Jean-Louis Giavitto Arshia Cont José Echeveste Julia Bondeau
and MuTAnt team members

August 2016

Contents

List of Tables

List of Figures



Antescofo is a coupling of a real-time listening machine with a reactive and timed synchronous language. The language is used for authoring of music pieces involving live musicians and computer processes, and the real-time system assures its *correct* performance and synchronization despite listening or performance errors.

These pages documents the language starting from version *0.5*. Users willing to practice the language are strongly invited to download Antescofo and use the additional Max or PureData tutorials (with example programs) that come with them for a sensible illustration of the language.

Antescofo scores can be edited, visualized and monitored by **AscoGraph**, a standalone program communicating with the Antescofo engine via OSC messages. See the related [Ascograph](#) documentation.

The documentation

- starts with a [User Guide](#) including the description of the [Antescofo Workflow](#)

- proceeds with a [Reference Manual](#)
- and is completed by the description of [Library Functions](#) and an [Antescofo How-To](#).

Accessing the information

These documents are available as [online web pages](#). The set of web pages for local use (when there is no internet connection) can be downloaded as a zipped file [AntescofoDocHTML.zip](#) (it unzips into a directory named *AntescofoDocHTML* and the file *index.html* in it can be opened in a browser). This file is also bundled with the *Antescofo package*. A [PDF version](#) of the above documentation is available but the layout, the handling of figures and the cross referencing is not satisfactory. The old documentation in PDF form is [still available](#).

Besides these documents, additional information on Antescofo can be found:

- on the [Project home page](#)
- on IRCAM's [ForumNet](#) where you can find tutorials to download with bundles for MAX and PureData
- on the web site of the [MuTant team-project](#) where you can find the scientific and technical publications on Antescofo
- on [Project Development Forge](#) where you can post a bug report

Your feedback is important

Please, send your comments, typos, bugs reports, suggestions, [use cases](#), [hints](#), and [tips](#) on the Antescofo [ForumUser](#) web pages. It will help us to improve the documentation and the *Antescofo* system.

Antescofo: A First User Guide

The *Antescofo* system couples machine listening and a specific programming language for compositional and performative purposes. It allows real-time synchronization of human musicians with computers during live performance.

This [User Guide](#) gives a bird's eye view of the *Antescofo* system:

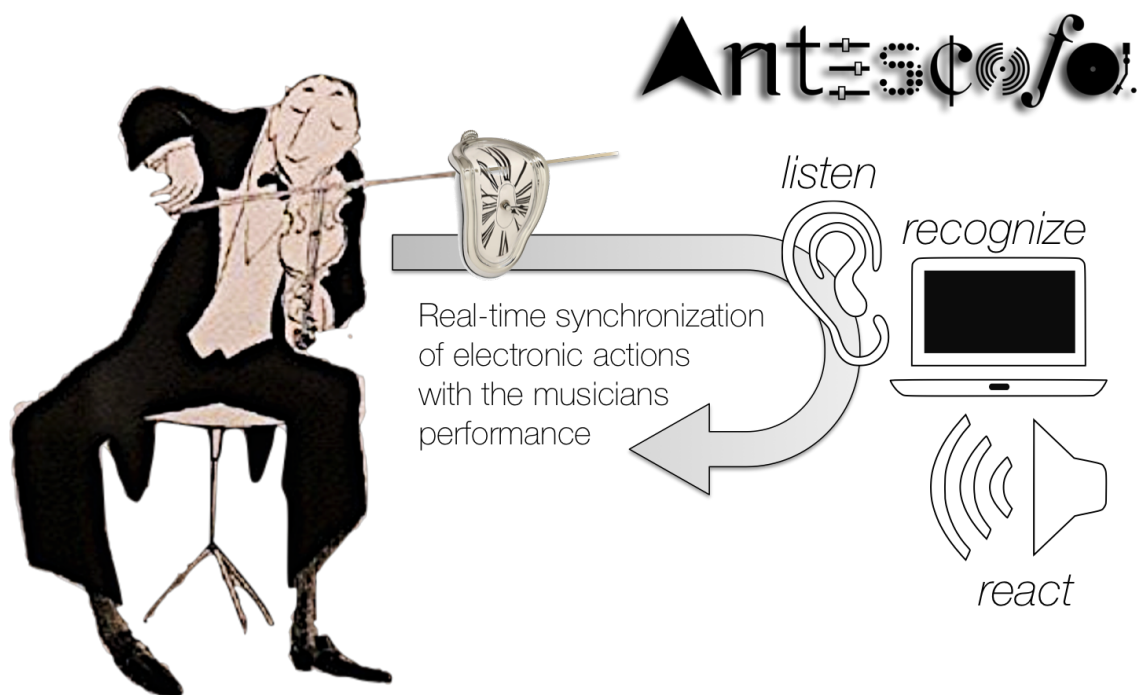


Figure 1: Antescofo principe

Introduction		
a brief introduction on Interactive Music Systems and <i>score following</i> (cf. below)	a presentation of the Structure of an Augmented Score in <i>Antescofo</i> which specifies the musical events that will be recognized in the audio stream together with the actions to trigger in time	a short introduction on Events and Actions , the basic elements of an augmented score

Overview

an overview of <i>Antescofo</i> Features	a digression on <i>Antescofo</i> Model of Time which is at
---	---

Workflow		
Authoring the Score	How to Interact with the Environment	and Preparing the Performance by tuning the listening machine, testing and debugging the system during rehearsals up to the final performance

Beyond score following || :—|:—- *Antescofo* is not limited to score following and been [used as an expressive programmable sequencer](#) dealing with multiple timelines, in interactive installations, for open and dynamic scores, *etc.* [Experience Yourself](#)

Additional information is available elsewhere: The [Reference Manual](#) offers a more detailed presentation of *Antescofo* features. The [Library Functions](#) list all predefined functions in the *Antescofo* library. The [Antescofo distribution](#) comes with several tutorial patches for [Max](#) or [PD](#) as well as the augmented score of actual pieces. The [ForumUser](#) is also a valuable source of information.

Interactive Music Systems

Mixed music (aka. *interactive music*) is the live association of acoustic instruments played by human musicians and electronic processes run on computers. Mixed music pieces feature real-time processes, as diverse as signal processing (sound effects, spatialization), signal synthesis, or message passing to multimedia software.

The specification of such processes and the definition of temporal constraints between musicians and electronics are critical issues in mixed music. They can be achieved through a program that connects music sheets and electronic processes. We call such a program an an **augmented score**.

Indeed, a music score is a key tool for composers *at authoring time* and for musicians *at performance time*. Composers traditionally organize the musical events played by musicians on a virtual time line (expressed in *beats*). These objects share temporal relationships, such as structures of sequences (*e.g.*, bars) or polyphony. To encompass all aspects of a mixed music piece, electronic actions have to share the same virtual time frame of the musical events, denoted in beats, and the same organization in hierarchical and sequential structures.

During live performance, musicians interpret the score with precise and personal timing, where the score time (in beats) is evaluated into the physical time (measurable in seconds). For the same score, different interpretations lead to different temporal deviations, and musician's

actual tempo can vary drastically from the nominal tempo marks. This phenomenon depends on the individual performers and the interpretative context. To be executed in a musical way, electronic processes should follow the temporal deviations of the human performers.

The Antescofo approach: coupling score following with a programming language

Achieving this goal starts by **score following**, a task defined as real-time automatic alignment of the performance (usually through its audio stream) on the music score. However, score following is only the first step toward musician-computer interaction; it enables such interactions but does not give any insight on the nature of the accompaniment and the way it is synchronized.

Antescofo is built on the strong coupling of machine listening and a specific programming language for compositional and performative purposes:

- The Listening module of Antescofo software infers the variability of the performance, through score following and tempo detection algorithms.
- And the Antescofo language
 - provides a generic expressive support for the design of complex musical scenarios between human musicians and computer mediums in real-time interactions
 - makes explicit the composer intentions on how computers and musicians are to perform together (for example should they play in a “call and response” manner, or should the musician takes the leads, *etc.*).

This way, the programmer/composer describes the interactive scenario with an *augmented score*, where musical objects stand next to computer programs, specifying temporal organizations for their live coordination. During each performance, human musicians “implement” the instrumental part of the score, while the system evaluates the electronic part taking into account the information provided by the listening module.

Brief history of Antescofo

The 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 rapidly became a system that couples a simple action language and machine listening.

The language was further used by other composers such as Jonathan Harvey, Philippe Manoury, Emmanuel Nunes and the system has been featured in world-class music concerts with ensembles such as the Los Angeles Philharmonic, New York Philharmonic, Berlin Philharmonic, 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 (whose PhD was on Antescofo *unique synchronization capabilities*) and Philippe Cuvilier (whose PhD was on the use of temporal information in the *listening machine*). The new team *MuTant* was baptized during early 2012 as a joint venture between Ircam, CNRS, Inria and UPMC in Paris.

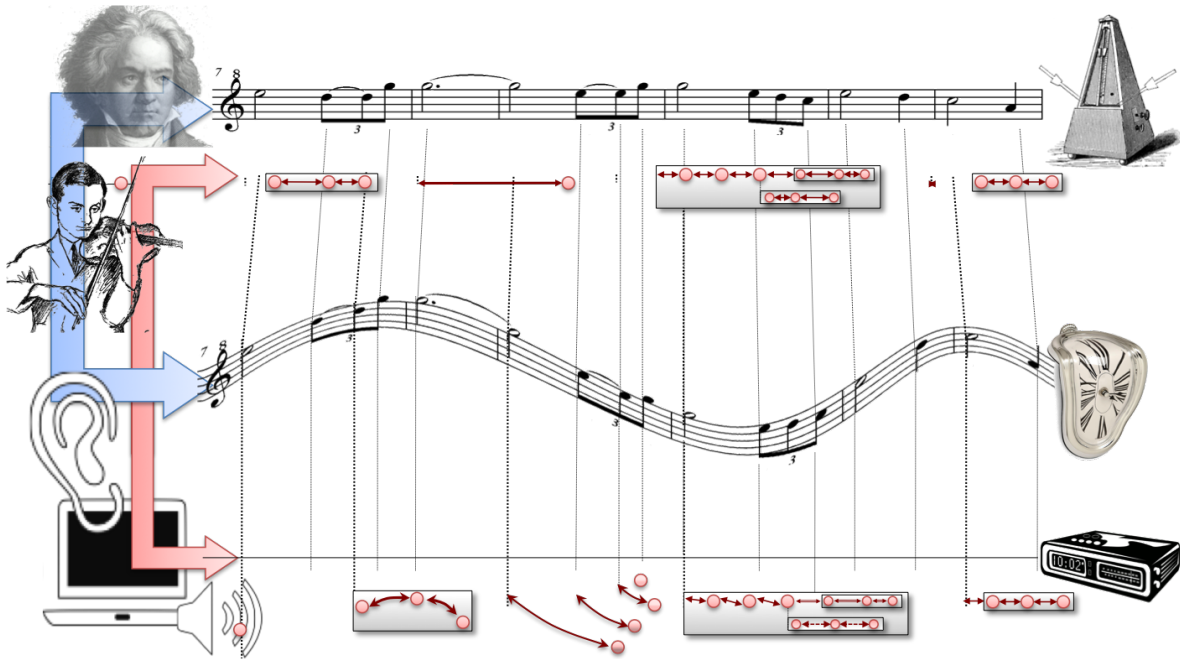


Figure 2: synchro score performance

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

A historical example: “Anthèmes 2” by Pierre Boulez.

Structure of an Antescofo Score

An Antescofo score is a text file that is used for real-time score following (detecting the position and tempo of live musicians in a given score) and triggering electronics as written by the artists.

Antescofo is thus used for computer arts involving live interaction and synchronisation between human and computerised actions.

An Antescofo score can be edited by any text editor. The users can find many syntax highlights on our Antescofo [ForumUser](#). The package for the *Sublime* editor is particularly usefull, as coherence is preserved in real-time between the score loaded in Max and the score edited by *Sublime*.

There is also a dedicated GUI, [AscoGraph](#), that can be used to edit an Antescofo score and to interact with a running Antescofo program. In this video, for example, we can see Antescofo playing with a musician and controlling both the electronic elements and the spatialization. The musician’s score (in a piano roll) and the electronic actions are both visible in the [AscoGraph](#) window.

An interweaving of musical events and electronic actions

An Antescofo score describes both the human actions to be recognized and the machine's reactions to environmental input. A 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 [Event](#).
- **ACTIONS** are elements to be undertaken once corresponding event(s) or conditions have been recognized. Actions in Antescofo extend the good-old *qlist* object elements in MAX and PD with additional features which will be described in this document.

The figure below shows a simple example from the Composer Tutorial on Pierre Boulez' "Anthèmes 2" (1997) for violin and live electronics as seen in Ascograph (open the picture in a new tab for a larger view).

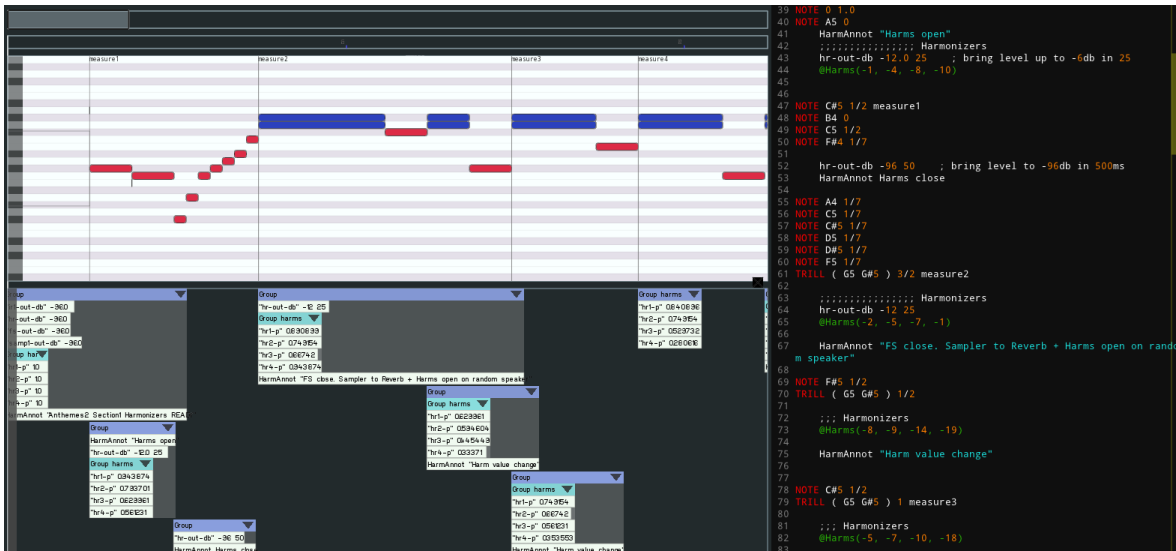


Figure 3: Antescofo Score Excerpt showing basic events and actions

The left window shows a visual representation of Events and Actions, whereas the right segment shows the raw text score. Events in the score describe expected notes, trills and grace notes from the solo Violin, and Actions specify messages to be sent upon recognition of each event. In this example, we showcase actions for four real-time *pitch shifter* (or harmoniser), whose general volume is controlled by the `hr-out-db` parameter, and each shifter parameter separately controlled by `hr1-p` to `hr4-p`. The values for pitch shifters are in pitch-scale factor. Their corresponding musical value is described in the text score as comments (any text following a semi-colon ‘;’ is ignored in the score).

This score shows basic use of actions and events. Red text in the text-editor correspond to reserved keyword for Events. For details regarding available events and their corresponding syntax, see section [Event](#). In this example, actions are basic message-passing to receivers in Max or Pd environments but with OSC, Antescofo can interact with many other softwares like CSound, Supercollider... Since they are isolated and discrete actions, we refer to them

as **Atomic Actions**. As will be shown later, actions in Antescofo can use delays expressed in various time formats, and further include dynamic (i.e. real-time evaluated) expressions, data-structures and more. Details on action structures are discussed in section [Actions](#).

The next figure shows a slightly more complex score corresponding to the beginning of “Tesla ou l’effet d’étrangeté” (2014) by composer Julia Blondeau for viola and live electronics as seen in AscoGraph. The graphical representation on the left is a visual interpretation of the text score on the right. For easier viewing, messages are represented here by white circles (AscoGraph option).

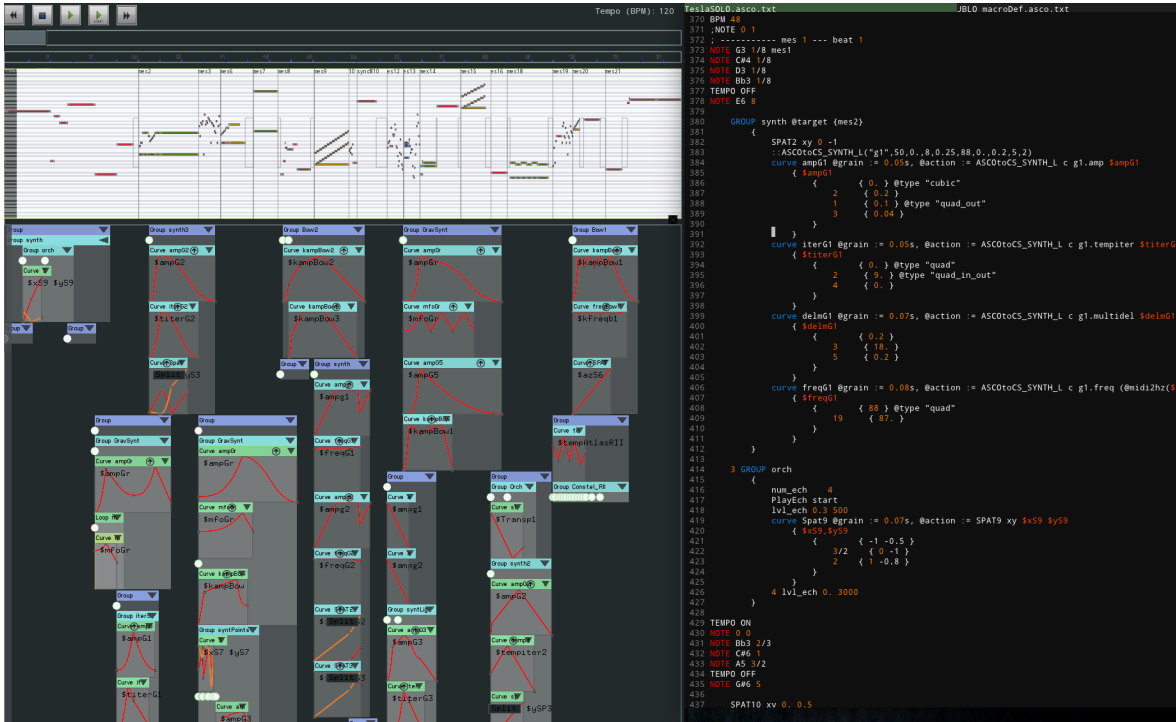


Figure 4: The beginning of *Tesla* (2014) by Julia Blondeau for Viola and Live electronics in *AscoGraph*

In the previous screenshot, the score for human musician contains many types of events (not all visible in text) with a mixture of discrete and continuous compound actions as written by the composer. At the moment you have to know that in Antescofo it exists 5 kinds of events :

- NOTE : in red in the piano roll
- CHORD : in green in the piano roll
- TRILL for classical trill or tremolo : in blue in the piano roll
- MULTI for glissandi (with one or more notes) : in yellow in the piano roll
- EVENT for symbolic events

Tesla makes use of **Compound Actions** which consist of parallel groupings of **Atomic Actions**, as well as continuous actions introduced by keyword *Curve*. These elements correspond to different electronic actions. We can therefore compose the electronic part in parallel with the instrument part, together in an “augmented score”. Effects, synthesis,

spatialization and others can be controlled by Antescofo. Functions, Processes, Actors (objects) and Macros can be used to abstract and reuse these controls.

We will see that with Antescofo, we can compose with many different kind of time. Sequential time for the atomic actions, continuous time for curves, cyclic time for loop or “composed time” with `processus` that will see in due course.

The file structure of an Antescofo augmented score

A textual score, or program, is written in a single file and loaded from there. The file itself can optionally include pointers to other score files, using the `@insert` feature:

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

The keyword can be capitalized: `@INSERT` as any other predefined `@`-identifier (*i.e.*, identifiers that start with a `@` character). An included file may includes other files. The `@insert` command is often used to store definitions and initialisation of the main Antescofo score. It will automatically create additional tabs in Ascograph text editor.

The `@insert_once` command is similar to `@insert` except that the file is included only once in the current score, when the directive is encountered the first time. The behavior makes possible to include a library of primitives in a set of files without the burden of taking care of their dependencies.

For the rest of this chapter, we will briefly introduce main elements in the language. Details will be left for dedicated chapters in the [reference manual](#).

Elements of an Antescofo Score

An *Antescofo* program is a sequence of *events* and *actions*. Events, recognized by the listening machine, are described in detail in chapter [Events](#). Actions, outlined in chapter [Actions](#) are computations triggered upon the occurrence of an event or of another action. Actions can be dynamically parameterized by *expressions* and data structures, evaluated in real-time and described in detail in the [Reference Manual](#).

Elements of the language can be categorized into six groups that correspond to various constructions permitted in the language:

- **Comments:** Any text starting by a semi-colon `;` or `//` is considered a comment and ignored by parser until the end of line (inline comment).
Block (multi-line) C-Style comments starting with `/*` and ending with `*/` are also allowed.
- **Keywords:** are reserved words that introduce either Event or Action constructions. Examples include `Note` (for events) and `Group` (for compound actions).
- **Simple identifiers:** denote Max or PD receivers and are also used to specify the label of a musical event or of an action.

- **@-identifiers:** are words that start with @ character. They either introduce a new definition or denote predefined functions, user-defined functions, user-defined macros, action attributes, or event attributes. The following @-identifiers are used to introduce new definitions: @abort @broadcast @fun_def @init @macro_def @obj_def @pattern_def @proc_def @track_def @whenever.
- ****-identifiers**:** are words that start with “ character. They correspond to user-defined variables or parameters in functions, processes, and object or macro definitions.
- **::-identifiers:** words starting with ::, obj::, pattern:: or track:: refer respectively to processes, objects (actors), patterns or tracks.

User defined score elements including macros, processes and functions can only be employed after their definition in the score. We suggest putting them at the beginning of the file or to put them in a separate file using the @insert command. They will be discussed in proceeding chapters.

Simple identifiers: *Antescofo* keywords and reference to the host environment

The Antescofo language comes with a list [Reserved Keywords](#) for defining elementary score structures. Reserved keywords are *case insensitive*. They can be divided in two groups:

- **Event Keywords** including NOTE, CHORD, TRILL and MULTI introduce musical events (see chapter [Event in Antescofo](#)) and are used to describe the music score to be recognised.
- **Action Keywords**, such as GROUP, LOOP and more, specify computations that can be instantaneous ([Atomic actions](#)) or *containers* for other actions that have a duration ([Compound actions](#)).

Here is an example:

```
NOTE 60 1/2
  rcvr1 harm1 60 87 0.5
  rcvr2 ampSy 0.8 2
NOTE 62 1
  rcvr1 harm1 87 78 1.5
  1/2 print HELLO
```

In the example above, rcvr1 harm1 60 87 0.5 and rcvr2 ampSy 0.8 2 are actions that are hooked to event NOTE 60 1/2, and 1/2 print HELLO denotes an action (sending to a receiver print in max/pd) with a delay of half-beat time. General syntax for atomic actions is described in chapter [Atomic Actions](#).

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

The example shows another use of a simple identifiers: rcvr1, rcvr2, harm1 ampSy and print are simple identifiers but are not reserved keywords. Here they refer to a receiver or a

symbol in Max/PD. Simple identifiers that are not reserved keywords are *case sensitive*. In case a score requires the user to employ a reserved keyword inside a message, the user should wrap the keyword in quotes to avoid clash.

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

Example: The figure shows two simple scores. In the left score, the second tempo change to 90 BPM will be effective starting on the event with label `Measure2` and as a consequence, the delay `1/2` for its corresponding action is launched with 90 BPM. On the other hand, in the right score the tempo change will affect the chord following that event onwards and consequently, the action delay of `1/2` beat-time hooked on note `C5` corresponds to a score tempo of 60 BPM.

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

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

@-identifiers: Functions, Macros, and Attributes

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

1. in the processing of a file, some commands directly affect the parsing of this file: the `@insert` command is used to insert another file, and the commands `@uid` and `@lid` are used to generate on-the-fly fresh identifiers;
2. to introduce new definitions (functions, processes, tracks, patterns, etc.);
3. to specify various attributes of an event or an action;
4. to call internal functions that comes with Antescofo language as listed in chapter [Library Functions](#);

5. and to call user-defined functions or macros.

Only ! ? . and _ are allowed as special (non alphanumeric) characters after the @.

Note that in the first three cases, @-identifiers are reserved identifiers and thus are *case insensitive*, that is @tight, @TiGhT and @TIGHT are the same keyword. Reserved @-identifiers are [listed here](#).

Users can define their own functions as shown in chapter [Functions](#). These @-identifiers are *case sensitive*. Predefined functions are [listed here](#).

\$-identifiers : Variables and Parameters

\$-identifiers like \$id or \$id_1 are simple identifiers prefixed with a dollar sign. Only ! ? . and _ are allowed as special characters after the \$. \$-identifier are used to give a name to variables (see section [variables](#) and as parameters for function, process and macro definition arguments. *They are case-sensitive*.

The figure below shows a rewrite of the excerpt of Pierre Boulez’ “Anthèmes 2” [given here](#) using a simple macro and employing basic @ and \$ identifiers. The harmoniser command is here defined as a [macro](#) for convenience and since it is being repeated through the same pattern. The content of the hr1-p to hr4-p actions inside the Macro use a mathematical expression using the internal function @pow to convert semi-tones to pitch-scale factor. As a result the Antescofo score is shorter and musically more readable. Variables passed to the macro definitions are \$-identifiers.

You can learn more on expressions and variables in chapter [expression](#) onwards.

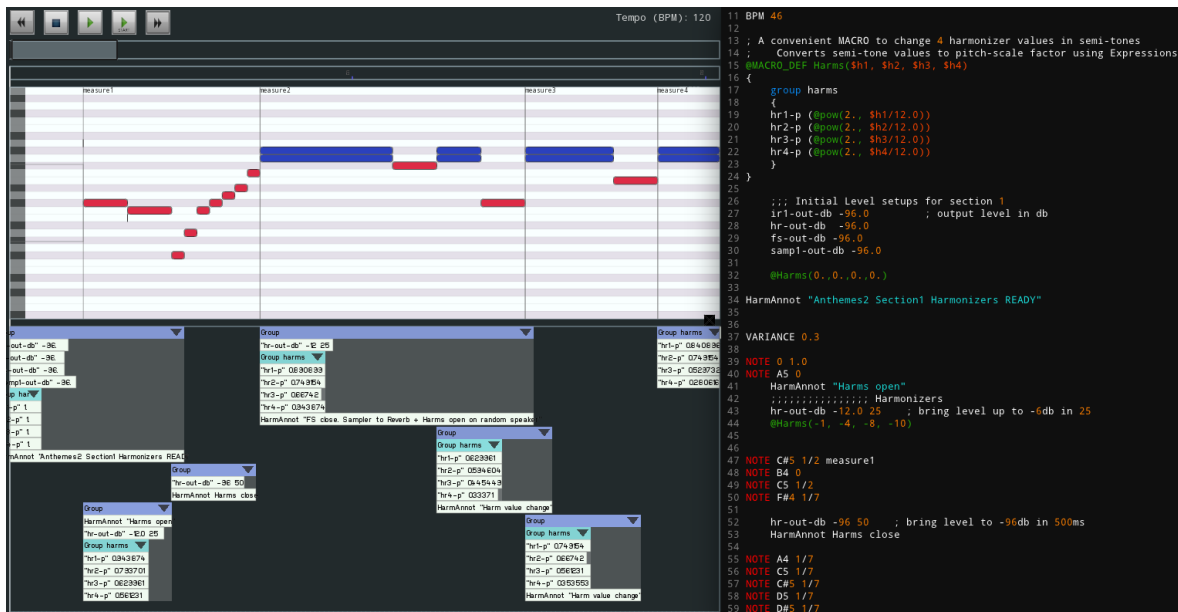


Figure 5: Rewrite of Figure [fig:a2-ex2] using a Macro and expressions

::-identifiers : Processes

::-identifiers like `::P` or `::q1` are simple identifier prefixed with two semi-columns. ::-identifiers are used to give a name to processus (see chapter [Process](#)).

Chapter 1

Events

An event in Antescofo terminology refers to elements that define what will probably happen outside your computers for real-time detection and recognition. In regular usage, they describe the *music score* to be played by the musician to follow. They are used by the listening machine to detect position and tempo of the musician (along other inferred parameters) which are by themselves used by the reactive and scheduling machine of Antescofo to produce synchronized accompaniments.

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

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

Event Specification

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

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

Here the '*' is a metacharacter meaning “zero or more repetitions” of the preceeding construction. Elements between square brackets '[' and ']' are optional but parentheses are literal elements that do appear in the code (they are not metacharacters).

TRILL and MULTI are examples of *compound events* organizing a set of NOTES in time. Thus they can accept one or several *pitch_lists*. *pitch_lists* in TRILL and MULTI are

distinguished by their surrounding parentheses. See the next section for a more musically oriented explanation.

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

There is an additional kind of event

```
EVENT d
```

also followed by a mandatory duration d , which correspond to a fake event triggered manually by the “nextevent” button on the graphical interface or by the “nextevent” message to the antescofo object in MAX/PD.

Parameters for event specification are described below.

Pitch

pitch (used in NOTE) can take the following forms:

- MIDI number (e.g. 69 and 70),
- MIDI cent number (e.g. 6900 and 7000),
- Standard Pitch Name (e.g. A4 and A#4).
- For microtonal notations, one can use either MIDI cent (e.g. 6900) or Pitch Name standard and MIDI cent deviations using '+' or '-' (e.g. NOTE A4+50 and NOTE A#4+50 or NOTE B4-50).

```
CHORD (A4+50 A#4+50 B4-50 Bx4-50 C##4+50 C##4-50)
print OK
```

- a minus sign - may precede the previous specification to specify that the current note is a continuation of a note with the same pitch in the preceding event:

```
CHORD (C4 D5) 1
CHORD (-C4 D3) 1/2
```

Pitch_list

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

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

Duration

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

Label

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

Events as Containers

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

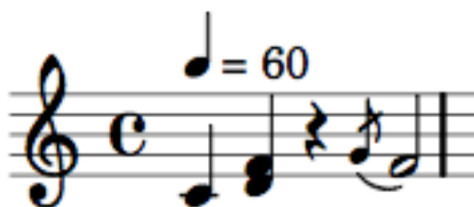


Figure 1.1: chord notation

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

The two additional keywords `TRILL` and `MULTI` also define 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, 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 below shows basic examples for Trill.

```
TRILL (A4 B4) 1.0
NOTE 0 1.0 ; a silence
TRILL ( (C5 E5) (D5 F5) ) 2.0
```

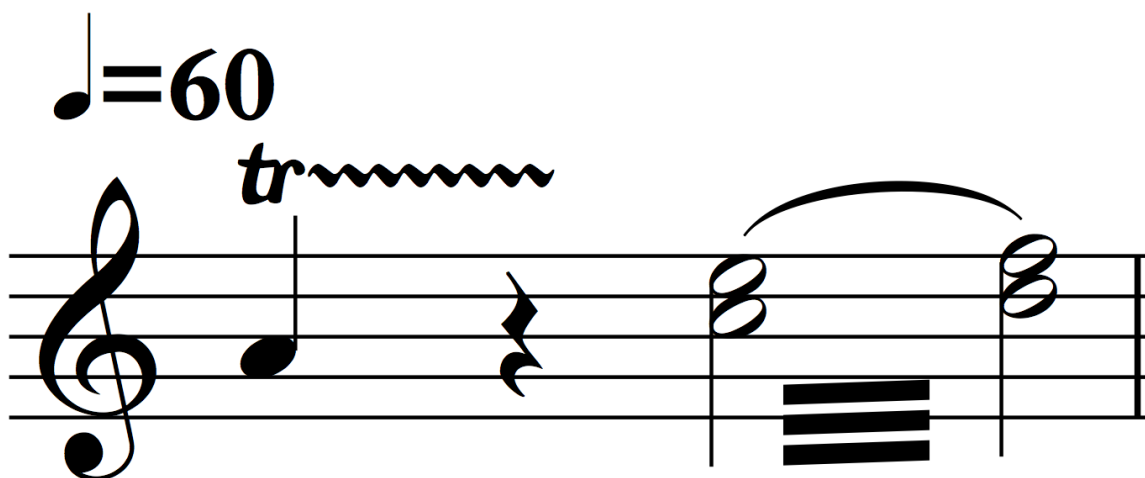


Figure 1.2: trill notation

MULTI

Similar to TRILL, a MULTI is a compound event (that can contain notes, chords or event trills) but where the *order* of actions are to be respected and decoded accordingly in the listening machine. They can model continuous events such as *glissando*. Additionally, a MULTI contents can be trills. To achieve this, it suffices to insert a character after the *pitch_list* closure. The next example shows a glissandi between chords written by MULTI.

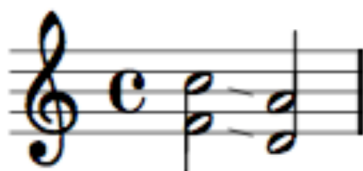


Figure 1.3: gliss notation

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

Compound Events

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

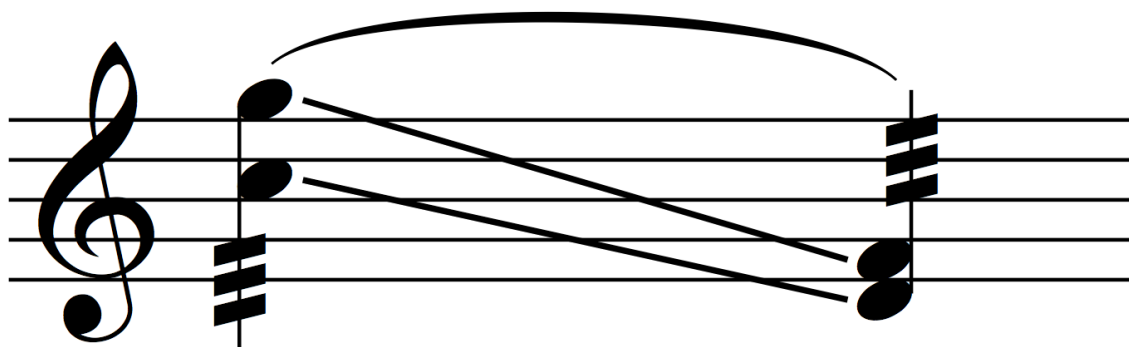


Figure 1.4: notation of a tremolo glissando

```
MULTI ( (C5 G5)' -> (D4 F4)' ) 2.0
```

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



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

Event Attributes

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

- The keyword *fermata* (or @fermata) specifies that this event has a *fermata* signature. A Fermata event can last longer and arriving and leaving it does not contribute to the tempo decoding of the performance.

- The keyword `pizz` (or `@pizz`) specifies that the event is a string *pizzicato*. This usually helps Score Follower stability.
- The keyword `hook` (or `@hook`) specifies that this event cannot be missed (the listening machine needs to wait the occurrence of this event and cannot presume that it can be missed).
- The keyword `jump` (or `@jump`) is followed by a comma separated list of simple identifiers referring to the label of an event in the score. This attribute specifies that the event can be followed by several continuations: the next event in the score, as well as the events listed by the `@jump`.

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

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

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

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

corresponds to 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`.

Event Label

A simple identifier or a string or an integer acts as a label for this event. There can be several such labels. If the label is a simple identifier, its $\$$ -form can be used in a expression elsewhere in the score to denote the time in beat of the onset of the event.

The @modulate Attribute

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

Chapter 2

Actions in Brief

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

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

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

Action Attributes

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

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

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

Some attributes are specific to some kind of actions. There is however one attribute that can be specified for all actions: *label*. It is described in section [Action Label](#). The attributes specific to a given kind of action are described in the section dedicated to this kind of action.

Delays

An optional specification of a *delay* *d* can be given before any action *a*. This 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 C3 2.0
    d1 action1
    d2 action2
NOTE D3 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 [Logical Instant](#).

Absolute and Relative Delay

A delay can be either absolute or relative. An absolute delay is expressed in seconds or milliseconds and refers 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 second after `a0` and `a2` occurs $(2*\$v)$ milliseconds after `a1`. If the qualifier `s` or `ms` is missing, the delay is expressed *in beats* and it is relative to the tempo of the enclosing group (see section [local tempo](#)).

Evaluation of a Delay

In the previous example, the computed value of `a2`'s delay may depend of the date of the computation (for instance, the variable 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 $d1 + d2 - 2.0$ 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 [synchronization strategies](#).

Label

Labels are used to refer to actions. Like events, actions can be labeled with

- a simple identifier,
- a string,
- an integer.

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

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

One action can have several labels. Unlike with event labels, the `$`-identifier associated with an action label cannot be used to refer to the relative position of this action in the score¹.

Compound actions have an optional identifier (section [compound action](#)). This is a simple identifier and acts as a label for the action without the burden to explicitly use the `@name` attribute.

Action Execution

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

- the occurrence of a musical event;
- the occurrence of a logical event (see the `whenever` construction page and the pattern specification at page [Whenever](#));
- the loading of the score (cf. the `@eval_when_load` construct at page [Eval when Load](#));
- the signal spanned by an `@abort` action (see abort handler at page [Abort](#));
- the sampling of a `curve` construct (page [Curve](#));
- the instance of an iterative construct (pages [Loop](#) and [ForAll](#));
- or the expiration of a delay starting with the triggering of another action.

¹The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

Chapter 3

A Brief overview of Antescofo features

This section introduces some features that are very useful, especially for composers. If you are interested to learn more, take a look at the corresponding chapters in the *Antescofo reference*. Remember that everything having to do with control in Max or PD could certainly be replaced by an Antescofo Program. You will see that you can easily manage tabs, lists and some features that you need to control all the parameters of a concert patch.

A useful action : the curve

One of Antescofo's useful Keywords is the curve. Many composers use automations in their sequencers or use some "line" in Max. In Antescofo, you can write a large ensemble of curves (with many interpolation types) and create a score that controls all the effects, spatialization and synthesis with this curves. All the receivers in your Max or PD patch can receive the updated variable controlled by a curve.

The example below shows 3 simple curves with different types of interpolation. A receiver named "print" will receive the variables \$x, \$y and \$z.

Make your life easier with macros !

Frequently, we use a same function that need some parameters in different moment of a piece. So you need to write many lines for a unique effect or synthesis. With a macro, you can denote that some parameters define a single musical entity. We think that is important to can choose between some writing style. Some different ways of thinking need different ways of writing !

The next example shows two ways to write a same thing.

```
; EXAMPLE 1 : the good old Q-list style !
NOTE 60 1
    SPAT_REV 0.2
    SPAT_X 1.
    SPAT_Y 0.8
```

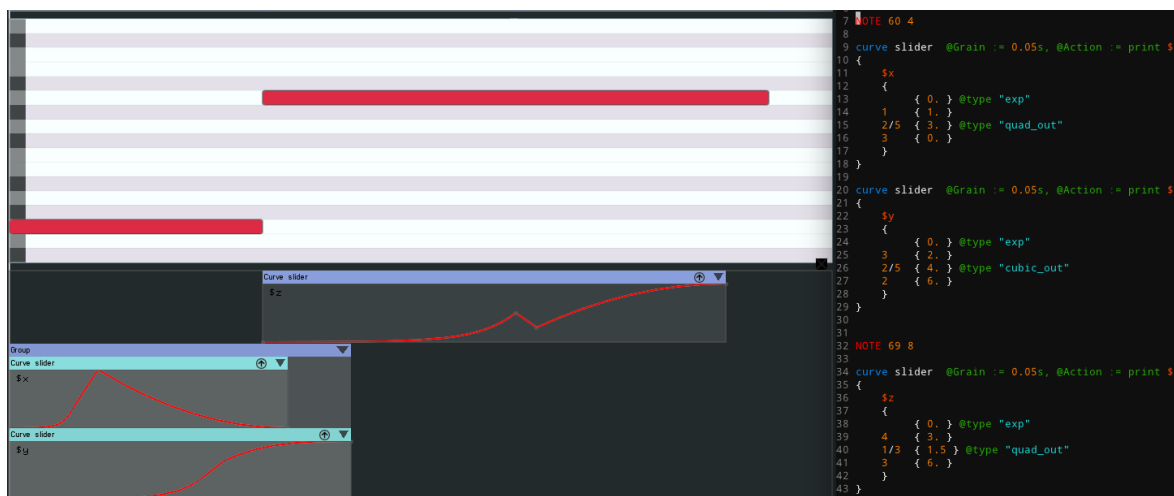


Figure 3.1: An example of curve

```

AddSynt_Hn 8
AddSynt_F0 888
AddSynt_Rev 1.4
AddSynt_Mod 0.1

```

NOTE 69 2

```

SPAT_REV 0.8
SPAT_X 0.
SPAT_Y 1.2

```

```

AddSynt_Hn 6
AddSynt_F0 1857
AddSynt_Rev 1.8
AddSynt_Mod 0.

```

NOTE 63 4

```

SPAT_REV 3.2
SPAT_X 2.
SPAT_Y 0.

```

```

AddSynt_Hn 14
AddSynt_F0 68.4
AddSynt_Rev 2.3
AddSynt_Mod 0.002

```

; EXAMPLE 2 : In another way....

```

@macro_def SPAT ($REV, $X, $Y)
{
    SPAT_REV $REV
    SPAT_X $X

```



```

        SPAT_Y $Y
    }
@macro_def AddSynt ($Hn, $F0, $Rev, $Mod)
{
    AddSynt_Hn $Hn
    AddSynt_F0 $F0
    AddSynt_Rev $Rev
    AddSynt_Mod $Mod
}

```

```

NOTE 60 1
    @SPAT(0.2, 1., 0.8)
    @AddSynt(8, 888, 1.4, 0.1)

```

```

NOTE 69 2
    @SPAT(0.8, 0., 1.2)
    @AddSynt(6, 1857, 1.8, 0.)

```

```

NOTE 63 4
    @SPAT(3.2, 2., 0.)
    @AddSynt(14, 68.4, 2.3, 0.002)

```

Tour the loop

Sometimes, the situation calls for a loop (ask to Steve Reich!). You can use many loops and imbricated loops in Antescofo. For example, you can include a curve in a loop and dynamically modify it while advancing the the loop time. There are many ways to end a loop. Guess what will happen in the following examples:

```

loop ForEver 1
    { print "Try again!" } ; an "infinite" loop...

```

```

3.5 abort ForEver ; ... that you have the power to finish !
    print "That's enough!"

```

```

;-----
$cpt := 0
loop L 1.5
{
    $cpt := $cpt + 1
    0.5 print a1
    0.5 print a2
} until ($cpt >= 3) ; A conditional end

```

```

;-----
; the same with an another type of abort
loop L 1.5

```

```

{
    print a0
    0.5 print a1
    0.5 print a2
} during [4.5]           ; A temporal constraint

```

Build your own world

In *Antescofo*, you can create your own functions (see `@fun_def`) if you frequently need to carry out the same task (to set a diapason, for example). There are many features that help create musical entities and facilitate electronic score writing.

Why do you need data structures...

Data structures come in handy in many situations. Some effects and syntheses involve a long list of parameters. In the *Data Structures* chapter, you will see many ways to create and manipulate different kinds of lists and data structures (see [map](#) and [tab](#)).

In the classical music notation, there are many symbols that each denote an ensemble of parameters. These symbols permit the musician to focus on the music and not on the parameters. You can have the same approach in *Antescofo* if you use macros, processes and data structures. For example, if you use a physical model for synthesis, it's very laborious to enumerate all the parameters in your score. In this case, you can use an ensemble of tabs as a “playing mode” library. So in your score, you will just have to write the name of the tab and not the ensemble of parameters. Like when you write *Sul ponticello* you don't have to describe to the musicians how to play that !

The figure shows a musical score for a string instrument. Above the staff, there are annotations for playing modes: 'ST' (Sul tasto), 'CLT' (Coda), 'CL' (Coda), and 'N' (Normal). A tempo marking '♩ = 72' and 'SP' (Sul ponticello) is also present. The score includes dynamics markings: *p* (piano), *f* (forte), and *pp* (pianissimo). There are also markings for triplets (3) and sixteenth notes (6).

The figure below shows a short library for a string physical model and a process that permits interpolation between two “playing modes”. Don't worry about understanding all the syntax (that's what the [reference manual](#) is for) but remember that it's possible!

```

; Declaration des "modes de jeu" pour interpolation
@global $SulPont,$Ord,$SulTasto,$SulPontVib1,$SulPontVib2,$SulTastoVib1,$SulTastoVib2,$OrdVib1,$OrdVib2

; pression : 0.2 - 5
; position : 0.028 - 0.22

$SulPont := [3.20, 0.032, 3.6, 0.] ; pression, position, freqVib, vibamp
$SulPontVib1 := [3.20, 0.032, 1.3, 0.002]
$SulPontVib2 := [3.20, 0.032, 7.1, 0.003]
$Ord := [2.5, 0.12, 1.3, 0.]
$OrdVib1 := [2.5, 0.12, 1.3, 0.002]
$OrdVib2 := [2.5, 0.12, 7.1, 0.003]
$SulTasto := [4.23,0.2, 1.3, 0. ]
$SulTastoVib1 := [0.9,0.22, 1.3, 0.002]
$SulTastoVib2 := [0.9,0.22, 7.1, 0.003]

; Numero du GenBow, [ModeJeu1,dur1,type1,ModeJeu2, ... , durn,typen,ModeJeun], Grain
; ex ::MdJ_bow("bow1",[$SulPont,1,"cubic",$OrdVib2,2,"linear",$SulTasto] ,0.2)
@proc_def ::MdJ_bow($numBow,$TabMdJ,$grain)
{
  if (@size($TabMdJ)>=4)
  {
    ::M_dj_bow1($numBow,$TabMdJ[0],$TabMdJ[3],$TabMdJ[1],$TabMdJ[2],$grain)
    ($TabMdJ[1]) ::MdJ_bow($numBow,(@cdr(@cdr(@cdr($TabMdJ)))),$grain)
  }
}

@proc_def ::M_dj_bow1($numBow,$MdJ1,$MdJ2,$durInterp,$type,$grain)
{
  @local $interpolMdJB, $x
  ; creation de la courbe entre les valeurs d'interpolations de chaque valeur des 2 tableaux
  $interpolMdJB := [ NIM{ 0 ($MdJ1[$i]), 1 ($MdJ2[$i]) EXPR{ $type } } | $i in @size($MdJ1) ]
  ; "tete de lecture" de la courbe
  curve interp @grain := $grain s,
  @action := { ASC0toCS_SYNTH_L c ($numBow+".kpres") ([ $nim($x) | $nim in $
    interpolMdJB][0] ) ; prend la 1ere valeur du tableau...
    ASC0toCS_SYNTH_L c ($numBow+".kpos") ([ $nim($x) | $nim in $
    interpolMdJB][1] )
    ASC0toCS_SYNTH_L c ($numBow+".kvibf") ([ $nim($x) | $nim in $
    interpolMdJB][2] )
    ASC0toCS_SYNTH_L c ($numBow+".kvibamp") ([ $nim($x) | $nim in $
    interpolMdJB][3] ) }
  { $x
    {
      { 0. }
      $durInterp { 1. }
    }
  }
}
}

```

You can use the same type of program to write a “spatialization” library (see the Stroppa/-Cont Library) where you can write a simple command in the score that will make a complex movement in the space.

In processes we trust

If you have to create complex processes that can heard some extra-parameters of the score (like audio descriptors, patterns...), *Antescofo* provides some dynamic features like the whenever or the processus that permit to write a real musical entity with some musical evolutions.

This process are different, in the way of thinking, of the classical score. The process is a kind of “daemon” that can be launched when it’s needed and that can be aborted at the good moment. The process move in parallel with the score but take account of the musician’s tempo (see \$RT_TEMPO).

In the figure below, a musician can choose a path in an ensemble of short extracts. Different zones are associated at this extracts. In the score, some atomic actions are classically played with the score of musician, in a sequential way. In parallel, a process is launched and evolve depending on area where the musician is. The process can be seeing as an entity that evolve

both with the musician and its own independant evolutions.

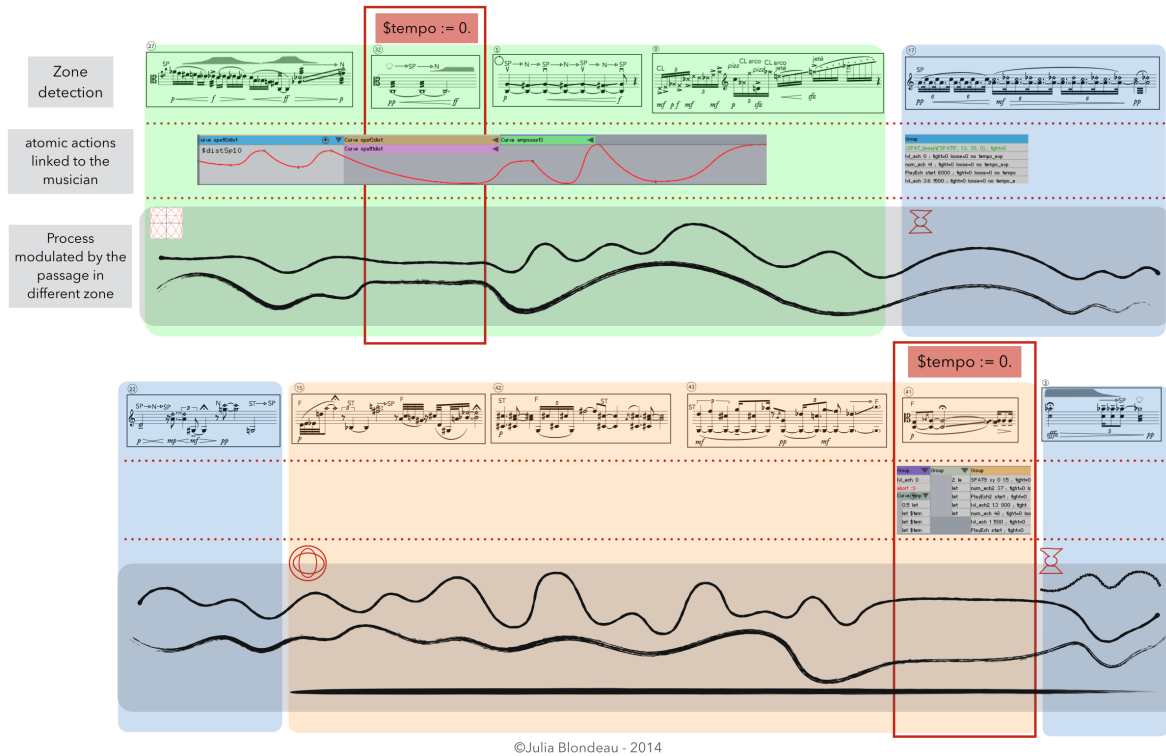


Figure 3.2: Process

Create your own process, macro and function library that you use in all of your pieces.

A conditional world

Sometimes we need to specify conditional actions. In *Antescofo*, the constructions `if` and `switch` are made for that. A conditional action is a compound action that performs different actions depending on whether a programmer-specified boolean condition evaluates to true or false.

You want launch a group of actions only if the musician plays at a particular amplitude? You have to use this kind of code :

```
if ($musAmp >= 1.2)
{
    synt_receiver bang
}
else
{
    print "Hey! You're playing too softly!"
}
```

But, note that this kind of `if` is evaluated when it is launched. So... it is useful but you maybe have to watch at variable during all the time of the performance. In this context, you

need a dynamic construction that look permanently the value of your variable. You need the `whenever` construction! In the same idea of before, if you want to know when your musician is playing too loud, you can write something like this :

```
whenever ($musAmp >= 1.2)
{
    synt_receiver bang
}
```

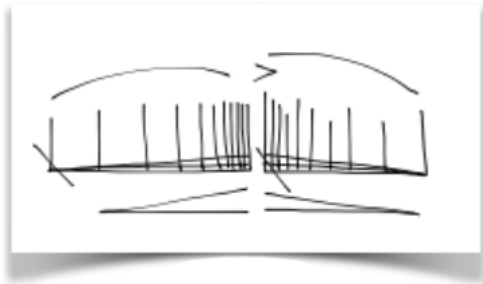
When the `whenever` statement is launched, the variable that it's given is permanently monitored and you will always know *when* the `whenever`'s condition is true.

It's as easy as pie!

Become the time master

In *Antescofo*, all the electronic actions are launched in the musician's time. The internal variable `$RT_TEMPO` give the tempo of the musician in real time. So, when you write your score, you can be sure that it will be synchronize with the musician (if you interested by the synchronization question, take a look at the chapter [Synchronization Strategies](#)).

It's great ! But... , perhaps you would like to impose *your* time ! And, maybe, you had written an electronic phrase that sound too steep and you would like to introduce more softness in the "electronic phrasing". You can want to write an *accelerando*, but write the absolute time values is so laborious. ...



So, in *Antescofo* language, all the group, `process`, `loop`, `curve`... can be "time controlled". This means that for each group, you can impose a tempo (BPM) or better: an evolution of tempo !

The attribute `@tempo` is made for that. If you see the previous example, you can see that for two periods of time, the process is like "time freezed" and you can see `::antescofo $tempo := 0` that is a simple example where you can stop for a moment any instance if you put its tempo at 0.

In the next example, a group `fusee` composed with atomic actions `::ASC0toCS_points`... is time controlled by the variable `$tempfusee`. This variable is modified in time by a curve that give a phrasing to the group.

```

GROUP tempfus @tempo := 80
{
  curve tempfusee @grain := 0.1s
  { $tempfusee
    {
      { 40 } @type "quart"
      4   { 130 } @type "quad"
      1   { 50 }
      3/4 { 80 }
    }
  }
}

GROUP fusee @tempo := $tempfusee
{
  ::SPAT_lissaj2inv("SPAT7",1.4,6,0)
  curve ampexplo @grain := 0.05s
  { $ampexplo
    {
      { 0.09 } @type "quad"
      2   { 0.15 } @type "quad_in_out"
      1   { 0.04 }
      1/2 { 0.17 }
      1/2 { 0.09 }
    }
  }
  ::ASC0toCS_points("i33",13/6,$ampexplo,0.9,76)
  11/6 ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,79)
  1/6  ::ASC0toCS_points("i33",1/8,$ampexplo,0.1,78)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,78.5)
  1/6  ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,77)
  1/6  ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,80)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,80.5)
  1/6  ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,86)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,86.5)
  1/6  ::ASC0toCS_points("i33",1/10,$ampexplo,0.7,91)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,91.75)
  1/8  ::ASC0toCS_points("i33",1/10,$ampexplo,0.7,90)
  1/8  ::ASC0toCS_points("i33",1/10,$ampexplo,0.7,89)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,89.5)
  1/8  ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,88)
  1/8  ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,87)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,86.5)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,86)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,85)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,85.4)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,62)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,70)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,91)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,74)
      ::ASC0toCS_points("i33",1/8,$ampexplo,0.7,74.5)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,85)
  1/8  ::ASC0toCS_points("i11",1/8,$ampexplo,0.7,84)
}

```

Chapter 4

Management of Time

The language developed in *Antescofo* can be seen as a domain specific synchronous and timed reactive language in which the accompaniment actions of a mixed score are specified together with the instrumental part to follow. It thus takes care of timely delivery, coordination and synchronisation of actions with regards to the external environment (musicians) using machine listening.

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

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 **timed system**, where computations are triggered at some date. The three main components of the system architecture are sketched in the figure below:

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

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

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

Logical Instant

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

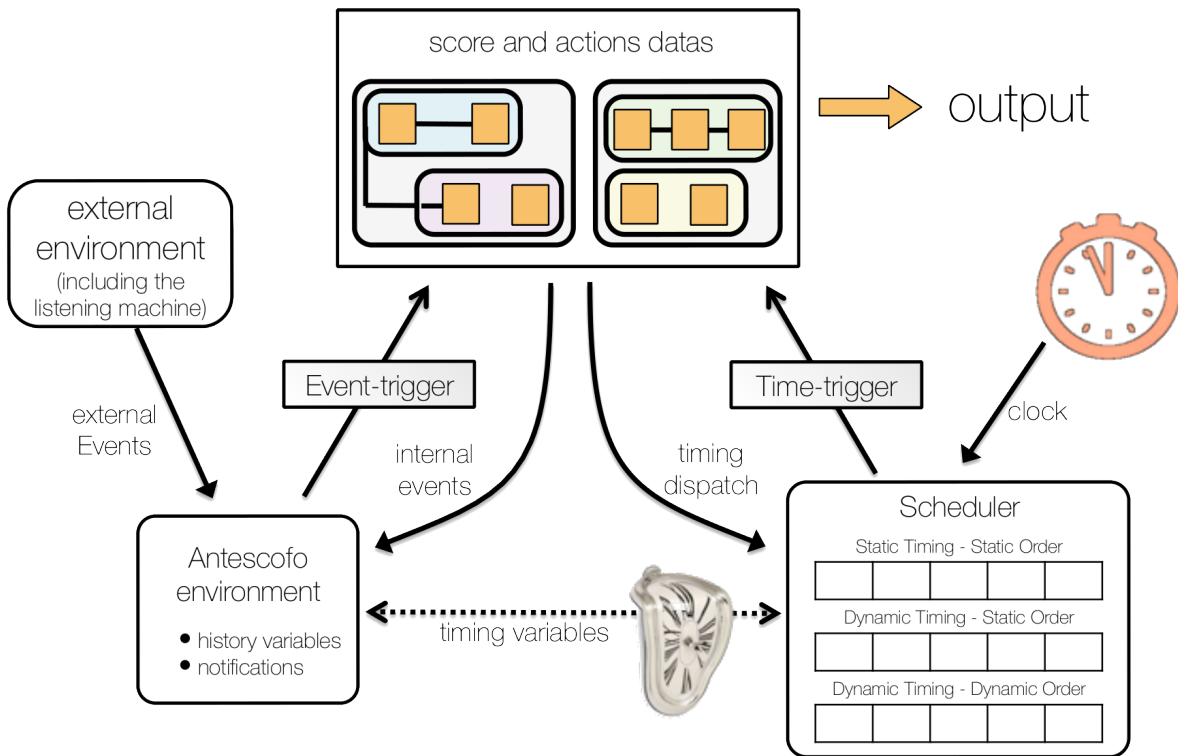


Figure 4.1: antescofo_architecture

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

Such an instant has a date (*i.e.* a coordinate) in each time coordinate system. The notion of logical instants is instrumental to maintaining the synchronous abstraction of actions (*i.e.* the idea that two actions may occur simultaneously) and to reduce temporal approximation. Whenever a logical instant is started, the internal variables \$NOW (current date in the physical time frame) and (current date in the relative time frame) \$RNOW are updated, see section [system variables](#). Within the same logical instant, synchronous actions are performed sequentially in the same order as in the score.

Computations are supposed to take no time and thus, atomic actions are performed inside one logical instant of zero duration. This abstraction is a useful simplification to understand the scheduling of actions in a score. In the real world, computations take time but this time can be usually ignored and does not disturb the scheduling planned at the score level.

In the figure below, the sequence of synchronous actions appears in the vertical axis. So this axis corresponds to the dependency between simultaneous computations. Notice that the (vertical) height of a box is used to represent the logical dependencies while the (horizontal) length of a box represents a duration in time. Note for example that even if durations of a_1 and a_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.

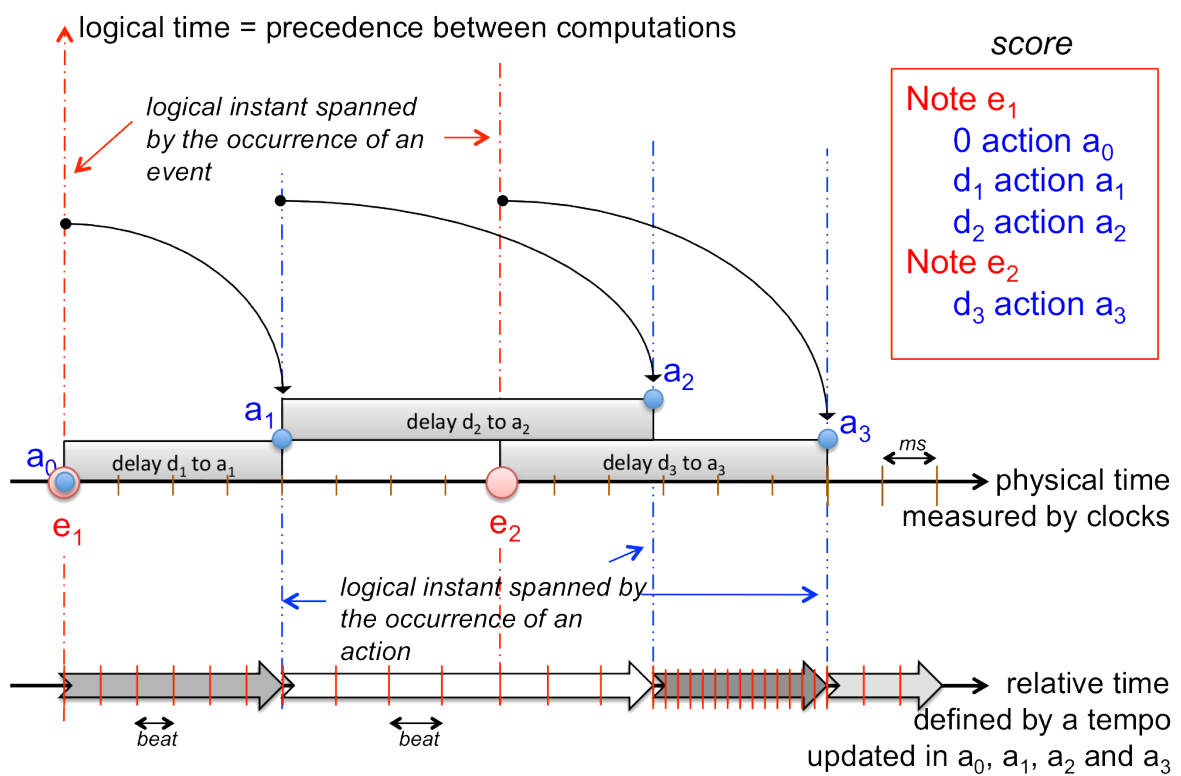


Figure 4.2: logical and physical time

Time Coordinate

A **time coordinate system** is 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 or in a sequence of actions. The position in the score is measured in beats. The relative time is relative to a tempo.

A tempo specifies the “passing of time” relatively to the physical time¹. In short, a tempo is expressed as a number of beats per minutes.

Programmers may introduce their own tempo local to a group of actions using a dedicated attribute `[@tempo]`. The corresponding coordinate systems is then used for all relative delays and datation used in the actions within this group. The tempo expression is evaluated continuously in time for dynamically computing the relationships specified by equation

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

linking 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} .

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 score². This variable is extracted from the audio stream by the listening machine, relying on cognitive model of musician behavior³. The corresponding time coordinate system is used when we speak of “relative time” without additional qualifier.

Locating an Action in Time

Given a time coordinate, there are several ways to implement the specification of the occurrence of an action. For instance, consider action a_2 in the above figure and suppose that $d_1 + d_2$ is greater than 1.5 beat (the duration of the event `NOTE e1`). Then action a_2 can be launched either:

- $(d_1 + d_2)$ beats after the occurrence of the event `NOTE e1`;

¹The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

²[Inbetweening or tweening](#) is the process of generating intermediate frames between two images to give the appearance that the first image evolves smoothly into the second image. The page [Tweeners](#) illustrates the standard [twens](#) to control the successive positions of a point, illustrating the use of tweens to control the apparent speed and to achieve different qualities of movement.

³The equivalent group given here is only an approximation because the grain d is dynamically computed and adjusted so that curve’s action is executed for each breakpoint boundaries (breakpoint’s duration are not necessary a multiple of the grain size).

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

(several other variations are possible).

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

Antescofo allows a composer to choose the right way to compute the date of an action in a time frame, to best match the musical context. This is the purpose of the [Synchronization Strategies](#).

Chapter 5

Antescofo Workflow

In this chapter, we will see how to work with *Antescofo*, from the edition of the score to the interactions with Max, PureData or other softwares through OSC.

Editing the Score

The first step in the development of an *Antescofo* Application is probably the authoring of the *augmented score*. An augmented score is a text program and can be edited with any text editor (like *Sublime*, *TextWrangler* or *Emacs* but *not* with a text processor like *Office/Word*).

However, *Antescofo* comes with a companion application: *AscoGraph*. *Ascograph* is a graphical tool that can be used to edit and then to control a running instance of *Antescofo* through OSC messages.

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

Ascograph is available in the same bundle as *Antescofo* on the IRCAM Forum. It is still far from being stable but can be extremely useful for authoring/visualizing complex scores and for monitoring your live performances. Your feedback is extremely welcome.

Importing Scores to *Antescofo* (import of Midi files and of MusicXML files)

Very often, the musical events to follow have already been specified through a score editor (**Finale**, **Sibelius** . . .), or exist as a MIDI score.

It is possible to automatically import MIDI or MusicXML scores to *Antescofo* format to spare the burden of rewriting the followed part. This feature is available by drag and dropping MIDI or MusicXML files into *AscoGraph*. For multiple instrument score, care should be taken to extract required part in a separate MIDI or MusicXML file. The result is an *Antescofo* score that can then be modified at will.

In the figure below you can see the **Antescofo importer** that appears when you drag your score in *AscoGraph*.

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

Importing MIDI scores to *Antescofo*

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

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

Importing MusicXML scores to *Antescofo*

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

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

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

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

Using *AscoGraph*

As you can see below (open the screenshot in another tab to have a larger view), the Ascograph interface contains 3 parts :

- the piano roll, where you can see the musician's score and the label at the top

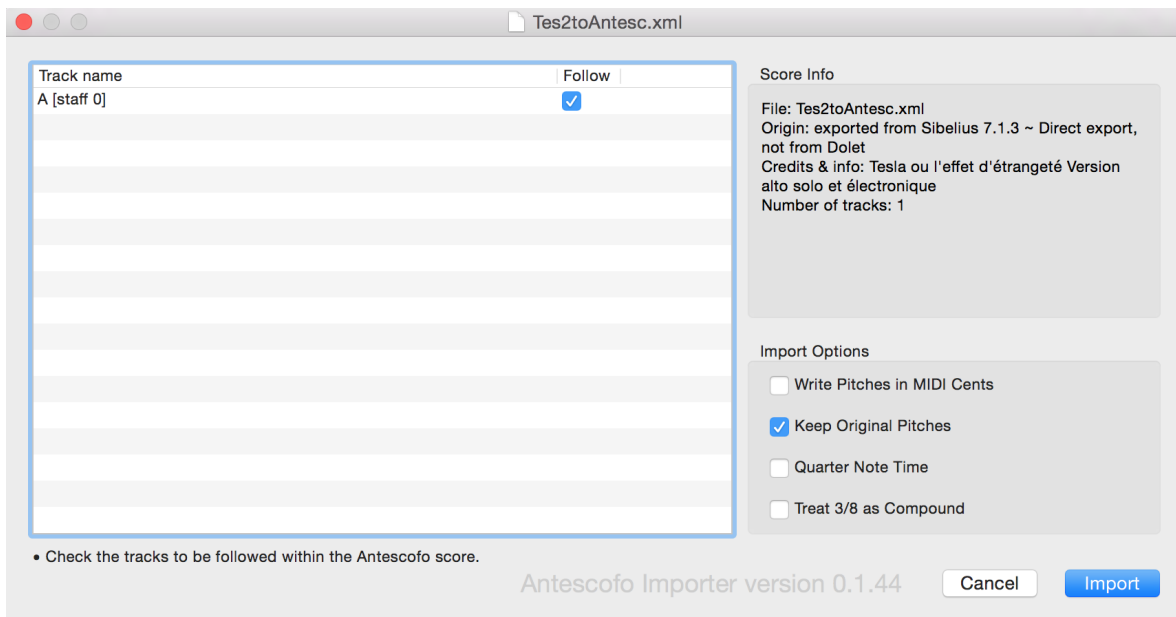


Figure 5.1: Antescofo Importer

- the action view, where you can see the graphical representation of the *electronic score* and where you can manually edit the curves.
- the text editor, where you can edit the musician's score and the electronic score.

At the top of the ascograph window, there are five buttons that correspond respectively to the Antescofo functions `previous`, `event`, `stop`, `play`, `start` and `next` event that you can launch directly in your patch.

The `play` button simply sequences from the beginning of the score to the end, using given event timing and internal tempi, and undertaking actions wherever available.

The `Play string` function (Menu/Transport/Play string) only plays the selected part of your score. This enables to test on the fly Antescofo fragment (only actions are performed, musical events are ignored) and even allows a limited form of live-coding.

The App Menu

The text Editor menu contains all the functions relevant to the text editor. Included is the display mode selector, where you can toggle between integrated, floating (default) and hidden. The View menu retains the functions pertaining to the visual editor.

Color Scheme

AscoGraph comes with a new color scheme for both the text and visual editor. By default, we provide dark backgrounds for both. The philosophy behind this design choice is the fact that many of us use *AscoGraph* during live concerts and bright backgrounds create too much light on your screens which make things annoying for you and your audience who are most of the time in the dark.

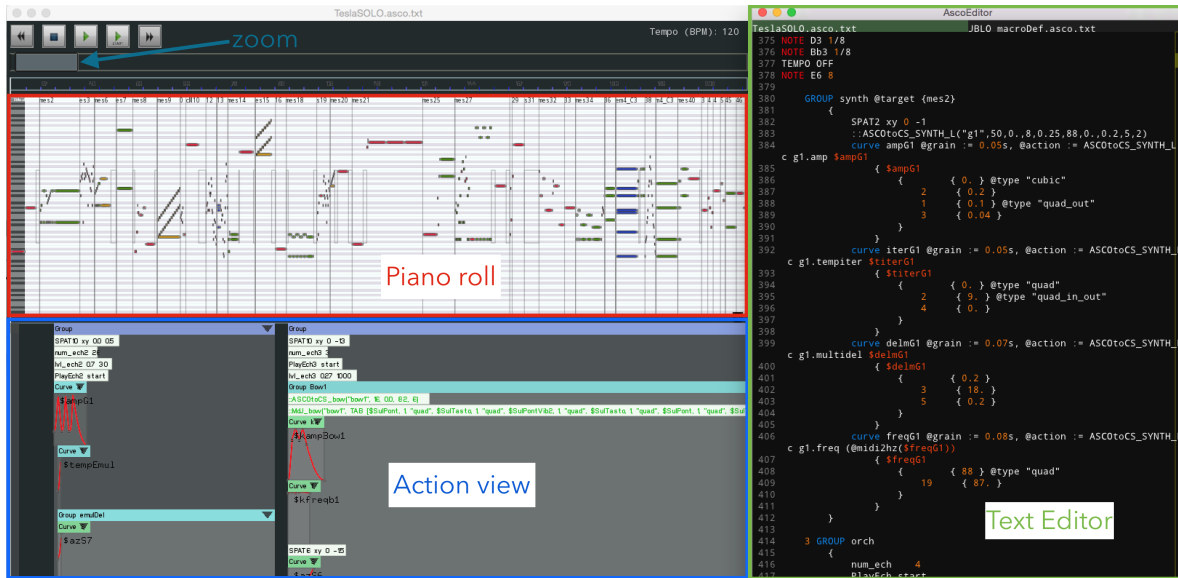


Figure 5.2: AscoGraph Interface

Many users however prefer working on white background while typing. You can easily Import a new Color Scheme for AscoGraph's Text Editor from the Text editor Menu. You can design your own color scheme by doing an Export Color Scheme from the Text Editor menu, modify the XML file and import it back. Users interested in classical white background text editor can import [Larry Nelson's Color Scheme](#) in particular.

Interaction Between Visual and Text Editors in *AscoGraph*

Clicking on a Musical Event or Action in the Visual Editor will bring the text editor to the corresponding text. Conversely, Right/Ctrl-click on an event in the text editor takes you to the corresponding place in the visual editor.

Vertical mouse-wheel (or double-finger on pad) gesture allows you to browse over the text-editor, as well as horizontal mouse-wheel (double-finger) gesture on the visual editor.

Shortcuts

Holding CMD + mouse scroll: zoom in/out (visual & text editors) CMD + <number> switches the text editor between the open files (main .asco score and any @INSERT-ed files). You can also switch using the tabs at the top of the editor.

Edit your curves

AscoGraph is also very useful to visualize and edit the curves constructs.

You can Edit *Antescofo* Curves easily from the Visual Editor. Applying graphical changes will automatically create the corresponding text into the right place. See Nadir B's very useful [YouTube Tutorials](#).

To start, you can create a curve after a note with the menu “create/actions/curve”. After saving the score, if you see the action view, below the piano roll, you can see your curve.

If you click on the arrow at the right of the curve band, the curve is moved on the piano roll, to see the superpositions and you can add or move the curve’s points. You can also choose the type of the interpolation between each points.

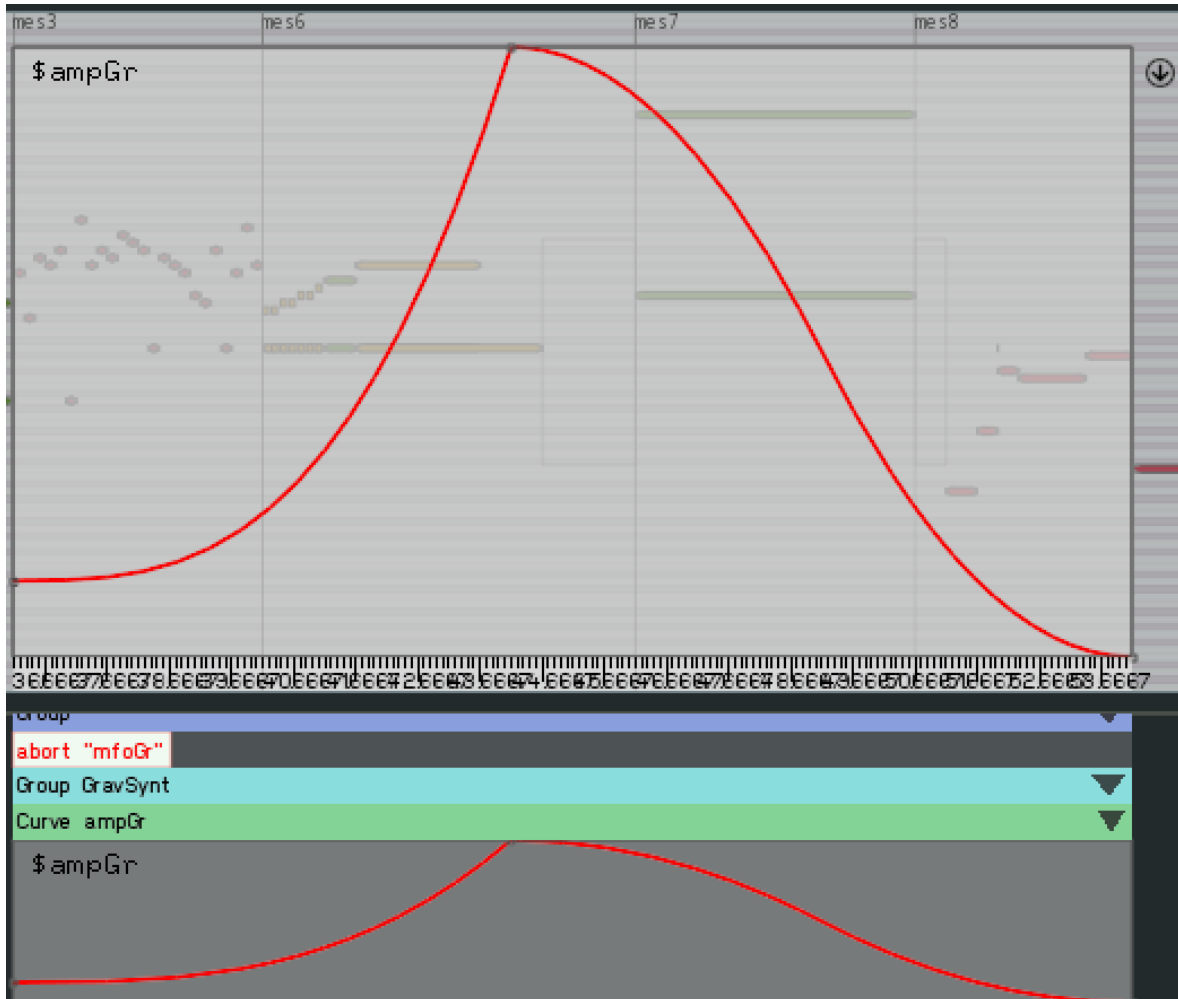


Figure 5.3: curve edition with Ascograph

Automatic Filewatch: Using Another Text Editor

Starting with *AscoGraph version 0.25*, there is an **Automatic Filewatch** integrated into the editor. This means that you can now use any of your favourite text editors instead of *AscoGraph*. The moment you save the Antescofo Score outside, it’ll be automatically loaded (and visualized) in the *AscoGraph* window without any intervention. And the Antescofo object will reload the score automatically !

So you can use in parallel the Piano Roll and action view and still use your favorite text editor.

The addition of the *Automatic Filewatch* feature just makes score editing with AscoGraph more coherent with other editors as they all integrate automatic filewatch as well.

Max users can also use the Antescofo Max Object Autowatch attribute. This means that if the loaded score is modified elsewhere, it'll be automatically reloaded in the Max object. Obviously, you'd want to turn this attribute off during live concerts! Autowatch is currently not available for PureData objects but upcoming.

For the people that prefer use their own beloved text editor, we have created some *syntax highlightings* for **Sublime Text**, **TextWrangler** and... **emacs** ! You have just to [download them in our Forum](#).

Styling your score

Antescofo scores can be pretty big and it is important to grasp the elements at first glance. A syntax highlighter is included in Ascograph and third party packages exist for Sublime and Atom, [look at our Forum](#).

Antescofo scores can also be embedded in markdown (as illustrated by this documentation) using the *Pygment* highlighter or in latex using the *lstlisting* with a dedicated style.

The automatic indentation provided by editors like Sublime for Antescofo score is basic. It is always possible to send a `printfwd` message to the antescofo object. This command will write a file with an automatically indented version of the score together with various additional information (depending on the current verbosity). This file can be loaded again in Antescofo. Notice however that macros are expanded in this score and that the initial comments are not reported.

Interacting with MAX/PureData

When embedded in MAX, the systems appears as an `antescofo~` object that can be used in a patch. This object presents a fixed interface through its inlets and outlets¹.

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 operator `@inlets`. The following hardcoded input types are recognized:

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

¹The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

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.

Outlets

By default, there 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.

Additional (predefined) outlets 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

Predefined Messages

The *Antescofo* object accepts predefined message sent to `antescofo-mess1`. These messages correspond to the internal commands described in section [internal command](#).

The `setvar` command

See section [setvar](#) in the Reference Manual.

Sending and Receiving OSC messages

See section [OSC message](#) in the Reference Manual.

Preparing the Performance, Rehearsals

Controlling *Antescofo* from *AscoGraph*

AscoGraph is very useful in a concert context, when you have to *follow the follower* to control the following part of Antescofo and to control what is launched in the Max patch.

The top of the Ascograph window has five buttons that correspond respectively to the Antescofo function `previous event`, `stop`, `play`, `start` and `next event` that you can launch also directly in your patch.

The `play` button simply sequences from the beginning of the score to the end, using given event timing and internal tempi, performing actions wherever available.

The `Play string` function (Menu/Transport/Play string) plays (evaluates) only the selected part of your score (restricted to a sequence of actions).

When *Antescofo* is in follower mode, and *AscoGraph* is up, the current position inferred by the listening machine is outlined both in the piano roll and in the *AscoGraph* text editor.

Moving in the Augmented Score

There are several commands (Max/PD messages that can be sent to *Antescofo*) that are very useful in rehearsals. The idea is to be able to take over the score deterministically (with correct electronics) if you are rehearsing for example from “Measure10” (a label in score) onwards and assuring that things are prepared (or not) once you start from there as if you have performed everything before. Here is a list of these commands:

Methods with no argument

- `start`: Sends initialization actions (before first event) and waits for follower
- `play`: Simulates the score (instrumental+electronics) from the beginning until the end or until STOP

Methods with label/symbol control

- `startfromlabel label` : Executes the score from current position to position corresponding to \$label in accelerated more WITHOUT sending messages. Waits for follower (or user input) right before this position.
- `scrubtolabel label` : Executes the score from current position to position corresponding to \$label in accelerated more WITH sending messages up to (and not including) \$label. And waits for follower (or user input).
- `playfromlabel label` : Executes the score from current position to position corresponding to \$label in accelerated more WITHOUT sending messages, then PLAYS (simulates) the score from thereon.
- `playtolabel label` : PLAY (simulate) score from current position up to \$label.
- `gotolabel label` : Position yourself on \$label without doing anything else! (backward compatibility)

NOTE: Most above commands would use “current position” and not “from the beginning”. In order to reset, one would precede one of the above with the STOP method.

Methods with float/beat-position control

Same as above, but taking a float value corresponding to cumulative beat-position in score (useful for interaction via notation editors such as AscoGraph and NoteAbility PRO). Namely: startfrombeat, scrubtobeat, playfrombeat, playtobeat, and gotobeat

Non-standard use of score navigation command

Like the others commands understood by *Antescofo*, these navigation messages can be issued from the *Antescofo* score itself. They can be used to develop open pieces, or to alter the flow of the performance on purpose.

Tuning the Listening Machine

In the default audio observation mode, you need to define the analysis parameters which are window size, and hop size in sample numbers. Antescofo automatically buffers audio and does the required analysis. Your choice of window size and hopsize could alter the performance and computation of the system! Default value is [2048 1024]. For example, when following piano music, you might want to set window size to 4096 (for better spectral resolution). If you want more exact timing, reduce the hopsize (and pay more CPU).

In your max/PD Patch, you can access to advanced performance controls through the audio menu.

Calibration

The quality of audio input is of utmost importance for any recognition system and Antescofo is no exception! If the audio input is too low or corrupted, Antescofo can no longer correctly detect events.

Antescofo comes with a built-in Calibration mode to help make sure there is enough audio level for the recognition system to work. You can use it by sending the “calibrate 1” message to Antescofo, start sending audio to Antescofo (no need to “start”) and watch the values sent from the left-most outlet with a prefix “calibration”.

Antescofo calibration values are always between 0.0 and 1.0 and can be interpreted as follows: Whenever there are “events” present in audio, these values should be relatively high (>0.75) and in the contrary (non-events such as silence, hall noise etc.) it should be relatively low (<0.5). It is best practice to test these conditions before any use. If these conditions do not meet, you have to tweak the input level (in Max or audio console) to Antescofo.

It is highly preferable in practice to separate live processing input from that of Antescofo to keep separate level.

Tuning

For pitch and polyphonic detection, Antescofo is tuned by default to 440.Hz (on A4). This can be changed with the message `tune $1`, where `$1` is the new tuning.

Harmonic Analysis Control

When run in polyphonic mode (default mode) Antescofo analyzes realtime audio and matches it to score pitches using a predefined number of harmonics. This can be controlled using the `nofharm $1` message and is 10 by default. Use it if you are confident on what you're doing !

Some instruments have strange harmonic structures ! For example, clarinet usually has odd harmonics only. Or vibraphones (depending on the manufacturer) might produce only several non-arranged harmonics. If such cases are systematic (and you can observe it clearly on the instrument spectrum), you can tell Antescofo to systematically use a certain harmonic series (which also accepts floating points) with the message `harmlist`. *We recommend you to be quite sure before changing this since it can severely affect behavior!*

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.

Printing the Parsed File

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

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

Tracing

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

The Outlet

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

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

Verbosity

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

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

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 `::antescofo $v` is interpreted as false (which will prohibit the triggering of the body).

Tracing the Evaluation of Functions

Function calls can be traced, see the description of the functions `[@Tracing]` and `[@UnTracing]`.

Chapter 6

Beyond score following. . .

Most people think *Antescofo* is exclusively used for mixed music and score following. That's partly true, but since the language exists and has been developed, *Antescofo* is also a great tool for electronic music.

Antescofo as a sequencer

Speaking about time is very easy in *Antescofo*. *AscoGraph* and an *Antescofo* score can be seen like a sequencer, where all the actions are organised in time (if you click on the `Play` button, *Antescofo* becomes a sequencer) that you specify in the score.

You can write an electronic score with the beat notation and decide after to change the tempo. In this case you don't have to rewrite all the durations. *Antescofo* does the translation. This allows you to change the tempo (with BPM) in any place in your score.

As we have shown before, you can also work with different times in a same moment with the `@tempo` attribute and write complex polyrhythms.

With `macros` you can greatly simplify your score to focus on the musical questions instead of the technical realization.

Hierarchical scores

With the system of `GROUP`, you can write a hierarchical score where the different voices are denoted by different groups. You can also create `tracks` (see `@track_def`) to filter any kind of musical object or focus on a particular voice.

With the action view of *AscoGraph* you can easily see how the different voices of an electronic polyphony interact with each other.

Open scores and installations

If you see the `process` and `whenever` constructions, you will see that these features can be easily used for installations or for open scores. You can use *Antescofo* without a "classical score", without `NOTES`, `CHORD`... It means that you can program a reactive environment

that “hears” sensors, audio descriptors...

If you compose open scores, you can take a look to the `@jump` keyword that 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`. So you can compose a non-linear score with choice points leaved to the performer. You even can modify during the execution the list of allowed jumps, achieving a kind of “multi-graph” score.

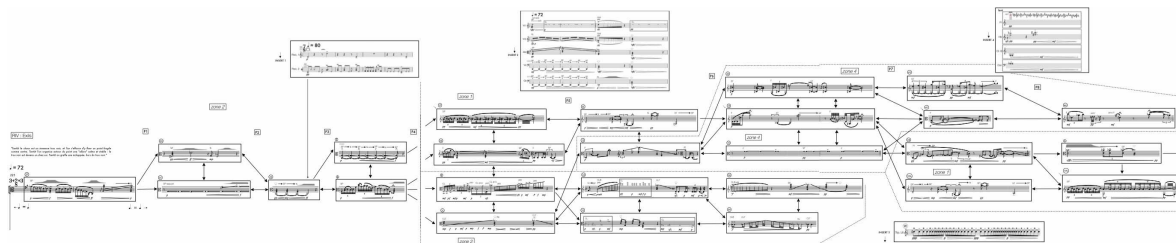


Figure 6.1: Open score in “Tesla ou l’effet d’étrangeté”, Julia Blondeau

Beyond Max and Pure Data...

We have shown many uses of *Antescofo* in Max. However, with OSC, you can use *Antescofo* as controller of any kind of software that supports OSC. For example, SuperCollider and CSound are great *Antescofo* fellow players ! In the two next examples, Max is a kind of “pipe” between *Antescofo*, CSound or SuperCollider, IRCAM’s Spat and audio descriptors. But *Antescofo* sends also many messages via OSC without Max.

A SuperCollider example

The figure below shows an environment created by José-Miguel Fernandez for his pieces where you can see 3 different softwares. In the background you can see the AscoGraph interface. At the left front you can see Max using *Antescofo* and descriptors (who are send to whenever and process). At the right you can see a SuperCollider interface where processes are generated “on the fly” and displayed on the screen. In this example, *Antescofo* manages all the control parts and sends the parameters needed for synthesis and effects to SuperCollider. Fernandez uses many chaotic functions included in *Antescofo* and several of its data structures (e.g., `map` and `tab`).

A CSound example

In the next example, *Antescofo* is used to send a score to CSound and to generate table for CSound. Here, *Antescofo* can replace the “.sco” file for CSound or the “score part” of the “.csd” in CSound. All the CSound synthesis is generated in real time and controlled by the object `antescofo~`.

The spatialization is controlled by *Antescofo* too with many `curves` and `process` that generate complex trajectory.

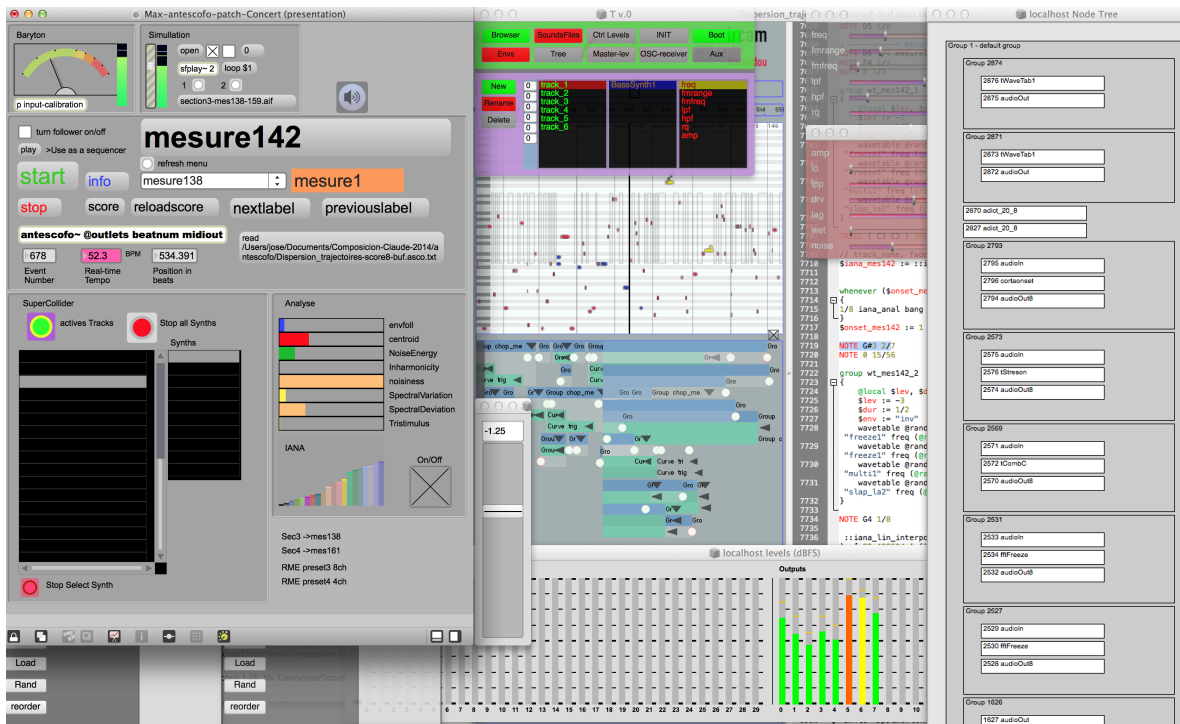


Figure 6.2: Utilisation of Antescofo with SuperCollider, works by José-Miguel Fernandez

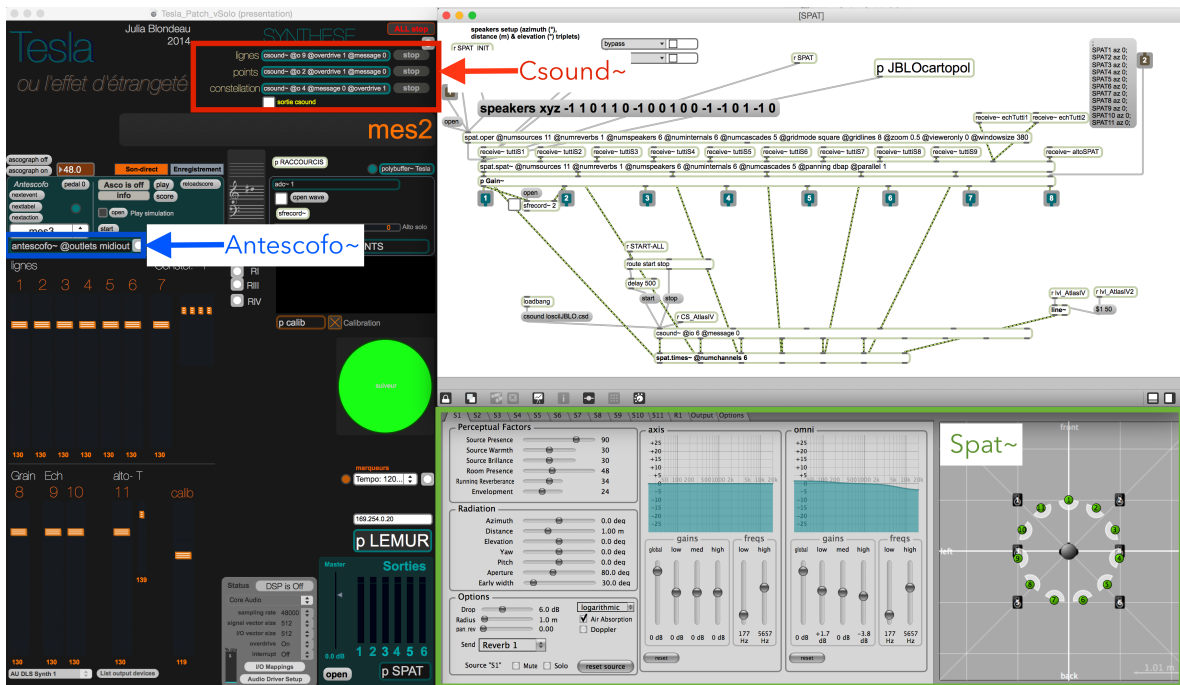


Figure 6.3: Utilisation of Antescofo with CSound, works by Julia Blondeau

OSC

Many people have been using Antescofo message passing strategies as defined above to interact with processes living outside MAX/PD (such as CSound, SuperCollider, etc.). To make their life easier, Antescofo comes with a built-in OSC host. The OSC protocol³ can be used to interact with external processes using the UDP protocol. It can also be used to make two Antescofo objects interact within the same patch. Unlike MAX or PD messages, OSC messages can be sent and received at the level of the Antescofo program. The embedding of OSC in Antescofo is done through 4 primitives.

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

```
oscrcv name port msg_prefix $v1 . . . $vn
```

see the [OSC message](#) chapter for more details.

Be adventurous !



Figure 6.4: DonQ

“De toute manière, le compositeur, lors de ses voyages d’exploration, se voit à la fois comme Christophe Colomb et Don Quichotte – il débarque sur une terre inconnue et/ou tombe de cheval et atterrit de manière peu glorieuse, et à chaque fois en tout cas là où il ne pensait pas aboutir ; c’est ainsi seulement qu’il fait l’expérience de lui-même, qu’il se transforme, qu’il vient à lui-même.”

LACHENMANN, *Ecrits et entretiens*.

“In any case, the composer, during his exploratory journeys, is seen as both Columbus and Don Quixote - he lands in an unknown territory and/or falls from his horse and lands

ungloriously, and each time in any case there where he did not think to reach; only in this way he does experience himself, turn, comes to himself.”

LACHENMANN, Writings and interviews.

We have provided a brief overview of Antescofo, but a WORLD remains to be discovered. . . and experimented with! In the [Reference Manual](#), you can more precisely research the previously addressed issues. You can decide to only take the examples given at the end of this documentation (snippets, How-to. . .) and copy them in your own score. . . or you can decide to roll up your sleeves and understand all the intricacies and secrets of Antescofo language to try to be an “aware Antescofo electronic composer!” ;-)

If you continue the adventure, you will see how to create a complex process or how to create a reactive environment where Antescofo can hear many things and react as you had composed! You will see how to manipulate time in Antescofo and how to synchronize your machine with a musician in real, musical way!

So. . . Try and fail, retry and be victorious, and don’t forget that your feedback is important to us. Please, send your comments, questions, bug reports, use cases, hints, tips & wishes using the IRCAM Forum discussion group at:

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

Antescofo Reference Manual

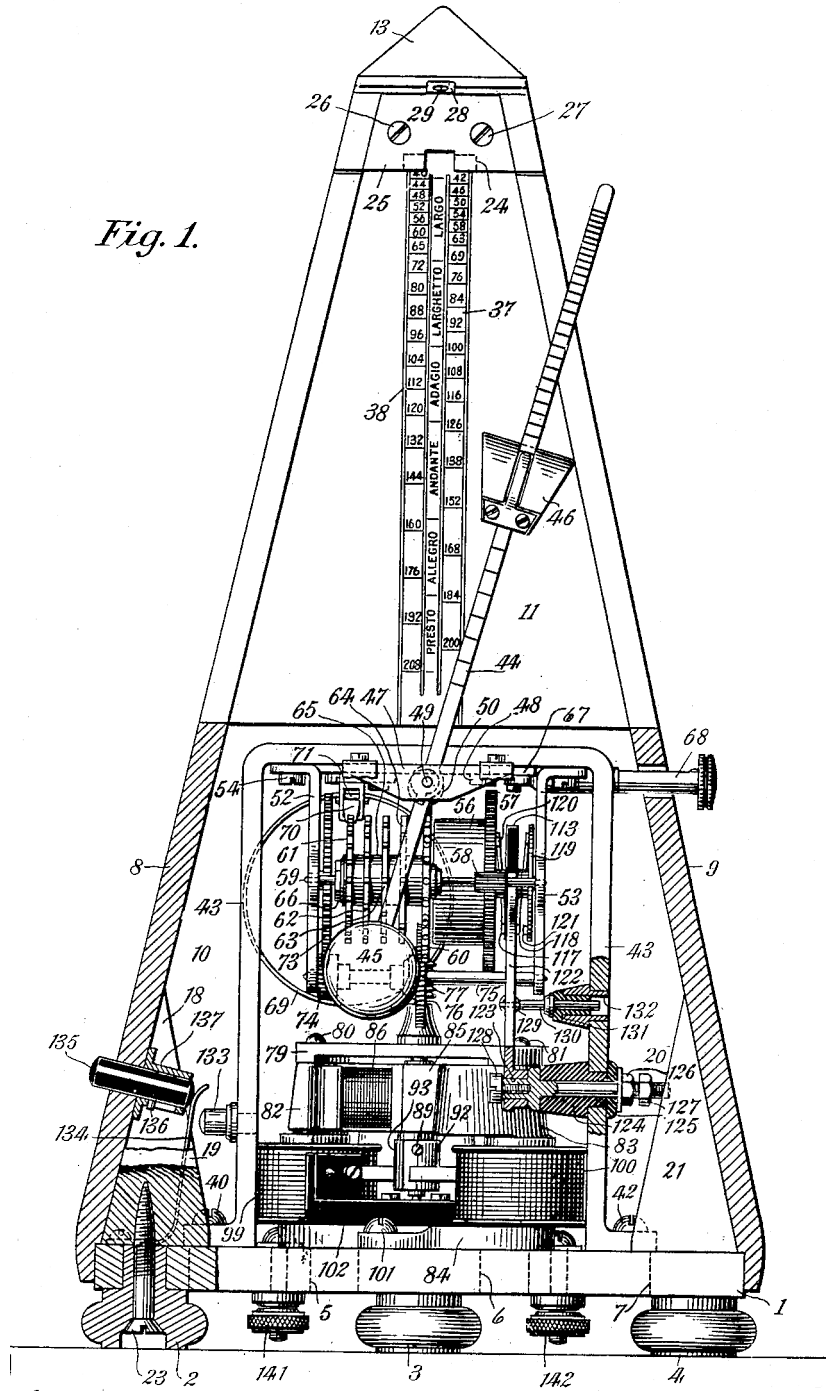


Figure 6.5: Frontispice metronom

Chapter 7

Introduction

This document is intended as a reference manual for the *Antescofo* programming language. It lists the language constructs, and gives their informal syntax and semantics through several chapters gathered in eight parts:

Syntax and score structure

- [Lexical elements](#) describes the lexical notation in an *Antescofo* program.
- [Program Structure](#) discusses the organization of an augmented score and present the interleaving of definition, events and actions.

Events

- [Event Specification](#) details the definition of musical events expected by the listening machine.

Actions

[Actions](#) are the computations launched by the system in reaction to the recognition of musical events or as the time passes. The introduction presents the notion of action sequence, actions' attributes and delays that are available for any kind of action. Then, the two kinds of actions are introduced:

- [Atomic actions](#) take no time to be performed;
- [Compound actions](#) represent durative activities.

Management of Time

The chapter [A strongly Timed Language](#) introduces the various notion of time at work in *Antescofo*:

- [the manufacturing of time](#) elaborates the temporal notions that organize the computations of actions;

- [the fabric of time](#) elaborates on the relationships between the potential timing of musical events expressed in the score and the actual time of the actions during the performance.

These two sections are followed by technical discussion of *Antescofo* temporal features:

- [Action Priority](#) explains the ordering of action that must be executed in the same instant.
- [Synchronization Strategies](#) details the constraints that can be specified between the actual timing of the performer and the timing of the electronic actions during the performance.
- [Error Strategies](#) presents the management of error from the action perspective.

Expressions

[Expressions](#) are used to parameterize actions. They are built from values and [variables](#), [conditional expression](#) and [functions](#).

Expressions are evaluated into values which are either:

- [Scalar values](#) represent indecomposable values, or
- [Data structures](#) that provide several ways to organize the data to manage during a performance.

Structure

The following chapters detail the main mechanisms that can be used to capitalize a piece of code and to re-use it:

- [Functions](#)
- [Processes](#)
- [Macros](#)
- [Actors](#)
- [Temporal Patterns](#)

Additional Features

Finally, several additional features that do not fit in the previous chapters are presented:

- [Tracks](#)
- [Files layout](#) of an *Antescofo* augmented score
- [Evaluation at load time](#)

Side-Notes

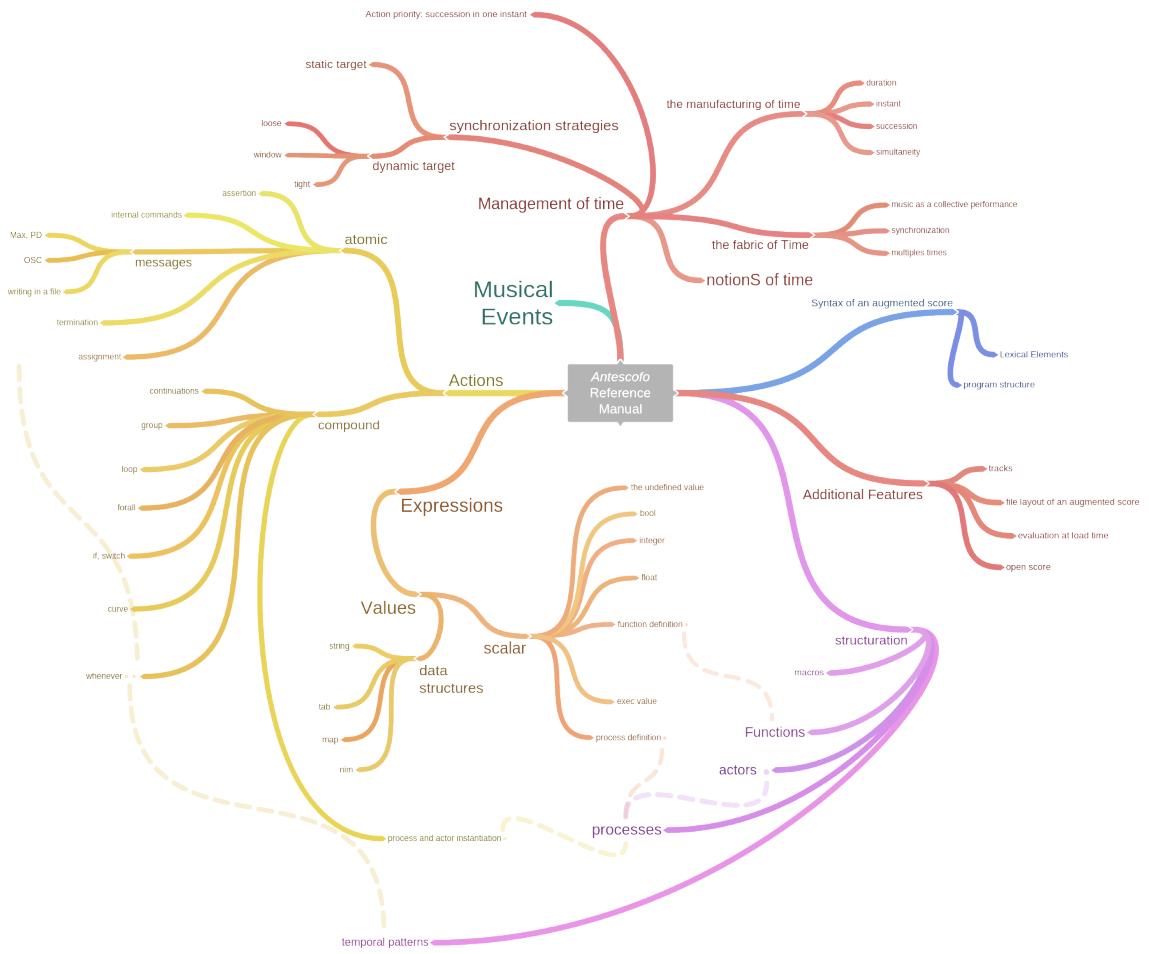
A few side-notes are used to present in detail some points of the *Antescofo* grammar and to dig deeper in some technical subjects:

- [Auto delimited expressions](#)
- [Simple expressions](#)
- a comparison between [Macro, Function and Processus](#)
- [Argument evaluation strategies](#)
- and the [grammar of object definition](#)

Other source of documentation

This [Reference Manual](#) is by no means a tutorial introduction to the language. A good working knowledge of the *Antescofo* application is assumed:

- The reader may refer to the [User Guide](#) for a general presentation of the system.
- This manual is completed by a [Functions Library](#) that describes all predefined functions in the *Antescofo* library.
- The [Antescofo distribution](#) comes with several tutorial patches for [Max](#) or [PD](#) as well as the augmented score of actual pieces.
- The [Antescofo Cookbook](#) is a valuable source of code snippets and *howto's*.
- And the [ForumUser](#) can be used to share information about *Antescofo*.



Chapter 8

Lexical Elements of an Antescofo Score

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

- **Comments:** Any text starting by a semi-colon `;` or `//` is considered as comment and ignored by parser until the end of line (inline comment).
Block (multi-line) C-Style comments starting with `/*` and ending with `*/` are allowed.
- **Keywords:** are reserved words introducing either Events or Action constructions. Examples include `Note` (for events) and `Group` (for compound actions).
- **Simple identifiers:** denote Max or PD receivers. They are also used to specify the label of a musical event or the label of an action.
- **`**_identifiers**` :** *are words that start with*“ character. They correspond to user-defined variables or parameters in functions, processes, object or macro definitions.
- **`::-identifiers`:** words starting with `::`, `obj::`, `pattern::` or `track::` corresponding respectively to processes, objects, patterns or tracks.
- **@-identifiers:** are words starting with a `@`. They either introduce a new definition or denote predefined and user-defined functions, user-defined macros, or action or event attributes.

User defined score elements like macros, processes, objects, tracks and functions can only be used after their definition in the score. We suggest putting them at the beginning of the file or to put them in a separate file using the `@insert` command. They will be discussed in later chapters.

Case-Sensitive and case-Insensitive Identifiers

Identifiers are sometimes case-sensitive and sometimes case insensitive. The rule is simple: reserved keyword or predefined identifiers are case insensitive. User introduced identifiers and predefined function names are case sensitive.

Comments

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

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

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

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

In this document, we use *railroad diagrams* to give a more precise description of the syntax. The railroad diagram specifying comment's syntax is:

```
{!BNF_DIAGRAMS/bnf_comment.html!}
```

Comments are ignored by the interpreter but not suppressed in normal processing. That is, they act usually as separators, *i.e.*

```
$x/**/y
```

is read as two consecutive tokens x and y . However, during macro-expansion comments are suppressed (not ignored) in the textual expansion: in a macro-expansion, the previous fragment is read as only one token xy by the interpreter (see chapter [Macros](#)).

Indentation

Tabulations are handled like white spaces. Columns are not meaningful so you can indent 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 myLab
Note C4 0.5 "some label with white space used to document the score"
1.0s
print "this is a Max message (to the print object)"
print "printed 1 seconds after the event Note C4..."
```

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

Splitting a line

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

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

Reserved Keywords

Reserved keywords can be divided in two groups:

- **Event Keywords** including NOTE, CHORD, TRILL, BPM, *etc.*, introduce musical events (see chapter [Events](#)) and are used to describe the music score to be recognised.
- **Action Keywords**, such as GROUP, LOOP and more, specify computations that can be instantaneous ([Atomic actions](#)) or *containers* for other actions that have a duration ([Compound actions](#)).

The current list of reserved keywords is :

```
<div class="kr">
abort &nbsp;
action &nbsp;
and &nbsp;
at &nbsp;
</div>
```

```
<div class="kr">
before &nbsp;
bind &nbsp;
bpm &nbsp;
</div>
```

```
<div class="kr">
case &nbsp;
    chord &nbsp;
closefile &nbsp;
curve &nbsp;
</div>
```



```
<div class="kr">
whenever &nbsp;
where &nbsp;
while &nbsp;
with &nbsp;
</div>
```

Notice that event keywords always occur at the top-level of the text score. Reserved keywords are *case insensitive*, that is,

```
note NOTE Note NoTe notE
```

refers to the same identifier. **However** simple identifiers which are not reserved keywords are *case sensitive*.

***Simple Identifiers : Antescofo* keywords and references to the host environment**

Simple identifiers are sequence of characters accepted by one of the three following regular expressions and that are not reserved keywords:

```
[[:alpha:]]_#?!~\xc3\xe2\x80-\xbf][[:alnum:]]_#'/./?!+~<>\-\Xc3\xe2\x80-\xbf]
[0-9]+[[:alpha:]]_'\xc3\xe2\x80-\xbf-}{2,}[[:alnum:]]_#?!~*/+.\-\xc3\xe2\x80-\xbf
[0-9]+[a-zA-Zt-zT-Z]+
```

[`:alpha:`] represents an alphabetic character, [`:alnum:`] represents an alphanumeric character, *i.e.* [`0-9a-zA-Z`] and the hexadecimal range `\xc3\xe2\x80-\xbf` represents a reasonable subset of UTF-8 accentuated characters and printable symbols.

These identifiers are much more general than the identifiers [`:alpha:`][`:alnum:`]* usually recognized in programming language. For example they can start by a number or include a relational operator like `<`, in order to represent the various Max or PD receivers found in current patches (identifier starting with a number are common when working with Max poly). It is however possible to encounter Max receivers that are not caught by the above regular expressions. In this case, just use a `string` instead of a simple identifier, for the receiver. For instance

```
"max receiver with white spaces" 1 2 3
```

refers to a receiver whose name includes white spaces.

\$-identifiers : Variables

`$-identifiers` like `$id`, `$id_1` are simple identifiers prefixed with a dollar sign. Only `! ? .` and `_` are allowed as non-alphanumeric characters. `$-identifiers` are used to give a name to variables and for function, process and macro definition arguments. *They are case-sensitive.*

You can learn more on expressions and variables in chapter [Expressions](#).

::-identifiers : Processes

::-identifiers like ::P or ::q1 are simple identifiers prefixed with two semicolons. ::-identifiers are used to give a name to processus (see chapter [Processes](#)).

Other ::-identifiers have a prefix before the :: and are used to give a name to various entities:

- obj:: identifies [Objects](#)
- track:: identifies [Tracks](#)
- pattern:: identifies [Patterns](#)]

__@-identifiers__ : Functions, Macros, and Attributes

A word preceded immediately with a ‘@’ character is called a @-identifier. Only ! ? . and _ are allowed as non alphanumeric characters after the @.

They have four purposes in Antecofo language. Note that in the first three cases, @-identifiers are *case insensitive*, that is @tight, @TiGhT and @TIGHT are the same keyword.

Parsing directives

Some commands affect directly the parsing of a file:

- the @insert command is used to insert another file,
- the @insert_once command is used to insert another file if it has not been already inserted;
- the command @uid generates on-the-fly a new (fresh) variable (\$-identifier)
- the @lid command generate on-the-fly the las generated identifiers

Definitions

The following @-identifiers are used to introduce new definitions:

- @abort specifies an *abort handler* linked to a compound action,
- @broadcast introduces a *broadcast method* in an object definition,
- @fun_def defines a new *function* or a function-method attached to an object,
- @init introduces a new *init section* in an object definition,
- @obj_def specifies a new *object* definition,
- @pattern_def declares a new *pattern* to be used in a whenever
- @proc_def starts a *process* or process-method definition

- @track_def declares a new *track*
- @whenever defines a *reaction*
- @macro_def is used to define new *macros*.

Events Attributes and Actions Attributes

Here is a list of the reserved @-identifiers used to specify an attribute of an action or an event. These @-identifiers are *case insensitive*.

```

@abort @action @ante

@back @back_in @back_in_out @back_out @bounce
@bounce_in @bounce_in_out @bounce_out

@circ @circ_in @circ_in_out @circ_out @coef @command
@conservative @cubic @cubic_in @cubic_in_out @cubic_out

@date @dsp_channel @dsp_cvar @dsp_inlet @dsp_link
@dsp_outlet @dump

@elastic @elastic_in @elastic_in_out @elastic_out
@eval_when_load @exclusive @exp @exp_in @exp_in_out @exp_out

@fermata

@global @grain @guard

@hook @immediate

@inlet @is_undef

@jump

@kill

@label @latency @linear_in @linear_in_out @linear_out
@local @loose

@modulate

@name @norec

@pizz @plot @post @progressive

@quad @quad_in @quad_in_out @quad_out @quart @quart_in
@quart_in_out @quart_out @quint @quint_in @quint_in_out

```

```
@quint_out  
  
@rdate @refractory @rplot  
  
@sine @sine_in @sine_in_out @sine_out  
@staccato @staticscope @sync  
  
@target @tempo @tempovar @tight  
@transpose @type
```

Naming Functions and Macros

@-identifiers are used to give a name to predefined functions, user-defined functions or user-defined macros. These predefined or user-introduced @-identifiers are *case sensitive*.

```
{!Library/Functions/functions.list!}
```


Chapter 9

Structure of an *Antescofo* Program

An *Antescofo* program, or **augmented score**, is a *sequence* of:

- macros, processes, functions, tracks, pattern and object **definitions**
- musical **event** specifications
- **action** specifications

written in a file. This file can include other files (inclusions can be nested arbitrarily) using the `[@insert](/Reference/file_structure/index.html)` and `[@insert_once](/Reference/file_structure/index.html)` commands, see section [file structure](#). File inclusion does not alter the organization of an augmented score: the interpreter sees one single flat file resulting from the textual inclusion of all included files before any evaluation.

```
{!BNF_DIAGRAMS/augmented_score_1.html!}
```

Events are recognized by the listening machine (described in detail in chapter [Event Specification](#)). Actions, outlined in chapter [Actions](#) and detailed in [Atomic Actions](#) and [Compound Actions](#), are computations triggered upon the occurrence of an event or of another action. Actions can be dynamically parametrised by [Expressions](#) and data structures, evaluated in real-time and described in detail in the sections [Scalar Values](#) and [Data Structures](#).

[Comments](#) may appear anywhere and are simply ignored by the interpreter.

Definitions

Definitions may appear anywhere between events and actions. This is only for the programmer convenience: they are handled when the score is loaded and their appearance do not alter the sequence of events, the sequence of actions or the interleaving of events and actions. When the sequencing of actions and events must be considered, definitions are simply abstracted away.

```
{!BNF_DIAGRAMS/definitions.html!}
```

There are nine kinds of entities that can be defined (see the above diagram). An entity (function, process, *etc.*) must be defined before being used. So we suggest gathering all definitions at the beginning of the program, even if they can appear anywhere in the file. That said, the position of a process definition in the file has an impact on its priority. See section [Action Priority](#).

When definitions are abstracted away, the structure of an augmented score is better viewed as a **first sequence of actions** followed by **reactions** made of an event followed by a sequence of actions :

```
{!BNF_DIAGRAMS/augmented_score.html!}
```

The First Sequence of Actions

Abstracting the definitions away, an augmented score starts by an optional sequence of actions. Actions in this section are evaluated at the beginning of the performance, as soon as the program is launched with a `start` or a `play` command and before the recognition of the first musical event.

An augmented score can consists of only this first sequence of actions. In this case, there is no musical event to recognize at all, and the actions correspond to a synchronous and temporized program (like a sophisticated cue list). This program is launched by the command `start` or `play`, may wait the elapsing of delays and can react to changes in the external environments through

- [OSC messages](#),
- [Whenever](#) watching variables set by the environnement using the `setvar` command
- [commands](#) send through the Max/PD interface to the *Antescofo object*.

Reactions: Events Triggering a Sequence of Actions

A reaction is the specification of a musical event followed optionally by a sequence of actions. This sequence of actions is triggered by the recognition of the musical event in the input stream (audio or midi).

Some elements categorized here as *events* do not act as true musical event to recognize in the input stream but as modifiers that affect the state of the listening machine for the recognition of the following ‘real’ musical events:

- BPM
- variance
- tempo
- transpose
- rubato (experimental)
- `napro_trace` (deprecated)

See also the section [Elements](#) in the user guide.

These elements can be abstracted away when considering the sequence of actions linked to an event.

Events are further described in the chapter [Event Specification](#). Actions are further described in the chapter [Actions Specification](#).

The Sequence of Reactions

Reactions appear in an augmented score in a *sequence*. The order in this sequence is fixed by the order of textual apparition in the file.

The score followed by the listening machine corresponds to the sequence of musical events extracted from the sequence of reactions. However, the `[@jump]` attribute can be used on musical events to evade the strict linear ordering of events and to specify arbitrary graph between events. In this graph a branching between events corresponds to an open score given the choice between several possible futures. See section [open scores](#).

An Example

Consider the following score (excerpt of an actual piece):

```

BPM 65

// some definition
@proc_def ::CS_solo_points($x, $y, $z, $u, $v) { /* ... */ }

@global $tSolo, $tabFreq

let $tSolo := 65
let $tabFreq := [ /* ... */ ]

// start of score
NOTE D1 1/8
NOTE C2 1/8
NOTE Db2 1/8
NOTE Ab1 1/8
NOTE A2 1/2
Curve tSolo @grain := 0.05s
{
  $tSolo
  {
    { $RT_TEMPO } @type "exp"
    1/2 { ($RT_TEMPO+30) } @type "cubic_out"
    2/4 { ($RT_TEMPO-20) }
  }
}
BPM 68
GROUP Solo @tempo := $tSolo
{
  ::CS_solo_points("i3",1/2,0.08,0.5,81)
  1/8 ::CS_solo_points("i1",1/2,0.09,0.5,91)
}

NOTE 0 5/2
NOTE G1 0
NOTE B1 0
NOTE D2 0

```

```

NOTE C1 1/2
NOTE 0 1/2
NOTE C1 1/2
NOTE G2 1

@fun_def @midi2hz($x)  {$diapason * @exp(($x-69.0) * @log(2.0)/12)}

    GROUP Solo2
    {
        ASCotoCS_SYNT4 c (@midi2hz($stabFreq[0]))
        ASCotoCS_SYNT4 c (@midi2hz($stabFreq[2]))
        ASCotoCS_SYNT4 c (@midi2hz($stabFreq[4]))
    }

NOTE 0 1/2
NOTE G1 0
NOTE B1 0
NOTE D2 0

```

With respect to the structure of the performance, the definition of process `::CS_solo_points` and of function `@midi2hz` can be abstracted away. The definitions are mandatory but they are processed when the file is loaded, not during the performance.

The `[@global]` clause is also a definition introducing global variables. Global variables are implicitly defined, so here this definition is used only for documentation purposes, to outline that two global variables will be used in this score.

The two BPM specifications are not real sonic events: they alter the recognition of the following musical events.

When we abstract these definitions and event modifiers away, the resulting score is:

```

let $tSolo := 65
let $stabFreq := [ /* ... */ ]

NOTE D1 1/8
NOTE C2 1/8
NOTE Db2 1/8
NOTE Ab1 1/8
NOTE A2 1/2
Curve tSolo @grain := 0.05s
{
  $tSolo
  {
    { $RT_TEMPO } @type "exp"
    1/2 { ($RT_TEMPO+30) } @type "cubic_out"
    2/4 { ($RT_TEMPO-20) }
  }
}
GROUP Solo @tempo := $tSolo
{
    ::CS_solo_points("i3",1/2,0.08,0.5,81)
    1/8 ::CS_solo_points("i1",1/2,0.09,0.5,91)
}

```



```

    }

NOTE 0 5/2
NOTE G1 0
NOTE B1 0
NOTE D2 0
NOTE C1 1/2
NOTE 0 1/2
NOTE C1 1/2
NOTE G2 1
GROUP Solo2
{
    ASCoToCS_SYNT4 c (@midi2hz($stabFreq[0]))
    ASCoToCS_SYNT4 c (@midi2hz($stabFreq[2]))
    ASCoToCS_SYNT4 c (@midi2hz($stabFreq[4]))
}

NOTE 0 1/2
NOTE G1 0
NOTE B1 0
NOTE D2 0

```

which is composed of a first sequence of actions

```

let $tSolo := 65
let $stabFreq := [ /* ... */ ]

```

followed by 17 reactions. These reactions corresponds to the following sequence of musical events looked in the audio stream by the listening machine

```

NOTE D1 1/8
NOTE C2 1/8
NOTE Db2 1/8
NOTE Ab1 1/8
NOTE A2 1/2
NOTE 0 5/2
NOTE G1 0
NOTE B1 0
NOTE D2 0
NOTE C1 1/2
NOTE 0 1/2
NOTE C1 1/2
NOTE G2 1
NOTE 0 1/2
NOTE G1 0
NOTE B1 0
NOTE D2 0

```

Only two reactions have actions associated to it: NOTE A2 1/2 and NOTE G2. A sequence of two actions (the curve `tSolo` and the group `Solo`) is associated to NOTE A2 1/2. Only

one action, the group `Solo2` is associated to `NOTE G2 1`. The groups are compound actions that gather together several others actions. Here they call processes or send `MAX/PD` messages.

Chapter 10

Event Specification

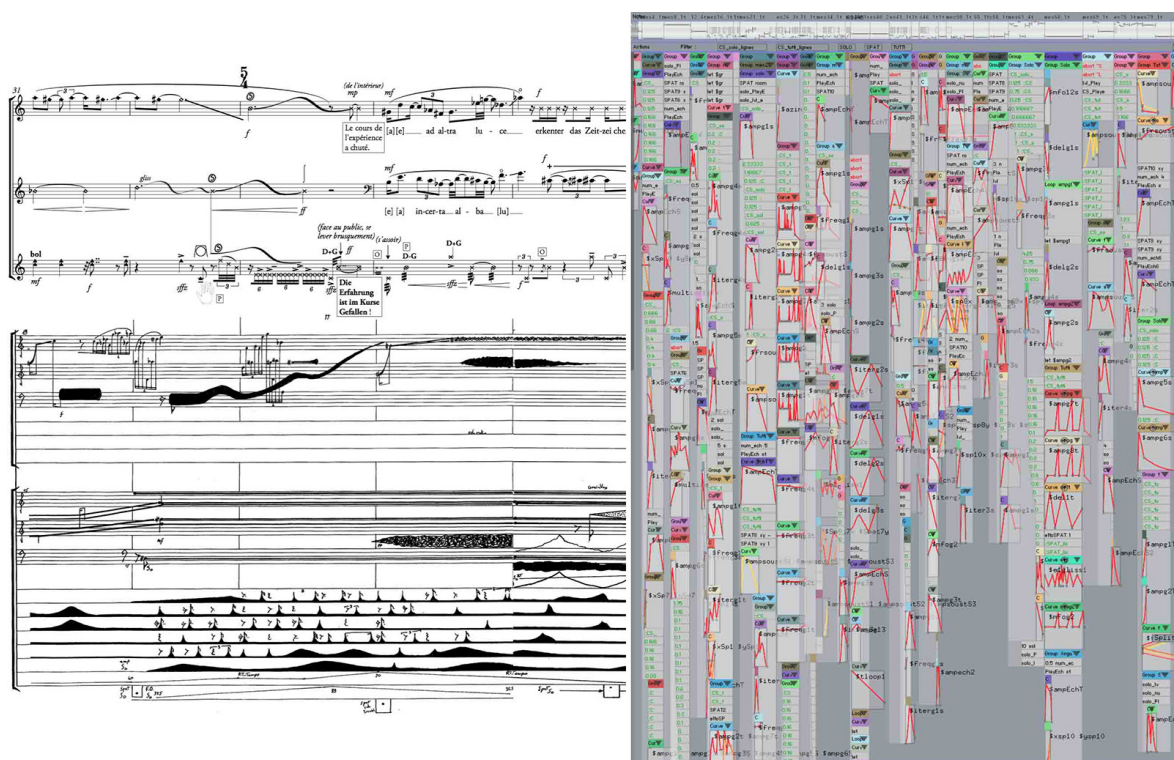


Figure 10.1: First measures of Nachtleben, Julia Blondeau, 2014.

First measures of Nachtleben, Julia Blondeau, 2014.

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. There are two basic event (NOTE and EVENT) and three event containers (CHORD, TRILL) and MULTI).

Here we give the general syntax of an event specification and then we details all its components.

Musical Event Specification

{!BNF_DIAGRAMS/event.html!}

Event definitions must end by a carriage return. In other words, you are allowed to define only one musical event per line.

TRILL and MULTI are examples of *compound events* organizing a set of NOTES in time. Thus they can accept one or several *pitch_lists*.

There is an additional kind of event

```
EVENT d
```

also followed by a mandatory duration *d*, which correspond to a fake event triggered manually by the “nextevent” button on the graphical interface or by the message ‘nextevent’ sent to the antescofo object.

We first detail the specification of a pitch, then the notation of a duration. We detail the various kinds of events before the presentation of the optional event’s attributes. The last section of this chapter is devoted to the command that alters the sequencing of the score.

Pitch Specification

{!BNF_DIAGRAMS/pitch_spec.html!}

A *pitch* (used in NOTE) can take the following forms:

- MIDI number (*e.g.* 69 and 70, 0 is silence),
- MIDI cent number (*e.g.* 6900 and 7000),
- Standard Pitch Name (*e.g.* A4 and A#4).
- For microtonal notations, one can use either MIDI cent (*e.g.* 6900) or Pitch Name standard and MIDI cent alteration using ‘+’ or ‘-’ (*e.g.* NOTE A4+50 and NOTE A#4+50 or NOTE B4-50).

A minus sign - may precede the previous specification to specify that the current note is a continuation of a note with the same pitch in the preceding event:

```
CHORD (C4 D5) 1
CHORD (-C4 D3) 1/2
```

A **pitch list** is a sequence of one or more pitches (without separator). They are used for instance to define content of a CHORD. For example, the following line defines a C-Major chord composed of C4, E4 and G4:

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

Duration specification

{!BNF_DIAGRAMS/duration_spec.html!}

Duration is a mandatory specification for all events. The of duration an event is specified *in beats* either by an integer (1), the ratio of two integers (like 4/3) or a float (like 1.5).

Events as Containers

Each event keyword in *Antescofo* 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. The figure below shows an example including simple notes and chords written in *Antescofo*:

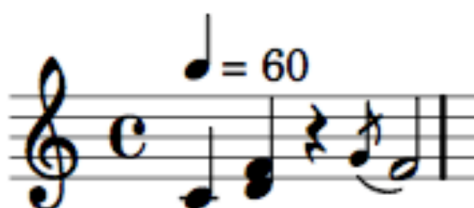


Figure 10.2: chord notation

```
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
```

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 below shows basic examples for Trill.

```
TRILL (A4 B4) 1.0
NOTE 0 1.0 ; a silence
TRILL ( (C5 E5) (D5 F5) ) 2.0
```

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

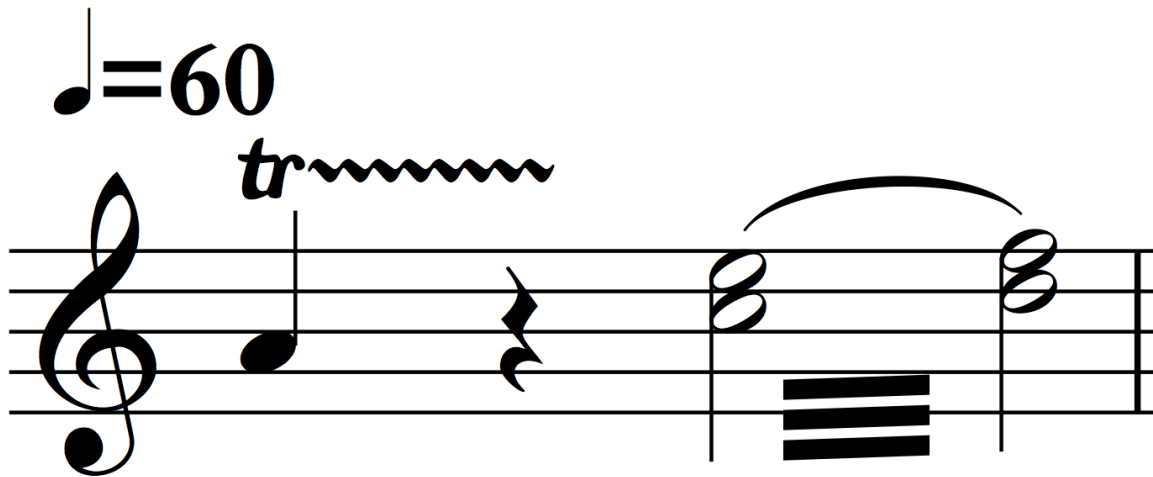


Figure 10.3: trill notation



Figure 10.4: trill notation

```

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

```

MULTI

{!BNF_DIAGRAMS/multi_spec.html!}

Similar to TRILL, a MULTI is a compound event (that can contain notes, chords or trills events) 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*.

A chord event inside a multi is specified through its *pitch list* between parenthesis.

To specify a trill event in a multi, it suffices to insert a ' character after the *pitch list* specifying the trill (between parenthesis). The figure below shows an example of glissandi between chords written by MULTI.

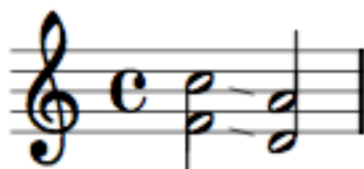


Figure 10.5: gliss notation

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

Event Attributes

{!BNF_DIAGRAMS/event_attributes.html!}

The attributes of a musical event are either a label or a @-keyword following the definition of the event. Attributes are optional.

Event Label

A simple identifier or a string or an integer acts as a label for this event. There can be several such labels. If the label is a simple identifier, its \$-form can be used in an expression elsewhere in the score to denote the time in beats of the onset of the event.

Label: Optionally, users can define *labels* on events as a simple identifier, a number or a *string*, useful for browsing inside the score and for visualisation purposes.

For example, `measure1` is an accepted label. If you intend to use *white space* or *mathematical symbols* inside your string, you should surround them with quotations such as `"measure 1"` or `"measure-1"`

Fermata, Pizzicato, Hook and Jump

There are four kinds of event attributes besides labels:

- The keyword `@fermata` specifies that this event has a *fermata* signature. A Fermata event can last longer and arriving and leaving it does not contribute to the tempo decoding of the performance.
- The keyword `@pizz` specifies the event is a string *pizzicato*. This usually helps Score Follower stability.
- The keyword `@hook` specifies that this event cannot be missed (the listening machine need to wait the occurrence of this event and cannot presume that it can be missed).
- The keyword `@jump` is followed by a variable or 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`. See below.

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

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

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

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

corresponds to 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`.

Open Score and Dynamic jumps

Open Score: specifying alternative follow-ups

Usually, musical events are matched in sequence: after an event e , the listening machine looks to match the event that is specified textually after e in the score. So, the score is deterministic: the expected future of each musical event is well defined¹.

The `[@jump]` attribute of an event is used to specify alternative “continuations” as a list of labels specifying the events that can be expected after e . This feature makes possible to escape the standard linearity of a score to specify *open score* where the musician may choose between several alternatives to proceed. The resulting score is not deterministic: after an event with a `[@jump]` attribute, one of the events listed in the jump list can be expected.

Obviously, the listening machine must be able to disambiguate the possible follow-ups: so, the first events of each continuation must be different.

Here is an example corresponding to the following open score organization:

```
{!BNF_DIAGRAMS/open_score.html!}
```

¹Even if the specified score is deterministic, the performance is not: musical events can be missed and the temporal relationships of the score are subject to the interpretation of the performer.


```

// INTRO
NOTE G3 1 INTRO
; ...
NOTE G4 1 @jump part2, part2_alt

// PART2
NOTE D2 1 part2
; ...
NOTE D3 1 end_part2 @jump next_part

// PART2 ALT
NOTE E3 1 part2_alt
; ...
NOTE E5 1 end_part2_alt

// NEXT_PART
; NOTE E5 next_part
; ...

```

Dynamic Jumps

Instead of a fixed list of labels, it is possible to use an *Antescofo* variable. This variable must refer to an event position (integer or float) or an event label (through a string) or a tab of them. This variable may change its value in the course of the performance. In this way, it is possible to achieve “*dynamic* open score” where the graph of the possibilities is updated during the performance, *e.g.* following the choices made by the musician, external events, internal computations, etc.

Dynamic changes in the score graph (through the variables appearing in the [`@jump`] attribute) impact the listening machine. The listening machine maintains a set of hypothesis about the potential events to recognize in the audio stream. If the changes in the score graph are anticipated enough with respect to the actual jumps, the listening machine will automatically accommodate these changes.

However, if the computation of the jumps are not far enough in time from the actual jumps, the listening machine must be explicitly warned to allow the revision of the hypothesis. This is done by setting `true` to the **system variable** `$JUMP_UPDATED` when the modifications are done. It is not easy to define what it means “far enough” because the temporal horizon used by the listening machine is adaptive.

Here is a toy example: the idea is to start with an `INTRO` sequence and then to finish with a `NEXT_PART` sequence, and in between playing at most once `PART2` or `PART3` followed by `INTRO`, in any order:

```

$jumps := [ "begin_part2", "begin_part3", "next_part" ]
$part2_done := false
$part3_done := false

// INTRO
NOTE G3 1 INTRO
; ...

```

```

NOTE G4 1 @jump $jumps

// PART2
NOTE D2 1 begin_part_2
    $part2_done := true
; ...
NOTE D3 1 end_part_2 @jump INTRO
    $jumps := if ($part3_done) { "next_part" }
            else { ["begin_part3", "next_part"] }
    $JUMP_UPDATED := true

// PART3
NOTE E3 1 begin_part_3
    $part3_done := true
; ...
NOTE E5 1 end_part_3 @jump INTRO
    $jumps := if ($part2_done) { "next_part" }
            else { ["begin_part2", "next_part"] }
    $JUMP_UPDATED := true

// NEXT_PART
; ...

```

In this example, the musician may choose to perform one of the following five scenarios:

```

INTRO --> NEXT_PART
INTRO --> PART2 --> INTRO --> NEXT_PART
INTRO --> PART2 --> INTRO --> PART3 --> INTRO --> NEXT_PART
INTRO --> PART3 --> INTRO --> NEXT_PART
INTRO --> PART3 --> INTRO --> PART2 --> INTRO --> NEXT_PART

```

Score statement

An *Antescofo* text score is interpreted from top to bottom. Score statements will affect lines that follow its appearance.

```
{!BNF_DIAGRAMS/score_statement.html!}
```

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

The `variance` statement is used to alter the variance parameter of the listening machine. A variance is associated to each musical event. The statement changes the variance of the next events for the specified quantity. Use at your own risk.

The `tempo` statement is used to disable or to enable the tempo inference. When the tempo inference is disabled, the nominal tempo specified in the score (through BPM statement, are used).

The `dummysilence` statement is used to insert a special “ghost event” is inserted between two musical events. This can improve in some contexts, the behavior of the listening machine (and degrade this behavior in other contexts).

The `pizzsection` statement can be used to define a sequence of musical events (between `pizzsection on` and `pizzsection off`) that have the `@pizz` attribute by default.

The `@transpose` statement is used to define a transpose factor used for all subsequent pitch definition.

The `rubato` statement is reserved for future specification.

Chapter 11

Actions Specifications

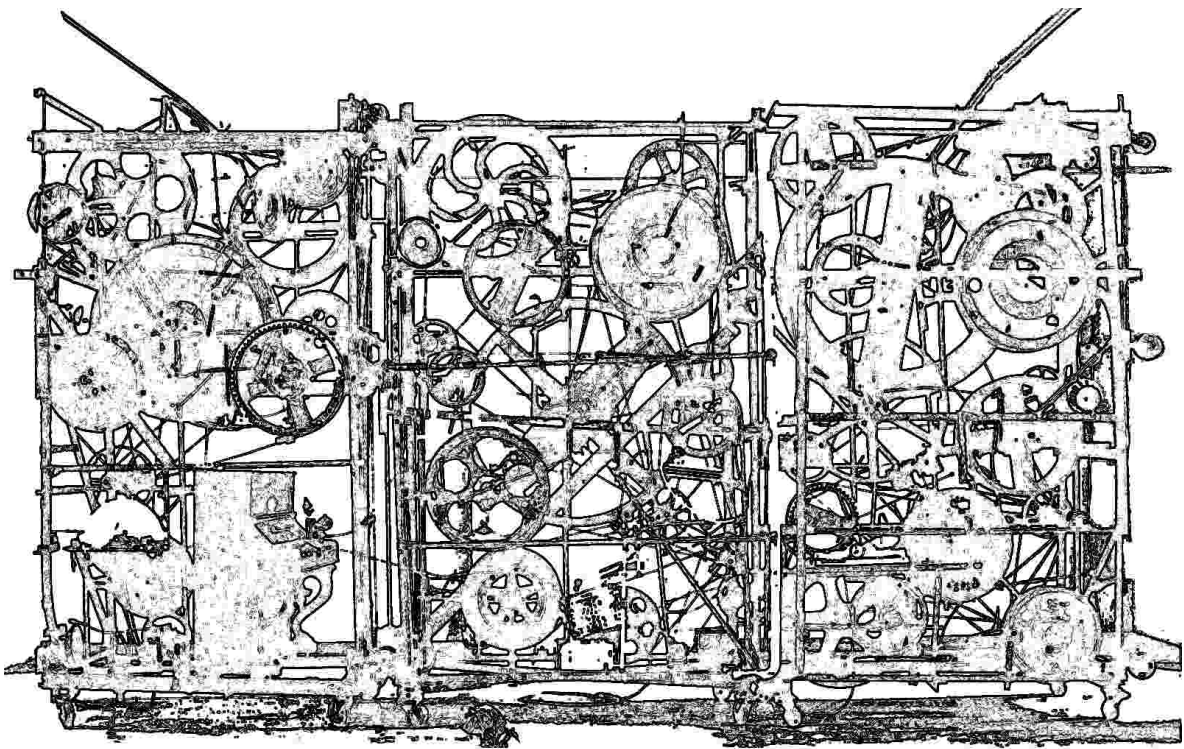


Figure 11.1: altered picture of a Tinguely machine

Actions are commands and computations performed by *Antescofo* at a certain point in time. Often, these actions are messages sent to a MAX/PD object to trigger other activities. Other actions correspond to internal computations. Actions can be also used to specify the temporal organization of subsequent actions.

Actions can be categorized as **instantaneous** or **durative**:

- an **instantaneous action** takes no time to be performed (see the [Synchrony hypothesis](#));
- a **durative action** is an action that takes times to be performed.

However, another relevant categorization is **atomic** or **compound**:

- An **atomic action** performs an elementary computation or a simple message passing that cannot be decomposed.
- A **compound action** groups other “child actions” allowing for *polyphony* (*i.e.* actions that are interleaved in time or that are performed in parallel), *loops* (*i.e.* actions that are iterated and repeated in time), conditional actions, *etc.*

An atomic action is always instantaneous. Usually, a compound action takes time to be performed. However, some compound actions may be instantaneous (for example, a conditional action that involves only one atomic action without delay).

Action Sequence

Actions always appear in sequences called **groups**. A group organizes the performance of its actions in time. Some groups are *explicit* when they are introduced with the **Group** construction (the fundamental compound action). Or they can be *implicit*:

- as the sequence of actions that appears after a musical event,
- as the children of other compound actions: **Loop**, **Whenever**, *etc.*,
- as the body of a process,
- as the [**@action**] clause of a **Curve**,
- as an [**@abort**] clause of a compound action,
- as an [**@init**] clause, a [**@whenever**] clause or as the body of a method in an object.

An action in an sequence of actions:

- starts with an optional **delay**
- is linked with the previous action through a **continuation operator**. They are currently three continuation operators:
 - ‘ ‘ (nothing, the two actions appears in sequence in the text) which specifies that the action that follows starts with the begining of the previous one;
 - ==> which triggers the next action with the end of the previous one;
 - and +=> which launches the next action at the end of the previous one including its children.

In addition, actions have optional **attributes** that are specified differently depending on whether the action is atomic or compound.

A Glimpse of Syntax

Actions Sequence

Sequence of actions appear after a musical event or as the body of a compound action. They are made explicit with the notion of **Group** which is used to specify additional properties of the sequence (*e.g.* synchronization attributes).

```
{!BNF_DIAGRAMS/general_actions.html!}
```

The `[@local]`, `[@global]` and `[@tempovar]` keywords introduce **variables declaration**. The local variables are local to the sequence of actions.

Atomic Action

These actions are further described in chapter **Atomic Actions**. They are performed instantaneously.

```
{!BNF_DIAGRAMS/atomic_actions.html!}
```

Compound Action

These actions are further described in chapter **Compound Actions**. They act as *temporal containers* organizing the temporal relationships of other actions.

```
{!BNF_DIAGRAMS/compound_actions.html!}
```

Action Attributes

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

```
Group G @att\ensuremath{_1}, @att\ensuremath{_2} := value { }
atomic_action @att\ensuremath{_1}, @att\ensuremath{_2} := value
compound_action @att\ensuremath{_1}, @att\ensuremath{_2} := value { ...
```

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

Some attributes are specific to some kind of actions. There are listed below and they are described in the section dedicated to this kind of action:

- `[@norec]` is relevant only for the **abort** atomic action,
- `[@action]` and `[@grain]` are meaningful only for the **Curve** construct,
- `[@abort]` and `[@exclusive]` have an impact on all compound actions,
- `[@immediate]` is relevant only for **Whenever**,
- `@staticscope` is a qualifier for process definition,
- synchronization attributes may alter all actions, including atomic ones (even if a `[@tempo]` specification on an atomic action is meaningless).

Curve Related Attributes {!BNF_DIAGRAMS/curve_attributes.html!}

Whenever Related Attributes {!BNF_DIAGRAMS/whenever_attributes.html!}

Process Definition Related Attributes {!BNF_DIAGRAMS/process_attributes.html!}

Abort Related Attributes {!BNF_DIAGRAMS/abort_attributes.html!}

Compound Actions Related Attributes {!BNF_DIAGRAMS/group_attributes.html!}

Synchronization Related Attributes {!BNF_DIAGRAMS/synchro_attributes.html!}

Labels

There is one additional attribute that can be specified for all actions: a **label**. The label of a compound action usually follows the keyword introducing the compound action, like the label `G` for the group in the example above. The label can also be specified with the `::antescofo @label` attribute:

```
action ... @label := a_label
action ... @label := "a label"
```

(this is the only way to give a label to an atomic action or to an `if` or a `switch`).

Labels are used to refer to an action, for instance to terminate it. Like events, actions can be labeled with:

- a simple identifier,
- a string,
- an integer.

There can be several labels for the same action. Unlike with event labels, the `$`-identifier associated to the label of an action cannot be used to refer to the relative position of this action in the score.

Delays

{!BNF_DIAGRAMS/delay.html!}

An optional specification of a *delay* can be given before any action *A*. This defines the amount of time between the previous event or action in the score and the computation of *A*. It can be expressed in seconds, milliseconds, or beats.

The delay countdown will begin to run from either the beginning or the end of the previous action, in accordance with to the continuation operator that precedes it. If the action is triggered by an event, the delay countdown must begin upon recognition of said event. See the section [Continuation Operator](#) for more information. For the rest of this section we suppose a default continuation: the delays are counted down from the beginning of the previous action.

Upon 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 C2 2.0
    d\ensuremath{_{1}} action\ensuremath{_{1}}
    d\ensuremath{_{2}} action\ensuremath{_{2}}
NOTE D2 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 C2 note, and `action2` will be triggered `d2` after the firing of `action1`. That is to say, `action2` is fired `d1 + d2` after the recognition of C2.

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 [Logical Instant](#).

Absolute and Relative Delay

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

```
    a\ensuremath{_{0}}
    1 s a\ensuremath{_{1}}
(2*$v) ms a\ensuremath{_{2}}
```

Action `a1` occurs one second 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.

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.

Delay vs. Expressions. The expression used in the specification of a delay, and more generally of a duration¹, must evaluate to a numeric (integer or float). There no specific type of value corresponding to a delay.

This means that `1 s` is not a value. The `s` or `ms` qualifier appears in the specification of a delay, but is not part of the expression defining the duration of the delay. A consequence is that you cannot pass `1 s` as the value of an argument (however, you can pass `1`).

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 chapter [Synchronization Strategies](#).

When an Action is Performed

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

- the occurrence of a musical event (detected by the listening machine)
- the change of a musical parameter (*i.e.*, the tempo)
- the start or the end of another action
- the expiration of a delay
- the reception of an [OSC message](#)
- a logical event (see the [whenever](#) construction and the chapter [Patterns](#)) triggered by an internal update (via `:=`) or an external update (via `setvar`) of a variable
- the reception of a [message](#) from the host environment (Max, PD)
- the loading of the score (cf. `[@eval_when_load]`)
- the signal spanned by an [abort](#) action (see `[@abort]` handlers)
- the sampling of a [Curve](#)
- the instance of an iterative construct [Loop](#) and [Forall](#)
- the launch of a process (cf. [Processes](#)) or the creation of an object (cf. [Objects](#))

In addition, for delays and for durative actions, the passing of time depends on a **temporal scope** which defines a tempo, a synchronization strategy and other temporal parameters. These notions are investigated in chapter [Synchronization](#).

¹used for the period of a [Loop](#) and in the breakpoint of a [Curve](#)

Chapter 12

Atomic Actions

Actions are divided into **Atomic Actions** which are performed **instantaneously** and cannot be decomposed and actions that gather others actions and that usually take time (described in next chapter **Compound Actions**).

In this chapter, we focus on **atomic actions** that is, actions that take no time to be performed. *Instantaneity* is an idealization of the actual computation, see chapter **Synchronization** for more detailed explanation on this abstraction.

An **atomic action** corresponds to

- an interaction with the environment through: message passing to MAX/PD receivers, OSC messages, or file writing,
- an assignment,
- the termination of another action,
- an internal command,
- the launching of a processus or the creation of an object,
- checking an assertion.

{!BNF_DIAGRAMS/atomic_actions.html!}

(Clicking on a box gives direct access to the corresponding documentation.)

Message passing to Max/PD

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

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

Since such actions are destined for interaction with external processes (in MAX/PD), we refer to them as *external actions*. External actions includes also sending an OSC message or writin data in some file, see below.

Message Receiver

A MAX/PD message starts by an optional delay followed by a symbol referring to a MAX or PD receiver. This identifier must be a simple identifier (that is, it cannot be a [reserved #-identifier] and must not be a [reserved keywords](#)) referring to **receiver object** in MAX/PD.

Alternatively, it is always possible to use an arbitrary string: the content of the string denotes the Max receiver.

For example, the following action attempts to send its message to a receiver called “print” in MAX/PD whose patch might look like the figure on its left:

```
<td>
<div>
! [simple print patch] (Figures/SimplePrintPatch.png)
</div>
</td>
<td>
```

NOTE C4 1.0

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

```
</td>
```

Message arguments

What follows a receiver is a comma-separated sequence of argument list. An argument list is simply a sequence of closed-expressions, simple identifiers and @-identifiers. *Antescofo* follows the Max convention: a message to the receivers is sent for each argument list, that is

```
print 1 2 3, 4 5 6, 7 8 9
```

is equivalent to the 3 messages

```
print 1 2 3
print 4 5 6
print 7 8 9
```

Message terminator

The specification of a message ends with a carriage-return (the end of the line) or a closing brace. This is important because the arguments of a message is a list of items, without separators, so a terminator is needed.

Writing a message with a lot of arguments on one line can be cumbersome. So a message can span several lines, but the intermediate lines must end with a backslash `\` which voids the following end of line.

For instance,

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

will print

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

In the second message of the previous example, there are 7 arguments: the first four are simple identifiers converted into the corresponding symbols, the fifth argument is evaluated into an integer and the last is a string. The backslash character has a special meaning and must be “backslashed” to appear in the string, see section [String](#).

Expressions in messages’ arguments

Expressions are evaluated to give the arguments of the message. To avoid ambiguities, an expression in a message must be a [closed expression](#), that is: a simple identifier, a scalar constant or an expression between parentheses.

In the previous example, the first `print` has two arguments: a string and a variable which evaluates to 1. Each value is converted into the appropriate MAX/PD values when the message is sent:

- a simple identifier is converted into a Max/PD symbol,
- a string is converted into a MAX/PD symbol
- an integer is converted into a long
- a float is converted into a float
- a boolean is converted into a long
- a tab of size n is converted into n arguments, one for each tab element
- other *Antescofo* values (*e.g.* `map` or `nim` or functions, processes, *etc.*) are converted into their string representation before being sent to Max/PD.

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

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

prints the following to MAX/PD console

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

Computing the receiver

Sometimes it is necessary, or just handy, to specify the receiver of a message as the result of a computation. In this case, the special construct `@command` is used:

```
@command(expression) argument_sequence
```

This action is performed in three steps:

- first the expression in the `@command` is evaluated.
- Then the resulting value is interpreted as a string.
- At last this string is used as the Max/PD receiver of the message.

Examples of such computations often involve string concatenation, as in

```
ForAll $num in (4)
{
    @command("spat" + $num) ($param[$num])
}
```

In this code, the [ForAll](#) construct iterates its body, which will send the four messages:

```
spat0 ($param[0])
spat1 ($param[1])
spat2 ($param[2])
spat3 ($param[3])
```

OSC Messages

Many people have been using message passing not only to control Max/PD objects, but also to interact with processes living outside MAX/PD such as **CSound**, **SuperCollider**, *etc.*

To make their life easier, *Antescofo* comes with a **builtin OSC server**. The [OSC protocol](#) can be used to interact with external processes using UDP messages. It can also be used to make two *Antescofo* objects interact within the same patch. The management of OSC messages is achieved in *Antescofo* through 4 primitives.

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
```

where:

- `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 or a string, like `localhost` or “`test.ircam.fr`”). 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 (*e.g.* between 49152 and 65535, see [List of TCP and UDP port numbers](#)).
- `msg_prefix` is the OSC address in the form of a string that will prefix the OSC message send to this channel.

As soon as the OSC channel is declared, it is started and can be used to send messages. Note that sending a message before the definition of the corresponding output channel is interpreted as sending a message to MAX.

Sending an OSC message takes a form similar to sending a message to MAX or PD:

```
name arg\ensuremath{_{1}} arg\ensuremath{_{2}} ...
```

This action construct and send the osc message

```
msg_prefix arg\ensuremath{_{1}} arg\ensuremath{_{2}} ...
```

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 OSC message prefixes can be used to identify the output channels. For instance, the declarations:

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

can be used to later invoke

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

OSCRECEIVE

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

```
oscrecv name port msg_prefix $v\ensuremath{_{1}} $v\ensuremath{_{2}} ...
```

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 port number where the is received.
- 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 assigned to the variables $\$v_1, \$v_2 \dots$. If there is less variables than arguments, the remaining arguments are simply thrown away (and an error message is emitted). Otherwise, if there is less arguments than variables, the remaining variables are untouched (and an error message is emitted).

Currently, *Antescofo* accepts only the following OSC types: bool, int32, int64, float, double, string and the arrays markers bracketing sequence of these values (nested arrays are allowed). These value are converted respectively into boolean, integer, float, string and bracketed sequence are converted into tab.

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.

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

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.

Conversion between OSC types and *Antescofo* types

Sending (or receiving) an OSC message implies the conversion of an *Antescofo* value into an OSC value (or the reverse). The conversion is applied following the following mapping

	<i>Antescofo</i> value	OSC value
bool		bool
int	int32	<i>see function</i> [<code>@set_osc_handling_int64</code>]
int	int64	<i>see function</i> [<code>@set_osc_handling_int64</code>]
Float	float	<i>see function</i> [<code>@set_osc_handling_double</code>]
Float	double	<i>see function</i> [<code>@set_osc_handling_double</code>]

	<i>Antescofo</i> value	OSC value
string		string
string		symbol
tab	<i>see function</i> [<code>@set_osc_handling_tab</code>]	

Remarks

- *Antescofo* value types that are not present in the table are not handled (e.g. `proc` or `nim`).
- The default handling of *Antescofo* integers is to send them as 32 bits integers, the only precision required by the OSC protocol. A call to `@set_osc_handling_int64(true)` switches this default behavior to send 64 bits integers instead. Notice that this feature is not implemented in all OSC packages. See function [`@set_osc_handling_int64`].
- The default handling of *Antescofo* float is to sending them as floats (32 bits representation), the precision required by the OSC protocol. A call to `@set_osc_handling_double(true)` switches this default behavior to send double (64 bits representation) instead. Notice that this feature is not implemented in all OSC packages. See function [`@set_osc_handling_double`].
- The handling of `tab` can be changed using the [`@set_osc_handling_tab`] functions. By default, *Antescofo* sends the elements of a `tab` as the successive arguments of a message, without using the OSC array facilities. A call to `@set_osc_handling_tab(true)` switches the behavior to rely on the array feature present in OSC *v1.1*. A call to `@set_osc_handling_tab(false)` switches to the default behavior (`tab` flattening).

Writing in a File

In the current *Antescofo* version, it is only possible to write an output file. The schema is similar to that of an OSC message: first, a declaration opens and binds a file to a symbol. This symbol is then used to write out in the file. Then the file is eventually closed. Here is a typical example:

```

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

```

The action `openoutfile` opens the file. Currently, there is only one possible mode to open a file: if it does not exist, it is created. If it already exists, it is truncated to zero at opening.

After `openoutfile`, the symbol `out` (a simple identifier) can be used to write in file `/tmp/tmp.txt` in a syntax that mimic messages. A “message” `out` is followed by a list of closed expressions, as for OSC or MAX/PD messages. Special characters in strings are interpreted as usual.

The optional integer `opt_int` at the end of the command `openoutfile` is used to minimize the impact of the i/o on the scheduling. It is interpreted as follows:

- If negative or null, the buffer associated to the file is shrunk to zero and the outputs are always flushed immediately to the file system.
- If positive, this number is used as a multiplier of the default file buffer system size. Factors greater than one increase the size of the buffer and thus reduce the number of effective i/o. The effect is usually negligible.

If the i/o's interfere with the scheduling, consider using the host environment to implement them (*i.e.* relying on Max or PD buffer to minimize the impact on time sensitive resources)¹.

The file is automatically closed at exit. Be aware that because of file buffering, the content of the file may be not entirely written on disk before closing it. If not explicitly closed, the file remains open between program load, start and play command.

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

Assignments

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

```
let $v := exp
```

Notice the assignation symbol is `:=`. The operators `=` and `==` denote the equality predicate. The keyword `let` is optional but more clearly distinguishes between the delay and the assigned variable:

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

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

A variable has a value before its first assignment: the **Undefined** value.

Expressions `exp` in the right hand side of `:=` are described in chapter [Expressions](#).

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

```
_ := exp
```

simply evaluates the right hand side and discards the result.

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

¹The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

Assignment to Vector Elements and to Scoped Variables

The left hand side of is not restricted to variables. As a matter of fact, there are three kinds of assignments:

Assignment to a variable

```
let $x := exp
```

the assignment of an element in a tab:

```
let e[i\ensuremath{\_1}, i\ensuremath{\_2}, ...] := exp
```

where e is an expression that evaluates to a tab and i_1, i_2, \dots , evaluate to integers (see sect. [mutating a tab element](#))

the assignment of a local variable in an *exec*:

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

where e is an expression that evaluates to an *exec* see [Exec](#) and [outsideScope](#).

The keyword `let` is mandatory when expressione is more complex than a variable, *i.e.* in the last two kinds of assignment.

Activities Triggered by Assignments

The assignment of a value to a variable may trigger some activities:

- the evaluation of a [whenever](#) that depends on this variable;
- the reevaluation of the delays that depend on a relative tempo that depend on this variable;
- the reevaluation of the synchronization strategies that depend on this variable.

As mentioned in section [Delays](#), 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 relative time, its conversion in physical time is dynamically adjusted when the corresponding tempo changes.

External Assignments

A global variable may be assigned “from outside *Antescofo*” in two ways:

1. using the message `setvar` to the *Antescofo* object in Max or PureData,
2. using an OSC message.

Section [OSCreceive](#) describes the assignment of variables upon the reception of an OSC message.

A simple patch using the `setvar` message is pictured below. The message takes as its first argument the name of the *Antescofo* variable to assign.

If there is only a second argument, this argument becomes the value of the variable. Max/PD integers, floats and strings are handled. If there are several remaining arguments, these arguments are put in a tab (see [and](#) and the variable is assigned with this tab value.

External assignments trigger the `whenever` that may watch the externally assigned variables, cf. [whenever](#). For example, with the patch pictured below, the program:

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

will write

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

on the console when the `prepend setvar ...` is activated.

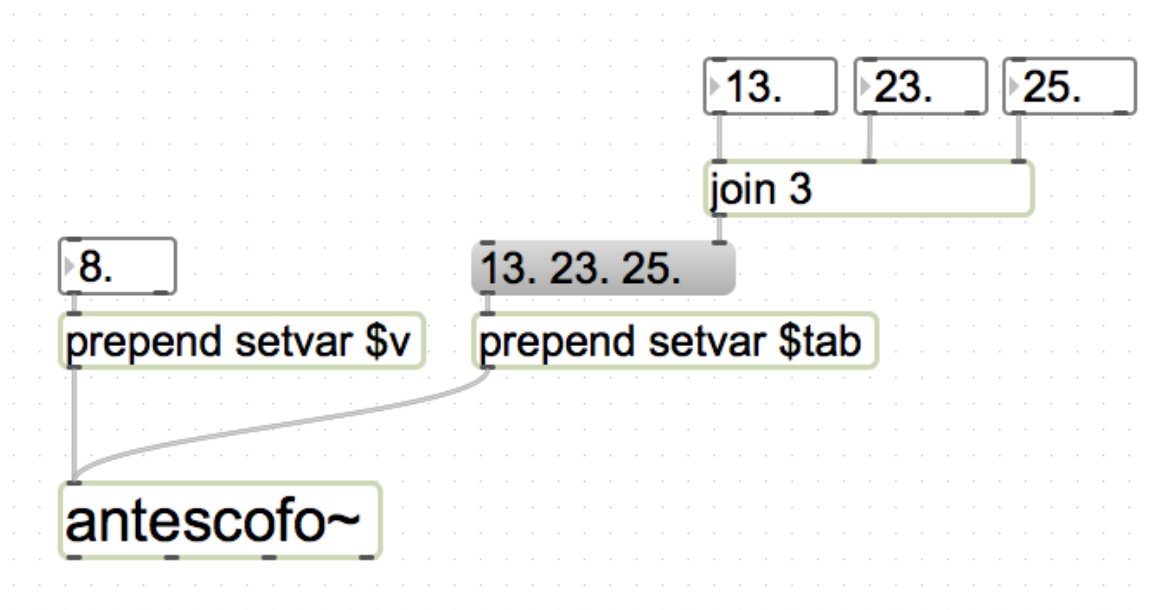


Figure 12.1: Example of `setvar`

Unassignable variables

System variables and special variables cannot be assigned:

```
$BEAT_POS      $DURATION      $ENERGY      $LAST_EVENT_LABEL
```

\$MYSELF	\$NOW	\$PITCH	\$RCNOW
\$RNOW	\$RT_TEMPO	\$SCORE_TEMPO	\$THISOBJ

These variables are *read-only* for the composer: they are assigned by the system during the performance. However, like usual variables, their assignment (by the system) may trigger some activities. See section [Variables and Notifications](#).

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 not fired. On the other hand, [Compound Actions](#) act as containers for others actions and thus span over a duration. Only compound actions can be aborted; however, the action of aborting is atomic (occurs in an instant)². We say that a compound action is *active* when it has been fired but some of its nested actions are still waiting to be fired. Aborting an action only has a visible effect while it is active.

Cancelling an action refers to another notion: the dynamic suppression of an action from the score. This feature is **deprecated** since version 0.9: a conditional action can often be used instead and is at the same time more expressive and more efficient.

Aborting an Action

After a compound action has been launched, it can be aborted, meaning that the nested atomic actions that remain will not be fired.

There is no harm to abort a non-active compound action. So an abort command can be issued multiple times and there is no need to check that the target action is till active.

There are two forms of termination:

```
abort name
abort expression
```

where `name` is the label of an action and `expression` evaluates to an [Exec](#).

Termination through a label

If the abort’s argument is a label, then active actions with that label (and their children) will be aborted. The command has no effect on atomic actions or compound actions with no children.

Above, the plural is used (“active actions”) because one label can be shared by several distinct actions: in this case, all active actions labeled by `name` are aborted together. One action can also occur several times (*e.g.* the body of [loop](#), or the body of a [whenever](#) statement). All occurrences of an action labeled by `name` are aborted.

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

²Because termination is an atomic action, the [abort](#) command is presented here but its understanding presumes some knowledge on [compound actions](#).

Termination through an exec

The argument of `abort` can be an expression. In this case the expression is evaluated and must return an `Exec`. Only this exec will be terminated by the command.

Abort and the hierarchical structure of compound actions

By default, the abort command applies recursively on the whole hierarchical structure of actions (cf. section [Compound actions](#)), even if some actions in the path are not active anymore.

Notice that the actions launched by a process call in a context *C* are considered as descendants of *C*.

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

```
group G1 {
  1 a\ensuremath{_{1}}
  1 group G2 {
    0.2 b\ensuremath{_{1}}
    0.5 b\ensuremath{_{2}}
    0.5 b\ensuremath{_{3}}
  }
  1 a\ensuremath{_{2}}
  1 a\ensuremath{_{3}}
}
2.5 abort G1
```

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

```
2.5 abort G1 @norec
```

then, actions *a*₂ and *a*₃ are aborted but not actions *b*₂ and *b*₃ which will be performed.

Abort handler

`Abort` commands can be issued from everywhere in the code, making difficult to express some dedicated actions to do when the compound action is terminated, actions that are not needed when the compound action reaches its natural end³.

A direct implementation of this behavior is provided by **abort handlers**. An abort handler is a group of actions triggered when a compound action is terminated by an `abort`. Abort handlers are specified using an `[@abort]` clause with a syntax similar to the syntax of the `[@action]` clause of a [curve](#):

³The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

```
CompoundAction ... @abort := { ... }
{
    ...
}
```

An handler can be defined for all compound actions. The scope of the handler is the scope introduced by the compound actions (if any): local variables introduced eventually by the compound action are accessible in the handler.

When a handler associated to a compound action is spanned by an `abort` command, the handler cannot be killed by further `abort` command (in other words, abort handler cannot be aborted).

Notice that `abort` commands are usually recursive, also killing the children spanned by the abort target. If these children have themselves handlers, they will be triggered when terminating the target. However, the order of activation is not specified and can differ from one execution to another.

Notice, if an action is not active, its termination through `abort` does not trigger the execution of its abort handler. However, if the abort is recursive (the default), the abort signal is correctly passed to the (active and non active) childs, until reaching the leaves of the hierarchy.

A Paradigmatic Example

A good example of the use of abort handlers is given by a `curve` that samples some parameter controlling the synthesis of a sound. On some event, the synthesis must be stopped, but this cannot be done abruptly: the parameter must go from its current value to some predetermined value, *e.g.* 0, in one beat. This is easily written:

```
Curve C
  @grain := 0.5
  @action := { print "curve: " $x }
  @abort := {
    print "Curve C aborted at " $x
    Curve AH
      @grain := 0.2
      @action := { print "handler curve: " $x }
      {
        $x { { $x } 1 { 0.0 } }
      }
  }
  {
    $x { { 0.0 } 10 { 10.0 } 10 { 0.0 } }
  }
```

When an `abort` is issued, the curve is stopped and the actions associated to the abort handler are launched. These actions create a new curve with the same control variable `$x`, starting from the current value of `$x` to 0.0 in one beat. A typical trace is (the command is issued at 1.5 beats):

```
print: curve: 0.
```

```

print: curve: 0.5
print: curve: 1.
print: curve: 1.5
print: Curve Aborted at 1.5
print: handler curve: 1.5
print: handler curve: 1.2
print: handler curve: 0.9
print: handler curve: 0.6
print: handler curve: 0.3
print: handler curve: 0.

```

Internal Commands

Internal commands correspond to the MAX or PD messages accepted by the `antescofo~` object in a patch. The “internalization” of these messages as primitive actions makes the control of the MAX or the PD `antescofo~` object possible from within the score itself. For example, it is possible to switch the follower on or off when a specific musical event is reached.

Internal commands are named

```
antescofo::xxx
```

where the suffix `xxx` is the head of the corresponding MAX/PD message recognized by `antescofo~`. The exhaustive list of internal commands is **given below** together with their arguments. But first we focus on a set of very important commands used to navigate the score. These commands are critical in rehearsal or during the design phase, as they evades the linearity of the score.

Controlling the Execution Flow

These commands are also messages (from the Max or PD patch) to the *Antescofo* object (in this case, without the prefix `antescofo::`). They control the execution flow of an *Antescofo* program :

- `antescofo::start` (no argument) Sends initialization actions (before first event) and wait for follower.
- `antescofo::play` (no argument) Simulates the score (instrumental+electronics) from the beginning until the end or until stop.
- `antescofo::startfromlabel` `string antescofo::startfrombeat` `float` Executes the score from current position to position specified by the argument (a label given by a string or a position in beats) in accelerated more WITHOUT sending messages. Then waits for follower (or user input) right before this position.
- `antescofo::scrubtolabel` `string antescofo::scrubtobeat` `float` Executes in accelerated more the score from current position to position specified by the argument WITH sending messages up to (and not including) the specified position. Then waits for follower (or user input). The position is given by a lable (a string) or a position in beats.
- `antescofo::playfromlabel` `string antescofo::playfrombeat` `float` Executes the score from current position to position specified by the argument in accelerated more WITHOUT sending messages, then PLAYS (simulates) the score from thereon.

- `antescofo::playtolabel` string `antescofo::playtobeat` float Play (simulate) score from current position up to position specified by the argument.
- `antescofo::gotolabel` string `antescofo::gotobeat` float Position the system on the position specified by the argument without doing anything else.

List of internal commands

- `antescofo::actions` string inhibits (string = "off") or enables (string = "on") the launch of actions on events recognition. This is different from muting a `track`: muting a track inhibit the sending of some Max/PD messages while this command inhibit the performance of all actions.
-

- `antescofo::add_completion_string` string defines a new completion string to a running and connected `Ascograph` (Mac only).
-

- `antescofo::analysis` int int specifies a new FFT windows length and a new hop size for the audio analysis.
-

- `antescofo::ascographwidth_set` int defines the width of the `Ascograph` window. Look also some parameters of the interface in the inspector of the `Antescofo` MAX object.
-

- `antescofo::ascographheight_set` int defines the height of the `Ascograph` window. Look also some parameters of the interface in the inspector of the `Antescofo` MAX object.
-

- `antescofo::ascographxy_set` int int defines the *x* and *y* position of the window. Look also some parameters of the interface in the inspector of the `_Antescofo_` MAX object.
-

- `antescofo::asco_trace` int turn on (int=1) or off (int = 0) the `Ascograph` tracking of the score position when is `Antescofo` is running in following mode.
-

- `antescofo::before_nextlabel` (no argument) force the progression of the score following up-to the next label, launching the actions between the current point and the next label, but still waiting its occurrence.
-

- `antescofo::bpmtolerance` `float` reserved command.
-

- `antescofo::calibrate` `int` turn calibration mode on (1) or off (1).
-

- `antescofo::clear` (no argument) clear all preloaded scores.
-

- `antescofo::decodewindow` `int` changes the length of the decoding window used in the inference of the position.
-

- `antescofo::filewatchset` `string` (Max only) watch the file whose path is given by the argument, to reload it when changed on disk (the file is supposed to be the source of the current score).
-

- `antescofo::gamma` `float` change an internal parameter of the score following engine.
-

- `antescofo::get_current_score` (no argument) send to a running and connected the source of the current score.
-

- `antescofo::get_patch_receivers` (no argument) reserved command.
-

- `antescofo::getlabels` (no argument) send to the first outlet the list of events label. Correspond to the `get_cues` message accepted by the object.
-

- `antescofo::gotobeat` `float` position the follower at a position specified in `beat` without doing anything else. See below for moving in score commands.
-

- `antescofo::gotolabel` `string` position the follower on an event specified by its label without doing anything else.
-

- `antescofo::harmlist` `float ...` (a list of floats corresponding to a vector) specify the list of harmonics used in the audio observation.
-

- `antescofo::info` (no argument) print on the console output various informations on the version, and the current status of the object. Usefull when reporting a problem.
-

- `antescofo::killall` (no argument) abort all running processes and actions.
-

- `antescofo::mode` `int` reserved command.
-

- `antescofo::mute` `string` `mute` (inhibits the sending) of the messages matched by a track.
-

- `antescofo::nextaction` (no argument) forces the follower to wait for the next event that has some associated actions. The actions triggered between the current position and the new one, are launched.
-

- `antescofo::nextevent` (no argument) forces the follower to wait for the first event that appears after the current position. The actions between the current position and the next event are launched.
-

- `antescofo::nextlabel` (no argument) forces the follower to wait for the next event that has a label. The actions between the current position and the next event are launched. The infered tempo is kept unchanged.

-
- `antescofo::nextlabeltempo` (no argument) same as but the elapsed time is used to adjust the tempo.

-
- `antescofo::nofharm` `int` number of harmonics to compute for the audio analysis.

-
- `antescofo::normin` `float` set some internal parameter of the score following.

-
- `antescofo::obsexp` `float` set some internal parameter of the score following.

-
- `antescofo::pedal` `int` enables () or disables () the use of a resonance model in the audio observation.

-
- `antescofo::pedalcoeff` `float` attenuation coefficient of the pedal model.

-
- `antescofo::pedaltime` `float` attenuation time of the pedal model.

-
- `antescofo::piano` `int` turn the follower in the piano mode (corresponding to a set of parameters adjusted to optimize piano observation).

-
- `antescofo::play` (no argument) simulates the score (instrumental+electronics) from the beginning until the end or until STOP.

-
- `antescofo::playfrombeat` `float` executes the score from current position upto the position given by the argument in accelerated more WITHOUT sending messages, then PLAYS (simulates) the score from thereon.

-
- `antescofo::playfromlabel` *string* executes the score from current position upto then event specified by the argument in accelerated more WITHOUT sending messages, then PLAYS (simulates) the score from thereon.

-
- `antescofo::playstring` *string* interpret the string argument as an sequence of actions and perform it. This command is used by to evaluate a text region highlighted in the editor. It can be used to evaluate on-the-fly actions that have been dynamically generated in an improvisation scenario. Due to OSC limitation, the string size cannot be greather than 1000 characters. But see next command.

-
- `antescofo::playstring_append` *string* this command is used to evaluate on-the-fly a string of size greather than 1000 characters. The sequence of actions must be broken in a sequence of strings each of size less than 1000. Each of these pieces are processed in turn by this command, except for the last one which uses

-
- `antescofo::playtobeat` *float* PLAY (simulate) score from current position up to position specified by the argument.

-
- `antescofo::playtolabel` *string* PLAY (simulate) score from current position up to the event specified by the argument.

-
- `antescofo::preload` *string string* preloads a score and store it under a name (the second argument) for latter use.

-
- `antescofo::preventzigzag` *string* allow or disallow zig-zag in the follower. In non-zig-zag mode, the default, the follower infer only increasing positions (except in the case of a jump). In zig-zag mode, the follower may revise a past inference. This ability does not impact the reactive engine : in any case, actions are performed without revision.

-
- `antescofo::previousevent` (no argument) similar to but looking backward in the score.

-
- `antescofo::previouslabel` (no argument) similar to but looking backward in the score.
-
- `antescofo::printfwd` (no argument) output the formatted print of the current program in a new window.
-
- `antescofo::printscore` (no argument) output the formatted print of the current program in a new window.
-
- `antescofo::read` `string` loads the corresponding score from the file specified by the argument.
-
- `antescofo::report` (no argument) reserved command
-
- `antescofo::scrubtolabel` `string` Executes the score from current position to the event specified by the argument, in accelerated more WITH sending messages. Waits for follower (or user input) right before this position.
-
- `antescofo::scrubtobeat` `string` Executes the score from current position to the position given by the argument, in accelerated more WITH sending messages. Waits for follower (or user input) right before this position.
-
- `antescofo::setvar` `string numeric` `antescofo::setvar` `string string` `antescofo::setvar` `string` `a1 a2 ...` assign the value given by the second argument to the variable named by the first argument. If they are more than two arguments, the a_i are packed into a tab. Using this command, the environment may notify *Antescofo* some information. For instance, *Antescofo* may react because the variable is in the logical condition of a `whenever`. See [external assignment](#).
-

- `antescofo::score` `string` loads the corresponding score (an alias of).

- `antescofo::start` `string` Sends initialization actions (before first event) and wait for follower.

- `antescofo::startfromlabel` `string` Executes the score from current position to position corresponding to in accelerated more WITHOUT sending messages. Waits for follower (or user input) right before this position.

- `antescofo::startfrombeat` `int` Executes the score from current position to the given position in accelerated more WITHOUT sending messages. Waits for follower (or user input) right before this position.

- `antescofo::static_analysis` (no argument) reserved command

- `antescofo::stop` (no argument) stop the follower and abort the runing actions.

- `antescofo::suivi` `int` enables or disables the follower. Even if the follower is off, actions may run and can be spanned and interaction with the environment may happens through and .

- `antescofo::tempo` `float` specify an arbitrary tempo.

- `antescofo::tempoint` `int` reserved command.

- `antescofo::temposmoothness` `float` adjust a parameter of the tempo inference algorithm.

- `antescofo::tune` `float` set the tuning base (default 440.0Hz).

- `antescofo::unmute` `string` unmute (allows the sending) of the messages matched by a track.

- `antescofo::variance` `float` set the variance parameter of the inference algorithm.

- `antescofo::verbosity` `int` specify the level of system messages emitted during execution.

- `antescofo::version` (no argument) print the version of the object and various build information on the console.

Assertion

The action

```
@assert expression
```

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, which is not convenient).

Chapter 13

Compound Actions

Compound actions act as *containers* for others actions. They serve as *temporal* containers: their purpose is to organizes actions in time, rather than in space (that is what data structures are for). They can thus also be seen as *control structures*.

The actions “inside” a container (we also say “nested in” or refer to “child actions”) are specified as a sequence of actions. Successive actions in the sequence are linked with one of the three possible sequencing operators. These operators are described in section [continuation operators](#).

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

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

We first present the [group](#) structure. It gathers several actions logically within a same block that share common properties of tempo, synchronization and error handling strategies. The [group](#) is the basic container: all other compound actions are variations on this structure

- a [loop](#) is a sequential iteration of a group,
- a [curve](#) is also a sequential periodic iteration of a group,
- a [forall](#) is a parallel iteration of a group,
- an [if](#) or a [switch](#) is a choice between several groups to launch, at some point in time,
- a [whenever](#) is a conditional choice in time of the launching of a group,
- a [process call](#) or an [object instantiation](#) is the launching of an abstract group.

All compound actions are guarded by an optional [end clause](#) that limits its scope of execution. The [[@exclusive](#)] attribute may also be specified to prevent the overlapping execution of the same action.

In a sequence of actions, two successive actions are related by a [continuation operator](#).

General Syntax of a Compound Action

{!BNF_DIAGRAMS/compound_actions.html!}

Loop

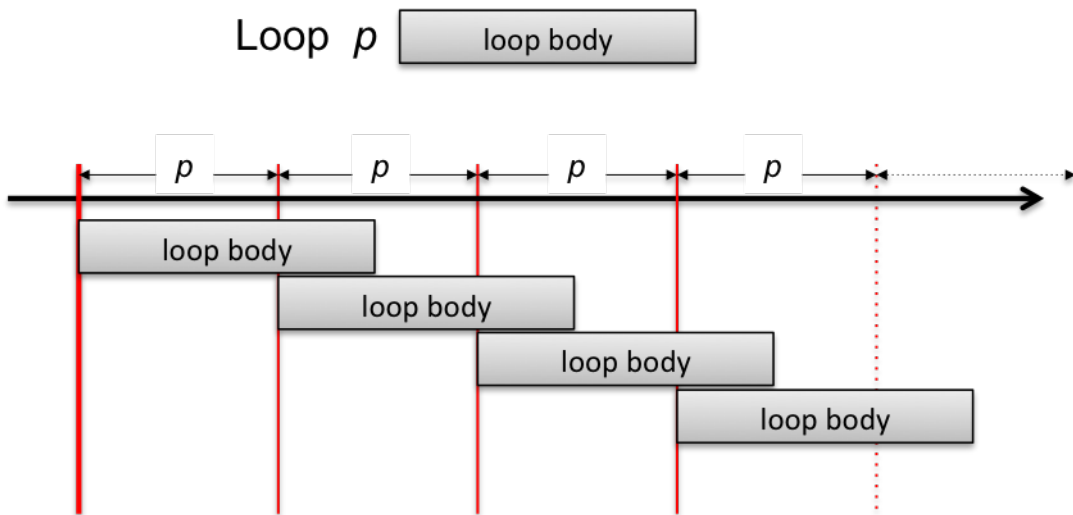


Figure 13.1: loop

ForAll

ForAll $\$x$ in X forall body($\$x$)

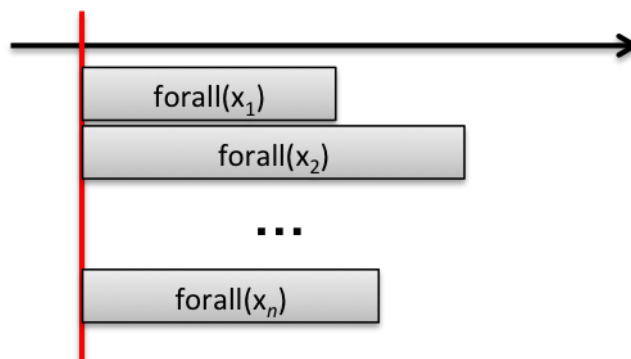


Figure 13.2: forall

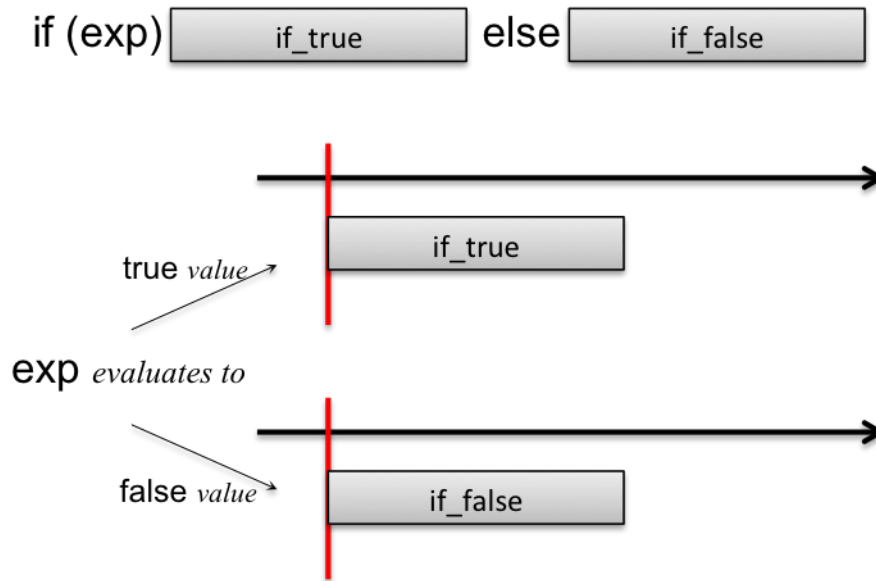


Figure 13.3: if

If

Whenever

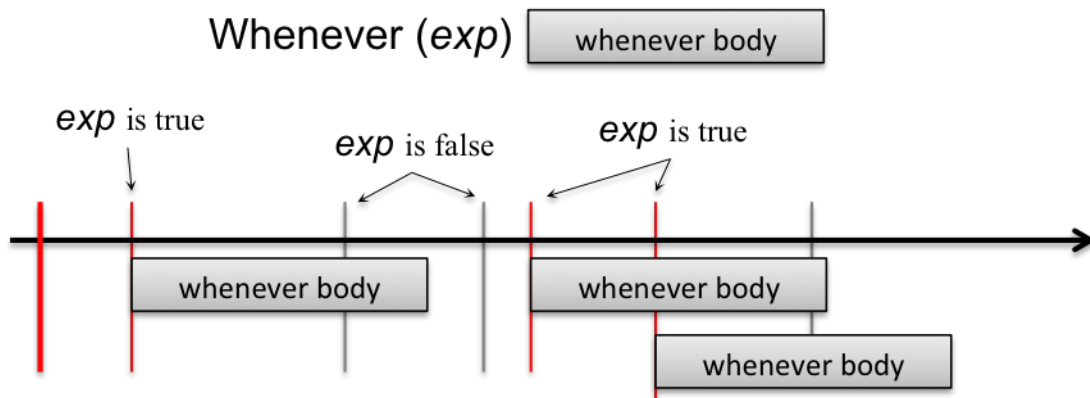
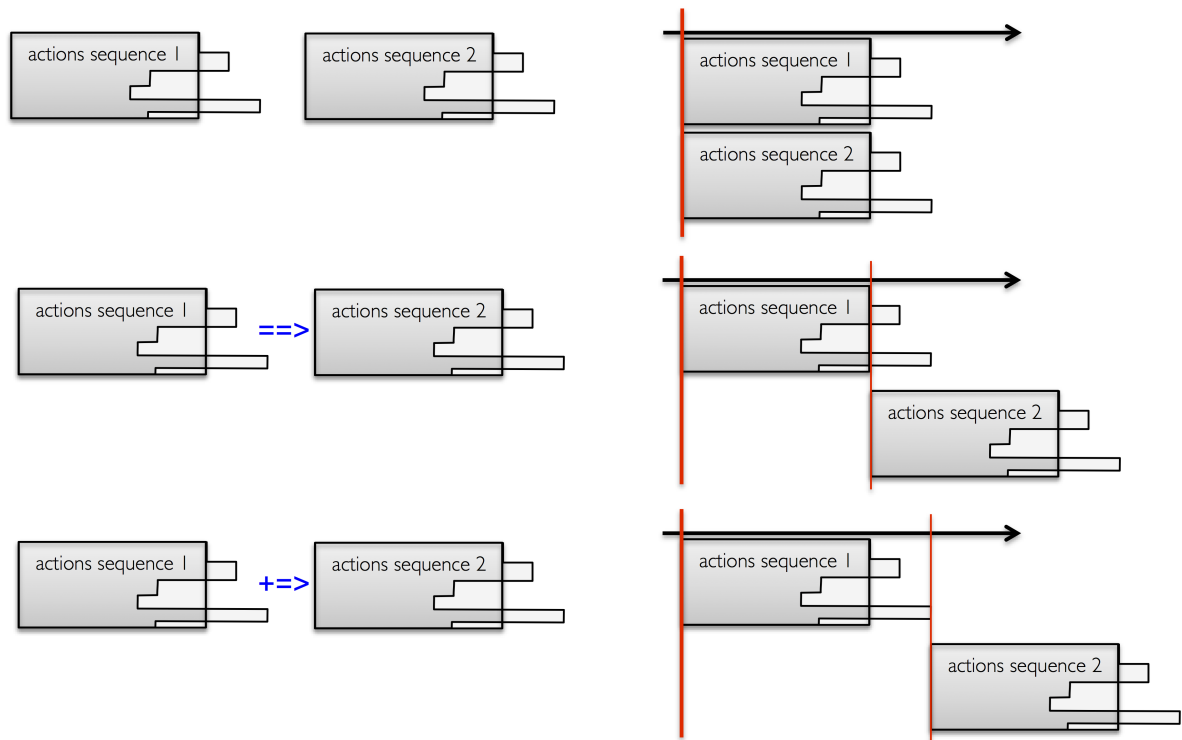


Figure 13.4: whenever

Continuation operators

The three **continuation operators** link two successive actions in a sequence of actions. It is not possible to specify an attribute for the resulting actions, but it is easy to embed this sequence as the body of a group and to specify the attributes for this group.



Group

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

The actions of a group are launched sequentially and the group organizes their precise temporal relationships: *a group is a **timeline** and the actions of the group are actions scheduled on this timeline*. Each group has a **temporal scope** which defines how time passes in the timeline. More generally, a temporal scope can be specified for all compound actions. Temporal scopes are implicitly defined through attributes of compound actions: `[@tempo]` and **synchronization attributes**. This notion will be developed in the next chapter **Management of Time**. In this chapter, we focus on the syntax and the hierarchical structure of compound actions.

The general syntax of a group is defined by the following diagram:

```
{!BNF_DIAGRAMS/group.html!}
```

The specification of the `label`, `Attributes` and `end` clause are optional. The `label` is a simple identifier that acts as a label for the group.

There is a short notation for a group without optional parameters and attributes: its actions are simply be written between braces. For example:

```
action\ensuremath{_{1}}
{ 0.5 action\ensuremath{_{2}} }
action\ensuremath{_{3}}
```

is equivalent to

```

    action\ensuremath{_1}
  Group {
    0.5 action\ensuremath{_2}
  }
  action\ensuremath{_3}

```

Implicit Groups

Some groups are implicit. For example, the actions following an event are members of an implicit group named `top_gfwd_xxx` where `xxx` is a unique number in the score ¹. And all child actions of a compound action take place in a implicit group, often called the *body* of this compound action (*e.g.*, the body of a loop, the body of a process, *etc.*).

By default, the implicit top-level groups are created with the `[@loose]` synchronization strategy. This behavior can be changed in favor of the `[@tight]` synchronization strategy, using the command `top_level_groups_are_tight` at the beginning of the score.

Action Sequence

The actions of a group are arranged in a sequence. Two consecutive actions in this sequence are launched together, in parallel. For instance

```

    let $x := 0
    print $x

```

will be launched in the same instant. **Nevertheless**, actions that occur in the same instant are ordered: this is the [synchrony hypothesis](#). So, in the previous example, a 0 will be printed.

The temporal scope of a group is used to interpret the relative delays that appears optionally in front of an action. For example:

```

    Group G @tempo := 120
  {
    1 action\ensuremath{_1}
    2 action\ensuremath{_2}
  }

```

With the launch of group `G`, the delay of the first action is evaluated into 1, and then nothing happens until 1 beat at tempo 120 is elapsed. At this moment `action1` is launched and the delay preceding `action2` is evaluated. *Etc.*

The quantity of physical time corresponding to a relative delay is specified by the tempo of the group. The way of counting this quantity depends of the synchronization strategy of the group.

The sequencing of actions in a group can be modified using **continuation operators**. The `==>` operator is used to launch an action after the end of the preceding one and `+>` is used

¹The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

to launches at the end of the previous one *including* its children. The end of a group is the launch of the last action in its action sequence (if this action has a delay, the group ends with the start of this delay). For example

```
let $start := $RNOW
Group G
{
  1 action\ensuremath{_{1}}
  2 action\ensuremath{_{2}}
  Group GG
  {
    1 action\ensuremath{_{3}}
    1 action\ensuremath{_{4}}
  }
}
==> print OK ($start - $RNOW)
```

will print OK 3 ([\\$RNOW](#) gives the relative time) because G ends at soon as GG is started and GG is started with `action2`. But, if the continuation operator `==>` is changed for `+>`:

```
let $start := $RNOW
Group G
{
  1 action\ensuremath{_{1}}
  2 action\ensuremath{_{2}}
  Group GG
  {
    1 action\ensuremath{_{3}}
    1 action\ensuremath{_{4}}
  }
}
+> print OK ($start - $RNOW)
```

then OK 5 will be printed because the `+>` operator will execute the print message at the end of all actions spanned directly or indirectly by G.

These features will be discussed more in depth in chapters [continuation](#) and [temporal scope](#).

The Nested Structure of Groups

Groups, and more generally compound actions, can be nested arbitrarily. We illustrate below the nesting of groups specified by

```
Group G
{
  action\ensuremath{_{0}}
  1 action\ensuremath{_{1}}
  1 Group G1
  {
```

```

        action\ensuremath{_{2}}
    1 action\ensuremath{_{3}}
        action\ensuremath{_{4}}
    }
2 Group G2
{
    3 action\ensuremath{_{5}}
        action\ensuremath{_{6}}
    }
action\ensuremath{_{7}}
Group G3
{
    action\ensuremath{_{8}}
    Group G31
    {
        2 action\ensuremath{_{9}}
            action\ensuremath{_{1}}\ensuremath{_{0}}
        }
    action\ensuremath{_{1}}\ensuremath{_{1}}
}
}
}

```

as a tree making explicit the father/child relationships

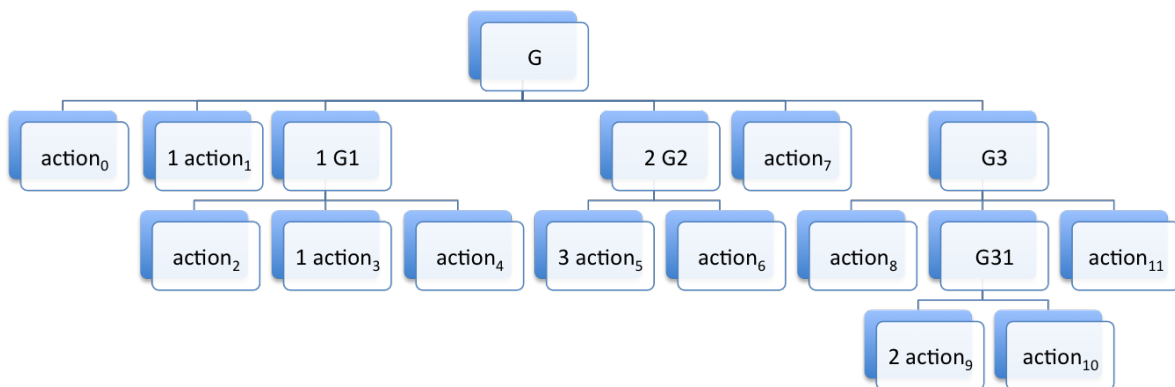


Figure 13.5: nested structure of a group

and as a timeline showing the temporal organization

In this last diagram, the width of an action $action_i$ (abbreviated a_i) is not relevant. A group is pictured as a rectangle containing its childs but this is merely a graphical convention: a group ends with the start of its last action.

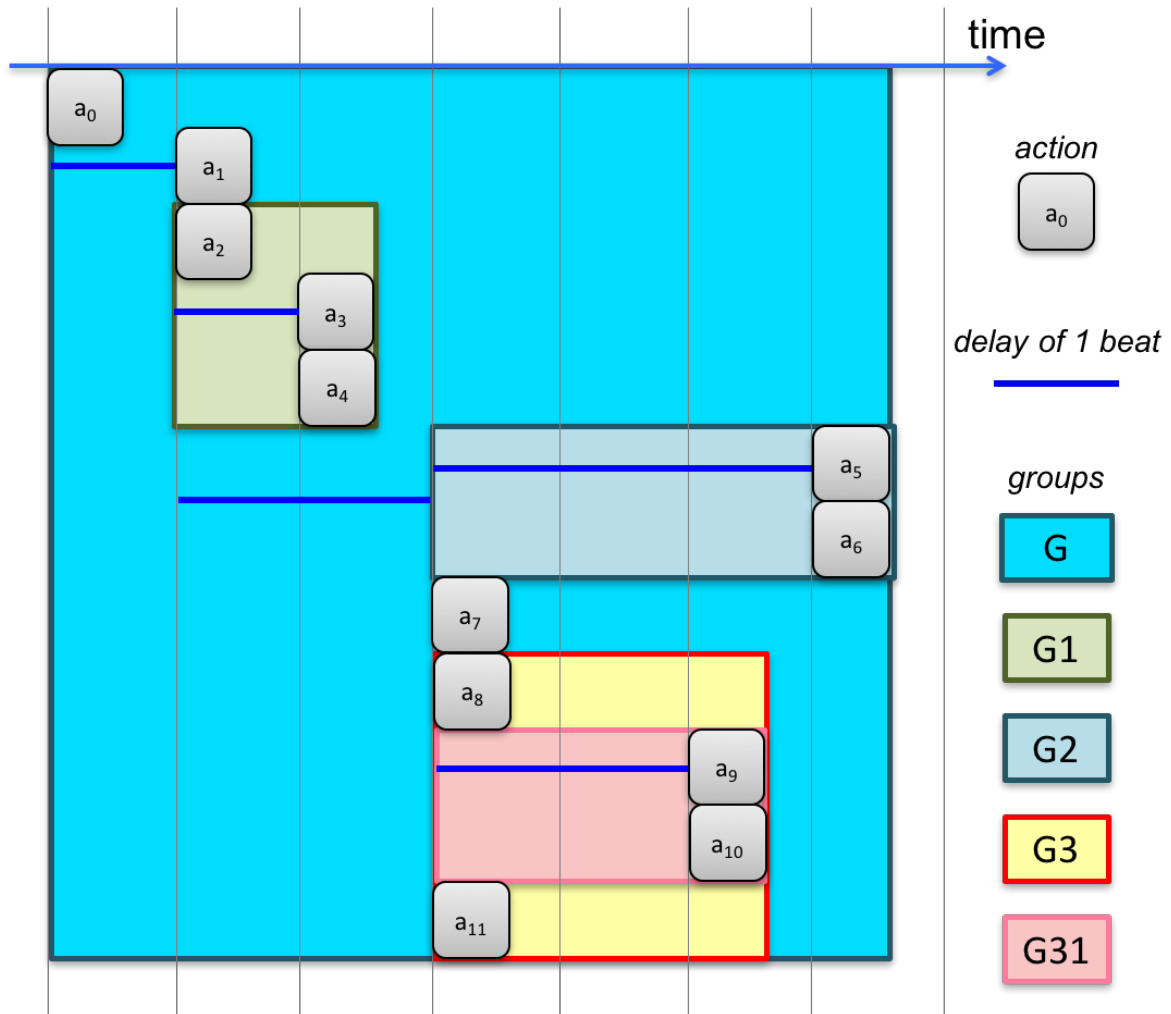


Figure 13.6: exemple temporal organization of nested groups

Instances of a Group

A group is related to either an event or another action. When the event occurs or the action is triggered, the group waits for the expiration of its delay before sequentially launching the actions that comprise it. We say that an **instance** of the group is created and launched. The instance is considered *alive* while there is an action of the group waiting to be launched. In other words, an instance expires when the last action of the corresponding group is performed. It is possible to refer to the instance of a group through a special kind of values called [Exec](#).

We make a distinction between the group and its instances because several instances of the same group can exist and can even be alive simultaneously. Such instances are created by `loop`, `forall`, *etc.* These constructions are described in the rest of this chapter.

Local variables

Variables local to a sequence of actions can be declared using the `[@local]` keyword. A `[@local]` declaration is not an action, and can appear anywhere in the sequence. The introduced variable is:

- local to each instance of the action sequence (two instances do not share the variable);
- its scope (where the variable's name is recognized) is the whole sequence where it is defined and all enclosed actions;
- and its lifetime (when the variable can be read and written) is the lifetime of the sequence and its children. The variable ceases to exist once the last nested action has expired.

Notice that a local variable can be safely accessed by a child action, even if the group where it has been defined has expired.

See section [Variables](#) for further information.

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,
- using a [until](#) (or a [while](#)) *logical clause*,
- using a [during](#) *temporal clause*

Note that when the name of a group is used in an [abort](#) action, all alive instances of this group are killed. It is possible to kill a specific instance using the [exec](#) that refers to this instance.

The two last mechanisms are called **end clauses**.

The `until` and the `while` Clause

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

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

The word `DONE` will never be printed because the group is aborted when `$x` becomes true. More exactly the expression `$x` is checked each time a action must be launched. And if true, the group is terminated instead of proceeding with the action. So, with the following program:

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

the word `DONE` will be printed even if the `$x` variable has been set to `true`. As a matter of fact, at date 3 beats, the variable is `false` again (notice that in `{ ... } Group G { }` there are two groups that are spanned in parallel).

There is another way to represent the `until` keyword: using the contrary `while` statement. Thus,

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

is equivalent to

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

The `during` Clause

A `during` clause specifies **temporal validity**, *i.e.* the time a group is active. When this time is exhausted, the group is aborted. This time can be specified

- in beats (relative time): `[d]`
- in (milli-)seconds (absolute time): `[d s]` or `[d ms]`
- or in number of logical instants: `[d #]`.

For instance:

```

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

```

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

The notation of a duration follows the notation used for the access to the [history of a variable](#). So

```

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

```

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

```

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

```

will execute the group only once. For example

```

Group GG
{
    print GG 1
    1 print GG 2
    1 print GG 3
    1 print GG 4
    1 print GG 5
    1 print GG 6
    1 print GG 7
} during [4#]

```

will print:

```

GG 1
GG 2
GG 3
GG 4

```

because 4 logical instants after its activation, the group GG is aborted.

This last logical duration may seems useless for a group, but is very convenient to specify the number of iterations of a [loop](#) or the maximal number of triggering of a [whenever](#).

The @abort clause

Every compound action may have an [abort handler](#) specified through the `[@abort]` attribute. The [abort handler](#) is a sequence of actions performed when the compound action is terminated via an abort. It is not performed when the group reaches its natural end or if it is terminated via an end clause.

The scope of the handler is the scope introduced by the compound actions (if any): local variables introduced by the compound action are accessible in the handler.

For example

```
Group G @abort { print "DONE" }
{
    print G 0
  1 print G 1
  1 print G 2
  1 print G 3
}
1.8 abort G
```

will print:

```
G 0
G 1
DONE
```

A typical example of an abort handler is illustrated in [section abort handler](#): they are used to stop a curve arbitrarily with a “fade” leading the parameter to reach a final value irrespectively of its value when the abort occurs.

The @exclusive Clause

The last figures of the [previous section](#) show that *multiple instances of the same group* spanned by a compound action may overlap in time. Sometimes it is necessary to avoid this behavior: this can be achieved using an `@exclusive` attribute on the compound action.

The effect of `@exclusive` is to [abort](#) any previous instances of the group (if they are still active) when a new instance is triggered. The termination includes the eventual children of the previous instances. [Abort handlers](#) are activated if there are any. The new instance is triggered after the termination process.

Synchronization Attributes

[Synchronization strategies](#) like `@loose` and `@tight`

```
group ... @loose ... { ... }
group ... @tight ... { ... }
```

and [error strategies](#) like `@local` and `@global`

```

group ... @global ... { ... }
group ... @local ... { ... }

```

can be specified for a group and also for every compound action 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 easily create nested hierarchies (groups inside groups) sharing homogeneous behaviors.

Synchronization strategies are described in chapter [Synchronization Strategies](#).

Local Tempo

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

```

group G @tempo := exp ... { ... }

```

exp is an arbitrary expression that defines the passing of time for the delay of the actions of that are expressed in relative time in the group, see chapter [Management of Time](#).

With a local tempo, you can create, for example, an accelerando. In the next example, we use a variable to specify a local tempo and we control this variable with a curve (see [Curve](#)). That way, we can write a group where all durations are equal. It's the variation of the local tempo variable that creates the accelerando.

```

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

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

```

Loop: Sequential Iterations

{!BNF_DIAGRAMS/loop.html!}

The loop construction

```

loop optional_label period { loop_body }

```

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

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

will print:

```
0
0.5
1
1.5
...
```

The instances of the loop body are evaluated as independent groups. So, if the period is shorter than the duration of the body of the loop, successive iterations will overlap:

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

will print:

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

Here, when the body of the loop is instantiated, the global variable `$i` is copied in the local variable `$j`: `$i` can then be updated to count the iteration but `$j` records the iteration number for a given loop body. The loop period is 1 beat and the duration of the body is 2 beats. So two successive instances of the loop body overlap and their printing are interleaved. Notice that the local variable is local to a loop body instantiation (they are as many `$j` as concurrent loop bodies).

The overlapping of two iterations of the loop body can be avoided, see `[@exclusive]` below.

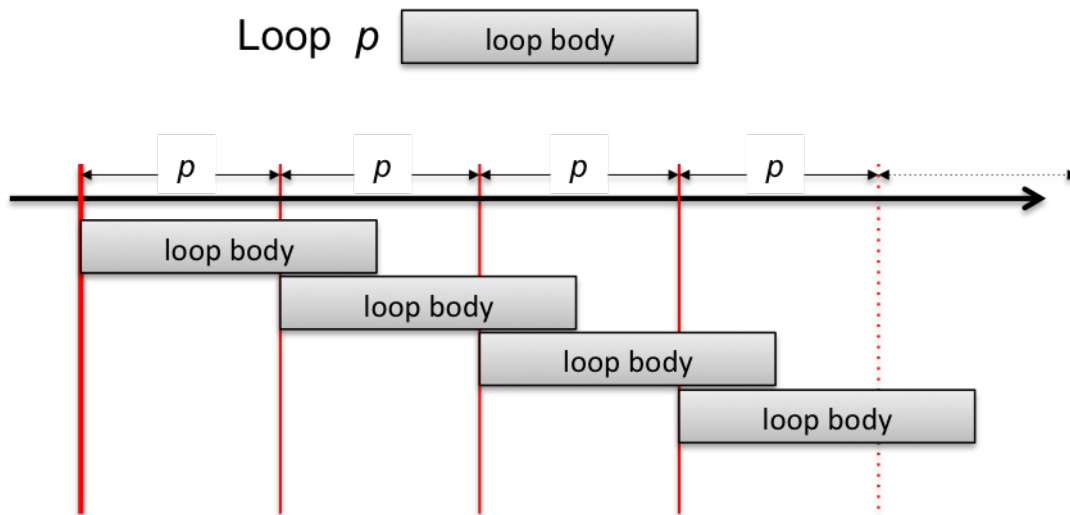


Figure 13.7: loop

Loop Period

The period of a loop is an expression *evaluated at each iteration* and is used to schedule the next iteration. So the duration between two iterations can change as the time progress and the iterations are not necessarily periodic.

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

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

will print:

```
0
1
3
6
10
15
...
```

When the loop is launched at time 0 second, the body is also launched for the first time and, in parallel, the next iteration is scheduled with the current value of the period (which at this

time is 1 second). A 0 is printed. After 0.5 seconds, the variable `$period` is incremented. At date 1 second, the period for the next iteration is evaluated (to 2) and the second iteration is launched (printing a 1). So after 1+2 seconds, the third iteration takes place and print a 3, etc.

In addition, the period expression can evaluate to a **tab** (*i.e.*, a vector): in this case, the elements of the vector are the successive periods of the loop. Note that the periods are taken cyclically in the vector. The `s` or `ms` qualifier can be used to specify that the tab elements are given in absolute time instead of relative time:

```
$p := [100, 200, 400, 800]

Loop $p ms
{
    print $NOW
}
```

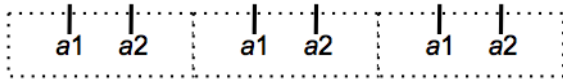
will print:

```
0
0.1
0.3
0.7
1.5
1.6
...
```

Stopping a Loop

The optional **until** or **while** clauses are evaluated at each iteration and eventually stop 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
{
    $cpt := $cpt + 1
    0.5 a\ensuremath{_{1}}
    0.5 a\ensuremath{_{2}}
}
until ($cpt >= 3)
```

The previous loop can also be written using a [during](#) clause. Logical times corresponds to loop iterations, so:

```
loop L 1.5
{
    0.5 a\ensuremath{_{1}}
    0.5 a\ensuremath{_{2}}
}
during [3#]
```

is equivalent to the previous loop. Because the loop period is 1.5, three loop iteration will last 4.5 beats, so it can be also written:

```
loop L 1.5
{
    0.5 a\ensuremath{_{1}}
    0.5 a\ensuremath{_{2}}
}
during [4.5]
```

If an [end clause](#) is not provided, the loop will continue forever but it can be killed by an explicit [abort](#) command:

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

In the case above, OK will only be printed three times.

Instantaneous Iteration

A period of zero (in relative or absolute time) is perfectly legal: all iterations take place in the same instant:

```
Loop 0ms
{
    print "OK at " $NOW
} during [100#]
```

will print 100 times OK at xxx at date xxx.

Instantaneous iterations can be used for instance to perform computations on a data-structure (but see also the [iteration expression](#) allowed in function definitions).

However, an infinite loop with a zero period implies to perform an infinite number of computations in finite time, which is not possible. For this reason, there is a run-time security: if there is no `end clause`, the run-time aborts the loop if the number of successive iterations with a period of zero reaches a predefined limit of 10000.

Avoiding Overlapping Iterations: `[@exclusive]`

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

```

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

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

2 antescofo::killall

```

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

```

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

```

```

iteration 1 at 0.0
iteration 1 at 0.25
iteration 1 at 0.5
iteration 1 at 0.75
iteration 2 at 1.0
iteration 1 at 1.0
iteration 1 at 1.25
iteration 2 at 1.25
iteration 1 at 1.5

```

```

iteration 2 at 1.5
iteration 1 at 1.75
iteration 2 at 1.75
iteration 2 at 2.0

```

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

See also the section [Priority](#) for the management of actions that take place at the same date.

Synchronization Attributes of a Loop

The loop body is an implicit group and the instances of the loop body are childs of the loop. So, synchronization attributes, like `[@tempo]`, defined at the loop level, are inherited by them.

Parallel Iterations

{!BNF_DIAGRAMS/forall.html!}

The `loop` construction iterates the triggering of a group (the loop body): one body instance is triggered after the other, with a given interval (the loop period). The action `ForAll` (for *parallel iteration*) instantiates a group *in parallel* for each element in an iteration set. The simplest example is the iteration on the elements of a `tab`:

```

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

```

will trigger in parallel a group for each element in the tab referred by `$t`. For each group, the *iterator variable* `$x` takes the value of its corresponding element in the tab. It is implicitly a local variable, not visible outside the `ForAll` body.

The result of this example is to print in sequence

```

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

```

The general form of a parallel iteration is:

```

forall $var in expression
{
    ; action sequence
}

```

where `expression` evaluates to a `int`, a `tab` or a `proc`:

- If the iteration set n is an `int`, the values of the iterator are the integers $0, \dots, (n - 1)$ if n is positive, and $(n + 1), (n + 2), \dots, 0$ if n is negative.
- If the iteration set is a `tab`, the values of the iterator are the `tab`'s elements.
- If the iteration set is a `proc` or an `obj`, the values of the iterator are the `exec` that correspond to the `proc`'s or `obj`'s instances.

Parallel iterations also accept a `map` for the iteration set. In this case, the syntax introduces two variables to refer to the keys and the values in the map:

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

will print:

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

There is also a parallel iteration expression allowed only in the context of a function definition, see the section [extended expressions](#).

Curve (continuous action)

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

We introduce Curves² starting with a simplified and familiar syntax of linear interpolation and move on to the complete syntax and showcase details of Curve construction.

Simplified Curve Syntax

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

²The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

Curve level 0.0, 1.0 2.0 s

In this example, the action constructs a line starting at 0.0, going to 1.0 in 2.0 seconds and sending the results to the receiver object level1. The initial point 0.0 is separated by a comma from the destination point. The destination point consists of a destination value (here 1.0) and the time to achieve it (2.0s in this case). At each sampling point x , a message level x is sent to the environment.

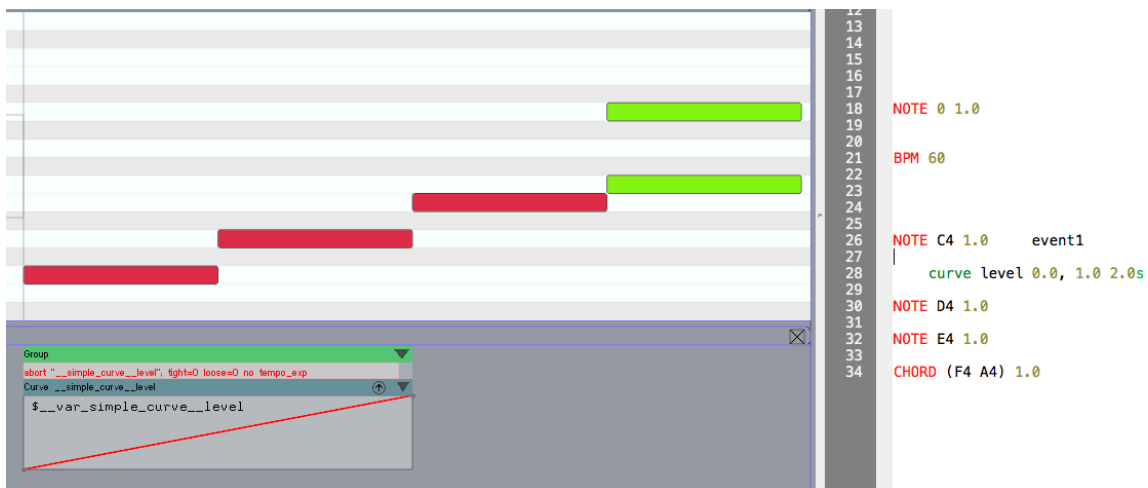


Figure 13.8: Simplified Curve syntax and its realisation in Ascograph

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

Curve level 0.5 1.0

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

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

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

The simplified curve is thus very similar to `line` object in Max or PD. That said, it is important (and vital) that the first call to *Simplified Curve* have an initial value. Otherwise, the departing point is unknown and you risk receiving *NaN* values (*not-a-number*) until the first destination point!

To summarize, starting a simplified curve is written:



Figure 13.9: Chaining Simplified Curve

```
curve max_receiver starting_point , final_point duration
```

and chaining a simplified curve is written:

```
curve max_receiver final_point duration
```

```
{!BNF_DIAGRAMS/simplified_curve.html!}
```

where `cexp` are closed expressions.

The action fired at the sampling point of a simplified curve is restricted to be a message with only one argument: the sampled value. Replacing the receiver with a bracketed `@command` construct, it is possible to have more general messages and even to compute the receiver of the message:

```
curve @command { receiver arg\ensuremath{_{0}} arg\ensuremath{_{1}} } start
```

where `receiver` and the `argi` are [closed expressions](#), will send the message:

```
@command(receiver) arg\ensuremath{_{0}} arg\ensuremath{_{1}} x
```

for each sampling point `x` of the curve (see the `[@command]` keyword for computing the receiver of a message). The chained version is similar:

```
curve @command { receiver arg\ensuremath{_{0}} arg\ensuremath{_{1}} } final
```

The simplified command hides several important properties of Curves from users and are there to simplify calls for simple linear and scalar interpolation. For example, the time-step for interpolation in the above curve is 30 milli-seconds and hard-coded. A complete [curve](#) allows for adjustment of such parameters, several kinds of multi-dimensional interpolations, complex actions, and more. From here we will detail the complete curve syntax.

Full Curve Syntax

A curve iterates a sequence of actions (specified with the [`@action`] attribute) on each sampling point (defined by the [`@grain`] attribute) of a [piecewise function](#). The piecewise function is defined by multiple sub-functions, each sub-function applying to a certain interval defined by **breakpoints**. Between two breakpoints, the function is defined by an interpolation type, the delay between the breakpoint and the value of the general function at the breakpoints.

Piecewise functions defined this way are also called **breakpoint functions** or **BPF**. They are implemented in *Antescofo* as [nim](#). [Nim](#) offers a powerful data structure to compute Piecewise functions and the curve construction acts as a **nim player** by sampling the [nim](#) in time.

In this section, we mainly discuss the curve construction that directly embeds the specification of the underlying BPF.

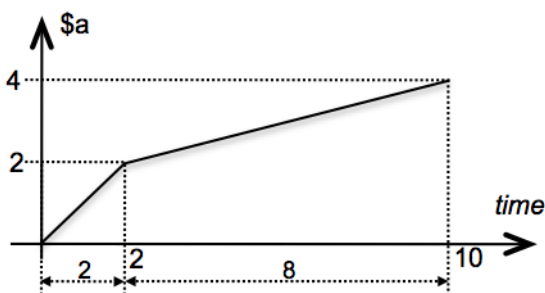
Scalar Curve

The example below shows a simple curve defined in two pieces. The Curve has the name C and starts at a value of 0. Two beats later, the curve reaches 2 and ends on 4 after 8 additional beats. Between the breakpoints, the interpolation is linear, as indicated by the string "linear" after the keyword `@type`. Linear interpolation is the default behaviour of a curve (hence it can be dismissed).

```

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

```



In the above example, variable `$a`, called the **curve parameter**, ranges over the curve. Its value is updated at a time-rate defined by attribute `@grain` which can be specified in absolute time or in relative time. Each time `$a` is updated, the `@action` sequence, which can refer to `$a`, is triggered.

Vectorial Curve

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

```

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

```

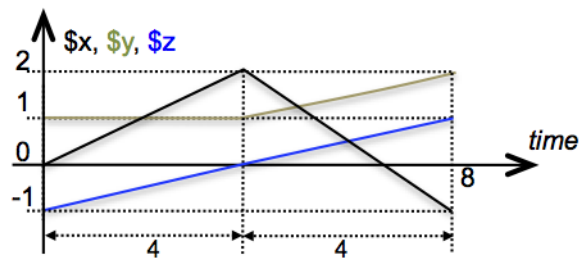


Figure 13.10: image

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

```

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

```

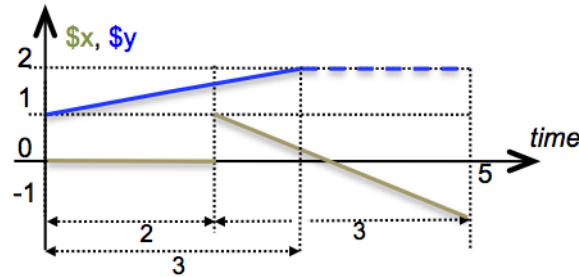



Figure 13.11: image

In the above example, curve parameters $\$x$ and $\$y$ have different breakpoints. The breakpoint definition on $\$x$ shows how to define a sudden change on a step-function with a zero-delay value. Incidentally, note that the result is not a continuous function on $[0, 5]$. The parameter 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 $\$x$.

The figure [1] below shows a simple 2-dimensional vector curve on *Ascograph*. Here, two variables $\$x$ and $\$y$ are used to sample the curve and are referenced in the action. They share the same breakpoints but can be split within the same curve. The curve is also aborted at event2 when the `abort` command is called with the curve's name.

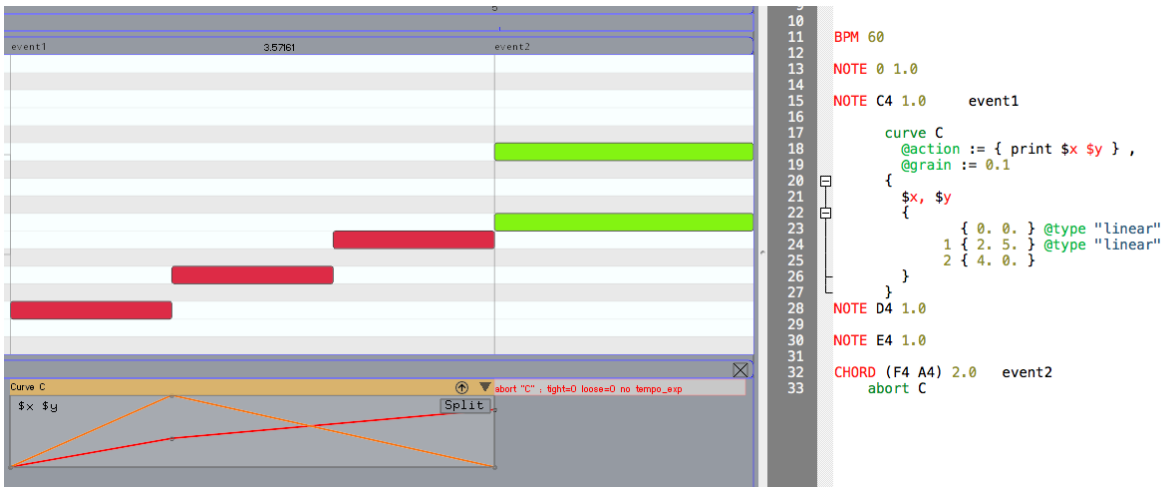


Figure 13.12: image

Editing Curve with Ascograph

Using *Ascograph*, you can graphically interact with curves, as long as the curve parameters are constant. If this is the case, it is possible to move breakpoints vertically (changing their values) and horizontally (time position) by mouse, assigning new interpolation schemes graphically (control-click on breakpoint), splitting multi-dimensional curves and more.

For many of these operations on multi-dimensional curves, each coordinate should be represented separately. This can be done by pressing the button on the Curve box in

which will automatically generate the corresponding text in the score. Each time you make graphical modifications on a curve in *Ascograph*, you need to press *APPLY* to regenerate the corresponding text.

The figure below shows the curve of the previous example [1] embedded on the event score, split and in the process of being modified by a user.

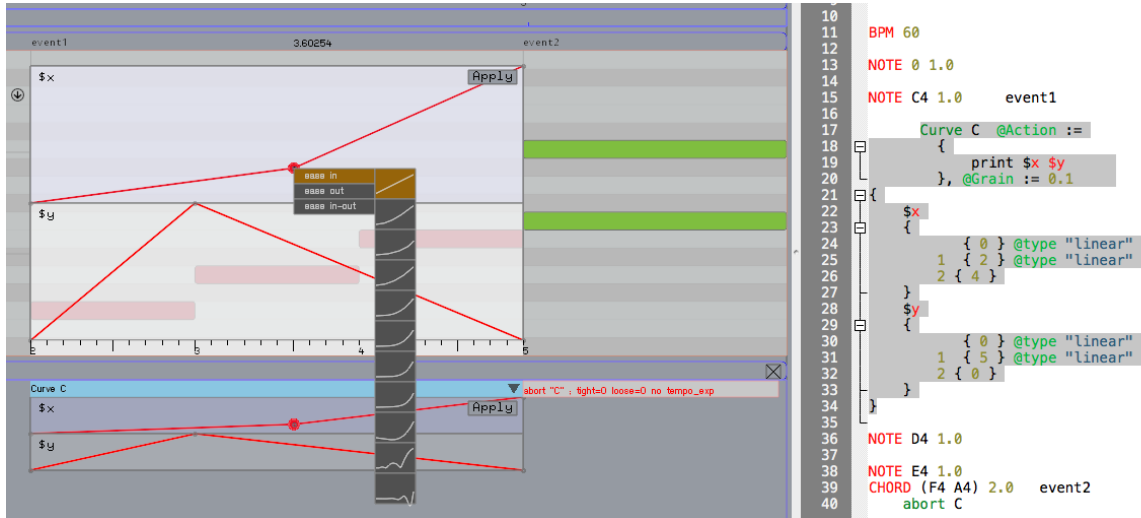


Figure 13.13: image

In the following sections we will get into details of Curve attributes: namely [*@action*], timing [*@grain*], and interpolation methods [*@type*]. But before that, we will describe the textual syntax. Knowing the textual syntax is important when defining curve whose parameters are defined by full expressions.

Textual Definition of a Full Curve

{!BNF_DIAGRAMS/full_curve.html!}

Actions Fired by a Curve

Each time a parameter is assigned, the action specified by the attribute [*@action*] is also fired. The value of the attribute is a sequence of actions. Usually, it is a simple message but arbitrary actions are allowed, for instance :

```
curve C
@action := {
    print $y
    2 action\ensuremath{$_1} $y
    1 action\ensuremath{$_2} $y
}
{ ... }
```

At each sampling point, the value of *\$y* is immediately sent to the receiver *print*. Two beats later *action₁* will be fired and one beat after that *action₂* will be fired.

This sequence of actions is an implicit group and cannot have attributes, but a `group` can be nested for that end:

```

curve C
@action := {
  Group @tempo := 120
  {
    print $y
    2 action\ensuremath{_{1}} $y
    1 action\ensuremath{_{2}} $y
  }
}
{ ... }

```

If the `@action` attribute is absent, the curve simply assigns the parameters specified in its body. This can be useful in conjunction with other parts of the code if the parameters are referred in expressions or in other actions. In the next example, a curve is used to dynamically change the tempo of a `loop`

```

Curve
@grain := 5ms
{
  $x { {60} 10 {120} }
}

Loop 1
@tempo := $x
{
  print loop $NOW
}
until ($x >= 105)

```

will print:

```

0.0
0.954669
1.83255
2.64963
3.41704
4.14287
4.83321
5.49282
6.12547
6.73421
7.32156

```

Grain, Duration and Breakpoints Specifications

In the previous example, the time step (the sampling rate of the curve) called *grain size* and specified with the `@grain` attribute, is expressed in absolute time while breakpoints' durations are expressed in relative time. However, durations and grains can be freely expressed in absolute or relative time.

The grain size can be as small as needed to achieve perceptual continuity. However, in the MAX/PD environments, one cannot go below 1ms (the temporal precision of the host).

The grain specifies only a maximal duration between two sampling points. This freedom is used by *Antescofo* to ensure that the actions fired by the curve will be fired for each breakpoints boundaries.

Grain size and duration, as well as the values at breakpoints, can be [closed expressions](#) too. Grain size is evaluated at each sampling point, which makes it possible to change dynamically the time step.

The values of the breakpoint are evaluated once: when the curve is fired.

Curve Playing a NIM

A single value can be used as an argument of the curve parameter. In this case, the expression is expected to evaluate to a [NIM](#), allowing the user to dynamically build breakpoints and their values as a result of computation. The syntax has already been described:

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

defines a curve where the breakpoints are taken from the value of the expression `e`. This expression is evaluated when the curve is triggered and must return a [nim](#) value. The [NIM](#) is used as a specification of the breakpoints of the curve. Notice that, when a NIM is “played” by a curve, the first breakpoint of the NIM coincides with the start of the curve.

For example

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

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

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

$stab := [ $nim1, $nim2, ..., $nim10 ]
```

```

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

```

A typical situation is to play a NIM chosen from a repertoire of NIMs in a specified time interval `$dur`. In this case, directly playing the NIM with the curve is not appropriate, because the NIM will be played with its natural length. Fortunately, processes like `NIMplayer` make the desired behavior easy to code.

```

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

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

@proc_def ::NIMplayer($NIM, $dur)
{
  curve readNIM
  @grain := 0.02s,
  @action := { print ($NIM($x)) }
  {
    $x
    {           { (@min_key($nim)) }
      $dur     { (@max_key($nim)) }
    }
  }
}

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

```

The playing of the NIM is controlled by a curve. The functions `[@min_key]` and `[@max_key]` are used to get the definition interval of the nim `$nim`.

Interpolation Methods

The specification of the interpolation between two breakpoints is given by an optional string. The keyword `@type` is mandatory only when a variable is used to specify the interpolation type. Using a variable makes it possible to compute the interpolation type, *e.g.* when a curve is embedded in a process and there is a need to parameterize the interpolation type.

A linear interpolation is used by default. *Antescofo* offers a rich set of interpolation methods, mimicking the standard **tweeners** used in flash animation³. There are 10 different types:

³[Inbetweening or tweening](#) is the process of generating intermediate frames between two images to give

- *linear, quad, cubic, quart, quint*: which correspond to polynomial of degree respectively one to five;
- *expo*: exponential, *i.e.* $\alpha e^{\beta t + \delta} + \gamma$
- *sine*: sinusoidal interpolation $\alpha \sin(\beta t + \delta) + \gamma$
- *back*: overshooting cubic easing $(\alpha + 1)t^3 - \alpha t^2$
- *circ*: circular interpolation $\alpha \sqrt{(\beta t + \delta)} + \gamma$
- *bounce*: exponentially decaying parabolic bounce
- *elastic*: exponentially decaying sine wave

With the exception of the linear type, all interpolations types come in three “flavors” traditionally called **ease**:

- **in** (the default) which means that the derivative of the curve is increasing with the time (usually from zero to some value),
- **out** when the derivative of the curve is decreasing (usually to zero),
- and **in_out** when the derivative first increases (until the midpoint of the two breakpoints) and then decreases.

The corresponding interpolation keywords are listed below and illustrated in the next figures. Note that the interpolation can be different for each successive pair of breakpoints. These interpolation methods are also available for [NIM](#) (but [NIM](#) includes a richer set of interpolation types).

```

"back"
"back_in"
"back_in_out"
"back_out"

"bounce"
"bounce_in"
"bounce_in_out"
"bounce_out"

"circ"
"circ_in"
"circ_in_out"
"circ_out"

"cubic"
"cubic_in"

```

the appearance that the first image evolves smoothly into the second image. The page [Tweeners](#) illustrates the standard [tweens](#) to control the successive positions of a point, illustrating the use of tweens to control the apparent speed and to achieve different qualities of movement.

```

"cubic_in_out"
"cubic_out"

"elastic"
"elastic_in"
"elastic_in_out"
"elastic_out"

"exp"
"exp_in"
"exp_in_out"
"exp_out"

"quad"
"quad_in"
"quad_in_out"
"quad_out"

"quart"
"quart_in"
"quart_in_out"
"quart_out"

"quint"
"quint_in"
"quint_in_out"
"quint_out"

"sine"
"sine_in"
"sine_in_out"
"sine_out"

```

Programming an Interpolation Method.

If your preferred interpolation method is not included in the list above, it can be easily programmed. 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 }
}

```

```
}
```

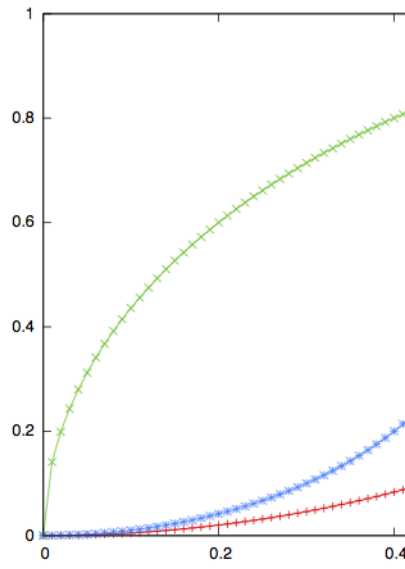
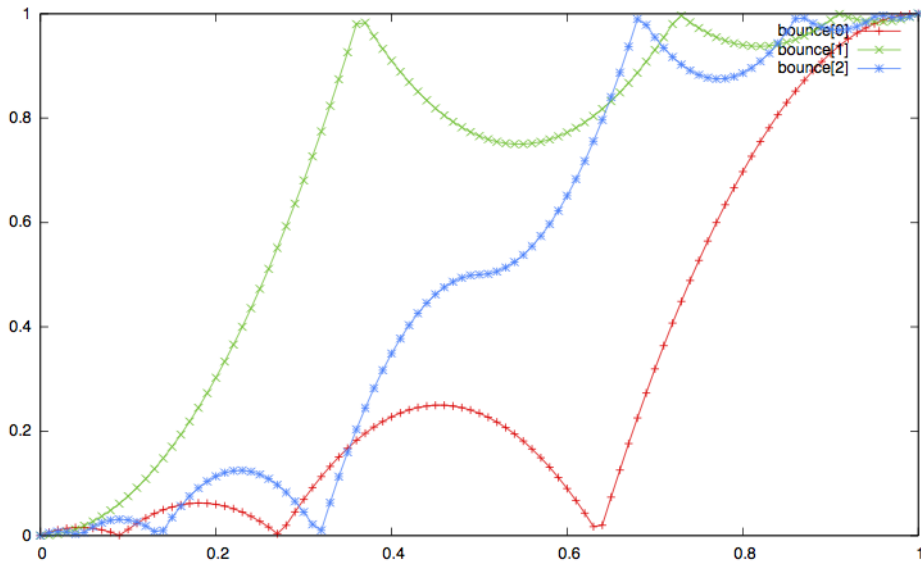
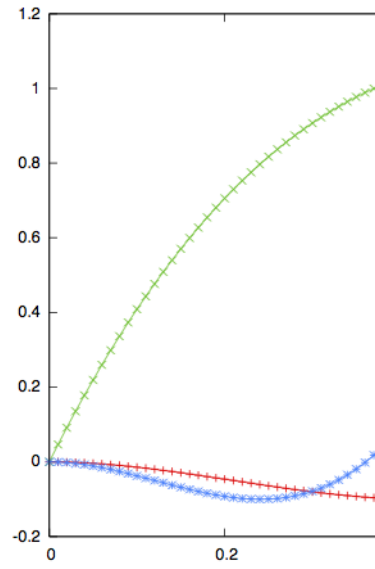
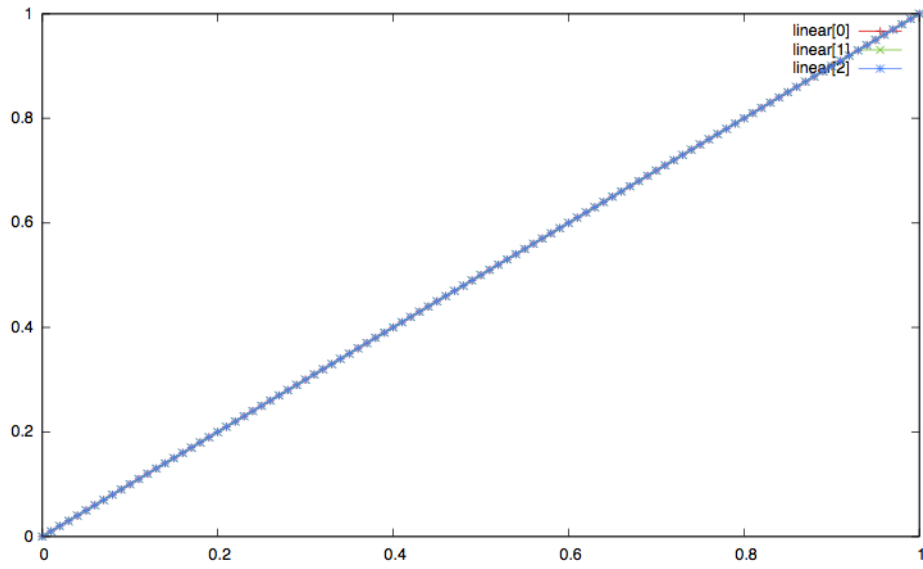
The curve will interpolate function `@f` between 0 and 1 after it starts, over the course of one second and with a sampling rate of 0.1 beats.

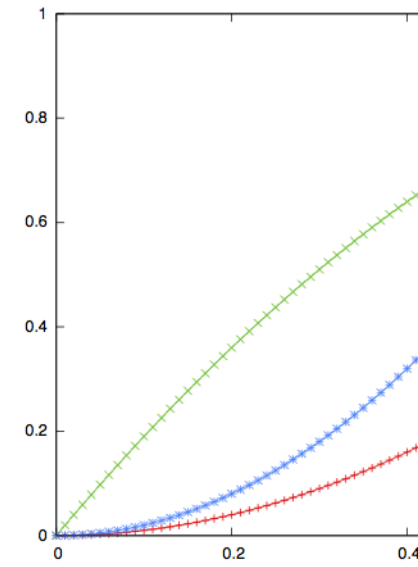
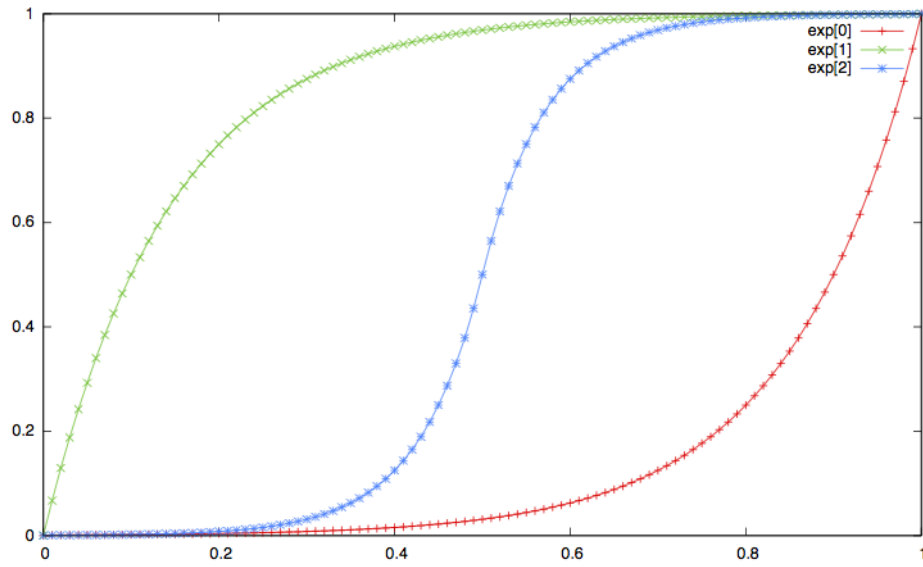
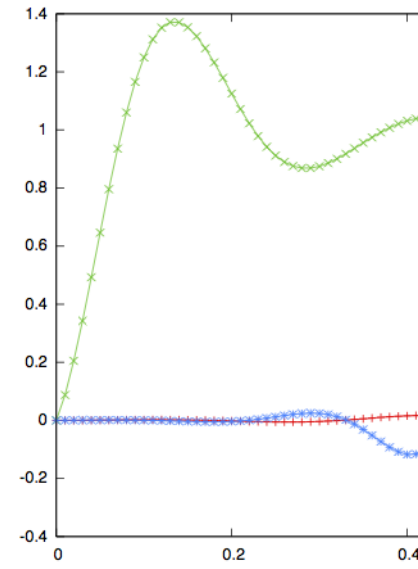
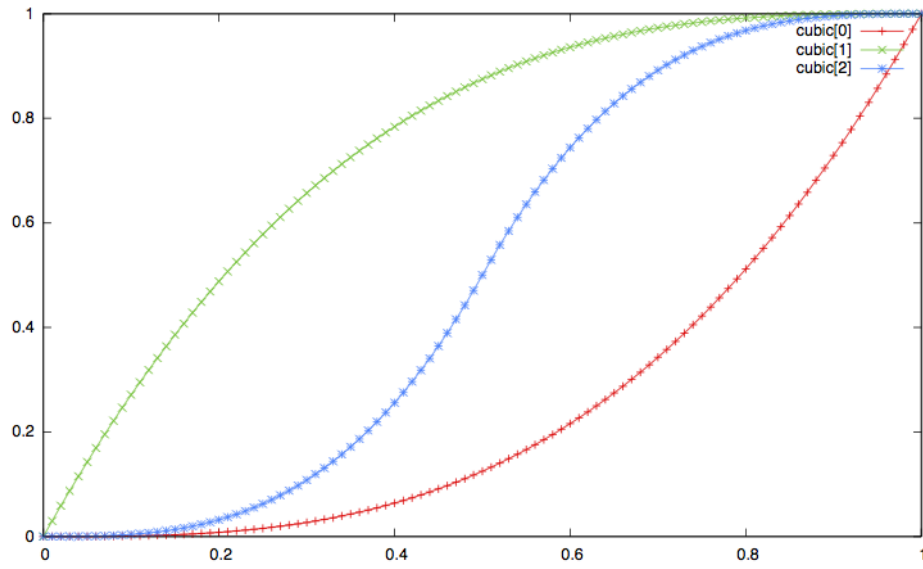
Examples of Interpolation Types

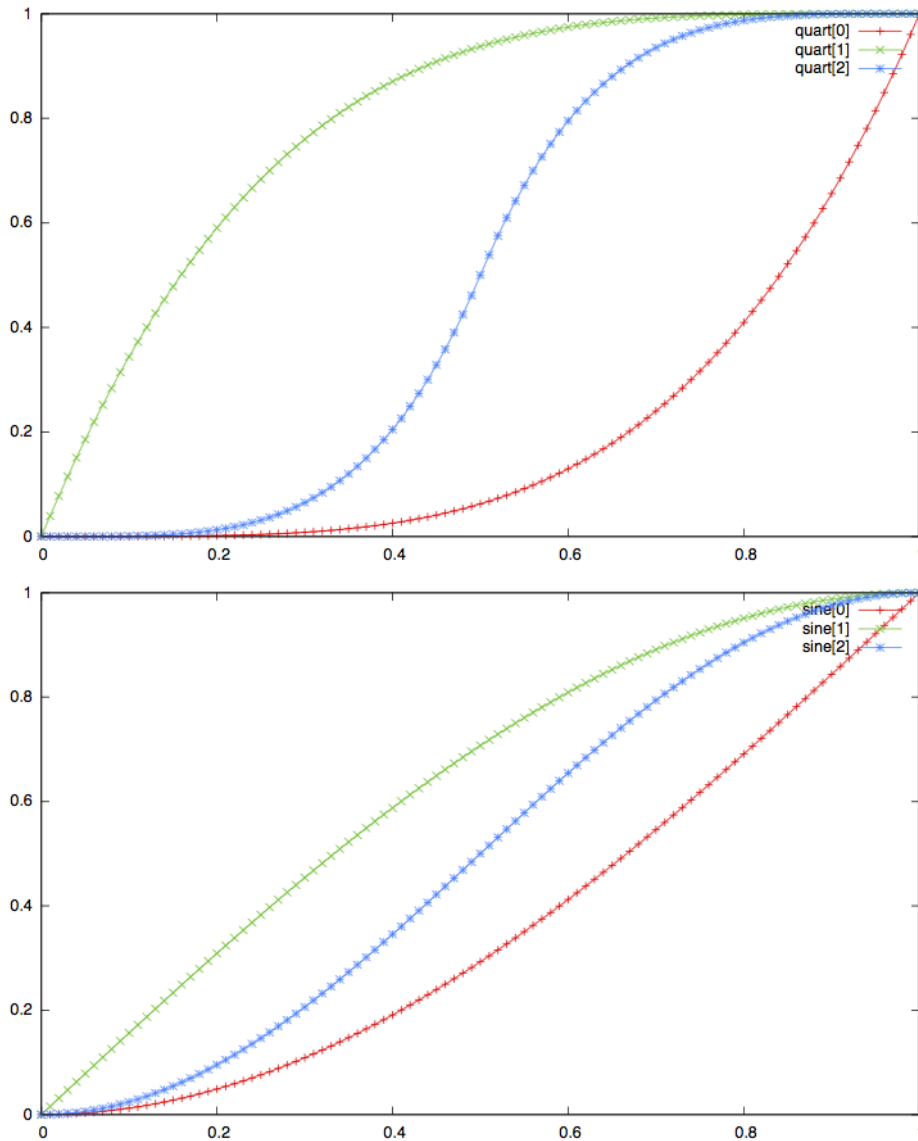
In the pictures below:

- The label `xxx[0]` corresponds to the ease *in*, that is to the type `"xxx_in"` or equivalently `"xxx"`.
- The label `xxx[1]` corresponds to the ease *out*, *i.e.* the interpolation type `"xxx_out"`.
- And the label `xxx[2]` corresponds to the ease *in_out*, *i.e.* interpolation method `"xxx_in_out"`.

Open the image on another window to enlarge the plot.







Curve Synchronization

Synchronization attributes apply to curve. To understand the effect of synchronization strategies on curve, it is useful to understand the curve as a group whose actions are the curve’s action iterated at each sampling point:

```

curve C
  @grain := d
  @action := { ... action\ensuremath{\_i} ... }
  { $x { {start} ... {end} } }
    
```

is really a shorthand for⁴:

⁴The equivalent group given here is only an approximation because the grain d is dynamically computed and adjusted so that curve’s action is executed for each breakpoint boundaries (breakpoint’s duration are not

```

Group G
{
    $x := start
    { ... action\ensuremath{_{i}} ... }
d $x := ...
    { ... action\ensuremath{_{i}} ... }
d $x := ...
    { ... action\ensuremath{_{i}} ... }
    ...
d $x := end
    { ... action\ensuremath{_{i}} ... }
}

```

The synchronization attributes simply apply to this group.

IF and SWITCH: Conditional and Alternative

IF: Conditional Actions

{!BNF_DIAGRAMS/if.html!}

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_expression)
{
    ... actions launched if the condition evaluates to true ...
}

```

or

```

if (boolean_expression)
{
    ... actions launched if the condition evaluates to true ...
}
else
{
    ... actions launched if the condition evaluates to false ...
}

```

Like 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 follow the conditional. For example

```

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

```

necessary a multiple of the grain size).

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. The attribute of `if` are used for these groups.

There are also conditional expressions (sections [Conditional Expression](#) and [Extended Expression If](#)) that share a similar syntax.

Any kind of *Antescofo* value can be interpreted as a boolean value, see sections [scalar values](#) and [data structures](#).

SWITCH: Alternative Actions

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

There are also alternatives expressions (section [switch expression](#)) that share a similar syntax in the context of function definitions.

Alternative Actions without Selectors

{!BNF_DIAGRAMS/switch.html!}

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

```
switch
{
    case e\ensuremath{_1}: a\ensuremath{_1}
    case e\ensuremath{_2}: a\ensuremath{_2}
    ; ...
}
```

(where $e_1, e_2 \dots$ are arbitrary expressions and $a_1, a_2 \dots$ are sequences of actions) can be rewritten in:

```
if ( e\ensuremath{_1} ) { a\ensuremath{_1} }
else
{
    switch
    {
        case e\ensuremath{_2}: a\ensuremath{_2}
        ; ...
    }
}
```

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

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

```
whenever ($PITCH)
{
    switch
    {
        case $PITCH < 80:
            $octave := 1

        case $PITCH < 92:
            $octave := 2

        case $PITCH < 104:
            $octave := 3
    }
}
```

uses a [whenever](#) to set the variable $\$octave$ to some value each time $\$PITCH$ is updated for a value below 104.

Note that the actions associated to a `case` are evaluated as if they are in a group. So the eventual delay of these actions does not impact the timing of the actions which follows the alternative. And like other actions, an alternative action can be prefixed by a delay.

Alternative Action with a Selector

{!BNF_DIAGRAMS/switch2.html!}

In this form, a selector is evaluated and checked in turn against each guard of the cases:

```
switch (s)
{
    case e\ensuremath{_{1}}: a\ensuremath{_{1}}
    case e\ensuremath{_{2}}: a\ensuremath{_{2}}
    ; ...
}
```

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

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

The evaluation start with e_1 and stops as soon as an action is performed for one of the e_i . If no guard checks true, no action is performed.

For example:

```

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

```

checks a variable `$x` and sets the variable `$zero` to true if `$x == 0` or `0.0` (because `0.0 == 0`). It then sets the variable `$empty` and `$zero` to false if `$x` refers to a non-empty tab or to a non-empty map (because function `[@size]` returns an integer which is 0 only if its argument is an empty tab or an empty map).

Reacting to logical events

{!BNF_DIAGRAMS/whenever.html!}

The [whenever](#) statement allows the launching of actions conditionally on the occurrence of a logical condition:

```

whenever optional_label ( boolean_expression )
{
    actions_list
}

```

The behavior of this construction is the following: The `whenever` is active from its firing until its end. In absence of an [end clause](#) or of an abort command, the `whenever` will be active until the end of the program execution.

When a `whenever` statement is active, each time a variable referred to by `boolean_expression` is updated, the expression is re-evaluated. If the condition evaluates to true, the body of the `whenever` is launched.

We stress the fact that only the variables that appear explicitly in the boolean condition are tracked. We say that these variables are **watched** by the `whenever`.

The boolean expression can be replaced by [temporal patterns](#) which ease the specification of complex events over time.

Nota Bene: multiple occurrences 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 a\ensuremath{_1}
    0.5 a\ensuremath{_2}
}

```

```

    0.5 a\ensuremath{\_3}
  } while ($cpt <= 3)

```

This example will produce the following schedule:

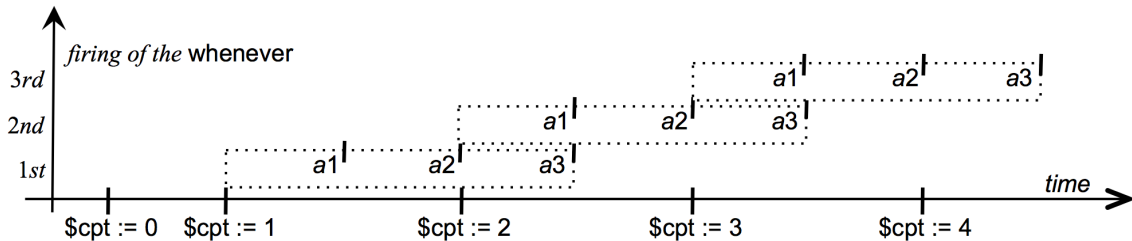


Figure 13.14: whenever schedule

Difference between conditional actions and whenever

Notice the difference between a [conditional action](#) and a [whenever](#): a conditional action is evaluated *once* when the flow of control reaches the action while the whenever is evaluated *as many times as needed* to track the changes of the variables appearing in the condition, between its firing and its end.

The whenever is a way to reduce and simplify the specification of the score, specifically when actions have to be executed each time some condition is satisfied. It also escapes the sequential nature of traditional scores. The actions resulting from a whenever statement are not statically associated to an event of the performer but dynamically at some point in time where a predicate becomes true.

The @immediate attribute

Note that the boolean condition is usually not evaluated when the whenever is fired because the variables that appears in the whenever are usually not assigned in the same instant.

To force the evaluation of the boolean expression when the whenever is fired, one can use the `[@immediate]` attribute. This attribute forces the evaluation of the boolean condition when the whenever is fired, in addition to the evaluation caused by an update of the watched variables.

Notice that if a watched variable is set in the same instant as the whenever is fired, it does not necessarily trigger the whenever's body, even if the condition is true: The watched variable must be set after the activation of the whenever. For instance

```

whenever W1 ($y) { print "OK whenever 1 at " $NOW }
let $y := true
whenever W2 ($y) { print "OK whenever 2 at " $NOW }
1s
let $y := true

```

will print


```

OK whenever 1 at 0
OK whenever 1 at 1
OK whenever 2 at 1

```

because whenever W1 is triggered two times (at time 0 and time 1) while W2 is triggered only once: at time 0, it is activated after the first assignment of `$y` and so cannot be triggered.

Instantaneous action that takes place in the same instant are executed sequentially: this is the [synchrony hypothesis](#). The order of execution within an instant depends on the [action priority](#).

Synchronization Attributes

Because the actions in the body of a `whenever` statement are not bound to an event or another action, synchronization and error handling attributes of the body's instances are the those of the `whenever`'s activation.

Avoiding Overlapping Instances of a Body

The activation of a `whenever` fires a new group and two such groups may overlap in time. Sometimes it is necessary to avoid this behavior. It can be done using an explicit `abort`:

```

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

```

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

```

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

```

This attribute may also be used on other compound actions. See also section [priority](#) for the management of actions that takes place at the same date.

Stopping a Whenever

An [end clause](#) can be defined for `whenever`. These clauses are evaluated each time the logical condition must be evaluated, irrespective of its or value. For example,

```

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

```

```

1.0 $X := false
1.0 $X := true
1.0 $X := true

```

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

Using a duration in relative time or in absolute time gives the a *interval of time* during which it is active. When the duration is elapsed, the `whenever` cannot longer fire its body.

The previous example with logical time shows how to stop the `whenever` after two updates of `$X` (whatever is the update). It is easy to stop it after a given number of bodies fire, using a counter in the condition:

```

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

```

This will print only one OK at relative time 2.0. Then the counter is set to 1 and the condition will always be false in the future.

However, the previous program will still leave the `whenever` active: the boolean condition is still checked at each update of `$cpt` or `$X`. So its is better to use a logical end clause to terminate the `whenever`

```

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

```

One can also use an `abort` command.

Watching Restrictions

The `whenever` watches variables, not values. This means that the construction monitors the updates of the variables that appear in the logical condition. When a variable is updated, the logical condition is (re)evaluated to decide (if true) to launch the `whenever`’s body.

Additionally, the set of watched variables is determined by a syntactic analysis of the boolean condition. Some systems' variables are not managed as ordinary variables and cannot be watched.

These constraints have several consequences that are reviewed below. Their rationale is to ensure that *Antescofo* scores remain causal and efficiently implementable.

Assignment of a tab

In

```
whenever ($t) { ... }
; ...
let $t[0] := ...
```

the assignment of a tab element does not trigger the whenever even if the `tab` is referred by a variable that appears in the whenever condition. As a matter of fact, the value of `$t` is mutable, the assignment mutates this value but the variable assignment: `$t` always refers to the same value.

Reference to a system variable

The three system variables `$NOW`, `$MYSELF` and `$THISOBJ` cannot be watched by a whenever.

The variable `$NOW` appears as continuously updated (there is no notion of quantum step in time progression, so watching this variable amount to execute infinitely often the whenever's body, in a finite time interval). Notice that this is *not the case* for `$RNOW` which is updated by discrete jumps at each musical events and meaningful actions.

Variables `$MYSELF` and `$THISOBJ` are not real variables: they are constants that denote some `exec` linked to the context where they appear.

Reference to a scoped variable

This limitation is rather subtle. Refer to [scoped variable](#) to fully appreciate the code below:

```
let $g := {
  @local $x
  $x := false
  whenever U ($x) { print "OK 1" }
  10
  print "end of G"
}
whenever V ($g.$x) { print "OK 2" }

2 let $g.$x := true ; [1]
```

The evaluation of this program will print

```
OK 1
end of G
```

because V does not watch the local variable $\$x$ in the group denoted by $\$g$. Only U is triggered by the assignment [1].

As a matter of fact, the variable watched by V is restricted to $\$g$: in the expression $\$g.\x , $\$x$ is only a *name*, not a variable. The determination of the variable denoted by expression $\$g.\x is dynamically computed and may change when $\$g$ is updated. The set of watched variables is statically determined. Dynamically changing this set is considered too costly and is not managed in the current *Antescofo* version.

One Activation per Instant

The variables watched by a whenever can be updated several times in the same instant. However, the whenever is fired at most once, with the first update that leads the condition to evaluate to true. For example:

```
$a := false
$b := false
$c := false

1
whenever( $a || $b || $c)
{
    print WHENEVER activated at $NOW $a $b $c
}

1
$a := false
$b := true
$c := true
```

will print only

```
WHENEVER activated at 1.0 false true false
```

because the whenever is activated at most once per instant, as soon as possible.

Causal Score and Temporal Shortcuts

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

```
let $x := 1
let $y := 1
whenever W1 ($x > 0)
{
```

```

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

```

When action `Start` is fired, the body `W1` of 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*:

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

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

The situation would have been much different if the `$y` assignments had been made after some delay. For example:

```

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

```

also generates an infinite stream of computations but with a viable schedule in time. If is fired at 0, then is fired at the same date but the assignment of will occur only at date 2. At this date, the body of is subsequently fired, which leads to the assignment of at date 3, etc.

$$\begin{array}{ccccccc} 0 : \text{Start} \rightarrow W1 \rightarrow & & 1 : \$y := 1+1 \rightarrow W2 \rightarrow & & 2 : \$x := 10+1 \rightarrow W1 \\ \rightarrow & & 3 : \$y := 2+1 \rightarrow W2 \rightarrow & & 4 : \$x := 11+1 \rightarrow W1 \rightarrow & & 5 : \dots \end{array}$$

Automatic Temporal Shortcut Detection

Antescofo automatically detects temporal shortcuts and stops the infinite regression. This behavior is a consequence of the rule “whenever are activated at most once per instant” and no warning is issued.

Process Creation

A process is a group of actions that is triggered dynamically. A process call takes the following form:

```
:: exp (a\ensuremath{_1}, a\ensuremath{_2}, ...)
```

where `exp` is an expression which evaluates into a `proc` and the a_i are expressions that are the parameters of the process instantiation.

A process call is a compound action: the duration of a call is the duration of the instantiated group⁵.

Synchronization attributes, which are inherited by the instantiated group, may be specified. However, the attribute `[@exclusive]` has no effect.

Refer to the chapter `Process` for an in-depth presentation.

Object Creation

The remarks for process creation also apply for object creation. Please refer to chapter `[Object]` for a thorough presentation on that topic.

Continuations

{!BNF_DIAGRAMS/continuation.html!}

Performing an action at the end of a group (or any other compound action) may be difficult if the delays of the group's actions are expressions or if some conditional constructs are involved. Even with constant delay and no control structure, computing the duration of a group, or a loop/whenever/forall body, can be cumbersome.

This observation advocates for the introduction of two additional sequencing operators that are used to launch an action at the end of the preceding one:

(no special name)	<i>followed-by</i>	<i>ended-by</i>
a b	a ==> b	a +=> b
b is launched together with a	b is launched at the end of a	b is launched at the end of a <i>and</i> its eventual children

The juxtaposition (launch synchronously/in parallel), the followed-by operator (launch at the end) and the ended-by operator (launch at child ends) are binary operators called *continuation combinators*. Delays are not continuations⁶: they are unary operators applied to an action to defer its start.

The effects of the three operators are illustrated by the figure below. The end of a compound action is represented by the bold outlined rectangle. The child actions may end earlier or later than the end of the top-level actions:

- The juxtaposition aligns the starts of two sequences;

⁵The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

⁶Because a delay can be used in front of the first action of a sequence, a delay does not necessarily links two successive actions.

- the \Rightarrow “followed-by” operator align the end of the first sequence with the start of the second;
- and the $+\Rightarrow$ “ended-by” aligns the latest end (the final launching of an action, including a child action) with the start of the second sequence:

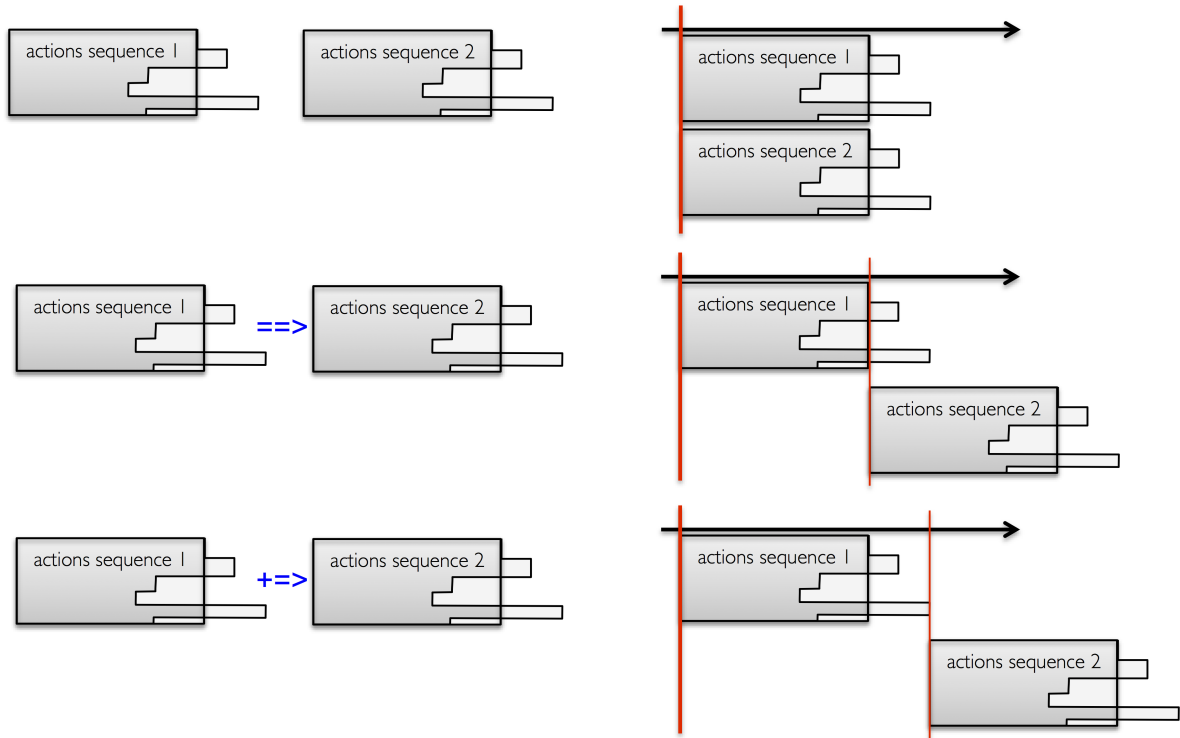


Figure 13.15: illustration of the continuation combinators

Continuation combinators do not change the scope of the local variables of their arguments. In other words, in $a \Rightarrow b$ the actions in b cannot access the local variables defined in a .

Continuation combinators freely compose between actions and are right associative. Here are some examples:

<i>Expression</i>	<i>Meaning</i>
$a \Rightarrow b \Rightarrow c$	is equivalent to $a \Rightarrow \{ b \Rightarrow c \}$ and specifies that $b \Rightarrow c$ starts at the end of a
$\{ a \Rightarrow b \} \Rightarrow c$	starts c at the end of $\{ a \Rightarrow b \}$, that is, with the end of b

<i>Expression</i>	<i>Meaning</i>
<code>{ a ==> b } c</code>	starts <code>c</code> with the start of <code>{ a ==> b }</code> , that is, with the start of <code>a a ==> b c </code> is equivalent to <code>a ==> { b c }</code> and starts <code>{ b c }</code> with the end of <code>a a b ==> c </code> is equivalent to <code>{ a b } ==> c</code> and starts <code>c</code> with the end of <code>{ a b }</code> , that is, the end of <code>b a +=> b ==> c </code> is equivalent to <code>a +=> { b ==> c }</code> and starts <code>{ b ==> c }</code> at the end of "a" and its children"

For instance, suppose we want to make an action after the end of a loop:

```

$cpt := 0
Loop 1
{
  print "tic" $cpt
  3 print "tac" $cpt
  $cpt := $cpt + 1
} during [3#]
+=> print "loop ended"

```

Here there will be 3 iterations of the loop. So, if the loop starts at date 0, the first iteration starts at 0 and ends at 3, the second one starts at 1 and ends at 4 and the last one starts at 2 and ends at 5.

Instead of explicitly computing these numbers to launch an action at the right time, we have used the continuation combinator `+=>` which waits the end of the loop and all the loop bodies, to trigger the print message: the message "loop ended" will appear at date 5.

As you can see, the *end of a loop is different* from the ends of the loop bodies: the loop in itself terminates when the last iteration is launched. As a matter of fact, the computation associated to a compound action `a` can be seen as a tree, with the sub-computations rooted at `a`. Thus, there is no need to maintain `a` after having launched the last sub-computation. So the end of `a` is usually not the same as the end of the last sub-computation spanned by `a` and this is why the operator is usually more handy than `==>`.

Nevertheless, the end of an action is always precisely defined although it can be only dynamically known:

- *atomic action*: the start and the end of the action coincide.
- *compound actions without duration*: are actions that launch other actions but they do not have a duration by themselves because they do not need to persist in time. Such

actions are the `if`, `switch`, and the `forall`. The start and the end of these actions coincide. However, these actions have children: the actions launched by these constructs.

- *compound actions with a duration*: the start and the end of these actions usually differ:
 - Group `G { a ... b }`: the start of `G` coincides with the start of `a`. The end of `G` coincides with the start of `b` (the last action in the group). The children of `G` are all actions launched directly (they appear explicitly in the group body) or indirectly (they are launched by a child of `G`).
 - Loop `L { a }`: the start of `L` coincides with the start of the first iteration of `a`. The end `L` of coincides with the last iteration of `a`. The children of `L` are the actions launched in the loop bodies.
 - Whenever `W { ... }`: there is no relationship between the start of `W` and the actions in it body. Usually, there is no end to a `whenever` except if there is a `during` or an `until` clause. In this case, the whenever terminates when the clause becomes true. The children of `W` are the actions launched by the instantiations of the whenever body.
 - A `process call` or an `object instantiation`: their end coincides with the end of the associated instance. An object has no end *per se* and must be aborted.

Continuation and abort

Abort handlers launch a group at the premature end of a compound actions. So they differ from the followed-by and ended-by operators that launches a group of actions at the (natural or premature) end of an action.

An abort handler, specified by the `[@abort]` attribute, is considered as a child of the associated action. So, when an abort handler exists and the associated action is aborted, the abort handler is launched with the followed-by continuation (if it exists). Because the abort handler is necessarily defined before, it happens before the followed-by continuation.

The ended-by continuation is launched after the end of the abort handler (because the abort handler is a child).

Continuations are not considered children of the continued action. So in `a ==> b`, `b` do not has access to the local variables of `a`, contrary to the `@abort` clause of `a`.

For example (note the bracketing of the process call):

```
@proc_def ::P()
@abort { print abort P $NOW }
{
    print start P $NOW
    10 print BAD END P $NOW
}

{ ::P() ==> print continuation P $NOW }

5
print "launch abort" $NOW
abort ::P
```

will give the following trace:

```
start P 0.0
launch abort 5.0
abort P 5.0
continuation P 5.0
```

The continuation is launched at date 5.0 because it is launched with the end of the instantiated process (here a premature end caused by the `abort` command). If the abort handler is replaced by:

```
@abort { 11 print abort P $NOW }
```

the corresponding trace is:

```
start P 0.0
launch abort 5.0
continuation P 5.0
abort P 16.0
```

because the followed-by continuation does not wait the end of the abort handler. If we replace the followed-by continuation by an ended-by continuation

```
{ ::P() +=> print continuation P $NOW }
```

the trace becomes:

```
start P 0.0
launch abort 5.0
abort P 16.0
continuation P 16.0
```

because the continuation takes place at the end of all of the action's children, including the abort handler.

Notions of TIME in *Antescofo*

Real musical time is only a place of exchange and coincidence between an infinite number of different times. Gérard Grisey

Antescofo claims to be a *strongly timed*⁷ computer music programming language, which means that:

⁷[ChucK](#) has introduced the term *strongly timed* to qualify programming languages that handle time explicitly as a first class entity. This is the case in *Impromptu*, where performative time is also considered in the context of *Live Coding*, refer to the article [Programming with time : cyber-physical programming with Impromptu](#). The real-time systems community has long advocated for the inclusion of time in the domain of discourse and not to consider it as an optimization problem or as a quality of service problem. See for example [Motivating Time as a First Class Entity](#) (1987). The reference [Computing Needs Time](#) is a documented presentation of the problematic.

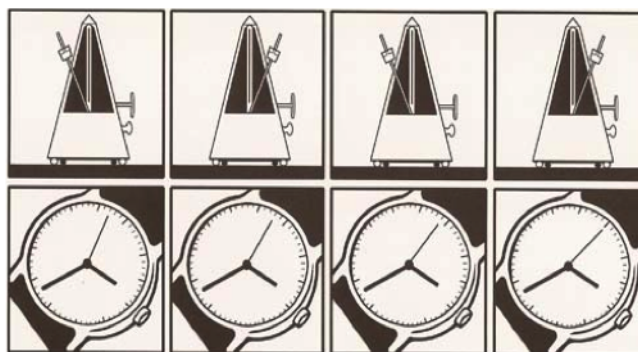
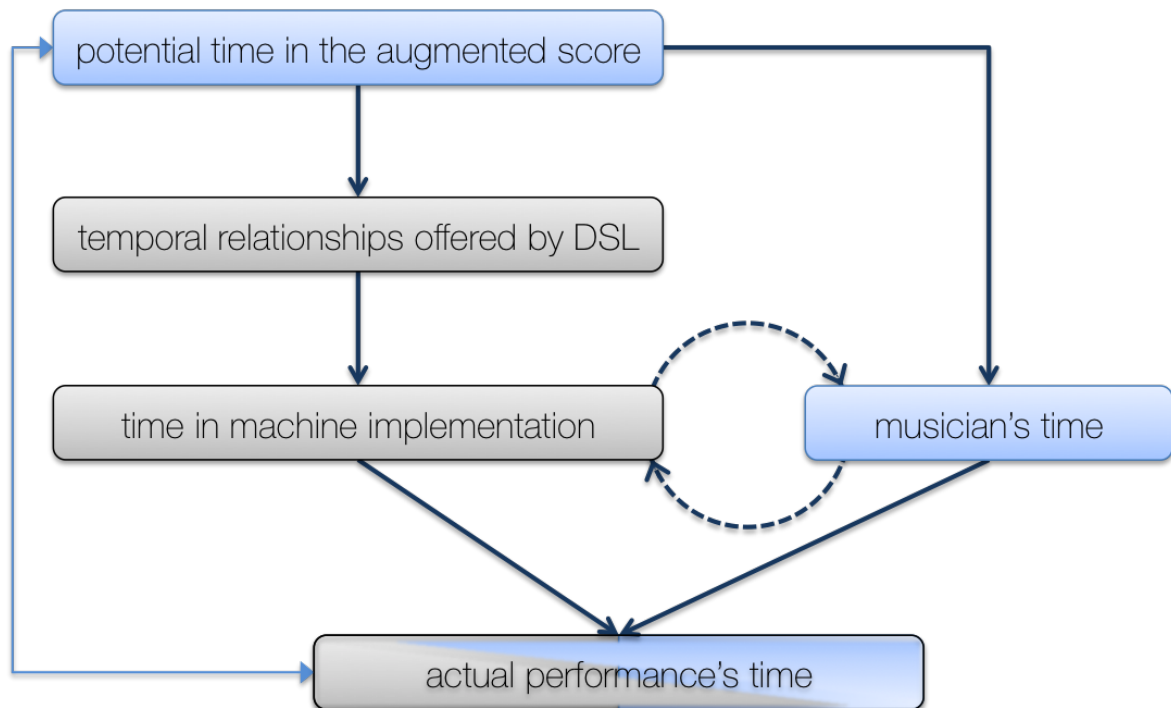


Figure 13.16: image of clocks and metronoms

- time is a first class entity in the language, not a side-effect of the computations performance, making time itself both controllable and explicitly computable;
- *when* a computation occurs is precisely specified and formalized⁸ guaranteeing behavior predictability, temporal determinism and temporal precision;
- the behavior of an *Antescofo* program is free from underlying hardware timing and nondeterministic scheduling in the operating system.

Several notions of time are at work in *Antescofo*. The composer defines *potential temporal relationships* between musical entities in an augmented score. These relationships are expressed through the temporal relationships between *Antescofo* actions and between actions and musical events. During the performance, the potential temporal relationships specified by the composer become *actual* relationships through the realization of musical events by the musician and the computation of electronic actions by the computer. A unique feature of *Antescofo* is that the composer is able to specify some constraints between the potential and the actual temporal relationships.

⁸Cf. for instance [José Echeveste's PhD thesis](#)



Several “layers of time” at work in *Antescofo*. What depends of the composer/human performer is in blue; what depends of the computer is in gray. Composers can express constraints between the potential timing expressed in the score and the actual timing of the performance. These constraints apply to the actual timing of the electronic actions.

This chapter presents the rich model of time provided by the DSL and the constraints that can be expressed between the potential and the actual timing. The two following pages do not regard specific technical or syntactic aspects of *Antescofo*, but they do contain essential ideas about the language’s use of time:

- [the manufacturing of time](#) elaborates on the temporal notions that organize the computations of actions;
- [the fabric of time](#) elaborates on the relationships between the potential timing of musical events expressed in the score, the actual timing of performer’s events and the actual timing of the actions during the performance.

The Manufacturing of Time

In philosophy, the analysis of time by Kant distinguishes between two temporal entities, **instant** and **duration**, that are linked by three temporal modes or relationships: **succession**, **simultaneity** and **permanence**.

This analysis can be used to classify programming languages and computer music systems by their handling of instant and duration:

- Dealing with succession and simultaneity of instants leads to the **event-triggered** or **event-driven** view, where a processing activity is initiated as a consequence of the

occurrence of a significant event. For instance, this is the underlying model of time in MIDI.

- Managing duration and permanence points to a **time-triggered** or **time-driven** view, where activities are initiated periodically at predetermined points in real-time and last. This is the usual approach in audio computation.

These two points of view⁹ are supported in *Antescofo* and the composer/programmer can express his own musical processes in the most appropriate style. We will elaborate on this point, along with some comparison with *ChucK* to better fix the idea. Indeed, *ChucK* exhibits a complete and coherent model of time, relevant to both audio processing and the handling of asynchronous events like MIDI and OSC messages or interactions with serial, and human interface devices. In the next section, [the fabric of time](#), we will discuss the unique capacity of *Antescofo* to specify and manage several time references.

Instants and Succession: Sequential Languages

Sequential programming languages usually deal only with instants (which are the location in time of elementary computations) and their succession. The actual duration of a computation does not matter, nor does the interval of time between two instants: these instants are *events*.

This model is that of MIDI: basic events are *note on* and *note off* messages. There is no notion of duration in MIDI: the duration of a note is represented by the interval of time between a *note on* and the corresponding *note off* and it has to be managed externally to the MIDI device, *e.g.* by a sequencer. In addition, two MIDI events cannot happen simultaneously. So we cannot say for instance that a chord starts at some point in time, because starting the emission of the notes of the chords are distinct sequential events.

Instants, Succession and Simultaneity: Synchronous Languages

In a purely sequential programming language, it is very difficult to do something at a given date. We can imagine a mechanism that suspends the execution for a given duration and wakes up at the given date, as in

```
sleep(12 p.m. - now()) ;
computation to do at 12 p.m.
```

or if we have a mechanism that suspends the execution until the arrival of a date or the occurrence of an event:

```
wait(12 p.m.) ;
computation to do at 12 p.m.

wait(MIDI message) ;
process received message
```

⁹[Event-Triggered versus Time-Triggered Real-Time Systems](#) In Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, Vol. 563. Springer, Dagstuhl Castle, Germany, 87–101.

Notice that the computation resumes *after* the date or the event. On a practical level, this is usually negligible (*e.g.* usually chords can be emulated in MIDI using successive events). However, at a conceptual level, it means that *simultaneity* cannot be directly expressed in the language, which will make the specification of some temporal behaviors more difficult.

To express simultaneity in the previous code fragment, we have to imagine that computations happen infinitely fast, allowing events to be considered atomic and truly synchronous. This is the **synchrony hypothesis** whose consequences have been investigated in the development of synchronous languages dedicated to the development of real-time embedded systems like Esterel, Lustre or Lucid Sychrone.

Synchronous Languages

Synchronous languages have not only postulated infinitely fast computations, allowing two computations to occur simultaneously, they have also postulated that **two computations occurring at the same instant are nevertheless ordered**. This marks a strong difference between simultaneity and parallelism (more on this below) and articulates, in an odd way, succession and simultaneity¹⁰.

However, a formal model such as [superdense time](#) shows that there are no logical flaws in the idea that actions occurring at the same date are performed in a specific order (see next paragraph). Much better, this hypothesis reconciles determinism with the modularity and expressiveness of concurrency: at a certain abstraction level, we may assume that an action takes no time to be performed (*i.e.* its execution time is negligible at this abstraction level) and we may assume a sequential execution model (the sequence of actions is performed in a specific and well determined order) which imply deterministic and predictable behavior. Such determinism can lead to programs that are significantly easier to specify, debug, and analyze than nondeterministic ones.

A good example of the relevance of the synchrony hypothesis in the design of real-time systems is the sending of messages, in MAX or PureData, to control some device. To change the frequency of a sine-wave generator, the generator must have already been turned on. But there is no point in postulating an actual delay between turning on the generator and changing its frequency default value. The corresponding two messages are sent in the same logical instant but in a specific order.

Another example is audio processing when computations are described by a global dataflow graph. From the audio device perspective, time is discretized in instants corresponding to the input and the output of an audio buffer. In one of these instants, all computations described by the global graph happen together. However, in this instant, computations are ordered, *e.g.* by traversing the audio graph in depth-first order, starting from one of several well-known sinks, such as `dac`. Each audio processing node connected to the `dac` is asked to compute and return the next buffer, recursively requesting the output of upstream nodes.

In section [Thickness of an Instant](#) we investigate the notion of an **action's priority** used to totally order the actions that occur within the same instant, even if they are not structurally related in the program.

¹⁰Given two actions, one always precedes the other, but some successive actions can be simultaneous. Actions that occur simultaneously occur in the same logical instant. Logical instants are ordered completely by succession, just like actions within an instant.

Superdense Time

For the curious reader, we give here a brief account of superdense time¹¹, a simple formal model of time supporting the synchrony hypothesis. This approach is used by the *Antescofo* language for implementing a total order between actions execution.

Given a model of time T that defines a set of ordered instants, a superdense time $SD_{[T]}$ is built on top of T to enable simultaneous but totally ordered activities on the same instant. An instant in $SD_{[T]}$ is a pair (t, n) where t is an instant of T and n is a *microstep* (an infinitesimally small unit of time occurring within a logical instant):

- t represents the date at which some event occurs,
- and n represents the sequencing of events that occurs simultaneously.

So, two dates (t_1, n_1) and (t_2, n_2) are interpreted as (*weakly*) *simultaneous* if $t_1 = t_2$, and *strongly simultaneous* if, in addition, $n_1 = n_2$.

Thus, an event at (t_1, n_1) is considered to occur before another at (t_2, n_2) if either $t_1 < t_2$, or $t_1 = t_2$ and $n_1 < n_2$. In other word, $SD_{[T]}$ is ordered lexicographically.

How does this relate to *Antescofo*? In the figure below, the sequence of synchronous actions appears in the vertical axis. So this axis corresponds to the dependency between simultaneous computations. Notice that the (vertical) height of a box is used to represent the logical dependencies while the (horizontal) length of a box represents a duration in time. Note for example that even if durations of a_1 and a_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.

A delay, a period in a loop or a sample in a curve, correspond to a progression on the horizontal axis. When these quantities are expressed in relative time, they depends on a tempo which can be dynamic (*i.e.*, it can change with the passing of time). Dynamic tempo correspond to shrink or to dilate the horizontal axis, but the order of events on the timeline is preserved.

Causality between computations (*e.g.* the evaluation of a sum must done after the evaluation of the arguments of the sum) corresponds to succession on the vertical axis. Causality is not enough to give a complete ordering of simultaneous action. For example, between two simultaneous assignments:

```
let $x := $x + $y
let $y := $x + $y + 1
```

the final result is not the same following the succession of assignments performed by the interpreter. **Action priority** is used to decide which one must be performed first. And in *Antescofo*, the first assignment occurring in the score is performed first (as in mainstream sequential programming languages).

Duration : Audio Processing Languages

In time-triggered system, activities are performed periodically, that is at time points predefined by a given duration. This duration matches the dynamics of these activities. In these systems,

¹¹Claudius Ptolemaeus, Editor, [System Design, Modeling, and Simulation using Ptolemy II](#), Ptolemy.org, 2014. (section 1.7.2)

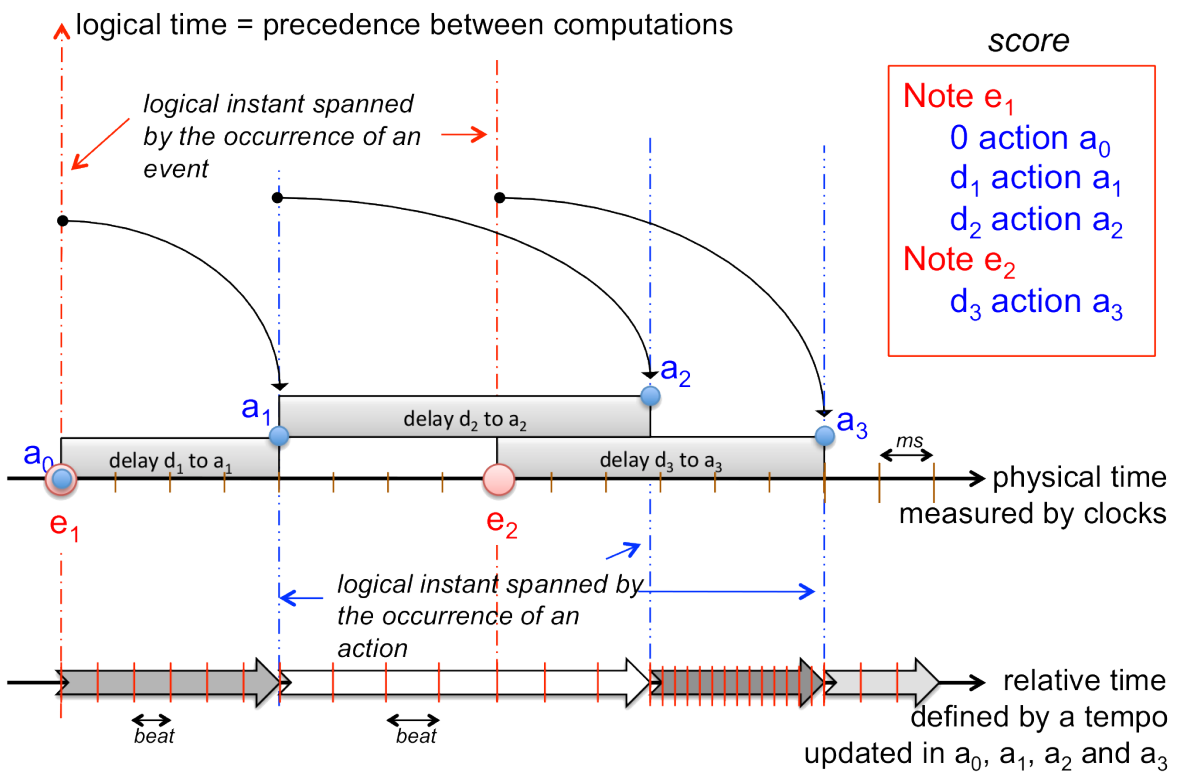


Figure 13.17: superdense time in Antescofo

the computation are driven by the passing of time, not by the occurrences of logical events like in event triggered systems.

Audio computations are very often architected as time-triggered systems. The audio signal is sampled periodically. Because of efficiency issues, block processing¹² is implemented by grouping samples into an audio buffer of fixed size matching a definite duration of the audio signal. Audio buffers are processed themselves periodically at an audio rate.

Duration also appears when a language offers the possibility to be woken up after some delay. In *Antescofo* it can be done using the delay before an action; in *ChucK* by *chucking* the delay to now:

```
// in ChucK
do something
5.8::ms => now // advance in time
do nextthing

// in Antescofo
do something
5.8 ms // wait 5.8 milliseconds
do nextthing
```

So, to trigger an activity periodically, such as by filling a buffer of 64 samples at 44100 Hz (5.8 ms), one can write:

```
while (true)
{
    do something
    5.8::ms => now
}

Loop 5.8 ms
{
    do something
}
```

These two code snippets in *ChucK* and in *Antescofo* seem very similar but their interpretations differ greatly. The *ChucK* program is a sequential program that is stopped for a given duration: any code between instructions to advance time can be considered atomic (*i.e.* presumed to happen instantaneously at a single point in time). The *Antescofo* code describes a parallel program where some actions have to be iterated every duration (iterations are supposed to take place independently, such that they can eventually overlap). These actions can be atomic or have their own duration.

This example exhibits one of the differences between the approaches of *ChucK* and *Antescofo*: *ChucK* takes a view where the computations happen **infinitely fast and sequentially**, while *Antescofo*'s approach is to view computation as **infinitely fast and in parallel**.

¹²The performance benefits of block processing are due to certain compiler optimizations, instruction pipelining, memory compaction, better cache reuse, *etc.*

This formulation seems absurd until one realizes that, here, parallelism refers to a logical notion related to the structure of the program evaluation and is not related to the time at which computations occur. A program is parallel if the progression of the computation is described by several threads (and each thread correspond to a succession of actions). In the case of ChuckK, all control structures are sequential, except the explicit thread creation operation `spork`. On the other hand, in *Antescofo* threads are implicitly derived from the nested structure of compound actions: every actions spans new threads for their child actions (except with the `==>` and `+=>` continuation operators). See table below:

	Implicit Threads	One thread	Explicit Threads
<i>instruction counter</i>	0	1	n
<i>examples</i>	PureData, <i>Antescofo</i>	C, Java, Python	Occam, C+threads, ChuckK
<i>thread creation</i>	implicit through data and control dependencies	—	through explicit operator <code>par</code> , <code>spork</code> , <code>fork</code> , ...

For example, the program

```

Group G
{
    d\ensuremath{d_1} a\ensuremath{a_1}
    d\ensuremath{d_2} a\ensuremath{a_2}
    d\ensuremath{d_3} a\ensuremath{a_3}
}

```

spans the three actions a_i in parallel: $(d_1 \ a_1) \parallel ((d_1+d_2) \ a_2) \parallel ((d_1+d_2+d_3) \ a_3)$

It is because of the cumulative delays that the group `:::antecofo G` seems a sequential construct. But without delays, it is apparent that the actions a_i are spanned in parallel. The ChuckK program that really mimics the *Antescofo* `Loop`, has to explicitly use *shreds* (*i.e.* ChuckK threads) to make the while bodies independent¹³:

¹³The dual question is the translation of the ChuckK `while (true) { ... }` construction in *Antescofo*. In *Antescofo* view, this construction should be avoided because it would lead to an infinite number of computations in finite time (if the body of the `while` is an instantaneous action). At the end of the day, throwing away all temporal abstractions, a computation takes finite physical time, so an infinite number of instantaneous computations will take an unbounded quantity of physical time, violating initial assumptions. Such behavior should be avoided. This explains why there is no `while` construct in *Antescofo*: a `while` construct makes the superdense time to progress on the vertical axis, and implementability requires finite height in the superdense time. On the other hand, a `loop` construct makes the time to progress on the horizontal axis, if the period is non null, and this axis can be unbounded. A `while (true) { ... }` construct (unbounded recursion) cannot be achieved in *Antescofo* using a loop with a period of zero `Loop 0ms { ... }` because the run-time imposes a finite number of consecutive iterations with a period 0 if there is no explicit `end clause`. If this limit is reached, the loop is aborted and an error is signaled. There is no danger from the `ForAll` construct: if it spans its body in parallel, the number of spanned groups is bounded by the size of a data structure (mimicking primitive recursion). However, there exist some means to specify an unbounded number of actions in *Antescofo*, and if these actions are all instantaneous (no delay, zero period, zero grain, *etc.*) it could potentially lead to an infinite number of actions in finite time. *Antescofo* allows loops with zero period if an `end clause` is specified, so `Loop 0ms { ... } while (true)` will hang the execution. It is also possible to define a recursive process calling itself before doing any others action or delays. It is also possible to specify an `expression` whose evaluation would lead to infinite computations in finite time (*e.g.* the call to a recursive function).

```
fun loop_body()
{
    do something
}
while (true)
{
    spork ~ loop_body
    5.8::ms => now
}
```

```
Loop 5.8 ms
{
    do something
}
```

Notice a benefit of the synchrony hypothesis: synchronous programs are sequential, even in the presence of implicit or explicit threads. So *there is no need for locks, semaphores, or other synchronization mechanisms*. Actions that occur simultaneously execute sequentially and without preemption, behaving naturally as *atomic transactions* with respect to variable updates.

Supporting Event and Duration

Real-life problems dictate the handling of both events and duration: music usually involves, in addition to audio processing, the handling of events that are asynchronous relative to DSP computations like MIDI and OSC messages or interactions with human interface devices.

Subsuming the event-driven and the time-driven architectures is usually achieved by embedding the event-driven view in the time-driven approach. The handling of events is delayed and taken into account periodically, leading to several internally maintained rates, *e.g.*, an audio rate for audio, a control rate for messages, a refresh rate for the user-interface, *etc.* This is the case for systems like Max or PureData where a distinct control rate is defined. Notice that this control rate is typically about 1ms, which can be finer than a typical audio rate (a buffer of 256 samples at sampling rate of 44100Hz gives an audio rate of 5.8 ms), but the control computation can sometimes be interrupted to avoid delaying audio processing (*e.g.* in Max). In Faust, events are managed at buffer boundaries, *i.e.* at the audio rate.

The alternative is to subsume the two views by embedding the time-driven computation in an event-driven architecture. After all, periodic activity can be driven by the events of a periodic clock. Thus, the difference between waiting for the expiration of a duration and waiting the occurrence of a logical event is that, in the former case, a time of arrival can be precomputed.

This approach has been investigated by Chuck where the handling of audio is done at the audio sample level. Computing the next sample is an event interleaved with the other events. It results in tightly interleaved control over audio computation, allowing the programmer to

at handle at the same time processing and higher levels of musical and interactive control¹⁴.

This approach is also the *Antescofo* approach¹⁵ where instants/events can be specified:

- by the performance of a musical event,
- by the reception of a message (OSC or Max/PD),
- by a duration starting from the occurrence of another event (the duration can be in absolute time or in relative time),
- by the start of an action,
- by the end of an action,
- by the satisfaction of a logical condition when a variable is assigned.

Duration appears in the delay preceding an action, in the period of a loop and in the sampling of a curve. *Antescofo* delays may be expressed in physical time (seconds, milliseconds) or in **relative time** (beat).

This feature is unique to *Antescofo*: in other computer music languages, a duration can be expressed in seconds, in milliseconds or even in samples, but these different units refer to the same physical time. Relative time in *Antescofo* is not linked to the physical time by a simple change of unit: it involves complex and dynamic relationships between the potential timing expressed in the score and actual timing of the performance. The correspondance between the two is not known *a priori* but builds incrementally with the passing of time during the performance. This problem is investigated in the next section.

The Fabric of Time

Music as a Collective Performance

Antescofo tackles two fundamental problems of mixed music defined as the association in live performance (in the context of written music) of human musicians and computer mediums interacting in real-time:

1. music as a performance,
2. and performance as a collective process.

¹⁴It can also be argued that ChuckK is a purely time-driven architecture, with a control rate equating to the signal sampling rate. Because the corresponding duration d is very small, and used both for the audio rate and the control rate, the distinction we made between the event-driven and the audio-driven architecture is blurred: one can understand d as the precision of locating an event in time.

¹⁵Audio processing in *Antescofo* is still experimental. Sample accuracy is achieved for control values corresponding a curve (irrespective of the [`@grain`] sample rate of the curve). Beyond that, our current research work is an attempt to consolidate sample accuracy and block computations through a notion of elastic audio buffer.

The first point refers to the divide between the score and its realization. Usually, notation does not specify all of the elements of music precisely, which leaves welcomed space for *interpretation*. The score can be thought of as a set of constraints that must be fulfilled by the interpretation but many score's incarnations may answer these constraints. The interpretation matters, conveying some meaning and assigning significance to the musical material. It is the performer's responsibility to choose/implement one of these possible incarnations. In doing so, the performer takes many decisions based on performance practice, musical background, individual choices and also because he is part of an ensemble: the music is played together with other musicians and the collective will dramatically affect the interpretation (our second point).

These two points challenge mixed music: how should various prescriptions of rhythm, tempo, dynamics and so on, be precisely realized within their permissible ranges by a computer? Computers cannot make these decisions out of the blue and, furthermore, have to take the other performers into account.

We restrict the rest of the discussion to the temporal relationships between various musical elements (to fix the idea, think about tempo). We qualify the temporal relationships specified in the score as *potential* and their realization in a performance as *actual*. A complete specification of the temporal relationships can be completely fixed by the composer, that is written in the score and definitive: the potential relationships are exactly the actual ones. For instance, it means that the tempo of the electronic action is explicitly specified in the score and implemented exactly during the performance. The interpretation problem is then avoided (there is no difference between potential and actual time), but then the human performer does not *play with* the machine: he or she *has to follow* the machine. This is no different from playing with a prerecorded tape, whereas the whole point of mixed music is to reintroduce the interpretation for the electronic part and to allow live interaction.

The obvious alternative is to let the composer to **fix the interpretation but relatively to the interpretation of the performer**. In this way, performers and computers play together, and there is still room for the human performer's interpretation. The situation is pictured below. For example, the electronic actions must follow the actual tempo of the performer, not the potential tempo possibly specified in the score.

This approach corresponds to a big shift of paradigm in mixed music and score following: electronic actions are not triggered on the occurrence of some musical events, but rather the timeline of the electronic is aligned (synchronized) with the timeline of the performer.

Antescofo follows this approach for temporal relationships. To implement it, *Antescofo* introduces several kinds of time:

- the potential time expressed in the score
- the actual time of the musical events performed on stage
- the actual time of the electronic actions implemented in real time.

and requests the composer to specify their relationships. These relationships, expressed as **synchronization strategies**, are presented in the paragraph [articulating time](#) below and discussed at length in section [Synchronization Strategies](#).

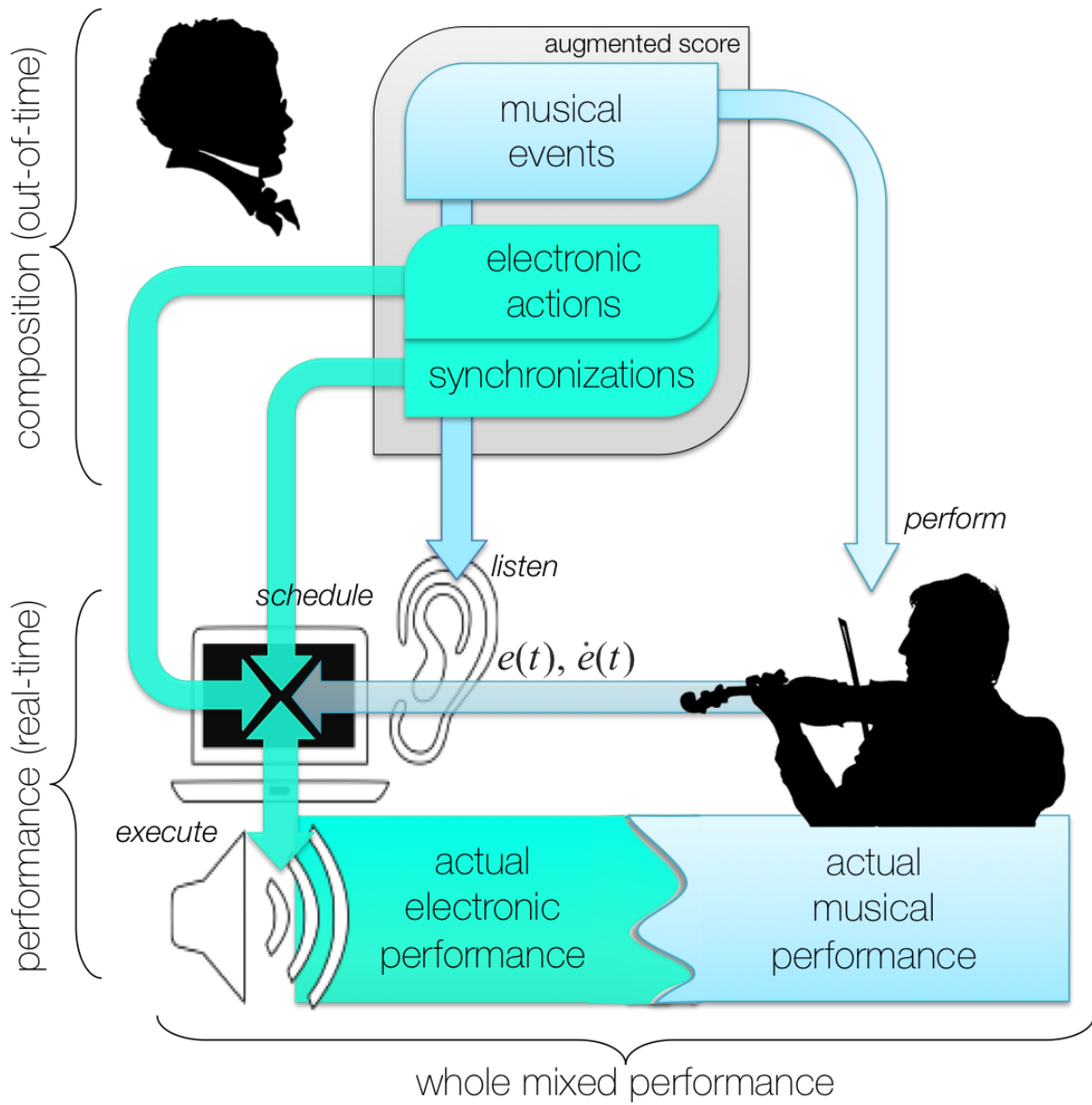


Figure 13.18: interpretation and synchronization

The Potential Score Time

The augmented score contains enough information to give a potential date to each event and action. These dates are specified through two pieces of information: the duration of each musical event and the tempo at each position in the score. The potential dating specified in the score can be represented as a curve picturing the advancement of the position in the score with respect to the passing of the physical time. See the plot below where the position in the score is measured in *beats*.

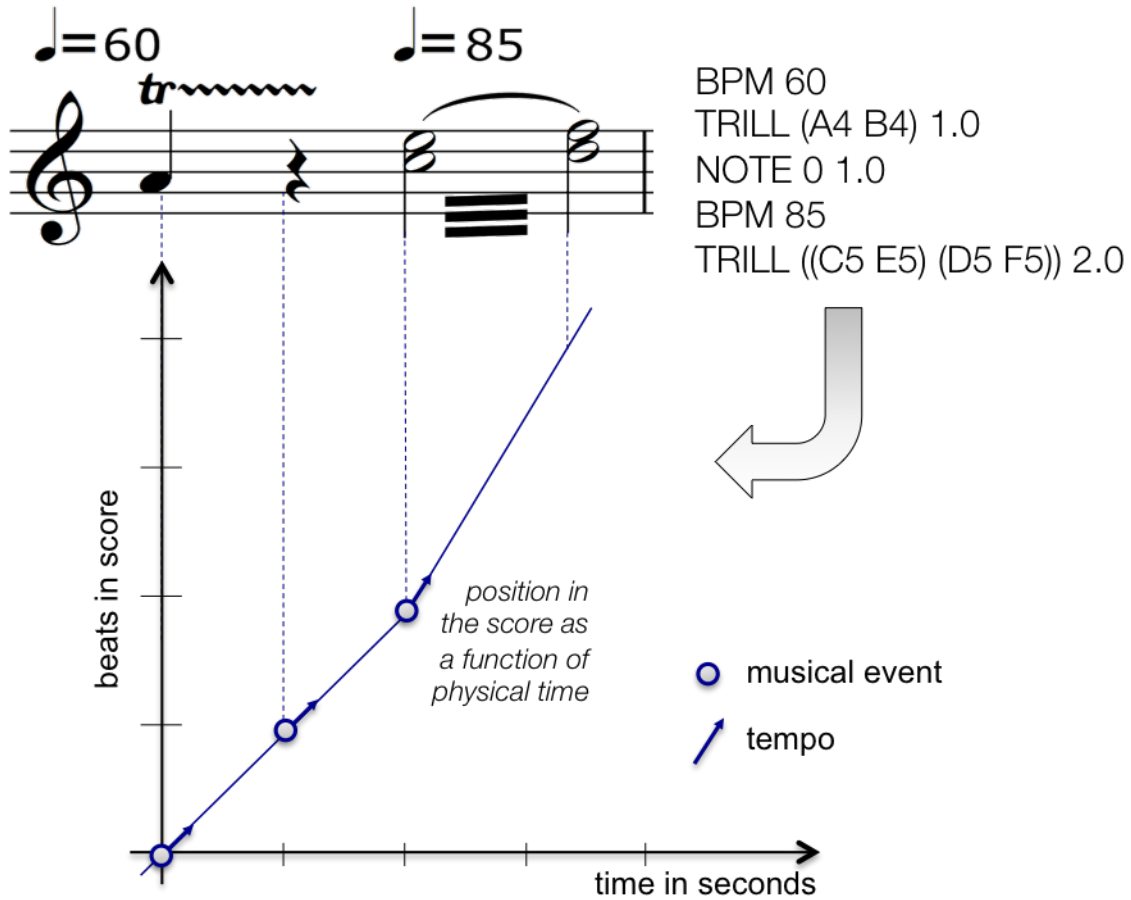


Figure 13.19: timing extraction

The occurrence of a musical event, represented by a circle and a vector, is used to give the tempo at this event. The vector represents a quantity measured in *beats per second*. The knowledge of the tempo can be used to compute the advancement in the score between two events, hence the potential date of each electronic action even if they are not synchronized with an event. For the sake of the simplicity, we suppose that the tempo is constant between two events¹⁶.

We call the previous map a **time-time** diagram because it relates the (potential) time in the score (in beats) with the physical time (in seconds). The previous map is completely

¹⁶This property is assumed in the current version (0.9x). Future versions will consider the specification of non piece-wise constant tempo like *accelerando*.

defined by the set of pairs (position of event, tempo at event)

$$\{ (position_1, tempo_1), (position_2, tempo_2), \dots \}$$

which formalizes what we called “time” in the previous paragraphs. The potential time extracted from the score enjoys an important property:

the potential position can be computed as the integral of the potential tempo

A consequence is that the time-time diagram of potential time is a continuous curve.

The Actual Musician Time

During the performance, musicians interpret the score with precise and personal timing, while the potential score time (in beats) is evaluated into the physical time (measurable in seconds). For the same score, different interpretations lead to different temporal deviations, and musician’s actual tempo can vary drastically from the nominal tempo marks. This phenomenon depends on the individual performers and the interpretative context.

The passing of time for the performer can be observed through the production of the musical events, so the information is restricted to the date of the occurrence these events. However, there are several methods to estimate the current tempo from the dating of the past events. The *Antescofo* approach is based on a study by Large and Jones¹⁷ but other approaches may still be relevant. In other words, the actual time of the musical events can be defined by a set of triples (date of event, position of event, tempo):

$$\{ (date_1, position_1, tempo_1), (date_2, position_2, tempo_2), \dots \}$$

From this information, a time-time diagram can be built to represent the passing of time for the performer (during the performance). But such diagrams will be merely formalities. There are indeed several ways to interpolate the positions between two events but no privileged way to choose one against the other, because there is no observation besides the musical events.

However, the tempo estimation at a particular event can be used to **forecast** the arrival of the next event¹⁸. The actual arrival may happen earlier or later than the predicted one: So, the relation between the actual position and the actual tempo is **non-newtonian**: the integration of the actual tempo gives only an approximation of the actual position.

This approximation can be seen as the result of the indetermination of the actual tempo at any instant. We advocate that this approximation is of a more fundamental nature: there is a divide between instantaneous discrete events (the onset of a note) and a general pace fixing the elapsing of a duration. The latter is a global and averaged quantity which does not prohibit the performer to advance or to postpone locally the occurrence of an event. Furthermore, the value of the tempo cannot be checked in between events.

Articulating Time

A unique feature of *Antescofo* is that it explicitly considers a time reference dedicated to the scheduling of the electronic actions. This time reference is specified by the composer and is

¹⁷E. Large and M. Jones. [The dynamics of attending: How people track time-varying events](#). *Psychological review*, 106(1):119, 1999. Other approaches may be considered.

¹⁸If an event e at position p happens at instant t with tempo T , then, the next event e' at position p' is predicted to happen at instant $t + \frac{p'-p}{T}$, *i.e.*, we suppose that the tempo remains constant between the two events and use a linear extrapolation.

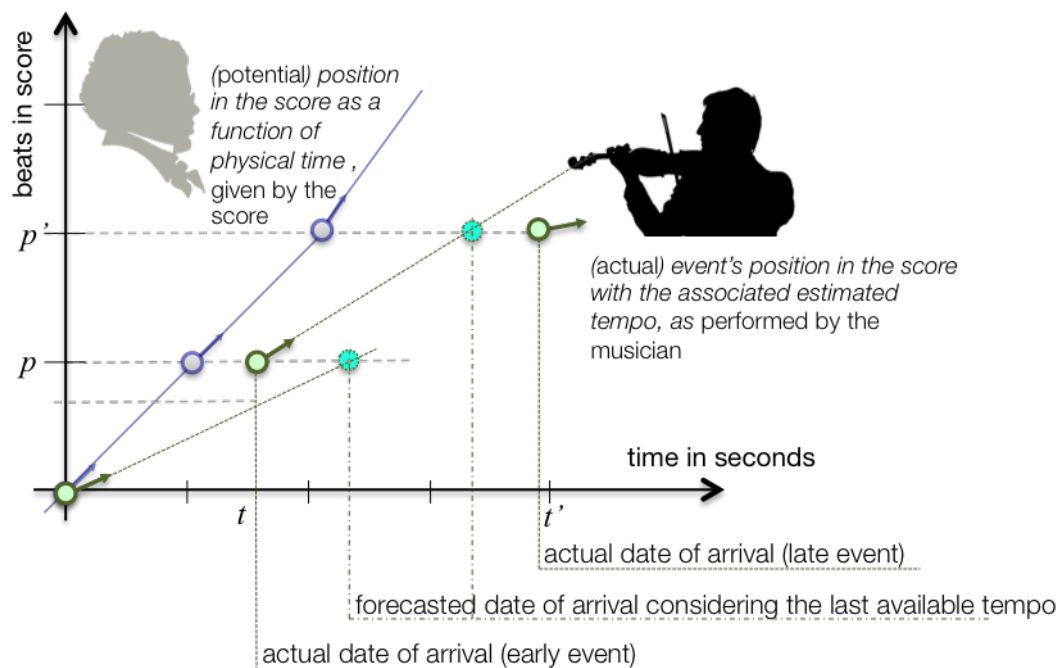


Figure 13.20: actual timing

dynamically computed during the performance, with respect to the potential time specified in the score and the actual time of the performer. This time-reference corresponds to a time-time map which is used to interpret beat positions, delays and durations involved in the actions.

This time reference is called a **temporal scope**. A temporal scope can be associated to each sequence of actions. By default, a sequence of actions **inherits** the temporal scope of its enclosing sequence of actions.

A temporal scope is defined by a **synchronization strategy** which defines how to “fill the gap” between the actual occurrence of events. There is a whole spectrum of synchronization strategies following the use of the information of position and the information of tempo. The interested reader will find a patch that can be used to compare the effect of the various synchronization strategies on a sequence of actions [at this page](#).

At one end of the spectrum, only the tempo information is used. This synchronization strategy is called `[@loose]` and illustrated below. With this strategy, the position of the successive events are not taken into account. Only the occurrence of the event triggering the sequence of actions is meaningful.

At the other end of the spectrum, the information of position is taken into account for each events. The tempo is only used to interpolate the change in position between two events. This is the `[@tight]` strategy. If an event happens earlier than expected, there is a jump from the current position p to the event's position p' . If an event happens later than expected, then the strategy qualifier `[@conservative]` freezes the position from time t (the date of the expected arrival) until the event's arrival at t' .

One can notice that the `[@loose]` strategy gives a smooth evolution of position with physical time, compared to the `[@tight]` synchronization strategy that may jump between position or may froze a position. The `[@tight]` strategy is relevant for actions whose progression must be

tempo
time-driven

position
event-driven

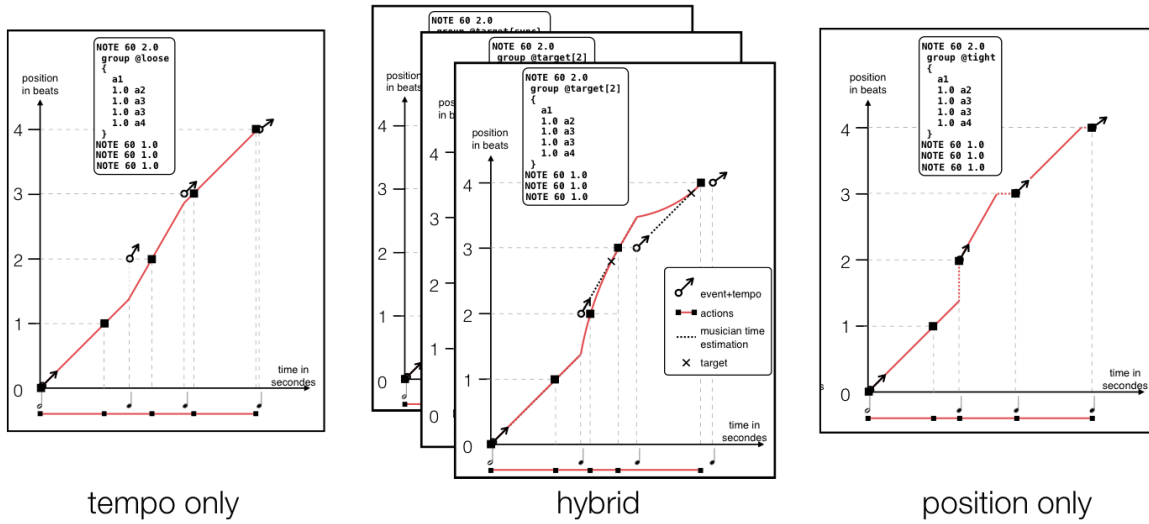


Figure 13.21: a spectrum of synchronization strategies

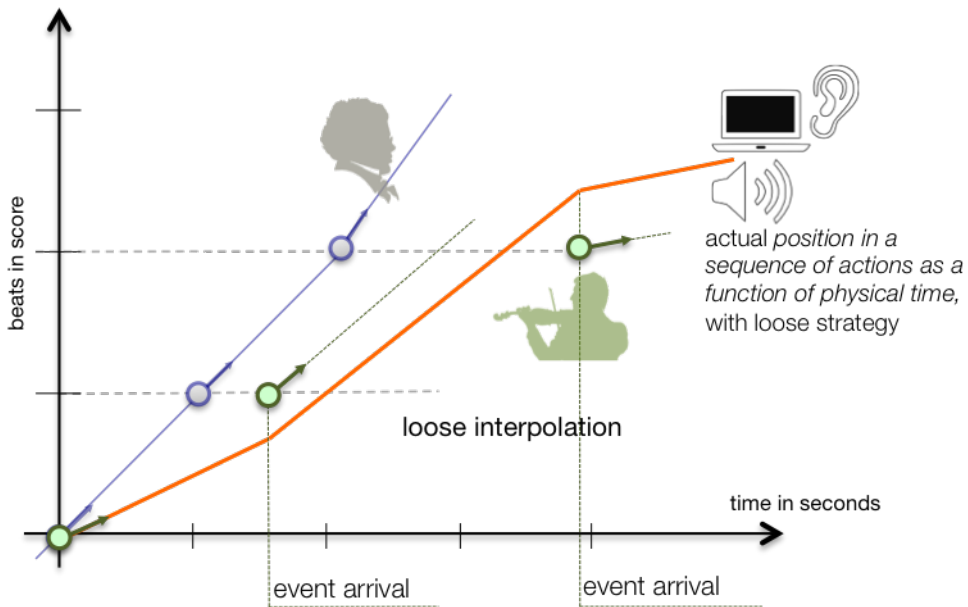


Figure 13.22: loose synchronization time-time map

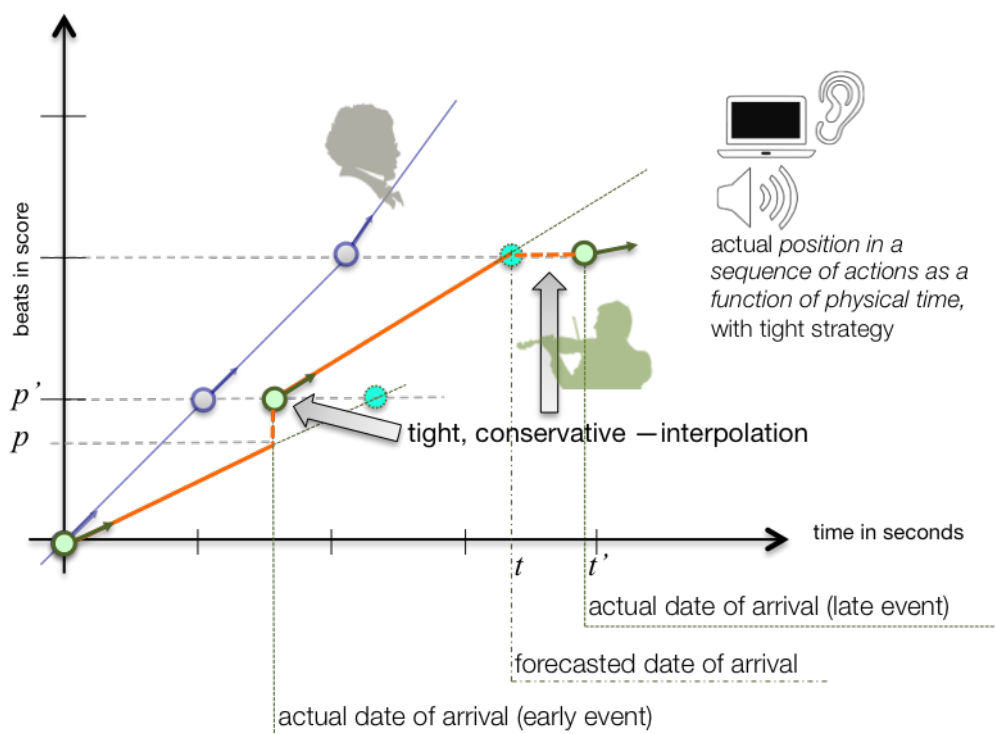


Figure 13.23: tight synchronization time-time map

synced with the onset of musical events. In between [*@loose*] and [*@tight*] behaviors, *Antescofo* offers strategies corresponding to an actual action time that catch up more or less smoothly with the musical events. They are all described in section [synchronization strategies](#).

The two examples of time-time diagrams for the actual timing of actions calls for several important remarks:

- Such a diagram can only be built in real-time, *i.e.* it is known only incrementally¹⁹ with the progression of the performance.
- At an instant t , the tempo T is known because there is a method to extract the tempo information from the past audio input²⁰. *Antescofo* assumes that the current tempo is known at each musical event and it is supposed to remain constant between events (in absence of specific [BPM] specification in the score).
- The position can be a *discontinuous and partial function of time*.
- The relation between the actual position and the actual tempo is **non-newtonian**: the actual position *is not* the integral of the actual tempo²¹.

¹⁹At some point in time, the change in position as time goes, is only a non-verifiable prediction until the occurrence of an observable event which can be used to fix the position.

²⁰E. Large and M. Jones. [The dynamics of attending: How people track time-varying events](#). *Psychological review*, 106(1):119, 1999. Other approaches may be considered.

²¹The last statement derives from the penultimate: the integral of a bounded quantity cannot be discontinuous.

They are indeed several ways to interpolate the positions between two events but no privileged way to choose one against the other. For instance, in the previous diagram, when an event happens later than expected, the position is frozen from the expected date of arrival t to the actual date of arrival t' . Another option, the `[@progressive]` attribute, would progress at the tempo rate, and jump back to the expected position when the event occurs (which means that the progression in the score is not monotonically increasing with physical time and makes a zig-zag).

Synchronizing with an Arbitrary Time

Our discussion focused on the specification of the actual time of electronic actions, by synchronization with a human performer. But for the *Antescofo* runtime, the human performer is simply a process that produces events associated with a tempo.

Such processes can be abstracted in *Antescofo* with a **tempovar**: a **tempovar** is a variable introduced with the `[@tempovar]` declaration. Assigning this variable corresponds to an event and a tempo is automatically derived using the algorithm used for the human performer by the listening module.

It is then possible to specify that a sequence of actions synchronized relatively to this tempovar. The only difference with the synchronization with the performer is that the performer follows an arbitrary score while a tempovar is supposed to be assigned periodically²².

A Side Note on Logical Time *versus* Actual Time

One conceptual advance in the field of real-time programming was the acknowledgment that time is a denotable entity, not an operational property: real-time programming language must include time in their domain of discourse.

As a consequence, modern programming languages that explicitly embed timing information within the code refer to a **logical time** decoupled from **physical time**. In this way, programs can be designed without the burden of external and operational factors, such as machine speed, portability, and timing behavior across different systems. It is then the responsibility of a compiler, an interpreter or a runtime to map this logical time with the physical time: in real-time systems, logical time aims to keep up with physical time (one logical second of logical time must take exactly one wall clock second); in non-real-time situations, logical time may run “as fast as possible” (*e.g.* in offline processing).

It is tempting to compare the relationships between the potential time of the score and the actual time of the electronics with the relationships between the logical time of a real-time program and the physical time of its realization. This analogy is misleading. The relationship between the potential time and the actual time of the electronic actions is not similar to the relationship between a specification and its implementation. The difference between the logical time expressed in a real-time program and the timing of its implementation accounts for the details that can be neglected in the realization. On the other hand, the potential time in the score is a partial specification. The actual time in which a sequence of actions takes place is built by a combination of three sources of information: (1) the potential time in the score, (2) the actual time of the musical events, and (3) the synchronization specifications given by the composer.

²²Experimental extensions are considered to remove this restriction.

From this point of view, *Antescofo* differs for all other music programming languages. All music programming languages we know support only *one logical time*. Languages may offer several time units, like seconds, milliseconds or samples. But these unit are *a priori* inter-convertible (we know once and for all that $1000ms = 1s$) and they refer to the same underlying time. The time used to schedule the electronic actions in this language is the logical time of the system.

The next section investigates the ordering of events in one instant. Then we present the synchronization strategies in depth. Finally, the last section of this chapter consider the handling of errors: as a matter of fact, our previous discussion neglected the fact that some musical events specified by the score may never happen in actual time because listening module's errors or performer's errors.

Action Priority

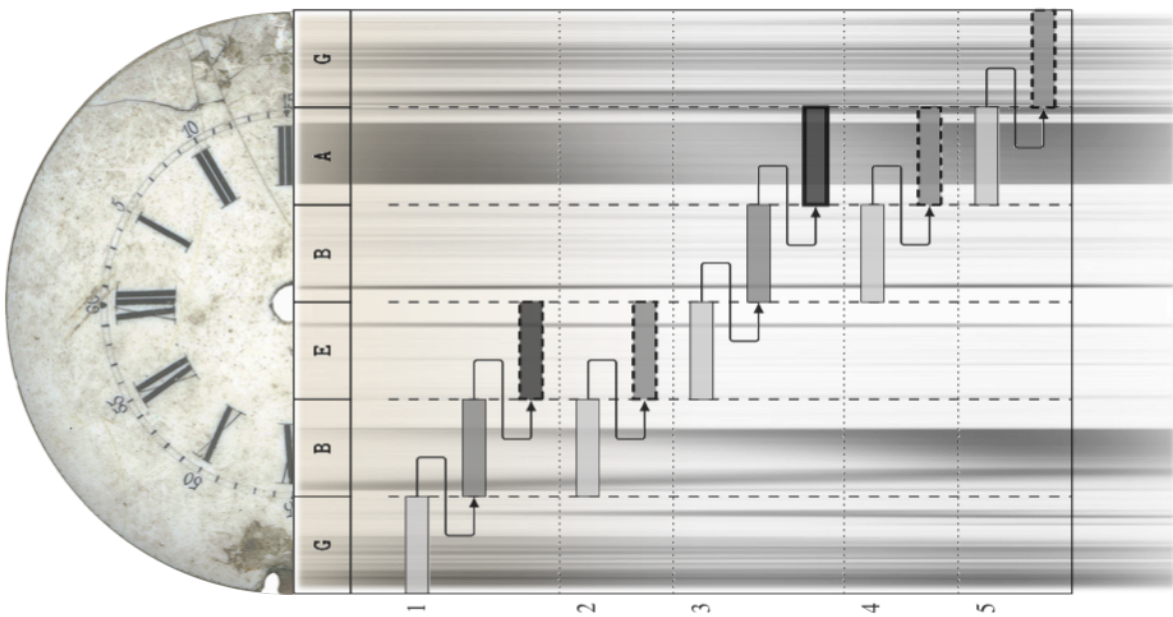


Figure 13.24: thickness of an instant

The Thickness of an instant

Each action performed by *Antescofo* occurs at some date and it may happen that several actions must be performed simultaneously “in the same logical instant”. This can be a problem. For example consider the fragment:

```
Group G1 { 1 $x := 0 }
Group G2 { 1 $x := 1 }
2 print $x
```

The two groups are launched in parallel and they schedule two incompatible assignments to be performed at the same date, after the expiration of a delay of one beat. The problem is to know what will be printed when we print the value of `$x`? If we assume an “ordinary” parallel execution, the outcome is not defined and the result is either 0 or 1 but not deterministically: it varies from one execution to the other.

The [synchrony hypothesis](#) used in the development of real-time embedded systems assumes that *the actions that occur at the same date are performed in a specific and well defined order*. *Antescofo* fulfills the synchrony hypothesis and the purpose of this section is to explain the execution order used to schedule actions at the same date. The rule is simple:

Two action instances that occur at the same date are ordered by their order of appearance in the score and if they are instances of the same action, they are ordered by seniority, except for the body of a whenever that are performed following their causal activation order as soon as possible*.*

Achieving a deterministic execution was a major goal for *Antescofo*. So we describe in great details the ordering of simultaneous actions. However, the rest of this section can be skipped in a first reading.

Same Execution Date

A first remark is necessary: in this section, when we speak about actions scheduled at the same date, we refer to two actions that must be performed at the same physical date, irrespective of their specification in the score.

Two actions that are specified at the same date in the score may well lead to two distinct execution dates. For example, in

```
NOTE C4 1
  Group H1 @loose
  {
    1 $x := 0
    ; ...
  }
  Group H2 @tight
  {
    1 $x := 1
    ; ...
  }
NOTE D3 1/2
```

the two assignments, which occur at the same date in the potential time of the score, may not happen at the same date during the performance because the groups H1 and H2 do not have the same synchronization strategy:

- the assignment in H2 is performed when note D3 occurs,
- while the assignment in H1 is performed 1 beat after the occurrence of C4 (and the conversion from beat to physical time rely on the tempo estimated at C4).

These two instants are not necessarily the same: event D3 may occur earlier or later than the specification given in the score. Thus, the value of $\$x$ depends of “external” events (the musical events produced on stage) even if they coincide in the score. These external events are not deterministic and do not depend on *Antescofo* itself.

Conversely, two unrelated actions may, by chance, occur at the same date. For example:

```
NOTE E4 0.3
2 Group I1 { 3 $x := 0 }
3 Group I2 { 2 $x := 1 }
```

The two assignment to $\$x$ occur at the same date because the sum of the delays occurring from the initial event (the occurrence of the musical event which triggers the actions) are the same ($2 + 3 = 3 + 2$). If they are really unrelated, their execution order probably does not matter. But there are other cases when two actions are clearly related in the score, are scheduled for the same date, and indeed are executed at the same date. In this case, order matters and the behavior of *Antescofo* must be easy to understand, deterministic and relevant.

The Syntactic Ordering of Actions

The presentation in this paragraph and the next, does not apply fully to the actions spanned by a *whenever* which will be discussed below.

With the exception of *whenever*, the execution order followed by *Antescofo* is simple: when two actions are scheduled at the same date, the *syntactic order* \prec of appearance in the program is used to determine which one is scheduled first. The syntactic order is roughly the order of appearance in the linear score but takes into account the nesting structure of compound actions.

More precisely, a vector of integers $w(a)$, called the *location* of a , is associated with each action a . This vector locates uniquely the action a in the syntactic structure of the score. Two actions a and a' scheduled at the same date are performed following the lexicographic order of their location $w(a)$ and $w(a')$.

In the next example, we write vectors by listing their element separated by a dot: 1.2.3 is the vector with the three elements 1, 2 and 3. The location $w(a)$ associated to an action a is build as follows:

- Top-level actions (appearing before the first musical event or associated to a musical event) are identified by their rank i of appearance in the score: $w(a) = i$.
- The i th action of a compound action G , is located at $w(G).i$.

The lexicographic order is best explained in an example. The localization of each action is given on the left

```
1           $i := 0
2           Loop L1 1
           {
2.1         print loop L1 iteration $i at $RNOW
2.2         $i := $i + 1
```

```

    }
3      $j := 0
4      Loop L2 1
    {
4.1          print loop L2 iteration $j at $RNOW
4.2          $j := $j + 1
    }

```

and the actual trace is

```

loop L1 iteration 0 at 0.0
loop L2 iteration 0 at 0.0
loop L1 iteration 1 at 1.0
loop L2 iteration 1 at 1.0
loop L1 iteration 2 at 2.0
loop L2 iteration 2 at 2.0
loop L1 iteration 3 at 3.0
loop L2 iteration 3 at 3.0
loop L1 iteration 4 at 4.0
loop L2 iteration 4 at 4.0

```

Nota Bene:

- The loop has a location which is distinct from the location of its body.
- The syntactic order does not take into account the fact that an action may have several occurrences (this will be handled in the next paragraph).

This program exhibits several actions that occur at the same date:

- The assignment to $\$i$ and the start of the loop appears at the same date. The assignment is performed first because $1 < 2$.
- For the same reason, the assignment to $\$i$ is performed before the assignment of to $\$j$ and before the start of the loop . The start of loop is executed before the assignment to $\$j$, *etc.*
- At time n , two prints occur together but the print message in L1 is issued before the print message in L2 because $2.1 < 4.1$.

A Full Temporal Address with 3 Components

The syntactic order is based solely on the syntactic structure of the score and neglects the difference between an action and the (multiple) realizations of this action (called *instances*): for example, an action a in a loop is performed at each iteration. All of these instances are associated to the same location $w(a)$. To compare these actions, that share the same location, we use their instance number.

This way, the temporal address of the execution of an action a has 3 components:

$(date, w(a), instance_number(a))$

Temporal addresses are *lexicographically ordered*:

- if two actions have the same date, then their locations (given order in the score) are used,
- and if two actions have the same date and the same location, they are compared using their instance number (which is assigned at runtime and not by the composer).

In other words: two action instances that occur at the same date are ordered by their order of appearance in the score and if they are instances of the same action, they are ordered by seniority.

Relevance

The resulting order \ll is total: two different actions a and b are always comparable and $a \ll b$ or $b \ll a$. Thus, this order entails a deterministic execution. The order \ll is not necessarily the order which is needed and there is no way to alter it in *Antescofo*. However, the corresponding scheduling seems relevant on several paradigmatic examples.

For instance, a classical problem is given by two nested loops:

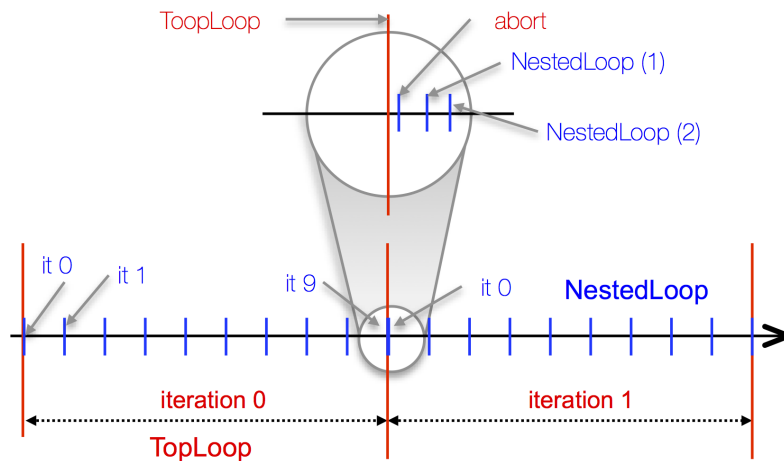


Figure 13.25: nested loop

```
$lab := 0
loop TopLoop 1
{
  abort $lab
```

```

$lab := {
  Loop NestedLoop 0.1
  {
    $X := $X + 1
  }
}

```

The loop `TopLevel` iterates a nested loop `NestedLoop` which assigns variable `$X`. The command launched at iteration n of the `TopLevel` loop is supposed to kill the `:::antescofo` `NestedLoop` spanned at the previous iteration to avoid two assignments of `$X` at the same date. The situation is pictured at the left of the program. Remark that the expected behavior can be achieved without an explicit `abort` using the `[@exclusive]` attribute.

Several actions share the same date:

- The 10th assignment to `$X` in the i th instance of `NestedLoop`. The temporal address of this assignment is $(i, 2.2.1.1, 10i)$.
- The first assignment to `$X` in the $i+1$ th instance of `NestedLoop`. The temporal address of this assignment is $(i, 2.2.1.1, 10i + 1)$.
- The `abort` command issued by the i iteration of `TopLoop`. The temporal address of this action is $(i, 2.1, 10i)$.

The final value of depends on the order of executions of these three instances. For example, this result differs if the `abort` command is issued after the two assignments or before. Because we have

$$(i, 2.1, 10i) \ll (i, 2.2.1.1, 10i) \ll (i, 2.2.1.1, 10i + 1)$$

the command `abort` is issued first and cancel the 10th assignment. So, when `:::antescofo` `TopLoop` is reiterated, there is only one assignment that corresponds to the first iteration of the new `NestedLoop`.

Scheduling of Whenevers

Whenevers span the execution of their body when activated by variable's assignments. Thus, if an activation occurs at the same date as the firing of another action a , the order of the two depends of relative order between the assignment and a : **the whenever body is activated as soon as the variable is assigned** and *if two whenevers are activated by the same variable, they are activated following their syntactic order*. The resulting ordering cannot be solely deduced from the syntactic structure of the score. It is however deterministic.

Here are several examples. In the following fragment:

```

whenever W1 ($x > 0) { print A }
whenever W2 ($x > 2) { print B }
let $x := 3

```

will print

A
B

the trace produced shows that W1 is activated before W2. Indeed, the two whenevers are activated by the same cause: the assignment to \$x. In this case, the whenevers are activated following the syntactic order explained above.

In this example

```
whenever W1 ($x) { print A }
whenever W2 ($y)
{
    print B
    let $x := 1
}
whenever W3 ($x) { print C }
let $y := 1
```

will print

B
A
C

the activation order is W2 W1 W3 because the activation of W1 and W3 are caused by the assignment in the body of W2: so they cannot appear before the activation of W2. Then, the activation of W1 and W3 is done in this order, following their syntactic order.

The next example shows that the order of activation is dynamic, *i.e.* it may depend of the values of the variables

```
whenever W1 ($x) { print A }
whenever W2 ($y) { print B }
if ($x > $y)
{
    $x := $x + 1
    $y := $y - 1
}
else
{
    $y := $y + 1
    $x := $x - 1
}
```

if \$x > \$y it will print

A
B

else it will print

B
A

In addition, do not forget that a **whenever** is activated at most once in a logical instant. So in the trace of the following fragment:

```
whenever W1 ($x || $y || $z)
{ print A }
whenever W2 ($x || $y)
{
    print B
    $z := true
}
$x := true
$y := true
```

will print

A
B

A and B appear only once.

Synchronization Strategies

The musician's performance is subject to many variations from the score. There are several ways to adapt the timing of the electronic actions to this musical indeterminacy based on the specific musical context.

An electronic phrase is written that specifies delays between each action in a block (group, loop, whenever, curve, *etc*). Through specific attributes, a particular **synchronization strategy** defines the temporal evolution of this phrase depending on the musician's performance. More generally, a synchronization strategy specifies the temporal relationships between the actual timing of a sequence of actions and the actual timing of a sequence of events, see the previous section **articulating time**. The relevant synchronization strategy is determined by the musical context and is at the composer or arranger's discretion²³.

From a synchronization perspective, the musical performance can be summarized by two parameters: the musician's position (in the score) and the musician's tempo. These two parameters are computed by the listening machine from the detection in the audio stream

²³The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

of the events specified in the score. Synchronization takes them into account. The observed position in the score, for example, can be used to fix the position in the sequence; the tempo estimation can be used to compute the evolution of an action's position between two events and also to anticipate the arrival of future events.

An [error handling strategy](#) defines what to do with the action associated to an event that is never recognized (the origin of this “non-recognition” does not matter).

Temporal Scope

The system maintains a **temporal scope** for each sequence of actions (groups, loops, curve, *etc.*). A temporal scope defines

- a *local position* (in beats): which represents the state of the progression when performing the sequence of actions;
- and a *local tempo* (in beat per second): which represents the pace of the progression in the sequence of actions.

The synchronization attributes associated with a compound action define the temporal scope of the action relative to another temporal scope. By default, the referenced temporal scope comes from the actual musician's performance. But it is possible to specify another using the `[@tempo]` attribute or using the `[@sync]` attribute referring to a variable introduced by a `[@tempovar]` declaration.

In absence of specifications, a temporal scope is inherited from the enclosing compound action. The sequence of actions at top-level, are implicitly synchronized with a `[@loose]` synchronization strategy with the musician (cf. below).

The synchronization attributes are described below. They define how the position and tempo in the sequence of actions depends of the musician's position and tempo:

- `[@loose]` uses only the musician's estimated tempo to synchronize the actions;
- `[@tight]` primarily uses the position information to synchronize the actions;
- `[@target]` is an intermediary between tight and loose strategies, aimed to dynamically and locally adjust the tempo of a sequence for a smooth synchronization with anticipated events.

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

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

Only one synchronization can be specified:

- the synchronization attributes `[@sync]`, `[@tempo]`, `[@loose]`, `[@tight]` and `[@target]` are mutually exclusive;

- [`@progressive`] is exclusive from [`@conservative`] but they can be combined with [`@target`] and [`@tight`] synchronization strategies;
- [`@latency`] can be used to correct some latency problems, independently of the chosen synchronization strategy;
- [`@ante`] and [`@post`] are experimental features not described here.

If no synchronization attribute are specified, then the corresponding group is [`@loose`].
 {!BNF_DIAGRAMS/synchro_attributes.html!}

Loose Synchronization

Once a “loose” 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 the default one but an explicit [`@loose`] attribute can be used. The implicit top-level group that encompasses the sequence of actions linked to a musical event is loose by default (this can be changed using the `top_level_groups_are_tight` command).

The [`@loose`] attribute can be followed by a list of events: in this case, the change in the musician’s tempo is considered only at these events (else they are considered on each musical event).

{!BNF_DIAGRAMS/loose.html!}

The figure below attempts to illustrate this within a simple example: the diagram 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.

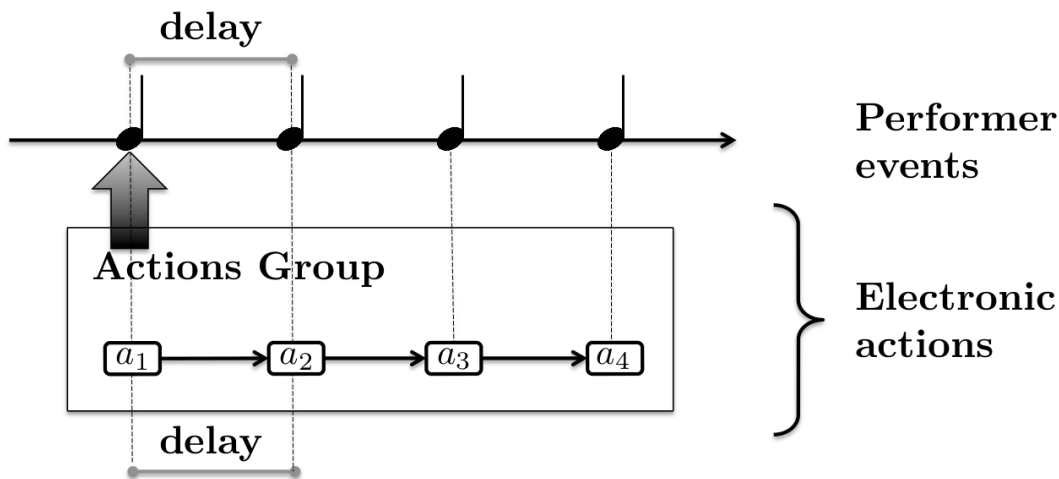


Figure 13.26: loose ideal synchronization

In a regular score following setting (*i.e.*, correct listening module) the action group is launched in synchrony with 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 the diagram below where the performer hypothetically accelerates the consequent events in the score.

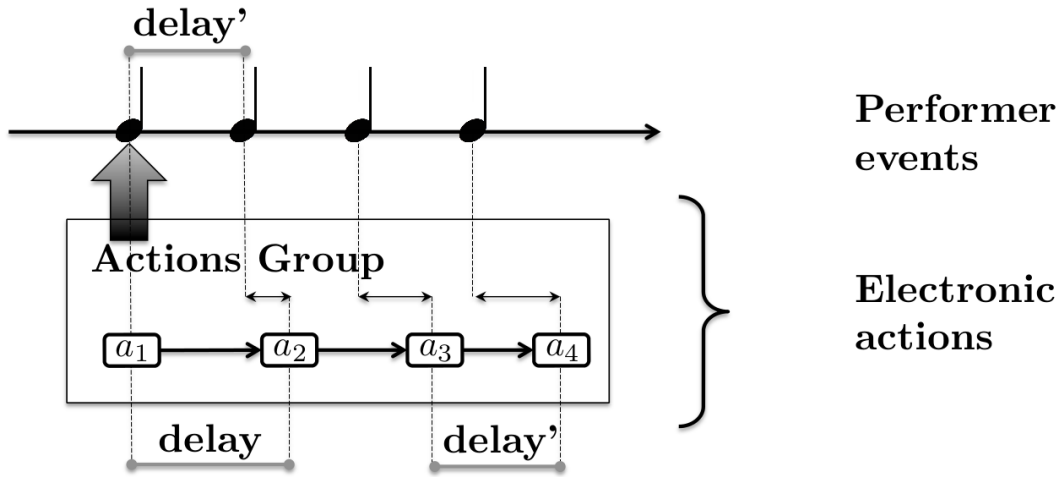


Figure 13.27: loose synchronization when accelerando

In this diagram the performer hypothetically decelerates the consequent events in the score.

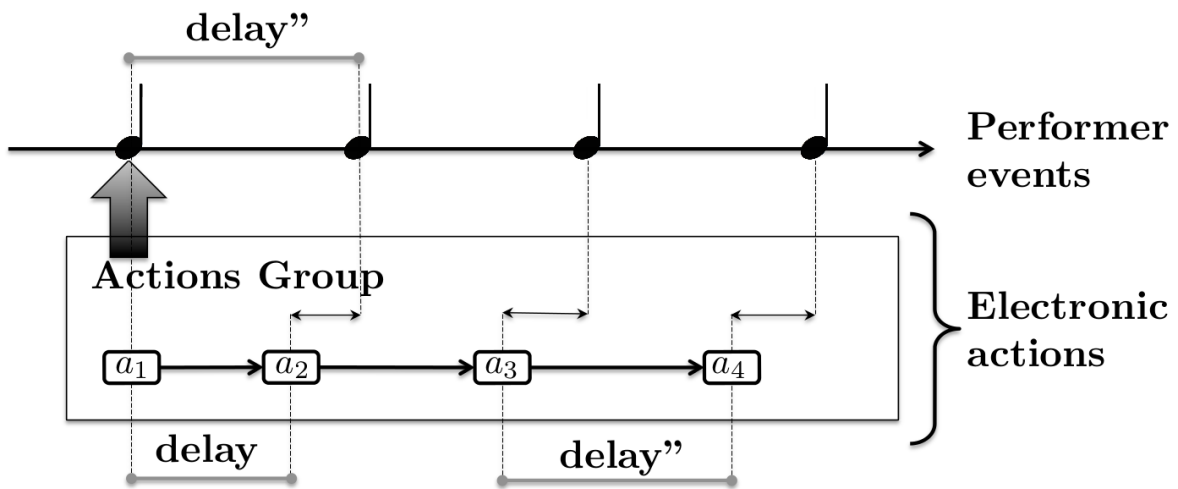


Figure 13.28: loose synchronization when rallentando

In these two cases, the delays between the actions will grow or decrease. The tempo inferred by the listening machine converges towards the actual tempo of the musician. Therefore, the delays, which are relative to the inferred tempo, will also converge towards the delay between the notes observed in the actual performance.

So, this 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.

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.

The [`@tight`] attribute without parameters considers all musical events to find the nearest event in the past. If parameters are provided, with the syntax `@tight := { label1, label2, ... }` only the musical events referred by their labels are considered.

```
{!BNF_DIAGRAMS/tight.html!}
```

Tight groups allow the composer to avoid segmenting the actions of the group into 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 behavior. During the execution the system synchronizes the next action to be launched with the corresponding event.

The implicit group that encompasses the sequence of actions linked to a musical event, uses the [`@loose`] synchronization strategy. This behavior can be changed to generate [`@tight`] groups by specifying the `top_level_groups_are_tight` keyword at the beginning of the score (the change of behavior is global for the entire score).

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

Target Synchronization

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

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

Static Targets

In some cases, a smooth time evolution is needed, but some specific events are temporally meaningful and must be taken into account. For example, this is the case when two musicians plays two phrases at the same time: they usually try to be perfectly synchronous (tight) on some specific events while other events are less relevant. These tight events can correspond to the beginning, or the end of a phrase or to other significant events commonly referred to as **pivot events** or **attractors**. *Antescofo* lets the composer list pivot events for a given block.

During the performance, the local tempo of the block is dynamically adjusted with respect to the actual occurrence of these pivots, cf. figures below. In the following example:

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

the local tempo of the group will be computed depending on the successive arrival estimations of events e_5 and e_{10} . Notice that the pivots are referred to by their label and listed between braces.

{!BNF_DIAGRAMS/static_target1.html!}

The second syntax $a \% b$ is used to specify that the pivots are the event located at current position $+ a * n + b$ beats (for $n \in \mathbb{N}$).

{!BNF_DIAGRAMS/static_target2.html!}

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

Dynamic Target

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

{!BNF_DIAGRAMS/dynamic_target.html!}

In the following example:

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

the tempo and the position of the actions converge to the tempo and the position of the musician. The convergence date is not an event (as in static target) but is fixed by the

following property: a difference of 1 between the position of the actions and the position of the musician is corrected in 2 seconds. The syntax [2] is used to specify a horizon in beats and [2#] to define a horizon in number of events.

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

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

Comparison between [`@loose`], [`@tight`] and dynamic [`@target`]

The figures below represent temporal evolution of an electronic phrase with several synchronization strategies. The time-time diagrams show the relationship between the relative time in beats and absolute time in seconds²⁴. The musical events are represented by vectors whose slopes correspond to the tempo estimation. The actions are represented by squares and the solid line represents the flow of time in the group enclosing these actions. From left to right and top to bottom, the strategies represented are : [`@loose`], [`@tight`], [`@target`]{sync}, [`@target`][2]. There is an illustrative patch that compares the effects of synchronization attributes²⁵.

How to Compute the Position in the Event of Conflicting Information

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

- With the [`@conservative`] attribute, the occurrence of events is trusted more than the tempo estimation to compute the musician's position. So, when the anticipated date of an event is reached, the computed position is stuck until the occurrence of this event.
- With [`@progressive`] attribute (the default), the estimation of the position will continue to advance even if the forecasted event is not detected.

Several system variables are updated by the system during real-time performance to record these various points of view in the position progression. They are used internally but the user can access their values. Variable \$NOW corresponds to the absolute date of the "current instant" in seconds. The "current instant" is the instant at which the value of \$NOW is required.

The variables \$RNOW and \$RCNOW are estimations of the current instant of the musician in the score expressed in beats. At the beginning of a performance,

$$\text{\$NOW} = \text{\$RNOW} = \text{\$RCNOW}$$

²⁴Read section [the fabric of time](#) for the notion of time-time diagrams.

²⁵The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

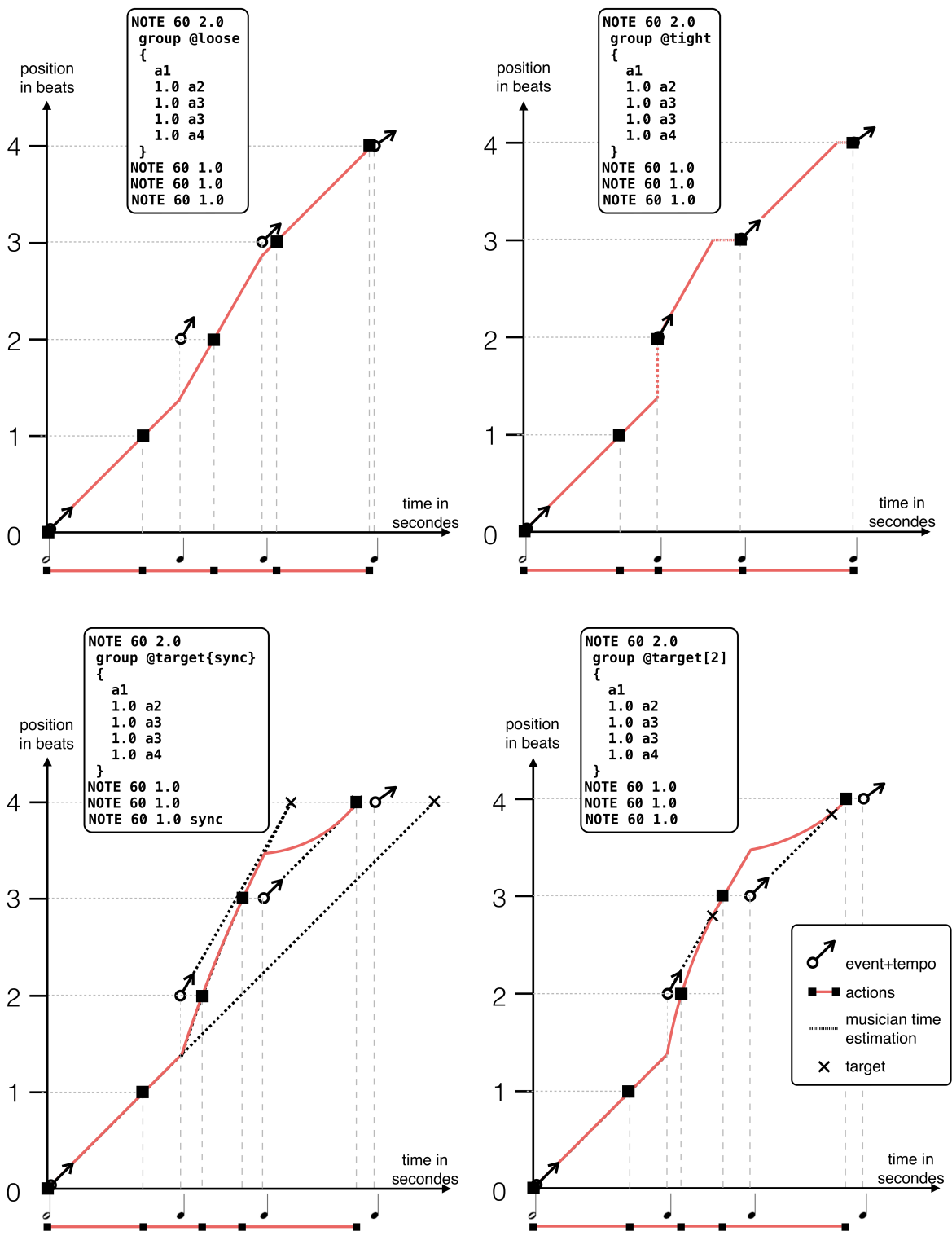


Figure 13.29: synchronization comparison

At other instants during the performance, let e_n be the last decoded event by the listening machine at time NOW_n , e_{n+1} the following event, p_n and p_{n+1} be their relative position in beats in the score, and del be the delay in beats since the detection of e_n . These variables are linked by the following equations:

$$del = NOW - NOW_n + \$RT_TEMPO / 60$$

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

$$\$RCNOW = p_n + del \quad \$RNOW = \min(\$RCNOW, p_{n+1})$$

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

From a musical point of view, the position estimation with variables is more reliable when an event is missed (the musician does not play the note or the listening module does not detect it) but sometimes the value has to “go back” when the prediction is ahead.

Specifying Alternative Coordination Reference

Explicit tempo specification [**@tempo**]

The tempo local to a sequence of actions can be specified by an expression. This makes the local position and the tempo of the sequence completely independent to that of the musician. For example:

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

will execute the child actions with a tempo of 70. The tempo can be defined by an expression. The variables of the expression are watched, much like the variable in the expression of a *whenever*. When these variables are updated, the expression is reevaluated, giving a new tempo value which is used to reevaluate on-the-fly all the pending delays

Here is an example:

```
Curve C1 @grain 0.05s
{ $t1 { {60} 5 {180} 5 {60} } }

Group G1 @tempo := $t1
{
  @local $x
  $x := 0
  Loop L 0.1 {
    $x := $x + 0.1
    plot $NOW " " $x "\n"
  }
}
```

The values of the variable $\$x$ in the loop are plotted in relative time at the left, and in physical time at the right. The linear progression in relative time is transformed into a quadratic progression made of two parabola, because the tempo variation is defined by a

piecewise linear function implemented by Curve C1 which goes from 60 to 180 and back to 60.

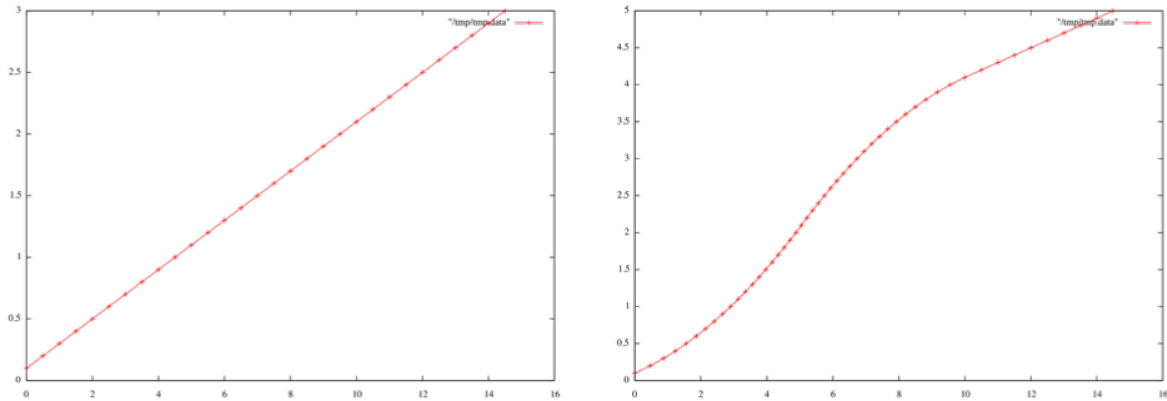


Figure 13.30: tempo specification

Synchronization on a temporal variable with `[@sync]`

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

Such variables are global variables introduced using the `@tempovar` declaration:

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

An update acts then as an event of duration 2 with a specified BPM of 60. The second argument of the declaration defines the increase in the “position of `$v`” each time the variable is updated. The first argument defines the initial “tempo” associated to this variable. This tempo corresponds to the expected pace of the updates. The position and tempo of a tempovar can be accessed using the dot syntax, cf. [temporal variable](#).

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

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

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

Latency Compensation

Latency compensation is an *experimental feature*. When attribute `@latency := 30ms` is specified for a sequence of actions, the runtime tries its best to anticipate the launch of each action by 30 milliseconds.

The anticipation is not guaranteed: it is taken on the delay preceding the actions in the sequence. So if the first action in the sequence is launched with no delay, the 30 ms cannot be compensated.

Missed Event Errors Strategies

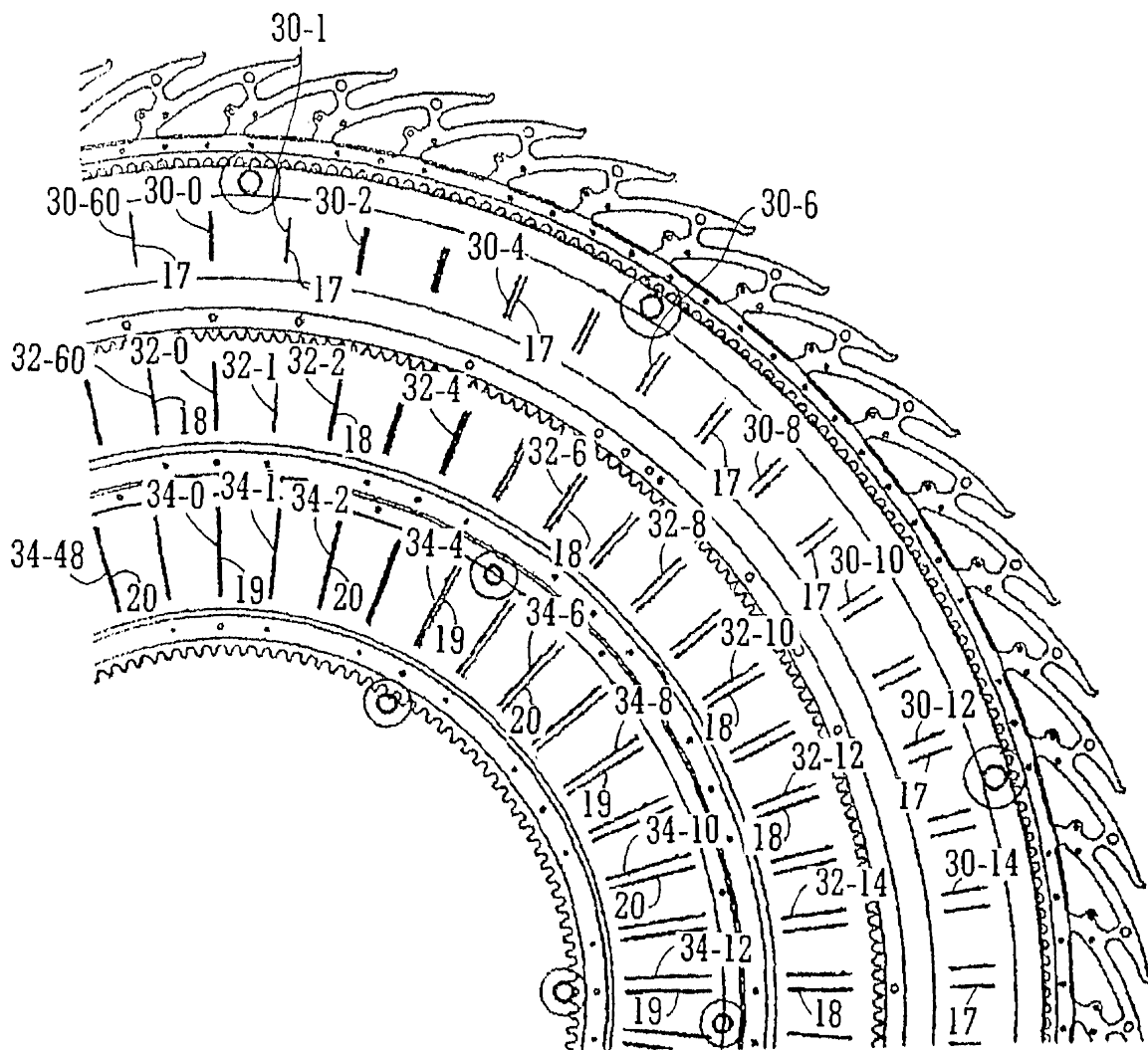


Figure 13.31: fragment of a clock mechanism

Some but not all of the errors during the performance (such as events missed by the performer, events that weren't detected by the listening machine, or events that the listening

machine mistakenly recognizes) are handled directly by the listening modules. The critical safety of the accompaniment part is reduced to the handling of missed events (whether missed by the listening module or human performer):

- In some automatic accompaniment situations, one might want to dismiss associated actions to a missed event if the scope of those actions does not bypass that of the current event at stake.
- On the contrary, in many live electronic situations such actions might be initialized for future actions to come.

It is the responsibility of the composer to select the right behavior by attributing relevant *scopes* to accompaniment phrases and to specify, using an attribute, the corresponding handling of missed events.

An action 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 an action being local or global is given to the discretion of the composer or arranger, through the specification of the `@local` or `@global` attribute. By default (*i.e.* without explicit specification), the actions at top-level are tagged `@global` and error behavior are inherited from the enclosing group.

Combining Synchronization and Error Handling.

The combination of the synchronization attributes and error handling attributes for a group of accompaniment actions gives rise to four distinct situations. Figure below attempts to showcase these four situations for a simple hypothetical performance setup.

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 action 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 an identity is similar to integral musical phrases that have strict starting points with *rubato* type progressions (free endings).

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

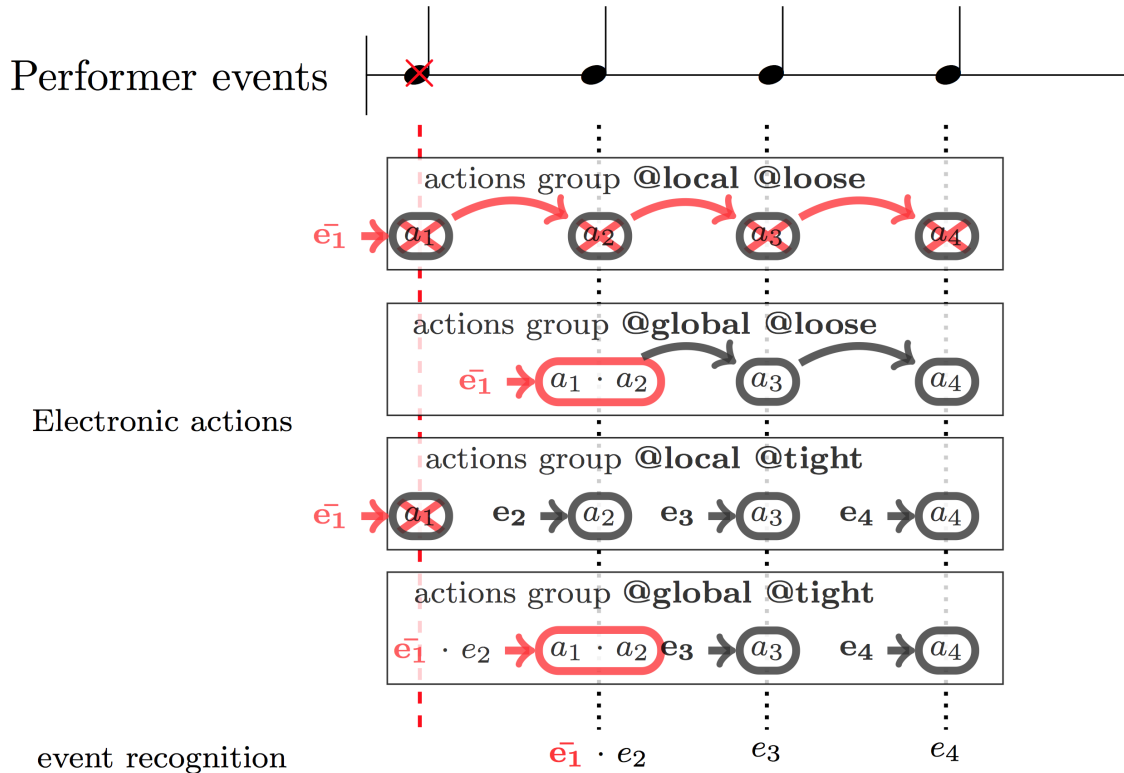


Figure 13.32: error management

To have a good understanding of the picture notice that:

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

Remarks:

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


```

NOTE C3 2.0
  d\ensuremath{_{1}} action\ensuremath{_{1}}
  d\ensuremath{_{2}} group G1
  {
    action\ensuremath{_{2}}
  }
NOTE D2 1.0

```

```

NOTE C3 2.0
Group @global @loose
{
  d\ensuremath{_{1}} action\ensuremath{_{1}}
  d\ensuremath{_{2}} group G1
  {
    action\ensuremath{_{2}}
  }
}
NOTE D2 1.0

```

During a performance, even in case of errors, if an action has to be launched it is fired at a date which is as close as possible to the date specified in the score. This explains the behavior of a group that is [`@global`] and [`@loose`] when its event trigger is recognized as missed. In this case, the actions that are still in the future are played at their “right” date, while the actions that should have been triggered are launched immediately (as a tight group strategy).

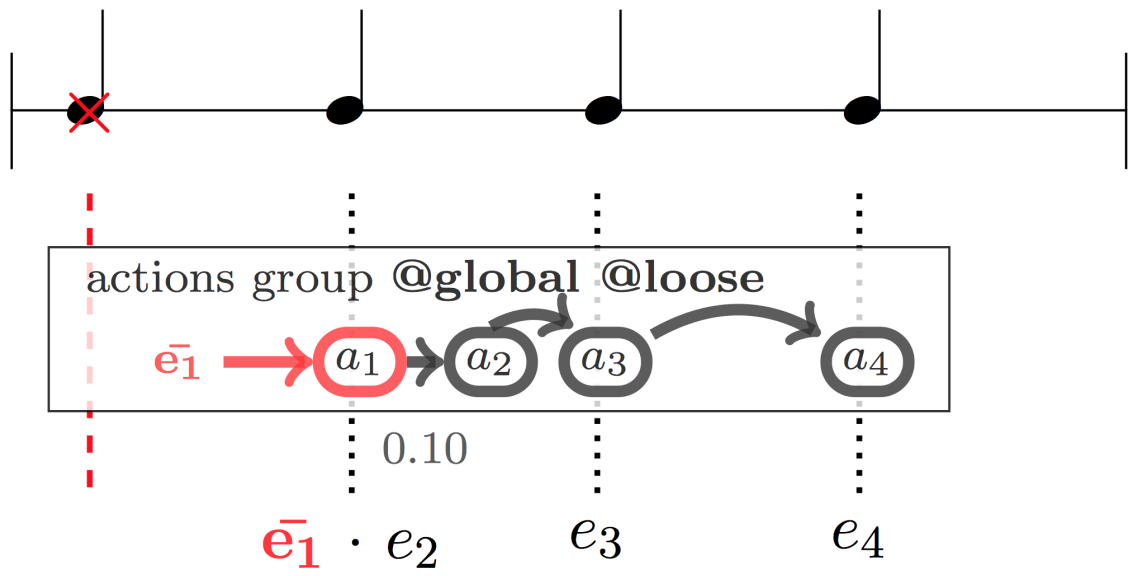
In the previous example, we remark delay variations (a_2 is directly fired for the `@loose @global` case and not 1.0 after a_1). This ‘tight’ re-scheduling is important if the a_2 action has a delay of 1.10, the action should effectively be fired at 0.10 beat after a_1 (next figure):

```

NOTE C3 1.0 e1
  group G1 @global @loose
  {
    a\ensuremath{_{1}}
    1.10 a\ensuremath{_{2}}
    1.0 a\ensuremath{_{3}}
    1.0 a\ensuremath{_{4}}
  }

NOTE D2 1.0 e2
NOTE D2 1.0 e3
NOTE D2 1.0 e4

```



Chapter 14

Expressions

details from the [Paul Klee notebook](#)

Expressions can be used to compute delay, period, local tempo, breakpoints in specification, and arguments of internal commands and external messages sent to the environment. Expressions are evaluated into [values](#) and this evaluation is supposed to take no time.

In this chapter

- we compare [actions and expressions](#) and we present [the three kind of expressions](#) that exist in *Antescofo*;
- we introduce the [general notion of a value](#) (each specific kind of value will be discussed at length in the following chapters);
- we detail [the notion of a variable](#);
- we explain the use of [variables to define a dynamic tempo](#) (a tempo that evolves in time);
- we present the [conditional expressions](#);
- and finally we explain [how to get the *exe* value associated to each running compound action](#).

Expressions *versus* Actions

Actions and expressions belongs to two clearly separated worlds in *Antescofo*:

- Expressions appear as parameters of actions and the evaluation of expressions is subordinated to the execution of actions.
- The evaluation of an expression does not last over time, thus the evaluation process can be more efficient than the execution of actions¹.

¹For instance, the implementation of a [Group](#) implies a state to maintain an environment accessible by the group's childs, it requires a scheduler to manage the delay, additional computation for the management of synchronization and the translation of relative delays into physical time, *etc.* Even if a compound action performs instantaneously, its execution is a little more costly than its corresponding expression.

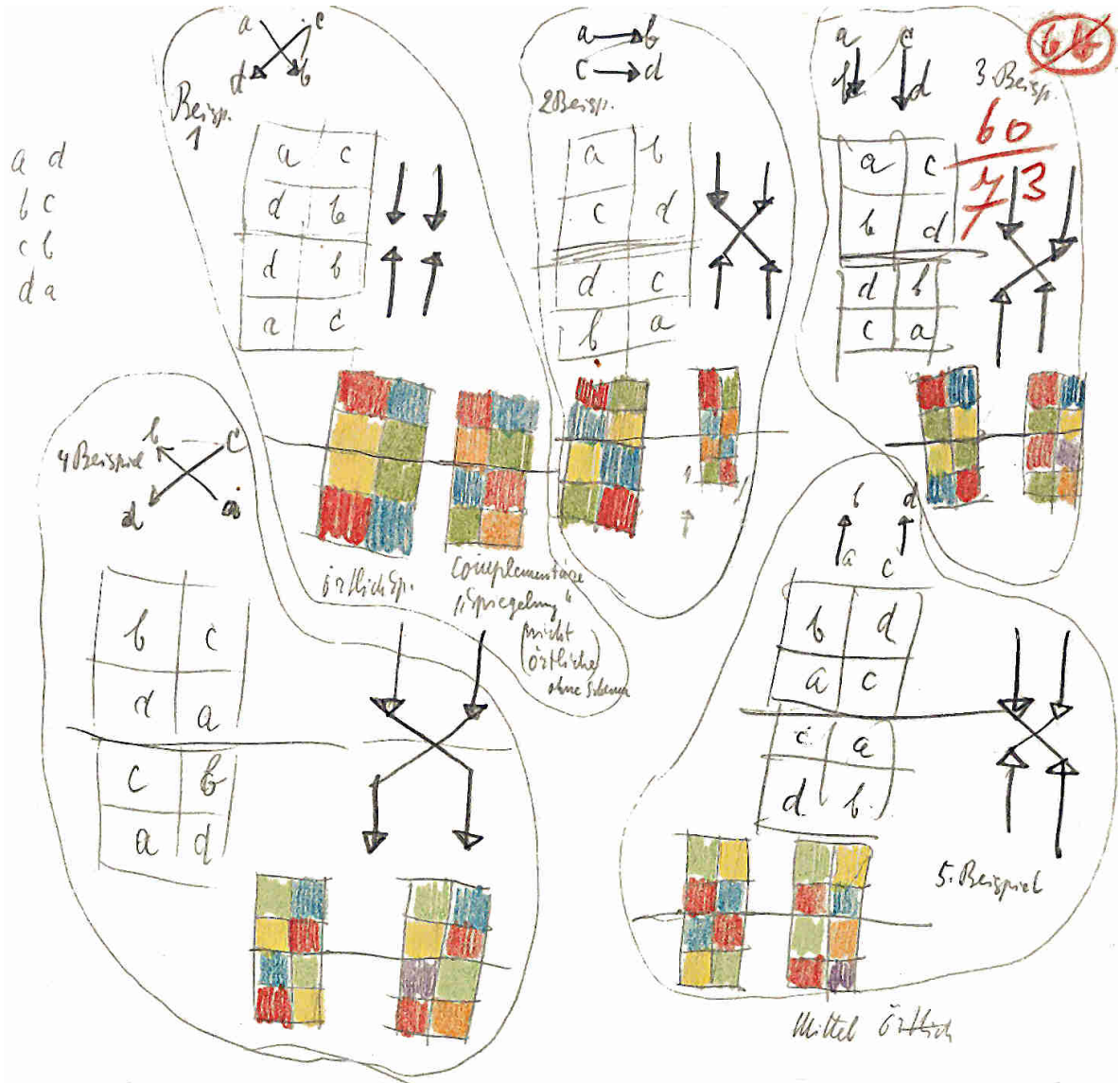


Figure 14.1: details from the Paul Klee notebook

This separation may appear sometimes somewhat artificial. For example, the assignment of a variable is an action (because it can trigger activities or alter synchronization or tempo information) but some variables are only introduced to store intermediate values in the computation of a complex expression. The `Loop` construct is an action but it is useful to implement iterative expressions. A process definition is a value (of type `proc`) and a handle to the current execution of a compound action is also a value (an `exec`). *Etc.*

To make the boundary between the two worlds more permeable, and to take into account the usual syntax of Max or PD message (they have no delimiters between arguments and the end of line is used as terminator), the *Antescofo* syntax distinguishes between three kinds of expressions. This distinction is only useful for syntactic reasons: they are not allowed to appear anywhere to make the parsing non-ambiguous but expressions “have the same rights” and are managed in the same way, irrespectively of their kind.

Three Kinds of Expressions

Expressions are categorized in three kinds of increasing generality, each including the previous one:

- **Closed expressions**² are best called *auto-delimited expressions*. These expressions are allowed in the specification of a delay, a breakpoint in a curve, the value of an attribute, *etc.* See section [auto-delimited expressions](#) below.
- **Simple expressions** are the usual expressions allowed anywhere else, for example in the right hand side of an assignment, as the argument of a function or process call, *etc.*
- **Extended expressions** are the expressions allowed in the body of a function. They enrich simple expressions with assignments, messages and instantaneous loops.

We stress again that these three categories exist only to make the parsing of an augmented score non-ambiguous. Expressions have the same status and are managed in the same way, irrespectively of their category. So it is possible to turn an expression of a more general category into an equivalent expression of a less general one:

- an extended expression can be used in place of a simple expression, simply by calling a function whose body is specified by the extended expression (a function call is a simple expression).
- a simple expression can be used where a closed expression is expected, simply by putting it between parentheses.

Auto-Delimited Expressions

Closed expressions, also called auto-delimited expressions, are expressions that are allowed in specific locations:

- the specification of a `delay`,

²The term *closed expression* usually refers to an expression that contains no free variables. This is not the meaning used here. A closed expressions refer here to expressions that can be put in sequence without ambiguity. See paragraph [auto-delimited expressions](#).

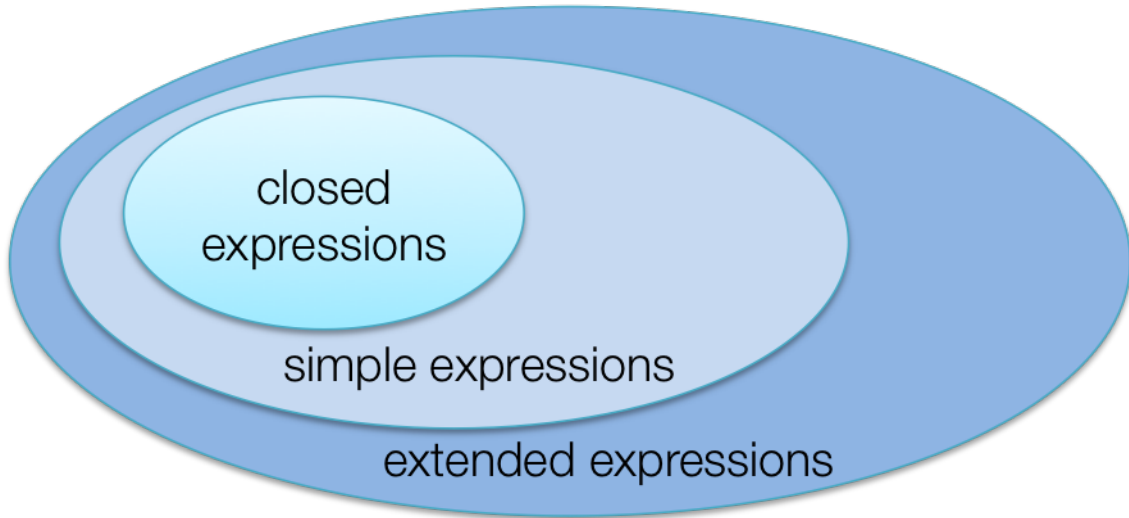


Figure 14.2: three kinds of expressions

- the arguments of a [message](#),
- the arguments of an [internal command](#),
- the list of breakpoints in a [curve](#),
- the specification of an [attribute](#) value,
- the specification of a [while](#) or [until](#) clause.

If a simple or an extended expression is provided where an auto-delimited expression is required, a syntax error is declared.

Numeric constants and strings are closed expressions, as well as map definitions. Tab definitions are closed expressions but the keyword TAB is mandatory. A variable is a closed expression too.

Notice that *every expression between parentheses is an auto-delimited expression*. So, a rule of thumb is to put the expressions in the contexts listed above between braces when the expression is more complex than a scalar constant or a variable.

The section [syntax of auto-delimited expressions](#) gives the full syntax of closed expressions and explains the motivations of these syntactic constraints.

Simple Expressions

Simple expressions include:

- closed and arithmetic expressions
- variables and constant values
- data structure definitions (tab, map, nim)

- function applications and process calls
- variable manipulation

and a combination thereof. A simple expression between parentheses is a closed expression (and a simple expression).

The grammar of simple expressions is defined in [simple expression grammar](#).

Extended Expressions

Writing large expressions can be cumbersome and may involve the repetition of common sub-expressions. Functions can be used to avoid the repeated evaluations of common sub-expressions. In addition, the body of a function is an *extended expression*, which is a sequence of simple expressions enriched with local variables, messages, assignments and loops.

Extended expressions enable a more concise and more readable specification of expressions. See chapter [Functions](#).

Next, we introduce the [general notion of value](#).

You may also go directly to:

- [the notion of variable](#);
- [temporal variables](#);
- [conditional expressions](#);
- [actions as expressions](#).

Values

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

- **scalar** or **atomic** values, described in chapter [Scalar Value](#), include the *undefined value*, booleans, integers, floats (IEEE double), symbols, function definitions, process definitions and running processes (*exec*);
- **data structures** or **compound values** like [strings](#) (sequences of characters), [tabs](#) (tables, vectors), [maps](#) (dictionaries), and interpolated functions [NIMs](#). Such data structures, described in chapter [Data Structures](#), can be arbitrarily nested, to obtain for example a dictionary linking strings with vectors of interpolated functions.

A compound value is referred to using a **handle** (or *pointer*). So, the same compound value can be shared between variables or shared between nested data structures (see [data structure](#)).

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

Functions can be used to combine values and build new ones. The programmer can define his or her own functions (see chapter [Function](#)), also having access to a large predefined [Functions Library](#). The figure below shows a simple score excerpt employing a simple expression and value. The text score on the right declares four expressions to be sent to receivers “hr1-p” to “hr4-p” (harmonisers) whose final value is being converted from semi-tones to pitch-scale factor. This graphical representation shows their evaluation.



Figure 14.3: principle

In this example we are able to use the final values of the expression in the graphical display of the score by [AscoGraph](#) since the arguments of the expression are constant. So these expressions are recognized itself as constant and their value is computed when the score is loaded (a mechanism known as “constant propagation”). If a variable were to be used, the expression would stay intact in the visual representation to be evaluated at run-time. Variables will be discussed in section [Variable](#).

Dynamic Typing

From a programming language perspective, *Antescofo* is a dynamically typed programming language: value types in *Antescofo* do not need to be explicitly specified; the type of values are checked during the performance and this can lead to an error at run-time.

When a wrong argument is provided to an operator or a predefined function, an error message is issued on the console and the returned value depends of the operators involved (often, the *undef* value). See section [Dealing with Errors](#) for useful hints on how to debug an augmented 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 to an *Antescofo* program fragment that

can be used to denote this value.

Checking the Type of a Value

Several predicates check if a value is of some type:

- [`@is_undef`]
- [`@is_bool`]
- [`@is_string`]
- [`@is_int`]
- [`@is_float`]
- [`@is_numeric`] (which returns true if the argument is either [`@is_int`] or [`@is_float`]),
- [`@is_map`]
- [`@is_nim`]
- [`@is_tab`]
- [`@is_fct`] (which returns true if the argument is an intentional function)
- [`@is_function`] (which returns true if the argument is either an intentional function or a [map](#))
- [`@is_proc`]
- [`@is_exec`]
- [`@is_obj`]
- [`@is_obj_XXX`] (where `XXX` is the name of an object definition).

Value Comparison

Two values can always be compared using the *relational operators*

< <= = != => >

and the `@min` and `@max` operators.

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

The various kind of values are reviewed in chapters [Scalar Values](#) and [Data Structures](#). *Antescofo* is a high-order language, so [Functions](#) and [Processes](#) are also values, as well as [Actors](#).

In the rest of this chapter, we review:

- [the notion of variable](#);
- [temporal variables](#);
- [conditional expressions](#);
- [actions as expressions](#).

Variables

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

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

The two forms are equivalent, but the `let` keyword is sometimes mandatory, see below.

An [assignment](#) is an action and like other actions, it can be done after a delay. We stress that variable assignments are actions and *not expressions*. However, they are instantaneous and they can appear in extended expressions in the body of a function.

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

Note that variables are not typed: the same variable may hold an integer and later a string, for example.

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

- the reception of an [OSC message](#),
- the message `setvar` or the internal command `antescofo::setvar`,
- the function `[@loadvar]`.

Also see the section [accessing scoped variable](#) below.

Histories: Accessing the Past Values of a Variable

Variables are managed in an 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 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 [variable declaration](#). So, only the n “last values” of the variable are recorded. Accessing the value of a variable beyond the recorded values returns an undefined value.

Dates functions

Two functions let the composer to know the date of a logical instant associated to the assignment of a variable:

```
@date([n#]:$v)
```

returns the date in absolute time of the n th to the last assignement of and

```
@rdate([n#]:$v)
```

returns the date in relative time (relative to the musician).

These forms mimic the form of functions but they are not; they are **special forms** and only accept a variable or the dated access to a variable.

Variable Declaration

Variables are **global** by default, that is, visible everywhere in the score, or they are declared **local** to a sequence of actions which limits its scope and puts a constraint on its lifetime.

For instance, the scope of a variable declared local in a `loop` body is restricted to one instance of the loop body, so two loop bodies refer 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 for the body of each [compound action](#). The declaration, may appear everywhere in the scope and takes a comma separated list of variables:

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

There can be several declarations in the same scope and all local variables can be accessed from the beginning of the scope, regardless of the location of their declaration.

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

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

will print

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

Lifetime of a Variable

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

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

The loop `::antescofo L` nested in the group runs forever and accesses the local variable after “the end” of the group `G` (the group ends with the launch of its last action, see [Action Sequence](#)). This use of `$x` is perfectly legal. *Antescofo* manages variables efficiently and the memory allocated for `$x` persists as long as needed by the children of `G` but no more.

History Length of a Variable

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

In a declaration, the specification of a history size for the variable takes the form:

```
n:$v
```

where n is an integer. This syntax specifies that variable has an history of length *at least* n .

To make it possible to specify the size of global variable's history, there is a declaration `[@global]`

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

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

The occurrence of a variable in an expression is also used to determine the length of its history. In an expression, the n th past value of a variable $\$v$ is accessed using the *dated access* construction (see above):

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

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

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

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

History reflected in a Map or in a Tab

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

- `@map_history($x)` returns a map where key n refers to the $n - 1$ 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 updates to the variable is less than the history length, the corresponding undefined values are not recorded in the map.
- `@tab_history($x)` is similar to the previous function but returns a tab where i th element refers to the the $n - 1$ to the last value of $\$x$.
- `@map_history_date($x)` returns a map where the value of key n is the date (physical time) of the $n - 1$ to the last update of $\$x$. The previous remark on the map size applies here too.
- `@tab_history_date($x)` builds a tab (instead of a map) of the dates in physical time of the of updates of the var $\$x$.
- `@map_history_rdate($x)` returns a map where the value associated to key n is the relative date of $n - 1$ to the last update of $\$x$. The previous remark on the map size applies here too.
- `@tab_history_rdate($x)` builds a tab (instead of a map) of the dates in relative time of the updates of the var $\$x$.

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

Plotting the history of a variable

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

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

Note that only the values stored in the history are plotted : so usually one has to specify the length of the history to record, using a `[@global]` or `[@local]` declaration.

The `@plot` and `@rplot` special forms expand to a call to the function `gnuplot`³. For example, the expression expands into

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

See description of `[@gnuplot]` in [Library Functions](#).

Accessing a Local Variable “From Outside its Scope of Definition”

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

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

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

³The `gnuplot` program is a portable command-line driven graphing utility for Linux, MS Windows, Mac OSX, and many other platforms. Cf. www.gnuplot.info. It must be installed on the system to have a working `[@gnuplot]` function.

The Dot Notation

To access the variable defined in one specific instance of a group, or more generally of a compound action introducing a scope ([@whenever], [loop](#), process call, etc.), one must use the dot notation through the *exec* referring to this instance. *Exec* are introduced in section [Exec](#).

It is possible to read the value of a local variable through the dot notation:

```
$p := ::P()
$x_of_p := $p.$x
```

Expression `$p.$x` get the value of the local variable `$x` in the process `::P` launched at the previous line. The instance of the process is accessed through its *exec*, see section [Exe](#).

The expression at the left of the dot operator may be more complex than just a variable:

```
$p := [ ::P() | (10) ]
$x_of_p2 := $p[2].$x
```

The first line launch 10 instances of process `::P` using a [tab comprehension](#). The second line get the local variable of the third instance of `::P`.

Assigning a Variable From Outside its Scope

As previously mentioned, a variable can be assigned from “outside”, see:

- the reception of an OSC message [OSCreceive](#),
- the message [setvar](#),
- the function [\[@loadvar\]](#),
- the assignment using the dot notation.

The [OSCreceive](#) and the [setvar](#) command can be used only for global variable. But local variable can be the target of the two other mechanisms.

The assignment of a local variable through the dot notation is similar to an usual assignment:

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

The expression at the left of the dot operator may be more complex than just a variable:

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

The first line launches 10 instances of process `:::atescofo ::P`. The second line sets the local variable of the third instance of `:::atescofo ::P`.

Notice the `let` keyword: it is needed in an assignment when the expression in the left hand side of the assignment is more complex than a variable.

These assignments are monitored by the [whenever](#) where the local variable `$x` appears. But an expression `$p.$x` does not monitor the local variable of the process. See section [Reference to a scoped variable](#)

Antescofo System Variables

System variables are internal variables managed directly by *Antescofo* and are updated automatically by the system. They are useful for interacting with, for example, the machine listener during performances and creating interactive setups.

The following variables are managed as ordinary variables:

- `$BEAT_POS` is the position of the last detected event in the score.
- `$DURATION` is the duration of the last detected event, as specified in the score.
- `$ENERGY` is the current *normalized energy* of the audio signal from the listening machine. The returned value is always between 0.0 and 1.0 and is equivalent to the *Calibration Output* of the Antescofo object in Max and Pd. **NOTE:** The variable is updated with high frequency (equal to the analysis hop size). Use it with care inside processes and *Whenever* constructs.
- `$LAST_EVENT_LABEL` is the label of the last event seen. This variable is updated only if the event has a label.
- `$PITCH` is the pitch (in MIDI Cents) of the current event. This value is well defined in the case of a NOTE and is not meaningful for the other kinds of event.
- `$RT_TEMPO` represents the tempo currently inferred by the listening machine from the input audio stream.
- `$SCORE_TEMPO` returns the tempo constant in the score at the exact score position where it is called.
- `$RCNOW` is the date in relative time (in beats) of the “current instant”. It can be interpreted as the current position in the score. This position is continuously updated between the occurrence of two events as specified by the current tempo. Thus, if the next event occurs later than anticipated, the value of `$RCNOW` will jump backward.
- `$RNOW` is the date in relative time (in beats) of the “current instant”. It can be interpreted as the current position in the score. This position is continuously updated between two events as specified by the current tempo. But, contrary to `$RCNOW`, the increase stops when the position in the score of the next waited event is reached and `RNOW` is stuck until the occurrence of this event or the detection of a subsequent event (making this one missed). Thus, cannot decrease.

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

Special Variables

These variables are similar to system variables, but they cannot be watched by a [whenever](#):

- `$NOW` corresponds to the absolute date of the “current instant” in seconds. The “current instant” is the instant at which the value of is required.

- `$MYSELF` denotes the *exec* of the enclosing compound action.
- `$THISOBJ` may appears in method definitions where it refers to the object on which the method is applied, or in the clauses of an object definition where it refers to the current instance.

Variables and Notifications

In *Antescofo*, a set of entities to be notified is associated to each variable. The notification mechanism is the core device used by the reactive engine to implement the computations.

Notification of events from the machine listening module drops down to the more general case of variable-change notification from an external environment. Actions associated to a musical event are notified through the variable `$BEAT_POS`. 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 is notified by all the variables that appear in its condition. The 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 reactive engine takes care of stopping the propagation when a cycle is detected. See section [Causal Score and Temporal Shortcuts](#).

The next section [temporal variables](#) investigates the use of a variable to track a process and to infer a tempo. Then we take a look at

- [conditional expressions](#)
- and [actions as expressions](#).

Temporal Variables

Starting version 0.8, *Antescofo* provides a feature to track the progression of any kind of performance P through the updates to a variable. The variable is used to infer the tempo of P and any sequence of actions can be synchronized with it, the same way a sequence of actions can be synchronized with the musician.

Such variables are called **temporal variables**. They abstract the “position” and the “speed” of P : each time P progress, it updates the associated variable, which corresponds to a predefined progression in the position of P . The “tempo of the variable” is computed using the same algorithm used by the listening machine to track the tempo of the musician.

Temporal variables are declared using the `@tempovar` declaration:

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

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

Temporal variables are regular variables that can be used in expressions. The value of a temporal variable is the last assigned value or undefined (as for an ordinary variable). In addition, a temporal variable stores the following internal information that can be accessed at any time:

- `$v.tempo` represents the internal tempo of `$v` as tracked by Large's algorithm. This attribute is initialized by the first parameter of the `@tempovar` declaration.
- `$v.position` represents current beat position of `$v`. This attribute is initialized to 0.
- `$v.frequency` represents frequency (period) of `$v`. This attribute is initialized by the inverse of the second parameter of the `@tempovar` declaration.
- `$v.rnow` represents relative time of `$v`

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

```
let $v.tempo := 55 // change the current tempo of $v
```

The `@sync` synchronization attribute

Using temporal variables, it is possible to define *synchronization strategies* of groups, loops, proc, etc., based on the progression recorded by a temporal variable, using the attribute `@sync` as below:

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

Temporal variables can be set by the environment, cf. `setvar`, allowing for easy tracking any kind of external processes and the synchronization on it. Temporal variables can also be used to set synchronization coordination schemes different than that of the human musician it follows.

Comparing score following and temporal variables

The table below compares the features offered by the score following (listening machine) and temporal variables:

features	score following	temporal variable \$v
event	musical event detected by the listening machine	update of \$v
elementary progression	an amount corresponding to the duration of the detected event	a fixed amount specified in the @tempovar declaration
position of last event	\$BEAT_POS	\$v.position
current position	\$RNOW	\$v.RNOW
progression speed	\$RT_TEMPO	\$v.tempo
progression speed estimation by	Large's algorithm	Large's algorithm
position is updatable	NO	YES
progression speed is updatable	NO	YES

Nota Bene:

- The value assigned to a temporal variable does not matter in the synchronization mechanism, nor in the position and progression speed tracking.
- Unlike a score, each event (update) of a temporal variable corresponds to a progression of the same quantity.
- Contrary to score following, the tracking parameter can be updated directly, directly impacting the progression of the sequence of actions synchronized with it.

Operators and Predefined Functions

The main operators and predefined functions are outlined in the following chapters together with the main data types involved. Here we sketch some operators and functions that are not linked to a specific type. The predefined functions are listed in annex [Library](#).

Conditional Expression

An important operator is the conditional *à la C*:

$$(\text{cond} \ ? \ \text{exp}\ensur{_1} \ : \ \text{exp}\ensur{_2})$$

returns the value of the expression `exp1` if the expression `cond` evaluates to `true` and else `exp2`. The parentheses are mandatory.

As usual, the conditional operator is a special function: it does not evaluate all of its arguments. If `cond` is true, only `exp1` is evaluated, and similarly for false and `exp2`.

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

```
if (cond) { exp\ensuremath{_{1}} } else { exp\ensuremath{_{2}} }
```

see chapter [Functions](#).

Beware not to confuse the [conditional action](#) and the conditionnal expression presented here. The body of the former is a sequence of actions; the latter, an expression. The former does not return a value, contrary to the latter.

@empty and @size

The predicate `[@empty]` returns true if its argument is an empty [map](#), [tab](#) or [string](#), and false otherwise.

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

- for aggregate values, the “size” of the arguments; that is, for a [map](#), the number of entries in the dictionary, for a [tab](#) the number of elements, for a [nim](#), the dimension of the nim and for a [string](#) the number of characters in the string.
- for scalar values, `[@size]` returns a strictly negative number. This negative number depends only on the type of the argument, not on the value of the argument.

Alphabetical Listing of *Antescofo* Predefined Functions

```
{!Library/Functions/functions.list!}
```

Actions as Expressions

An action can be considered as an expression: when evaluated, its value is an [exec](#). This can be a useful way to bypass *Antescofo*'s syntax constraints. To consider an action as an expression, the action must usually be enclosed in an `EXPR { ... }` construct.

However, the main use of this construct is to get the `exec` of an action. The action is fired when the expression is evaluated. The returned `exec` refers to the running instance of the action and can be used to kill this running instance or to access the local variables of the action. An atomic action (with a 0-duration run) returns the special `exec '0`.

Simplified Syntax

A number of shortcuts can be used to simplify the writing:

The surrounding `EXPR { ... }` is optional in the case of a process call.

The surrounding `EXPR { ... }` is optional in the body of a function, but only for messages and variable assignment, see [AtomicActionInExpression](#).

The keyword `EXPR` is optional in the right hand side of an assignment. For example:

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

is equivalent to

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

Example

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

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

Thus, one can kill the second instance with

```
abort $stab[1]
```

and one can access the local variable of the third instance through the dot notation:

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

In this case, the use of the `EXPR { ... }` avoids the definition of a process to encapsulate the loop.

Scalar Values

Antescofo offers a rich set of value types described in this chapter and those that follow. The value types examined in this chapter - `undef` (the undefined value), `bool` (booleans), `int` (integers), `float` (double floating point values), `fcn` (intensional functions), `proc` (processes) and `exec` (threads of execution) - are *indecomposable values*. Functions and processes are also covered in more depth respectively in chapters [Function](#) and [Process](#). The value types examined in the next chapter are data structures that act as containers for other values.

The Undefined Value

There is only one value of type `undef`. This value is written

```
<undef>
```

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.

Boolean Values

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

- the *logical negation* `!` (“not”) written prefix form⁴: for instance, `! false` returns `true`;
- the *logical disjunction* `||` (“or”) written in infix form: *e.g.*, `$a || $b`;
- the *logical conjunction* `&&` (“and”) written in infix form: for example, `$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`.

Integer Values

Integer values are written as expected. 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 `<`, `<=`, `==` (equal), `!=` (not equal), `>=`, and `>`: when an integer is compared against a float, it is first converted into the equivalent float.

In the context of a boolean expression, a zero is the false value and all other integers are considered to be `true`.

Float Values

As in the C language, Float (“floating point”) values, a data type that can store a long decimal number, are handled as IEEE doubles. The arithmetic operators, their priority and the usual conversions apply. Beware that two floats may be printed in the same way but may differ from a very small amount.

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

For the moment, there is only a limited set of predefined `{!Library/Functions/math_functions.list!}`

These functions correspond to the usual IEEE mathematical functions. They also accept integers.

User-defined Functions

An *intentional function* is a value that can be applied to arguments to achieve a function call. Like others kinds of values, it can be assigned to a variable or passed as an argument in a function or a procedure call.

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

⁴The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

to an @-identifier using the construct introduced in chapter [Functions](#). In an expression, the @-identifier of a function denotes the corresponding functional value. This value can be used for instance as an argument of higher-order functions (see examples of higher-order predefined function in section [map](#) for map building and map transformations).

Looking at functions as values is customary in functional languages like Lisp or ML. However, contrary to these functional languages, functions in *Antescofo* cannot be defined in a nested way, only from the top level like in C.

Some intentional functions are predefined and available in the initial environment like the IEEE mathematical functions. See annex [Library](#) for a description of predefined functions.

There is no difference between predefined intentional functions and user's defined intentional functions except for those in a Boolean expression, where a user's defined intentional function is evaluated to `true` and a predefined intentional function is evaluated to `false`. See also the predicates `[@is_fct]` and `[@is_function]`.

Intentional functions are described more in detail in chapter [Functions](#).

Intentional functions differ from *extensional functions*. Extensional functions h are defined by explicitly enumerating the pairs $(x, h(x))$ defining the function. [Maps](#) and [NIMs](#) are example of extensional functions and they are described in the next chapters.

Proc Values

Processes are for actions what functions are for expressions. And in the same way that functions are values, processes are values too.

The `::`-name of a process can be used in an expression to denote the corresponding process definition, in a manner similar to the @-identifier used for intensional functions. Such values are qualified as **proc values** and the corresponding type is named `proc`. Like intentional functions, proc values are first class values. They can be passed as arguments to a function or a procedure call.

They are two main operations on proc values :

- “calling” the corresponding process;
- “killing” all instances of this process.

Processes are described more in detail in chapter [process](#).

Exec Value

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

They are several ways to get an *exec*:

- The *special variable* `$MYSELF` always refers to the *exec* of the enclosing compound action.

- The *special variable* `$THISOBJ` always refers to the object referenced by a method (in a method definition) or in the clauses of an object definition.
- A process call returns the *exec* of the instance launched, see section [process call](#).
- Through the “action as expression” construct — see section [Action As Expression](#).

Exec as Coroutines or Lightweight Processes

An *exec* refers to a durative action (an action that lasts through time). They corresponds to the notion of *shred* in ChucK or more fundamentally, to the notion of **coroutine** used to structure and implement the reactive and timed part of the runtime.

The concept of coroutines was introduced in the early 1960s and constitutes one of the oldest proposals of a general control abstraction. The notion of coroutine was never precisely defined, but three fundamental characteristics of a coroutine are widely acknowledged:

- the values of data local to a coroutine persist between successive calls;
- the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage;
- they are non-nonpreemptive: coroutines transfer control among themselves in an explicit way with some control-transfer operations (there is no preemption, nor interruption).

As coroutines, *Antescofo*'s execs have several characteristic features: they are first-class objects, which can be freely manipulated by the programmer. There is only *one control transfer operation*: waiting a delay, see chapter [The Manufacturing of Time](#). This operation corresponds to the *yield* operation used to *suspend* a coroutine execution: the coroutine's continuation point is saved so that the next time the coroutine is resumed, its execution will continue from the exact point where it suspended. But here there is no explicit resume operation for execs. And contrary to usual coroutines, the creation of new coroutines corresponds to the principal control structures (whenever, loop, curve... implicitly each compound action specifies a coroutine).

An exec, much like a thread, represents an independent unit of execution which operates concurrently and can share data with other execs. But unlike conventional threads, whose execution is interleaved in a non-deterministic manner by a preemptive scheduler, an exec is a deterministic piece of computation and is naturally synchronized with all other execs via the shared timing mechanism, the synchronization constructs and the priority of actions.

Execs have a priority, so the sequence of execution of execs that run at the same date (in the same instant), is unambiguously determined. See chapters [The Manufacturing of Time](#) and [Action Priority](#).

Alive and Dead Exec

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

A conditional construct can be used to check the status of the *exec*:


```

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

```

Abort with Exec

Exec values can be used as an argument of an `abort` command. Notice that an *exec* refers to a specific instance of an action. So used in an `abort` command, it aborts solely the referred instance while using the label of an action will abort *all* the running instances of this action.

An `abort` command on a dead *exec* does nothing (and does not signal an error).

Accessing a Local Variable Through an *exec*.

Exec can also be used to access the local variables of the referred compound action. This is mostly useful for processes (cf. sect. [Assignment using the dot notation](#)).

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

Accessing a variable through the dot notation is a dynamic mechanism and the variable is looked first in the instance referred by the *exec*, but if not found in this context, the variable is looked up in the context of the *exec* itself, *i.e.* in the enclosing compound action, and so on, until it is found. If the top-level context is reached without finding the variable, an `undef` value is returned and an error message is issued. See sect. [procVariable](#) for an example.

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

Data Structures

Antescofo currently provides four kinds of data structures:

- a [string](#) is a *sequence of characters*,
- a [map](#) is a *dictionary* with arbitrary keys and values,
- a [tab](#) is a *vector* holding possibly heterogeneous values,
- a [nim](#) is an *interpolated function* defined by a sequence of breakpoints.

These data structures are described in the following sections.

String Value

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

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

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

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

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

Note that the result is a string with only one character. There is no specific type dedicated to the representation of just one character. The function `[@size]` can be used to get the number of characters in a string.

Notice also that strings are *immutable values*: contrary to tabs, it is not possible to change a character within a string. A new string must be constructed instead.

The operator `+` corresponds to string concatenation:

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

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

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

will output `"abc33"`. The value of the variable `$a` was automatically converted to type `String` before being concatenated.

There are several `{!Library/Functions/string_functions.list!}`

The function `[@explode]` can be used to convert a string into a [tab](#) (vector) of characters (represented as string of size one). This makes it possible to use the functions defined for tabs. If a tab contains multiple strings, they can be concatenated using the `[@reduce]` operator to build a new string:

```
$t := @explode("123abc")
@assert $t == ["1", "2", "3", "a", "b", "c"]
$t := @reduce (@+, $t)
@assert $t == "123abc"
```

Map Value

A map is a “dictionary” that associates a value to a key. The value can be of any type and so can the key:

```
map{ (k\ensuremath{_{1}}, v\ensuremath{_{1}}), (k\ensuremath{_{2}}, v\ensuremath{_{2}}), ... }
```

The `map` keyword is case-insensitive and is followed by a comma separated list of (key, value) pairs enclosed in braces. The previous construction is an expression and keys and values in the definition list are ordinary expressions. An empty map is specified by an empty (key, value) list:

```
MAP{ }
```

The types of the keys and values are not necessarily homogeneous. So a map may include an entry which associates a string to a number and later a map to a string, *etc.*:

```
map{ (1, "one"),
      ("dico", map{ ("pi", 3.14), ("e", 2.714), ("sqr2", 1.414) }),
      (true, [0, 1, 2, 3]),
      (1.234, 12 + 34)
    }
```

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

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

will print

```
first
<undef>
```

The `undef` value is returned for the second call because there is no corresponding key.

Extensional Functions

A MAP can be seen as a function defined by extension: an image (the value) is explicitly defined for each element in the *domain* (*i.e.*, the set of keys). [NIMs](#) are also *extensional functions*.

Extensional functions are handled as values in *Antescofo*. This is also the case for *intentional functions*, see chapter [Functions](#).

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

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

The computation of `$w` shows that a MAP is passed as an argument of the higher-order `@map` functions. Do not confuse the case-insensitive MAP keyword with the name of the function `@map`. This function applies its first argument to all elements of the `tab` passed as the second argument.

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)`. This is not the same as testing the value returned by `m(k)` is `undef` because the key can be present in the dictionary with the value `undef`.

The predefined `@is_integer_indexed` applied on a map returns true if all of its keys 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 satisfied and if every element in the range satisfies `@is_numeric`.

The functions `@min_key` and `@max_key` compute the smallest and largest value keys respectively amongst the keys of its map argument.

The functions `@min_val` and `@max_val` do the same for the values of its map argument.

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

The function `@domain` applied on a map returns the `tab` of its keys. In the returned `tab`, the keys are in increasing order. The function `@range` applied on a map returns the `tab` of its values. The order of the values reflects the order of their associated keys. For example

```
@domain({MAP{"zero", 0.0}, {"0", 0}, {"one", 1}}) --> ["0", "one", "zero"]
@range({MAP{"zero", 0}, {"0", 0}, {"one", 1}}) --> [0, 1, 0.0]
```

The functions `@count`, `@find`, `@member` and `@occurs` work on maps as well as on `tab` and `string`.

`@member(m, v)` returns `true` if there is a key `k` such that `m(k) == v` and returns `false` otherwise.

`@count(m, v)` returns the number of keys `k` such that `m(k) == v`.

`@occurs(m, v)` returns the first key `k` (for the `<` ordering) such that `m(k) == v` if such a key exists, else the `undef` value.

Finally, `@find(m, f)` returns the first key `k` (for the `<` ordering) such that `f(k, m(k))` returns true and the `undef` value if such an entry does not exist.

Constructing Maps

The operations described below act on a whole map to build new maps.

`@select_map` restricts the domain of a map: `@selec_map(m, P)` returns a new map `n` such that `n(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 `dico` with a new entry (no new map is created). Alternatively, the overloaded function `[@insert]` can be used: `[@insert]` can be used on tabs and maps; `[@add_pair]` is just the version specialized for maps.

`[@shift_map](m, p)` returns a new map n such that $n(x+p) = m(x)$

`[@gshift_map](m, f)` generalizes the previous operator using an arbitrary function f instead of an addition and returns a map n such that $n(f(x)) = m(x)$

`[@mapval](m, f)` composes function f with the map m : the results n is a new map such that $n(x) = f(m(x))$.

`[@map_compose](m, n)` builds a new map with keys taken in the images of m and values in n for all keys in the intersection of the keys of m and n . In other words, if

$$\begin{aligned} m &= \text{MAP}\{ (k\ensuremath{_1}, m\ensuremath{_1}), (k\ensuremath{_2}, m\ensuremath{_2}), \dots \} \\ n &= \text{MAP}\{ (l\ensuremath{_1}, n\ensuremath{_1}), (l\ensuremath{_2}, n\ensuremath{_2}), \dots \} \end{aligned}$$

constructs the map:

$$\text{MAP}\{ \dots, (m\ensuremath{_i}, n\ensuremath{_i}), \dots \}$$

if there exists an i such that $m(i) = m_i$ and $n(i) = n_i$.

`[@merge]` combines two maps into a new one. The operator is asymmetric, that is, if $m = \text{@merge}(a, b)$, then:

$$m(x) = \text{if } (\text{@is_defined}(a, x)) \text{ then } a(x) \text{ else } b(x)$$

`[@remove](m, k)` removes the entry of key k in map m (no new map is created). If k is not present in m , the command has no effect. This function is overloaded and also applies to tabs.

Extension of Arithmetic Operators

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

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

will print

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

If an arithmetic operator 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) }`.

Map Transformations

`[@clear]` erases all entries in the map.

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

```
m = MAP{ (3, 3), ("abc", "abc"), (4, 4) }
```

```
:::antecofo @listify(m) returns
```

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

because we have $3 < 4 < \text{"abc"}$.

Score reflected in a Map

Several functions can be used to reflect the events of a score into a map⁵:

- `[@make_score_map]` returns a map where the key is the event number (its rank in the score) and the associated value, its position in the score in beats (that is, its date in relative time).
- `[@make_duration_map]` returns a map where the key is the event number (its rank in the score) and the associated value, its duration in beats (relative time).
- `[@make_label_pos]` returns, like the following, returns a map whose keys are the labels of the events and whose values are the position (in beats) of the events.
- `[@make_label_bpm]` returns a map associating the event labels to the BPM at this point in the score.
- `[@make_label_duration]` returns a map associating to the event of a label, the duration of this event.
- `[@make_label_pitches]` returns a map associating a vector of pitches to the label of an event. A corresponds to a tab of size 1, a with n pitches to a tab of size n , *etc.*

These functions take two optional arguments, *start* and *stop*, to restrict where in the score the map is built. The map contains the key corresponding to events that are in the interval $[start, stop]$ (interval in relative time). Called with no arguments, the map is built for the entire score. With only one argument *start*, the map is built for the labels or the positions strictly greater than *start*.

Variable's History Reflected in a Map

The sequence of the values of a variable is kept in a history. This history can be converted into a map: see section [history reflected in a map](#).

⁵The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

List of `{!Library/Functions/map_functions.list!}`**Tables**

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

```
{!BNF_DIAGRAMS/tabdef_expr.html!}
```

The `tab` keyword is case-insensitive and optional. For example:

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

these statements assign a *tab* with 4 elements to the variable `$t`.

Tables are compared in [lexicographical order](#). In a boolean expression, an empty *tab* is false. Otherwise, it's true.

Elements of a *tab* can be accessed through the usual square bracket `... [...]` notation. Element indexing starts at 0, so `$t[n]` refers to the $(n + 1)$ th element of the *tab* referred by `$t`. Notice that the arguments of the square bracket are expressions⁶. Elements of a *tab* can be any kind of value, even a *tab*, which would create a multidimensional array. Additionally, one *tab* can contain multiple different types of objects.

The [ForAll](#) action can be used to iterate through all of the elements in a *tab*. A *tab comprehension* can be used to build new *tab* by filtering and mapping *tab* elements. There are also several predefined functions to transform a *tab*.

Multidimensional tab

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

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

is a 3×2 matrix that can be interpreted as 3 lines and 2 columns. For example, if `$t` is a 3×2 matrix, then the first element of the second line is accessed by the expression

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

The function `[@dim]` can be used to query the dimension of a *tab*, that is, the maximal number of *tab* nesting found in the *tab*.

A multidimensional array is a *homogeneous* *tab*: a *tab* of *tab* elements is an multidimensional array (of dimension 1) and a *tab* whose elements are multidimensional arrays of the same dimension and size are also multidimensional arrays.

If a *tab*'s argument is a multidimensional array, the function `[@shape]` returns a *tab* of integers wherein the element i represents the number of elements in the i th dimension. For example

⁶The arguments of the square brackets are expressions so one can write, *e.g.* `(@f($x))[$n]` to access the value of the n th element of the *tab* returned by a function `@f`. Parentheses are used to apply the brackets to the value returned by `@f` instead on `$x`.

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

The function returns 0 if the argument is not a well-formed (dimensionally consistent) array. For example

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

Note that for this argument, `@shape` returns `::antescofo 0` because the argument is a tab nested into a tab, but it is not an array because the element of the top-level tab are not homogeneous. The tab

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

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

Tab Comprehension

If a tab is specified by giving the list of its elements, the definition is said *in extension*. The tab `[...]` construction is an expression that defines a tab in extension.

A tab can also be defined *in comprehension*. A tab comprehension is an expression to build a tab from an existing tab or on some iterators. The general form of a tab comprehension is

```
[ output_expression | $var in input_set , predicate ]
```

where `output_expression` is an expression, `$var` is a variable identifier, `input_set` is an expression evaluating to a tab or an integer or the construction `e1 .. e2 : e3`, and `predicate` is a boolean expression. More precisely:

```
{!BNF_DIAGRAMS/tabcomprehension_expr.html!}
```

This construction follows the form of the mathematical [set-builder notation](#) (set comprehension). For instance

```
[ e | $x in t ]
```

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

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

The input set `t` may also evaluate to a numeric value `n`: in this case, take all the numeric values between 0 and `n` excluded by unitary steps:

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


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

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

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

The specification `start .. stop` specifies a range where `start` is included and `stop` is excluded. If the specification of the step is not given, its value is `+1` or `-1` following the sign of `(stop - start)`. The specification of `start` is also optional: in this case, the variable will start from 0. For example:

```
[ $s[$i] + $t[$i] | $i in @size($s) ]
```

creates a tab whose elements are the pointwise sums of `$s` and `$t` (assuming that they have the same size). Notice that expression `$i in (@size($s))` enumerates the indices of `$s`. Expression

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

generates a tab of 62 elements: `sin(-3.14), sin(-3.04), ..., sin(3.04)`.

Tab comprehension may specify a predicate to filter the members of the input set:

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

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

Tab comprehensions are ordinary expressions, so they can be nested. This can be used to create a tab of tabs. Such a data structure can be used to make matrices:

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

More Examples of Tab Comprehension

Here are some additional examples of tab comprehensions to illustrate the syntax:

```
$z := [ 0 | (100) ] ; builds a vector of 100 elements, all null
```

In this example, the iterator variable is absent. In this case, the input set is constrained to be an expression between parentheses and evaluating to either a number or a tab (it cannot be a range).

```
$s := [ $i | $i in 40, $i % 2 == 0 ] ; lists the even numbers from 0 to 40
$t := [ $i | $i in 40 : 2 ] ; same as previous
```

```

$u := [ 2*$i | $i in (20) ]           ; same as previous

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

$m := [ [1, 2, 3], [4, 5, 6] ]       ; builds a matrix of 3x2 dimensions
$m := [ [ @random() | (10) ] | (10) ] ; builds a random 10x10 matrix

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

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

$v := [ @random() | (10) ] ; builds a vector of ten random numbers
; matrix*vector product
[ @reduce(@+, $m[$i] * $v) | $i in @size($m) ]

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

```

Mutating a tab's element

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

```

let $t[0] := 33
$t[0] := 33 ; the 'let' is optional if the tab is denoted by a variable

```

changes the value of the first element of the tab referred by `$t` to the value 33. The general syntax is:

```
{!BNF_DIAGRAMS/tab_assignment.html!}
```

Unless the tab is referred to by a variable, the `let` keyword is mandatory. It is required when the expression in the left hand side of the assignment is more complex than a variable, a simple reference to an array element or a simple access to a local variable of an *exec*. See sect. [Assignment](#).

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

```

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

```

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

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

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

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

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

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

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

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

Sharing and copying a tab

In the introduction of section [Values](#), we mention that a compound value is referred through a handle (or pointer). So, the same compound value can be shared between variables or shared between nested data structures.

This can be illustrated by the following example:

```
let $t := [0, 0, 0]
$u := $t
let $t[0] := 333
print $u ; will print [333, 0, 0]
```

After the assignment to `$t`, the value referenced by `:::atescofo $u` has mutated, because the same tab is referenced by `$t` and `$u`.

Assignment of a tab, and more generally a compound value, does not imply the copy of the referenced value. The function `@copy` can be used to create a fresh value:

```
let $t := [0, 0, 0]
$u := @copy($t)
let $t[0] := 333
print $u ; will print [0, 0, 0]
```

The sharing of data structures is useful: the same tab can be referred to from many different places in the score and one update is visible by all the tab referrers. However, it may be troublesome. The following code is intended to create a 3×3 null matrix `$m0`

```
let $row := [0, 0, 0]
let $m0 := [ $row, $row, $row ]
```

but this is not exactly the case. As a matter of fact, if we want to turn `$m0` in the identity matrix using assignment:

```
let $m0[0, 0] := 1
let $m0[1, 1] := 1
let $m0[2, 2] := 1
```

what we obtain is a 3×3 matrix full of 1: because all the row of `$m0` refers to the same tab.

Assignment *versus* Mutating a tab's element

Changing the value of a tab's element *is not a variable assignment*: the variable has not been “touched”, it is the value referred by the variable that has mutated.

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

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

where it is apparent that no variable at all is involved. The previous expression is perfectly legal: it changes the second element of the tab `[0, 1, 2]`. This change will have no effect on the rest of the program because the mutated tab cannot be referred elsewhere but this does not prevent the action to be performed.

An *important consequence* is that mutating a tab elements does not trigger a [whenever](#) even if a variable is involved in the assignment. It is however very easy to trigger a [whenever](#) watching a variable referring to a tab, after the update of an element: it is enough to assign it to itself:

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

We can mutate the first element of `$t` but this does not trigger the [whenever](#). To do so, we assign the variable to itself. As a matter of fact, a [whenever](#) watches the assignment of a set of variables, NOT the mutation of the values referred by these variables.

Listable Operators

Usual arithmetic and relational operators are **listable** (cf. other listable functions in annex [Library](#)).

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

```

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

```

Tab manipulation

Several functions exist to manipulate tabs *intentionally*, *i.e.*, without referring explicitly to the elements of the tab. We briefly describe some of these functions. The [Library](#) exhaustively describes all tab related functions (click on function name to access the page dedicated to the function).

- `[@car]`(τ) returns the first element of τ if it is not empty, else it returns an empty tab.
- `[@cdr]`(τ) returns a new tab corresponding to τ deprived of its first element. If τ is empty, returns an empty tab.
- `[@clear]`(τ) shrinks the argument to a zero-sized tab (no more elements in τ , which is modified in-place).
- `[@concat]`(τ_1, τ_2) returns the concatenation τ_1 of and τ_2 .
- `[@cons]`(v, τ) returns a new tab made of v in front of τ .
- `[@count]`(τ, v) returns the number of occurrences of v in the elements of τ . Also works on string and maps.
- `[@dim]`(τ) returns the dimension of τ , *i.e.* the maximal number of nesting in the elements of τ . If τ is not a tab, the dimension is 0. A “flat” tab (a vector) has dimension 1.
- `[@drop]`(τ, n) gives a copy of τ with its first n , elements dropped if n is a positive integer, and with its last elements dropped if n is a negative integer. If n is a tab of integers, returns the tab formed by the elements of whose indices are not in n .
- `[@empty]`(τ) returns true if there is no element in τ , and false elsewhere. Also works on maps.
- `[@find]`(τ, f) returns the index of the first element of that satisfies the predicate f . The first argument of f is the index of the element and the second is the element itself. Works also on strings and maps.
- `[@flatten]`(τ) builds a new tab where the nesting structure of τ has been flattened. For example, `@flatten ([[1, 2], [3], [], [4, 5]])` returns `[1, 2, 3, 4, 5, 6]`.

- `[@flatten]` (`t`, `l`) returns a new tab where `l` levels of nesting have been flattened. If `l == 0`, the function is the identity. If `l` is strictly negative, it is equivalent to `[@flatten]` without the level argument.
- `[@gnuplot]` (`t`) plots the elements of the tab as a curve using the external command `gnuplot`. See the description `[@gnuplot]` in the index for further information and variations.
- `[@insert]` (`t`, `i`, `v`) inserts “in place” the value into the tab after the index `i`. If `i` is negative, the insertion takes place in front of the tab. If `i ≤ 0` the insertion takes place at the end of the tab. Notice that the function is overloaded and applies also on maps. The form `[@insert]` is also used to include a file at parsing time.
- `[@is_prefix]`, `[@is_suffix]` and `[@is_subsequence]` operate on tabs as well as on strings. Cf. the description of these functions in [library](#).
- `[@lace]` (`t`, `n`) returns a new tab whose elements are interlaced sequences of the elements of the `t` subcollections, up to size `n`. The argument `n` is unchanged. For example: `@lace([[1, 2, 3], 6, ["foo", "bar"]], 12)` returns `[1, 6, "foo", 2, 6, "bar", 3, 6, "foo", 1, 6, "bar"]`.
- `[@map]` (`t`, `f`) computes a new tab where the `i`th element has the value `f(t[i])`.
- `[@max_val]` (`t`) returns the maximum element among those of `t`.
- `[@member]` (`t`, `v`) returns true if `v` is an element of `t`. Also works on string and map.
- `[@min_val]` (`t`) returns the minimum among the elements of `t`.
- `[@normalize]` (`t`, `min`, `max`) returns a new tab with the elements normalized between `min` and `max`. If `min` and `max` are omitted, they are assumed to be 0 and 1 respectively.
- `[@occurs]` (`t`, `v`) returns the first index whose value equals the second argument. Also on string and map (the returned value is the corresponding key in a map).
- `[@permute]` (`t`, `n`) returns a new tab which contains the `n`th permutation of the elements of `t`. They are `s!` permutations, where `s` is the size of `t` (and `!` is the factorial function). The first permutation is numbered 0 and corresponds to the permutation which rearranges the elements of `t` in an vector `t0` such that elements are sorted increasingly. The tab `t0` is the smallest element⁷ among all tab that can be done by rearranging the element of `t`. The first permutation rearranges the elements of `t` in a tab `t1` such that `t0 < t1` for the lexicographic order and such that any other permutation gives a tab `tk` lexicographically greater than `t0` and `t1`. *Etc.* The last permutation (numbered `s! - 1`) returns a tab where all elements of `t` are in decreasing order.
- `[@push_back]` (`t`, `v`) pushes `v` to the end of `t` and returns the updated tab (`t` is modified in place).

⁷ *smallest* for the `<` ordering. On tabs, this ordering corresponds to the lexicographic ordering.

- `[@push_front](t, v)` pushes `v` to the beginning of `t` and returns the updated tab (`t` is modified in place and the operation requires the reorganization of all elements).
- `[@reduce](t, f)` computes $f(\dots f(f(t[0], t[1]), t[2]), \dots t[s-1])$ where `s` is the size of `t`. If `t` is empty, an undefined value is returned. If `t` has only one element, this element is returned. Otherwise, the binary operation `f` is used to combine all the elements in `t` into a single value. For example, `@reduce(@+, t)` returns the sum of the elements of `t`.
- `[@remove](t, i)` removes the element at index `i` in `t` (`t` is modified in place). This function is overloaded and also applies to maps.
- `[@remove_duplicate](t)` keeps only one occurrence of each element in `t`. Elements not removed are kept in order and `t` is modified in place.
- `[@replace](t, find, rep)` returns a new tab in which a number of elements have been replaced by another. See full description at `[@remove_duplicate]` in [library](#).
- `[@reshape](t, s)` builds an array of shape `s` with the element of tab `t`. These elements are taken circularly one after the other. For instance `@reshape([1, 2, 3, 4, 5, 6], [3, 2])` returns `[[1, 2], [3, 4], [5, 6]]`.
- `[@resize](t, s)` increases or decreases the size of `t` to `s` elements. If `s` is greater than the size of `t`, then additional elements will be `<undef>`. This function returns a new tab.
- `[@reverse](t)` returns a new tab with the elements of `t` in reverse order.
- `[@rotate](t, n)` builds a new array which contains the elements of `t` circularly shifted by `n`. If `n` is positive the elements are right-shifted, otherwise they are left-shifted.
- `[@scan](f, t)` returns the tab `[t[0], f(t[0], t[1]), f(f(t[0], t[1]), t[2]), ...]` of the partial result of the reduction (see `[@reduce]`). For example, the tab of the factorials up to 10 can be computed by: `@scan(@*, [$x : $x in 1 .. 10])`.
- `[@scramble](t)` returns a new tab where the elements of `t` have been scrambled (their order is randomized). The arguments are unchanged.
- `[@size](t)` returns the number of elements of `t`.
- `[@slice](t, n, m)` gives the elements of `t` of indices between `n` included up to `m` excluded. If `n > m` the element are given in reverse order.
- `[@sort](t)` sorts in-place the elements into ascending order using `<`.

- `[@sort]` (`t`, `cmp`) performs an in-place sort on the elements, putting them in ascending order. The elements are compared using the function `cmp`. This function must accept two elements of the tab as arguments and returns a value converted to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second.
- `[@sputter]` (`t`, `p`, `n`) returns a new tab of length `n`. This tab is filled as follows. The process starts with the first element in `t` as the *current* element. Successively for each element `e` in the result, a random number p' between 0 and 1 is compared with `p`: if it is lower, then the current element becomes the value of `e`. If p' is greater, the element after the current element becomes the new current element and this element becomes the value of `e`.
- `[@stutter]` (`t`, `n`) returns a new tab whose elements are each elements of `t` repeated `n` times. The argument is unchanged.
- `[@take]` (`t`, `n`) gives the first elements of `t` if `n` is a positive integer and the last elements of `t` if `n` is a negative integer. If `n` is a tab of indices, it gives the tab of elements whose indices are in `n`.

Lists and Tabs

Antescofo's tabs may be used to emulate lists

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

In particular, the operators `[@cons]`, `[@car]` and `[@cdr]` can be used to destructure and to build a tab. They can be used to define recursive functions on tabs in a manner similar to that of recursive functions on lists. *However*, it builds a new tab unlike the operation `cdr` on list in Lisp. A tab comprehension is often more convenient and usually more efficient than a recursive definition.

The [Library](#) documents all `{!Library/Functions/tab_functions.list!}`

New Interpolated Map

Interpolated maps are values representing a piecewise function defined by *interpolation* between a sequence of *breakpoints*.

They have been initially defined as piecewise linear functions which have been superseded by NIM (an acronym for **N**ew **I**nterpolated **M**ap). NIM extends the idea of interpolation between breakpoints to vectors and to a superset of the interpolation types available in the [curve](#) construct.

A NIM is an aggregate data structure that records the breakpoints of the piecewise interpolation and the interpolation type (as in a curve). A NIM is considered an [extensional function](#) like a [map](#). It can be applied to a numerical value to return the corresponding image.

NIMs can be used as an argument of the [curve](#) construct which allows the breakpoints to be dynamically built, “playing” the function later in time. See section [Curve Playing a NIM](#).

Continuous and Discontinuous NIM

NIMs can represent continuous or discontinuous functions.

Continuous NIM are defined by an expression of the form:

```
NIM { x\ensuremath{0} y\ensuremath{0}, d\ensuremath{1} y\ensuremath{1}
    , d\ensuremath{2} y\ensuremath{2} "linear"
    , d\ensuremath{3} y\ensuremath{3} "bounce"
    , ...
    , d\ensuremath{i} y\ensuremath{i} "t\ensuremath{i}"
    , ...
}
```

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

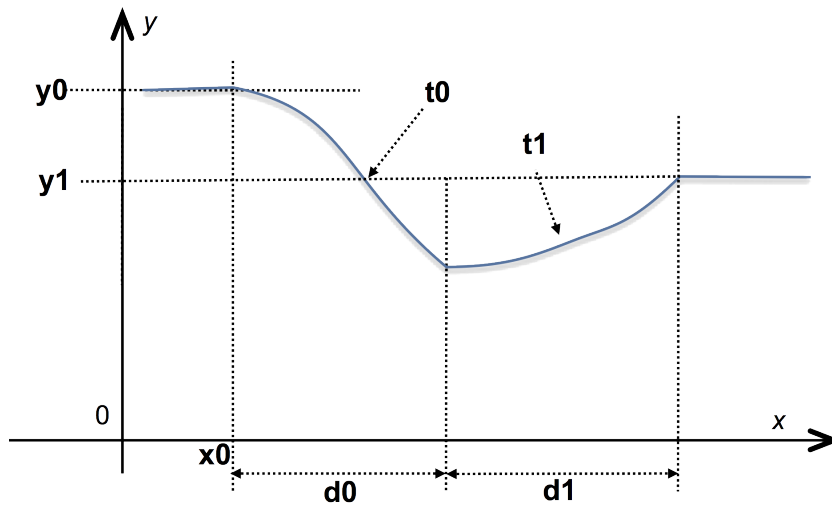


Figure 14.4: continuous ni

The previous definition specifies a continuous function because the value of f at the beginning of $[x_i, x_{i+1}]$ is also the value of f at the end of the previous interval.

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

```
NIM { x\ensuremath{0}, y\ensuremath{0} d\ensuremath{0} Y\ensuremath{0}
```

```

, y\ensuremath{_{1}} d\ensuremath{_{1}} Y\ensuremath{_{1}} "linear"
, y\ensuremath{_{2}} d\ensuremath{_{2}} Y\ensuremath{_{2}} "bounce"
, ...
}

```

The definition is similar to the previous form except that on the interval $[x_i, x_{i+1}]$ the function is an interpolation between y_i and Y_i . See illustration:

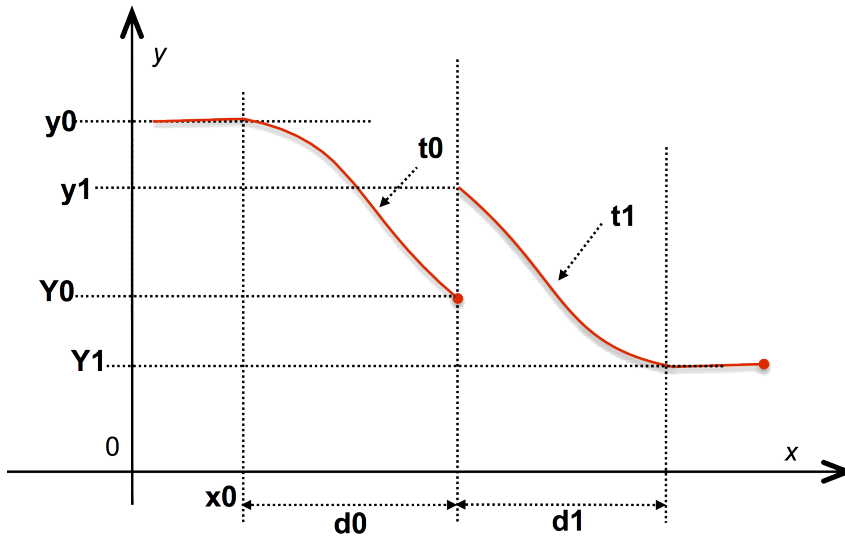


Figure 14.5: discontinuous nim

The NIM keyword is *case-insensitive* and introduces an expression. The parameters x_i, y_i, d_i, Y_i, t_i are expressions.

Definition of the NIM outside the breakpoints

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

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

However, the functions $[@min_key]$ and $[@max_key]$ return x_0 and x_n respectively (these functions also perform similarly on maps: maps are functions on a discrete range while NIMs are functions on a continuous range).

Interpolation type

The type of the interpolation is either a constant string or an expression. In the latter case, however, the expression must be enclosed in parentheses. The names of the allowed

interpolation types are the same as those of a [curve](#) (see [curve interpolation methods](#)), and the interpolation types are illustrated on [these figures](#). Other experimental interpolation methods are currently under development.

The specification of an interpolation type is optional. When not defined, the interpolation type is assumed to be linear.

Vectorial NIM

The NIM construct accepts tabs as arguments: in this case, the result is a vectorial function. For example:

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

defines a vectorial of two variables:

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

where f_1 is a cubic interpolation between 0 and 1 for x_1 going from -1 to $-1 + 2 = 1$ and f_2 is a linear interpolation between 10 and 20 for x_2 going from 0 to $0 + 3 = 3$.

The NIM construction is “smart”: the parameters of a vectorized NIM may mix scalar s and tabs. In this case, the scalar is extended implicitly into a vector of the correct size whose elements are all equal to s . This is the case even for the specification of the interpolation type. For example:

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

defines the function $\vec{f} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$ where

$$\begin{aligned} f_1(x) &= 0 \text{ if } x < 0 \\ &= 10 \text{ if } x > 1 \\ &= 10x \text{ elsewhere} \end{aligned}$$

and

$$\begin{aligned} f_2(x) &= 0 \text{ if } x < 0 \\ &= 10 \text{ if } x > 2 \\ &= 5x \text{ elsewhere} \end{aligned}$$

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

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

The function `[@dim]` returns the dimension of the image of a NIM, that is, 1 for a *scalar* NIM and $n \neq 1$ for a *vectorized* NIM, where n is the number of elements of the tab returned by the application of the NIM.

The function `[@size]` returns the number of breakpoints of the NIM (which is not equal to the dimension of the NIM). Note that a nim with zero breakpoints is the result of a bad definition.

Heterogeneous breakpoints in vectorial NIM

A vectorial NIM can be seen as the aggregation of several scalar NIMs.

The functions `[@projection]` and `[@aggregate]` can be used to respectively extract a scalar nim from a vectorial nim and to aggregate several (scalar or vectorial) nims of dimension d_1, \dots, d_p into a nim of dimension $d_1 + \dots + d_p$.

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

This process can be used to build vectorial NIMs whose breakpoints are different, as in the specification of [vectorial curves](#). There is no syntax to directly specify such NIMs and the function `[@aggregate]` must be used to build such NIM values that are said to have *heterogeneous breakpoints*. When printing such a NIM, the function `[@aggregate]` is used to list its many components:

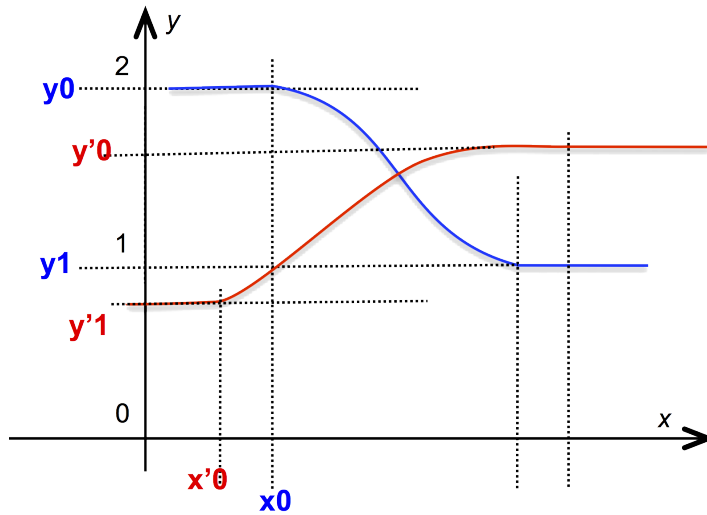


Figure 14.6: aggregate nim

```
@aggregate( NIM{0 0, 1 10 "linear",
              1 0 "sine_in_out"},
            NIM{-1 10, 3 0 "sine_in_out"} )
```

The function [`@projection`] can be used to extract a component from a vectorial NIM:

```
@projection($nim3, 1) == $nim2 --> true
```

Extending a NIM

The function [`@push_back`] can be used to add a new breakpoints to an existing NIM. They are several variations:

```
@push_back(nim, d, y1)
@push_back(nim, d, y1, type)
@push_back(nim, y0, d, y1)
@push_back(nim, y0, d, y1, type)
@push_back(nim\ensuremath{_1}, nim\ensuremath{_2})
```

The four first forms modify the NIM “in-place”. The last form builds a new NIM:

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

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

NIM Transformation and Smoothing

NIMs can be managed through `{!Library/Functions/nim_functions.list!}`

See their descriptions in the [library](#). Here, we describe the notions that appear when a NIM is restructured.

Sampling, homogeneization and linearization

It is possible to convert a NIM with an arbitrary interpolation type to a NIM with a linear interpolation type.

The function [`@sample`] takes a NIM `nim` and a number `n` and returns a NIM with `n` breakpoints and linear interpolation type that approximates `nim`. The process is illustrated in the following diagram: the NIM `$nim4` in input

```
@sample($nim3, 5) -->
@aggregate( nim{0 0, 0.333333 3.33333,
```

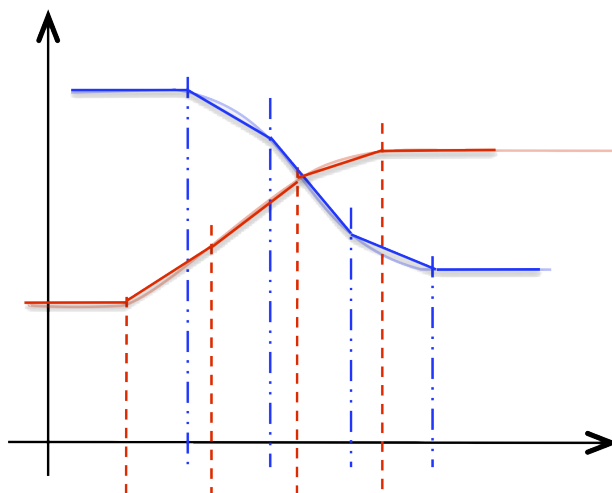


Figure 14.7: sampling a nim

```

0.333333 6.66667,
0.333333 10,
0.333333 7.5,
0.333333 2.5,
0.333333 0},
nim{-1 10, 0.5 9.33013,
    0.5 7.5,
    0.5 5,
    0.5 2.5,
    0.5 0.669873,
    0.5 0} )

```

As can be seen, in the case of a vectorial NIM with heterogeneous components, the approximation is done component by component and the result has heterogeneous breakpoints.

The function `[@align_breakpoints]` can be used on a NIM with linear interpolation type to obtain an equivalent NIM with homogeneous breakpoints:

```
let $nim4 := @sample($nim3, 5)
```

will compute a NIM with a heterogeneous breakpoint printed as

```
@align_breakpoints($nim4) -->
NIM { [-1, -1],
      [0, 10] [0.5, 0.5] [0, 10]
      [0, 9.33013] [0.5, 0.5] [0, 9.33013]
      ; ...
      [0, 0] [0.333333, 0.333333] [0, 0] }
```

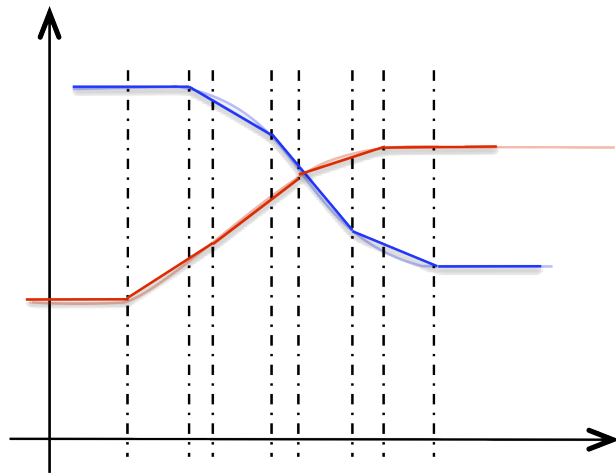


Figure 14.8: align breakpoints

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

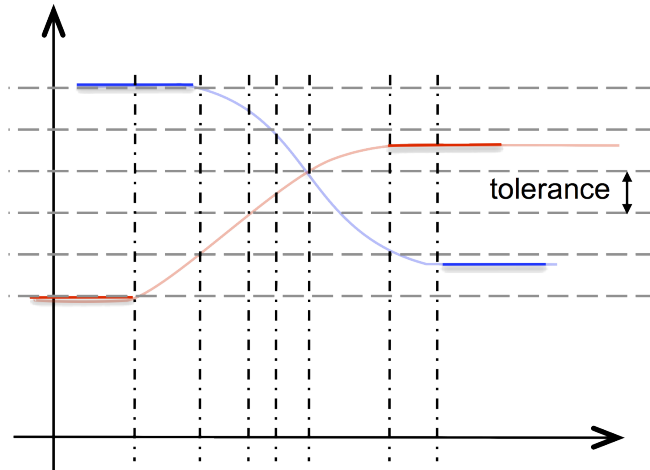


Figure 14.9: linearize nim

```
@linearize($nim3)  -->
NIM { [-1, -1],
      [0, 10] [0.609946, 0.609946] [0, 9.01426],
      [0, 9.01426] [0.270743, 0.270743] [0, 8.02012],
      ; ...
      [0.563462, 0.0636838] [0.152573, 0.152573] [0, 0] }
```

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

NIM simplification

NIMs can be used to record a temporal series of data arriving (through time) from the environment, using `setvar` or `OSC messages`. It is then often useful to simplify the NIM, in order to have a more compact representation. The underlying idea is that the NIM approximates a curve known by a series of points, the breakpoints of the NIM, and that this curve can be approximated by fewer points. The simplified NIM consists of a subset of the breakpoints that defined the original.

Several functions can be used to achieve a reduction of the number of breakpoints of a NIM with linear interpolation.

All the simplification functions apply to scalar as well as vectorial NIMs, but consider only the “ x -coordinate” given by the breakpoints of the first component⁸. In case of a vectorial NIM with heterogeneous breakpoints, this can be a drawback. In this case, an equivalent NIM with homogeneous breakpoints can be first built using the `[@align_breakpoints]` function.

The three following functions are inspired from polyline simplification functions developed in computer graphics⁹:

- `[@simplify_radial_distance_t](nim, d)` simplifies each component of the NIM independently by reducing successive breakpoints that are clustered too closely to a single breakpoint. Because the simplification works on each component independently, a breakpoint in this context is simply a point (x, y) in the plane. The simplification process starts from the first breakpoint and is iterated until it reaches the final breakpoints. The first and the last breakpoints are always part of the simplification. All consecutive breakpoints that fall within a distance from a kept breakpoints are removed. The first encountered breakpoints that lies further away than is kept.
- `[@simplify_radial_distance_v](nim, d)`: this simplification function is similar to the previous one but instead of working on each component independently, the vectorial nature of the nim is taken into account and the “ x -coordinate” is ignored. The NIM is seen as a sequence of points, the image of the NIM at coordinates given by the x part of the breakpoint of the first component. It is this series of points that is simplified to build the new NIM. The distance is thus taken in a n -dimensional space, where n is the dimension of the NIM.
- `[@simplify_lang_v](nim, tol, d)`: this simplification function follow the same approach as the previous one and consider a sequence of points in a n -dimensional space, where n is the dimension of the nim and the points are given by the image of the

⁸If the recorded value is a time series, then the x -coordinate corresponds to time and the NIM will be homogeneous as it is incrementally built using the function `[@push_back]`. The only way to build a non-homogeneous nim is to use the function `[@aggregate]`.

⁹The following references gives technical details on polyline simplification. Description and evaluation of several algorithms in a cartographic context are given in [this paper](#). The [Ramer–Douglas–Peucker algorithm](#) follows an idea similar to the Lang algorithm used by `[@simplify_lang_v]` but the shrinking of the current interval is dichotomous. This project provides an [implementation of the main algorithms](#) in C++.

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

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

Smoothing and Transformation

The functions described here work independently on each component of a nim. They see each component as a sequence of points y (the image of the nim) that are smoothed in various way. The resulting points are used to build a new nim with linear interpolation type. The number of breakpoints is preserved, as well as their x -coordinate. Only the y -part of the breakpoint (the image) is affected.

- `[@filter_median_t](nim, n)` smoothes the y by replacing every image by the median in its range- n neighborhood. Notice that the median is taken in a sequence of $2n + 1$ values. The first n points and the last n points are left untouched.
- `[@filter_min_t](nim, n)` filters the y by replacing every image by the minimum value in a sequence of length $2n + 1$ centered on y . The first n points and the last n points are left untouched.
- `[@filter_max_t](nim, n)` filters the y by replacing every image by the maximum value in a sequence of length $2n + 1$ centered on y . The first n points and the last n points are left untouched.
- `[@window_filter_t](nim, coef, pos)` replaces every y by the scalar product of `coef` (a tab) with a sequence of n values, where n is the size of `coef` and `pos` is the position in this window of the current y (numbering starts with 0). The first `pos` values and the last `n - pos` values are left untouched.

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

Process, Macros, Actors and Patterns, which recognize those patterns through user-defined entities and arguments.

Functions are described in this chapter, *processes* in the next one and *macros* in chapter [macro](#). Functions and processes are first class values in the language: they can be elements in a tab or a map, passed as arguments to a function or a process, *etc.* See paragraph [Macro versus Function versus Process](#) for a comparison of the three constructions. [Actors](#) and [Patterns](#) are not primitive entities: they are internally rewritten respectively in process and nested whenever.

Functions live in the domain of expressions and values:

- they are defined by an expression that specifies how the values provided as arguments in a function call are transformed into a(nother) value;
- they are themselves values (see section [Function As Value](#));
- a function call can appear only inside an expression (as a sub-expression) or where an expression is expected (for example in the attributes of an action);
- and the call of a function takes “no time” (see sect. [synchrony hypothesis](#)).

These properties do not hold for macros nor processes (see page [Macro versus Function versus Process](#) for a comparison of the three constructs).

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

In this chapter, we will focus on *intentional functions*. Intentional functions f are defined by arbitrary rules (*i.e.* by an expression) that specify how an image $f(x)$ is associated to an element x .

Some intentional functions are predefined and available in the initial environment like the IEEE mathematical functions. See [Library](#) for a description of more than 190 predefined functions. There is no difference between predefined intentional functions and user’s defined intentional functions except that in a Boolean expression, a user’s defined intentional function is evaluated to `true` and a predefined intentional function is evaluated to `false`.

Intentional functions can be defined (only at the top-level of the score) and associated to an @-identifier using the `@fun_def` construct. For example, the following code shows a convenient user-defined function that converts MIDI pitch values to Hertz. Any call to (for example) `@midi2hz(69)` anywhere in the action language where an expression is allowed (inside messages, *etc.*) will be replaced by its value at run-time.

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

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

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

Function definition

The two examples above are show simple functions whose bodies are just one expression:

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

The name of the function can be a simple identifier or an @-identifier. But in the rest of the score, the function must be referred through its @-name. For the moment, there is no-anonymous functions in *Antescofo*, no optional arguments, no named arguments.

Writing a large function can become cumbersome and may involve the repetition of common sub-expressions. To face these problems, since version 0.8, the body of a function can also be an *extended expression*. See [three kinds of expressions](#) for a presentation of the three classes of expressions.

{!BNF_DIAGRAMS/function_def.html!}

Extended expressions

An extended expression is an optional local variable declaration introduced by `[@local]`, followed by an arbitrary sequence of

1. simple expressions optionally preceded by the `return` keyword
2. local or global variable assignments using `[:=]` (right hand side is a simple expression)
3. iteration expressions `Loop` and `ForAll` whose sub-expressions are extended expressions
4. extended conditional expressions `if .. else ...` and `... whose` sub-expression are extended expressions
5. Max/PD messages
6. abort actions.

This structure is formalized by the diagram below where 'cexpr' refers to [closed expressions](#), 'sexpr' to [simple expressions](#) and 'Exp' to extended expression.

{!BNF_DIAGRAMS/extended_expressions.html!}

Rationale of Extended Expressions. An extended expression is allowed only in the body of a function. This is not because they have something special: they are no more

than “ordinary” expressions. The only motivation behind this constraint is to avoid syntactic ambiguities when the score is parsed. With extended expressions, function definitions are similar to C function definitions that mix expression and statement (but they are differences, see below). As a matter of fact, *Antescofo*’s functions mix expressions and (a few kind of) actions. Only a limited set of actions are allowed in functions: some of the actions that have zero-duration. The rationale is the following: a function call must have no extent in time and the evaluation must be more efficient than a process call.

After some simple introductory examples, we detail these extended constructions.

First Examples

The function definition

```
@fun_def polynomial($x, $a, $b, $c, $d)
{
    @local $x2, $x3
    $x2 := $x * $x
    $x3 := $x2 * $x
    return $a*$x3 + $b*$x2 + $c*$x + $d
}
```

computes ax^3+bx^2+cx+d : the extended expression specifying the function body introduces two local variables used to factorize some sub-computations. The result to be computed is specified by the expression after the `return` statement.

In function

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

the extended conditional is equivalent to but more readable, than the [conditional expression](#):

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

Notice however that, despite the syntax, this `:::atescofo if` is definitively NOT the action described in [conditional action](#): the branches of this are an extended expression, not a sequence of actions.

Remark that the `@fact` function is defined by *recursion*: the definition of `@fact` calls `@fact` itself.

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

```
@fun_def fact($x)
{
```

```

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

```

(but see the predefined functions [`@Tracing`] and [`@UnTracing`] below for easier tracing of function calls). Assignments of global variables, Messages and aborts are instantaneous actions that can be used in extended expressions. They are used for the side-effect they achieve. In an extended expression, such actions cannot specify attributes.

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

```

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

```

which can also be written

```

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

```

Again, the `loop` construction involved here is an expression, not an action. So, one cannot specify a period (expression are supposed to evaluate in the instant), they cannot have attribute, and `end clause` is mandatory to specify the number of iterations of the loop (but it cannot be a duration).

Function's Local Variables and Assignations

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

- their lifetime is limited to one instant, the instant of the function call,
- so it is neither necessary nor possible to refer to these variables outside of their definition scope (*i.e.*, the nearest enclosing extended expression).

As a result, their implementation is optimized (for example, we know that these variables cannot appear in the clause of a `whenever` so the run-time does not need to monitor their assignments, they do not have a history, *etc.*). The cost of accessing a function's local variable is the same as accessing a function argument. Comparing to a global or local variable in groups the gain in the memory footprint and in housekeeping the environment is noticeable.

Local variables are introduced using the keyword `@local` in the **first statement of an extended expression**. Every variable that appears in the left-hand-side of an assignment and whose name does not appear in a clause is assumed to be a global variable¹.

Extended expressions can be nested (through `if`, `loop`, `forall` and `switch` expressions). Each of these nested extended expressions may introduce their own local variables. A variable local to an extended expression is visible only to the sub-expression of this extended expression.

The initial value of a local variable is `<undef>`. Then, the value referred by a local variable is the last value assigned to this variable during the evaluation process. For example, with the definition

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

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

```
@fun_def g($x)
```

¹There is no chance that function would refer to a local variable introduced by a compound action. As a matter of fact, functions are defined at the top-level of the score, where only global variables are visible. Thus, only global and local variables introduced by the extended expression can be referred to in a function body.


```

{
  @local $y
  $y := 2*$x
  $y := $y + 1
}

```

will return '0 when called with x . See section on [exec values](#).

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

The **return** Statement

The value of an extended expression is the value of the last `return` encountered during the evaluation. The `return` is not necessarily the last statement of the sequence. If there is no `return` in the extended expression, the returned value is the value of the last expression in the sequence. If there are multiple `return` at the same expression level, a warning is issued and only the last one is taken into account.

For example:

```

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

```

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

A common pitfall

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

```

@fun_def pitfall($x)
{
  if ($x) { return 0 }
  return 1
}

```

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

```

@fun_def work_as_expected($x)
{
  if ($x)

```

```

        { return 0 }
      else
        { return 1 }
    }

```

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

Extended Conditional Expressions and Iteration Expressions

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

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

These expressions are qualified as *pseudo-actions*. They have been introduced in [extended expressions](#) because loops can be used to specify iterative expressions, and conditionals are useful for controlling the flow of an expression’s evaluation.

The Extended Expression `If`

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

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

So it is mainly used because it improves readability (the branches are extended expressions and may introduce their own local variables). The `else` branch is optional. If `cond` evaluates to `false` and the branch is missing, the returned value is `undef`.

The Extended Expression `Switch`

The syntax of the `switch` expression follows *mutatis mutandis* the syntax of the [switch action](#). For instance, the Fibonacci recursive function can be defined by:

```
@fun_def fibonacci($x)
{

```

```

switch ($x)
{
  case 0: return 1
  case 1: return 1
  case @<(1) :
    @local $x1, $x2
    $x1 := $x - 1
    $x2 := $x1 - 1
    return @fibonacci($x1) + @fibonacci($x2)
}
}

```

Recall that there are two forms of the `switch` construction. In this example, we use the form that compares the selector to the values `::antescofo` 0 and 1. The third value is a predicate² which is applied to the selector, and if true, the attached expression (here an extended expression) is evaluated. Here, for the sake of the example, the `switchexpression` handles the integer 0, 1 and all integers strictly greater than 1.

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

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

The Extended Expression Loop

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

The value of a loop is `undef`. Thus, a `Loop` expression is used for its side effects. For example, the computation of the square root of a strictly positive number p can be computed iteratively using the Newton's formula³:

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

by the following function:

```

@fun_def @square_root($p, $error)
{
  @local $xn, $x, $cpt
  $cpt := 0
  $x := $p
  $xn := 0.5 * ($x + 1)
}

```

²Here the predicate is the partial application of `@<` to 1. The result is a function that compares its argument to 1. We use the infix notation of the relational operator `<` because partial applications are possible only on prefix notation (and `@<` is the prefix notation of the infix `<`). See the [curried functions](#) section in this chapter.

³The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

```

Loop
{
    $x := $xn
    $cpt := $cpt + 1
    $xn := 0.5 * ($x + $p/$x)
} until (($cpt > 1000) || (@abs($xn -$x) < $error))

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

return $xn
}

```

We stress the fact that a *return inside the loop is useless*. As explained before, a `return` is not the indication of a non-local exit but the specification of the value returned by the nearest enclosing extended expression. A `return` in the loop body will specify the value of the body, which is then thrown away by the loop construct that always returns `undef`. This is why the exit of the loop is controlled here by an `until` clause and a `return` at the end of the function body is used to return the correct value.

The Extended Expression `ForAll`

This expression mimics the `ForAll` action. As an expression, it is used for its side-effect. The expression makes iteration possible over the elements of a tab, a map, or a range of integers. The return value is always `undef`.

Atomic Actions in Expressions

Some *atomic actions*, actions with zero-duration, are directly allowed in an extended expression: messages, abort and assignment to a global variable. Such actions may have neither a label nor other action attributes. The value of these actions in an extended expression is `'0`, that is, the value returned by the action-as-expression, see section [Action As Expression](#).

Note that one can launch an arbitrary action within a function body, using the `EXPR { ... }` construction. This construction is a backdoor that can be used to “*inject*” arbitrary actions into the world of expressions⁴.

This possibility is not without danger because it introduces durative action into an instantaneous context. For example, it makes it possible to access a function’s local variable that no longer exists:

```

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

```

⁴In the reverse direction, it is possible to “*inject*” arbitrary expressions into the world of actions, using the `_ := exp` construction which allows for the evaluation of a simple arbitrary expression without other additional effects.

```

}
$res := @pitfall2()

```

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

```

Error: Vanished local variable or function arguments bad access at line 1
Did you try to access an instantaneous variable from an action spanned

```

Indeed, the action spanned in the function body happens one beat after the evaluation of the function call itself, which is instantaneous. So when the action is performed, the local variable corresponding to the call does not exist anymore.

Function Call Evaluation Strategy

Antescofo functions implement *call-by-value* strategy, but this must be tempered by the fact that data-structures are referred to through a pointer. See the side page [Argument Passing Strategies](#). So, *Antescofo* functions can be *impure* (they can have *side effects*).

Argument evaluation order is not specified and is subject to change from one implementation of the language to the other.

And *Antescofo* functions are *strict*: all arguments are fully evaluated before evaluating the function body. (So, logical operators `&&`, `||` are not functions, they are *specials forms*.)

Functions as Values

In an expression, the `@`-identifier of a function denotes a functional value that can be used, for instance, as an argument of higher-order functions (see for example functions `[@map]`, `[@reduce]`, `[@scan]`, *etc.*). This value is of type *intentional function*.

The `@`-identifier of a function is not the only way to denote a functional value. The partial application of a function returns a function through a mechanism called *currying*, described in the next paragraph.

Curried Functions

In *Antescofo*, intentional functions can be *partially applied*. Partial function application says “if you fix the first arguments of the function, you get a function of the remaining arguments”. This notion is related to that of *curried functions*, introduced and developed by the mathematician Haskell Curry. The idea is seeing a function that takes n arguments as equivalent to a function that takes only 1 argument, with $0 < p < n$, and that returns a function that takes $n - 1$ arguments⁵.

Consider for instance

⁵Sometimes a subtle distinction is made between currying and partial function applications. A curried function is a function of arity 1 eventually returning a function which is also curried (expecting one argument). In contrast, partial function application refers to the process of fixing a number p of arguments to a function of arity n , producing another function of smaller arity $n - p$.

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

This function takes 3 arguments, so

```
@f(1, 2, 3)
```

returns 14 computed as: $1+2*2+3*3$. The idea of a curried function, or partial application, is that one can provide less than three arguments to the function `@f`. For example

```
@f(11)
```

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

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

is a function still awaiting one argument, z , to compute finally $55 + 33z$.

Curried functions are extremely useful as arguments of higher-order functions (*i.e.*, functions taking other functions as arguments). An example has been given in the definition of `@fibonacci` to provide a predicate to the case.

For a more appealing example, consider the function `@find(t, f)` that returns the first index i such that $f(i, t[i])$ is true. Suppose that we are looking for the first index whose associated value is greater than a . The value a will change during the program execution. Without relying on currying, one may write

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

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

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

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

All user defined functions are implicitly curried and almost all predefined functions are curried. The exceptions are the special forms and overloaded predefined functions that take a flexible number of arguments, namely: `@dump`, `@dumpvar`, `@flatten`, `@gnuplot`, `@is_prefix`, `@is_subsequence`, `@is_suffix`, `@normalize`, `@plot`, `@rplot`, and `@sort`. When a predefined function does not support partial application, an error message is emitted when an incorrect application occurs.

Tracing Function Calls

It is possible to (un)trace the calls to a function during the program run with the two predefined functions: `[@Tracing]` and `[@UnTracing]`. The trace is emitted on Max or PD console (or on the output specified by the `--message` option for the standalone).

The two predefined functions admit a variety of arguments:

- no argument: all user-defined functions are traced/untraced.
- the functions to trace/untrace: as in `@Trace (@in_between, "@fib")`, will trace/untrace the call and the returns to the listed functions. Notice that the function to (un)trace can be specified with their name or via a string.
- a tab that contains the functions to (un)trace through their name or through strings.

Here is an example:

```
@fun_def @fact($x) { if ($x < 1) { 1 } else { $x * @fact($x-1) } }
_ := @Tracing(@fact)
_ := @fact(4)
```

which generates the following trace:

```
+--> @fact($x=4)
|   +--> @fact($x=3)
|   |   +--> @fact($x=2)
|   |   |   +--> @fact($x=1)
|   |   |   |   +--> @fact($x=0)
|   |   |   |   |   +<-- 1
|   |   |   |   |   +<-- 1
|   |   |   |   +<-- 2
|   |   |   +<-- 6
|   +<-- 24
```

Infix notation for function calls

A function call is usually written in prefix form:

```
@drop($t, 1)
@scramble($t)
```

It is possible to write function calls in *infix* form, as follows:

```
$t.@drop(1)
$t.@scramble()
```

The `@` character is optional in the naming of a function in infix call, so we can also write:

```
$t.drop(1)
$t.scramble()
```

This syntax is reminiscent of the function/method call in *SuperCollider*. The general form is:

```
arg\ensuremath{_{1}} . @fct(arg\ensuremath{_{2}}, arg\ensuremath{_{3}}, ...)
arg\ensuremath{_{1}} . fct(arg\ensuremath{_{2}}, arg\ensuremath{_{3}}, ...)
```

The arg_i are expressions. Notice that the infix call, with or without the @ in the function name, is not ambiguous with the notation `exe.$x` used to refer to a variable x local in a compound action from the `exe` of this action, because the name of a function cannot start with the \$ character.

The infix notation is less general than the prefix notation, because in the prefix notation, the function can be given by an expression. For example, functions can be stored into an array and then called following the result of an expression:

```
$t := [@f, @g]
; ...
($t[$x])()
```

will call @f or @g following the value of the variable x . This cannot be achieved with the infix syntax: only function names (with or without @) are accepted in the infix notation, not expressions. In addition, a function without arguments cannot be called in infix form.

The use of this notation will become apparent with the notion of *method* presented in chapter [Actors](#).

Process

Sequences of actions, such as the body of a [group](#), of a [whenever](#), of a [loop](#), *etc.*, are the specification of a thread of execution⁶ owning its own temporal properties and managed independently by the run-time. An *Antescofo* process simply abstracts this notion by parameterizing a sequence of actions. A process can be instantiated multiple times by giving a value to the parameters (*aka* process call).

Processes are similar to functions: after its definition, a function can be called to compute a value from its body (an expression). After its definition, a process can be called and run from its body which is a *group* of actions. This group is called the *instantiation of the process*. Notice that there can be several instantiations of the same process that run in parallel.

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

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

⁶See the notion of [exec](#) and [coroutine](#). A sequence of actions may span multiples threads, *e.g.* the body of a [loop](#) or of a [whenever](#).

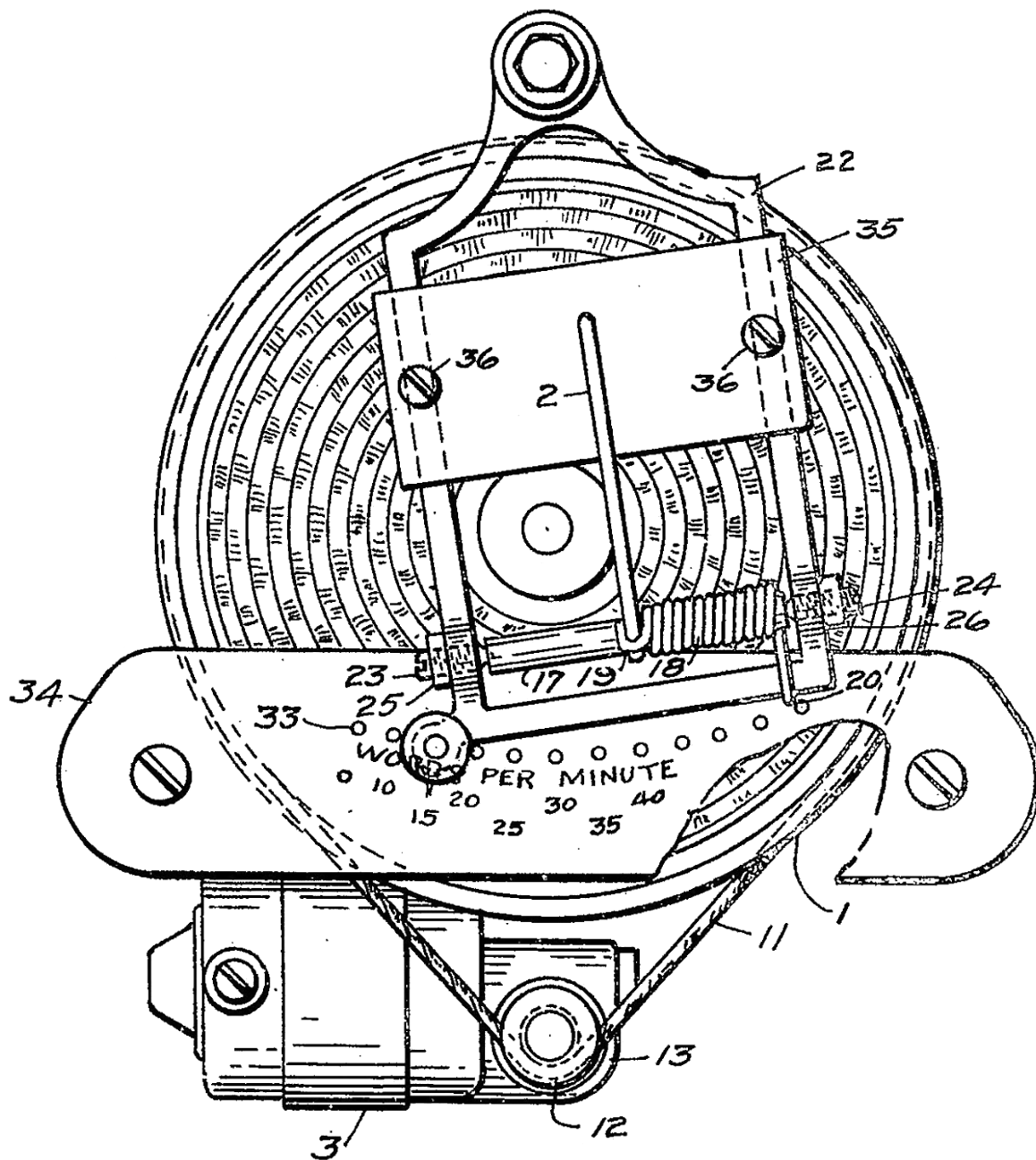


Figure 15.2: metronom

The name of the process denotes a *proc value* (see [Proc Values](#)), and is used to call the process.

Calling a Process

A process call is a mechanism similar to a function call: its parameters are expressions listed between parentheses and their values are bound to the arguments before running the body.

A process can be called as an action or as an expression.

Calling a Process as an Action

Here, calling a process is an action:

```
NOTE C4 1.3
  ::trace("C4", 1.3)
  ; more actions
NOTE D3 0.5
  ::trace("D3", 0.5)
```

In the previous code, the process is referred to through its name, a `::`-identifier, so it is apparent that a process is called. But the call can also be indirect, as in:

```
NOTE C4 1.3
  $x := ::trace
  :: $x ("C4", 1.3) ; calling the process assigned to $x
```

the process `::trace` is a value (of type [proc](#)) assigned to variable `$x`. To call the process referred by `$x`, we use the syntax `:: expression (...)` where `expression` must evaluate to a [proc](#).

A process call is a compound action equivalent to the insertion of the process body at the place of call as a group. So the previous code fragments launch the following behavior:

```
NOTE C4 1.3
  group trace_body1 {
    print begin "C4"
    1.3 print end "C4"
  }
  ; more actions
NOTE D3 0.5
  group trace_body2 {
    print begin "D3"
    0.5 print end "D3"
  }
```

Calling a Process as an Expression

A process can also be called in an expression with the same syntax. The instantiation mechanism is similar: a group starts and runs in parallel. However, an *exec value* is returned as the result of the process call. See section [exec value](#). This value refers to the running group launched by the process instantiation. It can be used in the computation of the surrounding expression.

The *exec* value corresponding to the process instance can also be accessed within the process body itself through a *special variable*

```
$MYSELF
```

This variable is read-only and is managed by the run-time.

Calling a process in an expression does not require the `::` marker, that is the expression: `antescofo $x(1, 2, 3)` can be either a process call or a function call, depending on the value of `$x`. It causes no trouble. The `::` keyword is mandatory only in actions, because allowing the syntax of function calls would lead to ambiguities in the specification of actions.

Recursive Process

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

```
Loop L 10
{
    ; actions ...
}
```

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

```
@proc_def ::L()
{
    Group iterate { 10 ::L() }
    ; actions ...
}
```

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

Process as Values

A process definition is a [proc](#) value and can be the argument of another process. For example:

```
@Proc_def ::Tic($x) {
    $x print TIC
```

```

}

@proc_def ::Toc($x) {
    $x print TOC
}

@proc_def ::Clock($p, $q) {
    :: $p(1)
    :: $q(2)
    2 ::Clock($p, $q)
}

```

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

Aborting a Process

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

```
abort ::P
```

will abort all the active instances of `::P`. The *active instances* of the process are the instantiation of the process body that are still [alive](#).

It is possible to kill a specific instance of the process using its *exec* value:

```

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

```

Using the special variable `$MYSELF`, it is possible to implement self-destruction on a specific condition `e`:

```

@proc_def ::Q()
{
    @local $PID
    $PID := $MYSELF
    ; ...
    whenever (e) { abort $PID }
    ; ...
}

```

The value of `$MYSELF` is stored in the local variable `$PID` because the value of `$MYSELF` is the *exec* value of the *immediately surrounding group*. If we replace `$PID` with `$MYSELF` in the `whenever`, we kill the running instance of the `whenever` body, not the process instance.

Processes and (local) Variables

Processes are defined at the top-level. So, the body of the process can only refer to global variables and to local variables introduced in the body. The process parameters are implicit local variables but other local variables can be introduced explicitly using the `@local` statement.

Process parameters are local variables

The parameters of a process are local variables that are initialized with the value of the arguments given in the process call. These variables are local to the process instance. These variables can be set in the process body, as in:

```
@proc_def ::P($x)
{
    whenever ($x == $x) {
        print "$x = " $x
    }
    ; ...
    1 $x := $x + 1
}
```

One beat after the call `::P(0)`, the text `$x = 2` appears on the console: the variable `$x` local to this process instance has been incremented by one which has triggered the `whenever` outputting the message. We stress that *values* are passed to the process, not variables or expressions, which is sometime a cause of pitfalls, see paragraph [What to Choose Between Macro, Functions and Processes](#).

Other than the initialization, there is no difference at all between the process parameters and a local variable introduced explicitly using `@local` statement.

Local variables

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

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

The left hand side of the infix operator `.` must be an expression whose value is an active *exec*. The right hand side is a variable local to the referred *exec*. If the left hand side refers to a dead *exec* (cf. `exec`), the evaluation of the dot expression raises an error.

In the previous example an instance of `::DrunkenClock` is recorded in variable `$dclock`. This *exec* is then used to access to the variable `$tic` which is local to the process. This variable is incremented with a random period varying between 0.75 and 1.25.

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

Dynamically Scoped Variable

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

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

$x := "x at top level"

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

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

::Q("Q at top level")
```

will print:

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

Assignment using the dot notation

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

```

@proc_def ::P()
{
    @local $x
    whenever ($x) { print "$x has changed for the value " $x }
    ; ...
}
$p := ::P()
; ...
$p.$x := 33

```

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

Notice that the features described here specifically for a process instance work for any *exec* (see sections [Action As Expression](#) and [exec value](#)).

Process, Tempo and Synchronization

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

```

@proc_def ::P() @tempo := ...
{ ... }

```

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

For example:

```

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

```

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

If not explicitly specified, the tempo of a process instance is inherited from the call site. This is *also true for the other synchronization attributes* (targets, strategies, etc.).

Actors

Actors are built on processes to provide autonomous objects that can react to external signals and able to answer requests initiated from other program parts. Actors do not belong to the *Antescofo* core: they do not add fundamental mechanisms, they rather provide *syntactic sugar* for the sake of the programmers and they are internally rewritten as processes. They are described in chapter [actors](#).

Chapter 16

Macros

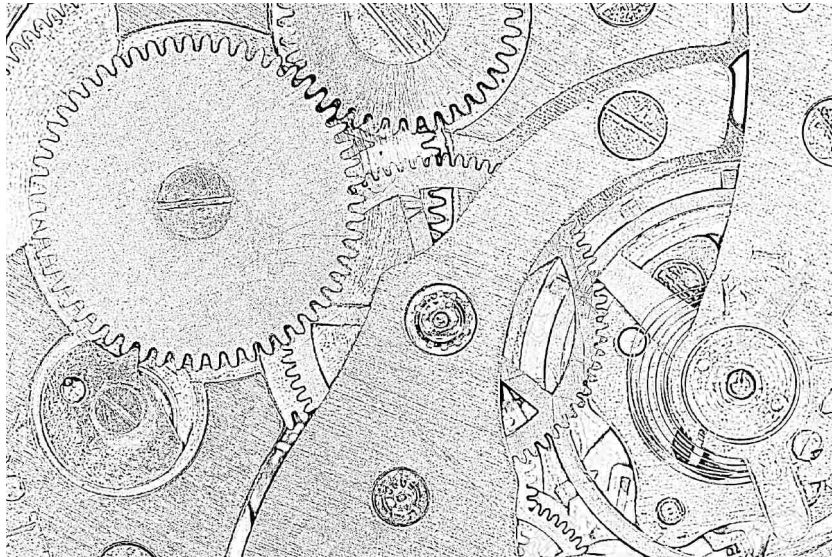


Figure 16.1: clock mechanism

Functions are used to abstract some variables over an expression and to repeatedly evaluate this expression with a change to the value referenced by these abstracted variables. Processes play a similar role, with a group of actions instead of an expression.

Macros can play both roles because the abstracted object is a text without *a priori* semantics. The mechanisms is then more primitive but may have some advantages. Refer to the side page [Macro versus Function versus Process](#) for a comparison of the three constructs. Usually, if a process or a function can do the same job as a macro, there are advantages to using them over macros and you should give them priority.

Macro Definition and Usage

Macro definitions are introduced by the `@macro_def` keyword. Macros are called by their `@-name` followed by their arguments between parentheses.

```
@macro_def @b2sec($beat) { $beat / (60. * $RT_TEMPO) }
```

A call to a macro is simply replaced by its definitions and given arguments in the text of the score: this process is called *macro-expansion* and is performed **before** program execution. The macro-expansion is a syntactic replacement that occurs during the parsing and before any evaluation. Macros are thus evaluated at score load and are NOT dynamic. The body of a macro can call other macros but macros cannot be recursive¹.

Macro names are @-identifiers. For backwards compatibility reasons, a simple identifier can be used in the definition but the @-form must be used to call the macro. Macro arguments are formal parameters using \$-identifiers (but they are not variable!). The body of the macro is between braces. The white spaces, tabulation and carriage-returns immediately after the open brace and immediately before the closing brace are not part of the macro body.

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

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

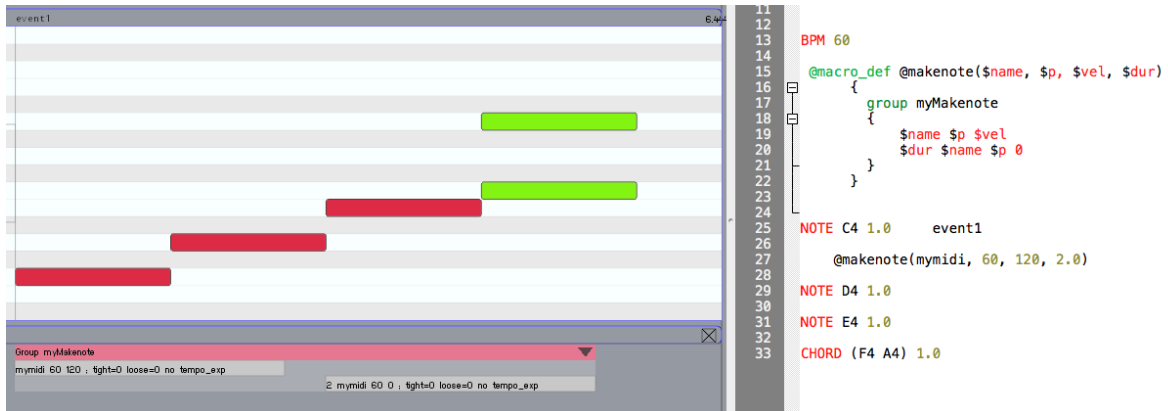
The figure below shows the above definition with its realization in a score as shown in *AscoGraph*. The call to the macro can be seen in the text window on the right, and its realization on the graphical representation on the left. Since Macros are expanded upon the loading of the score, you can only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

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

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

results in the following code after score is loaded:

¹A recursive macro definition will lead to an infinite expansion.



The screenshot shows a music notation software interface. On the left, a piano roll view displays a score with several notes represented by colored bars (red, green, and blue) on a grid. On the right, a code editor shows the following code:

```

11
12
13 BPM 60
14
15 @macro_def @makenote($name, $p, $vel, $dur)
16 {
17   group myMakenote
18   {
19     $name $p $vel
20     $dur $name $p 0
21   }
22 }
23
24 NOTE C4 1.0 event1
25
26 @makenote(mymidi, 60, 120, 2.0)
27
28 NOTE D4 1.0
29
30 NOTE E4 1.0
31
32
33 CHORD (F4 A4) 1.0

```

Below the piano roll, a group named 'myMakenote' is expanded, showing two instances of the macro definition with their respective arguments.

Figure 16.2: Example of a Macro and its realisation upon score load

```

5 group {
6   1 print One
7   2 print Two
8 }

```

Macros can accept zero arguments. In this case, there is no list of arguments at all:

```

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

```

Expansion Sequence

The body of a macro @m can contain calls to other macros, but they will be expanded *after* the expansion of @m. Similarly, the arguments of a macro may contain calls to other macros, but beware that their expansion takes place only *after* the expansion of the enclosing 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 tokens².

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).

Generating New Names

The use of macro often requires the generation of new names. As an alternative, consider using local variables that can be introduced in groups. Local variables enable the reuse of identifier names and are visible only within their scope.

However, local variables are not always a solution. In this case, there are two special macro constructs that can be used to generate 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(id)`. For instance

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

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

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

²This behavior differs, for instance, from the behavior of the macro-processor used for C or C++ where `1234@concat (56, 78)` would have been expanded into `12345678`. The difference is that `cpp-macro` expansion takes place *before* any parsing, at the raw level of the stream of characters, while *Antescofo* macro-expansion take place *during* the parsing, as a phase of the lexical analysis at the level of the stream of tokens.

```

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

```

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

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

```

@macro_def @Gen($x, $d, $action)
{
    group @name := Gengroup$x
    {
        $d $action
        $d $action
    }
}

```

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

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

will expand into

```

group @name := Gengroupone
{
    5 print Ok
    5 print Ok
}

```

Another trick is to know that comments are removed during the macro-expansion, so you can use comment to concatenate text after an argument, as with the C preprocessor:

```

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

```

With these definitions,

```

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

```

is replaced by

```

$yyysuffix
3.14159265

```

What to choose between macro, functions and processes

Capitalizing some code fragment to reuse it several times raises the recurring questions: should we use a macro, a function or a process? The side page [Macro versus Function versus Process](#) compares the three mechanisms. If the purpose of the code fragment is to produce a value, in the same instant as the call, then a function should be considered. If it is to perform actions, especially actions that take time, then a process should be considered. In other cases, such as if the code fragment must be parameterized by a variable name, then a macro must be considered.

The last point deserves some development. Consider the following process definition:

```
let $myVar := 0

@proc_def ::P($x)
{
    whenever ($x) {
        ; do something
    }
}

::P($myVar)
; ...
let $myVar := 1
```

If the intention of the programmer is to activate the `whenever` in the process `::P` each time the variable `$myVar` is set, the previous approach is incorrect: the `whenever` in `::P` is activated each time the local variable `$x` is set. When the process is called, the argument is evaluated and it is the value of `$myVar` which is passed to the process, *not* the variable itself³. This is why a process can be called with constant arguments:

```
::P(0)
```

With this call, it is apparent that the `whenever` in `::P` watches the local variable `$x` because there is no other variable involved.

To achieve the intended behavior, the name of the variable to watch must explicitly appear in the condition of the `whenever`. Macros are handy for that, because they are expanded literally:

```
let $myVar := 0

@macro_def @P($x)
{
    whenever ($x) {
        ; do something
    }
}
```

³See the side note [argument passing strategies](#) for more details.

```
@P($myVar)
; ...
let $myVar := 1
```

is expanded into

```
let $myVar := 0

whenever ($myVar) {
    ; do something
}
; ...
let $myVar := 1 ; this time, "do something" will be triggered
```

which achieves the desired behavior.

Chapter 17

Actors (objects)

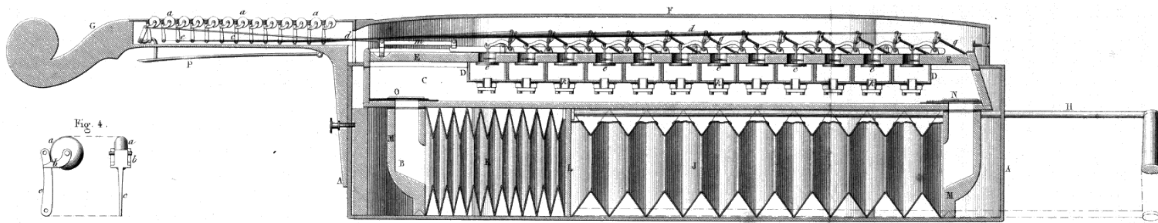


Figure 17.1: melophone

The notion of an *object* is now widespread in programming. This concept is used to organize code by gathering values together into a *state* and making the possible interactions with this state explicit through the notion of *methods*.

A related and less popular notion is the concept of an *actor*. The actor model of programming was developed in the beginning of the '70s with the work of Carl Hewitt and languages like Act. Later, Actor programming languages included the Ptolemy programming language and languages offering “parallel objects” like Scala or Erlang.

While objects focus on reuse with mechanisms like inheritance, method subtyping, state hiding, *etc.*, actors focus on the management of concurrent activities of autonomous entities.

In *Antescofo* we use the words *object* or *actor* interchangeably to refer to some kind of process used to encapsulate a state and to the concurrent, parallel and timed interactions with this state. The specification of these objects and the interactions with them are supported by some specific syntactic constructs. However, these dedicated constructions are internally rewritten in more fundamental mechanisms of functions, processes and whenever statements.

This chapter supposes a knowledge of the notions of objects and methods. The next section compares the notion of object with the notion of process. Then we describe the *Antescofo* notion of actors.

Introduction: Process as Object

A process instance can be used as a kind of autonomous entity encapsulating some data. In fact, a running process can be seen as an object or as an actor:

- a process instance is similar to the instance of a class: the process is the class and calling a process corresponds to class instantiation;
- the interval of time between the process call and the process end corresponds to the lifetime of the object;
- the *exe* of the instance corresponds to a reference to the object;
- the state of the object corresponds to the values of the local variables of the process instance;
- interactions with the object can be achieved by assigning its local variable (see [dot notation](#)).

Local variable assignments act as messages and in response to a message that it receives, a running process can make local decisions, create more processes, send more messages, and determine how to respond to the next message received.

For example, suppose we want to design an object of a class `channel` supposed to control some audio channel. The object iterates periodically over a list of parameters to be sent to a harmonizer. During the object lifetime, it is possible to add a new parameter and to reset the parameter list. We can implement this notion through a process `::channel`. By assigning the local variable `$velocity`, a new parameter is added to the list. By assigning it to 0, this list is reset to its initial value.

```
@proc_def ::channel($ch, $init, $period)
{
    @local $velocity, $stab, $i
    $velocity := 0
    $stab := $init
    $i := 0

    whenever ($velocity == 0)
    { $stab := init }

    whenever ($velocity != 0)
    { _ := @push_back($stab, $velocity) }

    Loop $period
    {
        $i := ($i + 1) % @size($stab)
        @command{ "harmony" ++ $ch } ($stab[$i])
    }
}
; ...
```

```

$P0 := ::channel(0, [12, 15], 1)
$P1 := ::channel(1, [10, 8, 9, 11], 1.5)
; ...

$P0.$velocity := 7
; ...

```

The dot notation is efficient but does not make the interactions with the object (running process) apparent. Functions can be used to make these interactions more explicit. Suppose we want to simultaneously change the period and reset the parameter list to its initial value. We can write a function:

```

@fun_def reset($pid, $per)
{
    $pid.$period := $per
    $pid.$velocity := 0
}

```

then we can call the function :

```
@reset($pid, 1.5)
```

and using the infix notation for function calls introduced in section [Infix Notation for Function Calls](#):

```
$pid.reset(1.5)
```

This last form is in line with the usual notation used to call an object's method: we ask the object (specified through its *exe* \$pid) to perform the *method* with parameter 1.5.

Actors

The previous idea — relying on processes to achieve a kind of concurrent object oriented programming — is pushed further with the @obj_def construction. An @obj_def definition is internally expanded into a process definition and into function definitions following the line sketched by the previous example. So, there is no fundamentally new mechanism involved. However, the dedicated syntax makes the programming more readable and reusable.

An *obj* definition is introduced by the keyword @obj_def and consists in a sequence of clauses. The order of the clause may be relevant. A clause of a given type may appear several times in an object definition. There are 8 kind of clauses, introduced by a keyword:

- @local introduces the declaration of the *fields* (also known as the *attribute*) of the object.
- @init defines a sequence of actions that will be launched at object instantiation.

- `@method_def` or `@fun_def` specifies a new method, *i.e.* a function that can be run on a specific obj. Such methods are also named *instance method* or *object method* because they involve a specific instance of an object. The body of a method is an extended expression.
- `@proc_def` specifies a new method, which is similar to the previous construction, except that the body of a routine is a sequence of actions (not an extended expression). These methods are sometimes called *routines*. They can have a duration and multiple instances of the same routine can be simultaneously active for the same object.
- `@broadcast` declares a function that performs simultaneously on all instances of an object.
- `@whenever` introduces a *daemon* which triggers a sequence of actions when some logical expression becomes true.
- `@react` is similar to the previous construct but triggers the evaluation of an extended expression when some logical expression becomes true.
- `@abort` defines an abort handler that will be triggered when the object is killed.

An object definition plays a role similar to a *class* in object-oriented programming, except that there is no notion of class inheritance in the current *Antescofo* version. Another difference is that objects run “in parallel” and their actions are subject to synchronization with the musician or on a variable, they can be performed on a given tempo, *etc.* As a matter of fact, as previously mentioned, objects are processes with some syntactic sugar.

{!BNF_DIAGRAMS/object_def.html!}

[\(click here for a larger view\)](#)

After a motivating example, we will detail the various clauses of an object definition.

A Basic Example

Here is a first example of an object definition:

```
@obj_def Metro($p, $receiver)
{
    @local $period, $trigger, $body

    @init {
        $trigger := false
        $body := 0
    }

    @whenever ($trigger)
    {
        $body := { Loop $period { @command($receiver) TOP } }
    }
}
```

```

@init {
  $period := $p
  $trigger := true
}

@broadcast reset()
{
  abort $body
  $period := $p
  $trigger := true
}

@method_def current_period() { return $period }
@method_def set_period($x)   { $period := $x }

@abort { print "object " $THISOBJECT "is killed" }
}

```

The object is called `obj::Metro` and corresponds to a new *type* of value. This type is a subtype (a specialization) of `proc`. It can be instantiated by giving the expected arguments for the object creation. These arguments are specified between parentheses after the object's name.

```

$metro1 := obj::Metro(2/3, "left_channel")
$metro2 := obj::Metro(1, "right_channel")

```

An object of type `obj::Metro` is created with an initial period. The purpose of this object is to send a message TOP to a receiver each period. The loop implementing the periodic emission of the TOP is triggered by a `whenever` controlled by a field (a local variable) `$trigger`. The exec of this loop is saved in field `$body` and used to abort the loop when the broadcast `@reset` is emitted.

Two methods are provided: `@set_period` is used to change the value of the period (the change is taken into account at the end of the current period) and `@current_period` is used to query the period actually used by a `obj::Metro`. The broadcast `@reset` can be used to reset the period of all running instances of a to their initial value (the value given at creation time). Here are some examples:

```

_ := $metro1.set_period(2 * $metro2.current_period())

```

sets the period of the first object to twice the period of the second object. All periods are reset calling the broadcast:

```

_ := @reset()

```

Note that a broadcast corresponds to an ordinary function. This function launches simultaneously, for all active instances, the code associated to the broadcast.

Notice that the definition specifies two `@init` clauses: the first one takes place before the `@whenever` and initializes the fields of the object. The second `@init` clause is used to launch the loop when the object is created and after the start of the `@whenever`.

An object lives “forever”. It can be killed and the command `abort`. When killed, the object will execute its abort handler. In the example, the abort handler uses the system variable `$THISOBJ` that refers, in the scope of an object clause, to the current instance of the object.

In the next paragraphs, we detail the various clauses present in an object definition.

Field Definition: `@local`

The clause has the same syntax as the declaration used to introduce local variables in a compound action. Here each “local variable” is used as a field of the object and corresponds to a local variable in the process that implements the object. The values of the fields/local variables represents the state of the object. *Antescofo* is a dynamically typed programming language, so the fields of an object have no specified type and can hold any kind of values in the course of time.

Several `@local` clauses can be defined and their order and placement is meaningless. Object fields are present from the start and initialized with the `undef` value.

Note that the argument of an object corresponds to implicitly defined fields. So, in the previous example, the state of the object is given by 5 variables: the initial period `$p`, the receiver `$receiver`, the current period `$period`, a control variable `$trigger` and the exec to the running loop that implements the object behavior `$body`.

A reference to a field may appear anywhere in a clause and always refers to the corresponding local variable. A variable identifier that is not declared as a local variable, refers to a global variable.

Performing an Action at the Object Construction: `@init`

Fields are initialized in `@init` clauses. Init clauses are interleaved with `@whenever` clauses and this order is preserved in the implementation, which makes possible to control the order of evaluation and the triggering of the whenever clauses.

In our example, the assignment of `$trigger` in the second `@init` clause will trigger the `@whenever` previously defined.

Specifying an Object Method: `@method_def` and `@proc_def`

Methods are functions or processes that are associated to an object. They represent some behaviors specified as

- an expression: such methods are introduced by the keyword `:::atescofo @method_def` or equivalently by `@fun_def` (because such methods are similar to functions);
- or a sequence of actions: such methods are introduced by the keyword `@proc_def` (because such methods are similar to processes). Such methods are called *routines* when we want to distinguish them from the previous type of methods.

Methods are called on objects and may involve some additional arguments. They have several advantages over bare functions or processes:

- A method can be overloaded, that is, the same method name can be used for a different object.
- When called, a method checks implicitly that it is called on a live instance of an object.
- When called, a method checks implicitly that it is called on an object of the expected type.
- A method has direct access to the object's fields.

These benefits come at some cost:

- A method can be called only through the infix call notation. A side consequence is that a method call is an expression (even if the method is a routine).
- Methods are not values, they are just simple names. For instance, you cannot pass them as arguments¹.

Ambiguity Between a Method Call and a Function Call in Infix Form

There is a possible ambiguity between infix function calls and method calls. This ambiguity arises if a function `@f` is defined and takes a first argument (an object) on which a method `f` is also defined. Then the call

```
obj . f (a\ensuremath{\_i})
```

is ambiguous: is it a *method* call to `f` or a function call `@f(obj, ai)`?

In this case, the rule is to call the method (if `obj` is alive). If you want to call the function, use the `@`-identifier in the call:

```
obj . @f (a\ensuremath{\_i})
```

Calling a Method on a Dead Object

If a method is called on a dead process, *Antescofo* looks for an ordinary function with the same identifier. If this function exists, it is called with the same arguments, instead of calling the method. If this function does not exist, an error is signaled.

Here is an example

```
@obj_def obj::Account()
{
  @local $deposit
  @init { $deposit := 0 }
  @fun_def credit($x) { $deposit := $deposit + $x }
  @fun_def debit($x) { $deposit := $deposit - $x }
}
```

¹However, you can apply them partially and use the partial application as a value (the value is of type (partially applied) *function*).

```

@fun_def @credit($x)
{ print "cannot credit a non-longer existant account" }

; ...
$joe := obj::Account()
_ := $joe.credit(100)

; ...
abort $joe

; ...
_ := $joe.credit(100)

```

The last assignment will trigger the evaluation of `::atescofo @credit(100)` because `$joe` no longer exists. Notice that methods are called using the `_ := action`, because they are expressions. They have the same status as a function call. So they cannot appear directly as actions.

Calling a Method on an Object of Incorrect Type

Method calls check that the method is defined on the object given as an argument. If this not the case, then a function with the same name is looked at and applied on the arguments. If such function does not exist, an error is signaled.

Calling a Method Within a Method

All method calls in the object definition which refer to the current object instance can be written in an abbreviated infix form that omits the receiver:

```
. method_name(a\ensuremath{_1}, a\ensuremath{_2}, ...)
```

instead of

```
$THISOBJ . method_name(a\ensuremath{_1}, a\ensuremath{_2}, ...)
```

the special variable `$THISOBJ` refers to the object instance on which the method is called, (see below). Obviously, the full syntax to call a method must be used if the receiver is not the current instance.

Accessing Object Fields in Methods

An object's field can be accessed directly through its `$-name` in the body of a method, as showed in the example by the body of the method `credit`. It is always possible to access to an object's field from "outside its methods" through the dot notation:

```
obj . $field
```

where `obj` is an expression evaluating to the object reference (an exec).

Local variables in methods

Methods defined through `@fun_def` are specified using [extended expressions](#). Such an expression may introduce local variables.

Routines are defined through a sequence of actions that may involve local variables. These variables are local to the routine instance (they cannot be accessed by others methods nor other routines). Actions in the routine body may have duration. Thus, several instances of the same routine may be active at the same moment.

Referring to the object: `$THISOBJ`

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

This variable is special: it has a meaning only in the scope of a method or in the clauses of an object. It cannot be watched by a [whenever](#). Assigning this variable leads to unpredictable result.

In a method body, a reference to an object field

```
$THISOBJ . $field
```

can be abbreviated in

```
$field
```

and a method called on the same object

```
$THISOBJ . method (...)
```

can be abbreviated in

```
. method_name(a , a , ...)
```

Specifying a Broadcast: `@broadcast`

Each broadcast clause defines a function with the same name. The syntax is similar to that of a function definition, except that it is introduced by the keyword `@broadcast`. Calling this function will execute the body of the function for each active instance of the object. The value returned is [undef](#).

Here is an example where a broadcast is used to count the number of instances:

```
@global $MyObj_count

@obj_def MyObj()
{
```

```

        @broadcast count() { $MyObj_count := $MyObj_count + 1 }
    }

    @fun_def countMyObj()
    {
        $MyObj_count := 0
        @count()
        return $MyObj_count
    }

    ; ...
    $number_alive_MyObj := @countMyObj()

```

A global variable `$MyObj_count` is used to add the number of instances. In the body of the broadcast, the fields of the object are accessible. A function `@countMyObj` is defined to reset the global variable, to broadcast the counting method and to return the result. This approach can be used to implement any function that do a global operation over all instances of an object.

Admittedly, using a global variable to share information between the various applications and the object instances can be troublesome. The broadcast mechanism will be extended to face this kind of problem in the next version of the language.

Specifying a Reaction: `@whenever` and `@react`

`@whenever` and `@react` clauses can be used to define the triggering of some actions *or* some expressions when some logical conditions occur. They are similar to (and implemented by) a `whenever`.

This construct makes it possible to define **daemons** that automatically respond to some events. There are two ways to do this. To launch actions, the syntax is:

```
@whenever(expression) { actions }
```

and to launch an extended expression, the syntax is

```
@react(expression) { extended_expression }
```

The second version is appropriate if the reaction consists in state update and instantaneous computations. The first form can be used to launch child processes and other durative actions. Note that because some actions are allowed in extended expressions, it is often possible to use one both forms interchangeably. In either case, it is possible to use termination guards as in

```
@react($x == $x) { $cpt := $cpt + 1 } until ($cpt > 3)
```

[Patterns](#) can be used to define complex condition in time.

Nota Bene: there is a daemon active for each object's instance.

Specifying an Abort Handler: @abort

The @abort clauses are gathered together and are launched when an object instance is killed. Object instances “live forever” (until a stop command) and they must explicitly be killed by an `abort` action.

Checking the Type of an Object: @is_obj and @is_obj_xxx

An instance of an object is implemented by a process, so it is of the `exec` type and the predicate `@is_exec` returns true on an object.

The predicate `@is_obj` can be used to distinguish between `exec` (object instances and process instances and more generally, instances of compound actions).

In addition, each time an object `obj::xxx` is defined using `@obj_def`, a predicate `@is_obj_xxx` is automatically defined. This predicate returns true if its argument is an object instance of `obj::xxx`.

Object Instantiation

An instance of an object is created using a syntax similar to that of a process call²:

```
$metro1 := obj::Metro(2/3, "left_channel")
```

creates an object of type `obj::Metro` with a parameter `$p` that sets to `2/3` and a parameter `$receiver` that sets to `"left_channel"`.

When an object is created, the `@init` clauses and the reaction clauses are performed in the order of their definition. Then, the object is alive and ready to interact. Interactions can happen through

- method calls,
- broadcasts,
- direct assignments of the object fields (using [dot notation](#))
- changes in logical condition of reactions
- and killing the object.

Concurrency Between Method Applications

Methods defined by `@fun_def` are evaluated instantaneously. In accordance with the [synchrony hypothesis](#), their runs “cannot overlap” and correspond implicitly to atomic region. So there is no need to use [semaphore](#), [mutex](#) or any other dedicated mechanism to implement [mutual exclusion](#) between method applications: mutual exclusion is given automatically.

²and is actually implemented by a process call

Routine executions may persist in time. So, two routines called on the same object may run in parallel and may concurrently access the same fields. This may lead to consistency problems. However, sequences of atomic actions without delay are always executed instantaneously and in mutual exclusion with other such sequences: they are natural [critical sections](#).

Object Expansion into Processes and Functions

The previous constructions are internally expanded in a process definition and in several function definitions, so there are no new evaluation mechanisms involved with the object constructions. However, they help in structuring the score³.

Keep in mind that an object, like any other process, can be synchronized. In particular, object inherits their synchronization from the synchronization strategy defined at their creation.

³Objects are a new feature in *Antescofo* and are expected to evolve to integrate new mechanisms: we plan to integrate reactions linked to the reception of OSC messages, object fields shared by all instances, the unification of the broadcast mechanism with a map-reduce mechanism. In the long term, we want to develop inheritance mechanisms and the specification of more sophisticated concurrency constraints between routines.

Chapter 18

Patterns

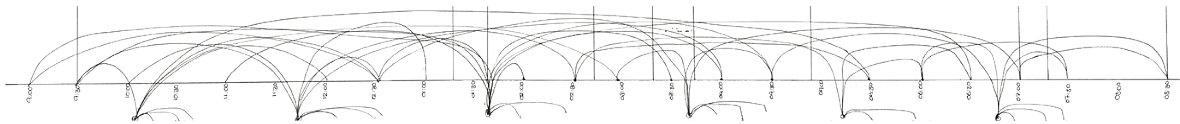


Figure 18.1: header figure

Patterns are a simple way to define complex logical conditions to be used in a [whenever](#). A pattern is a sequence of *atomic patterns* that describe the evolution through time of logical conditions. There is three kinds of atomic patterns: Note, Event and State.

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

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

Note: Patterns on Score

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

```
@pattern_def pattern::P
{
    Note C4 0.5
    Note D4 1.0
```

```
}
```

defines a pattern that can be used later as the argument of a [whenever](#):

```
whenever pattern::P
{
    print "found pattern P"
}
```

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

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

Pattern Variables

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

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

    Note $h
    Note $h
}
```

The pattern `Note` matches a note in the input followed by the listening machine. The pattern `pattern::Q` defines a repetition of two successive notes with the same pitch (their respective duration do not matter).

The wildcard, or *pattern variable* `$h`, is specified in the clause at the beginning of the pattern definition using a `@local` declaration. Every occurrence of a pattern variable must refer to the same value. Here, this value is the pitch of the detected note (given in midicents).

Pattern variables are really local variables and their scope extends to the body of the `whenever` that uses this pattern. So they can be used to parametrize the actions to be triggered. For example:

```
whenever pattern::Q
{
    print "detection of the repetition of pitch " $h
}
```

Specifying Duration

A pattern variable can also be used to restrict durations in the same manner. The value of a duration is the value given in the score (and *not* the actual duration played by the musician).

Specifying Constraints

There are two parameters for a note: its pitch and its duration. These parameters may be specified by

- a constant
- a pattern variable specifying an unknown value determined at matching time
- an (ordinary) variable specifying a specific value (the value of the variable at matching time)

For pitches, the constant is an integer or the ratio of two integers, or a symbolic note. These constants specify the expected pitch in midicents.

For duration, the constant specifies the expected duration in relative time (in beat) or in absolute time (in second with the *s* unit appended or in millisecond with *ms* appended). Notice however that such specification is probably useless: there is few chance that the actual note duration is exactly equal to the specified one.

A pattern variable can be used as a kind of wildcard. Declared using an `:::antescofo @local` declaration, the variable takes its actual value with the matching of its first occurrence. The following occurrence of the variable constraint the corresponding matching to take the same value.

An ordinary *Antescofo* variable can be used to specify a pitch or a duration. In this case, only a note with the specified pitch or duration is matched by the *note* pattern. The specified value is the value of the variable at the time of matching and this value can be changed dynamically (with an assignment). This mechanism can be used to adapt a pattern to a given context.

Here is an example involving ordinary variables in a pattern :

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

specifies a sequence of two notes. The first one must have a pitch equal to the value of the variable *\$X* (at the time where the pattern is checked). The pitch of the second one is *C4*, and the duration of the first is irrelevant while the duration of the second must be equal to the value of *\$Y*. As for *\$X*, this variable is updated elsewhere and the value considered is its value at the time where the pattern is checked. These two variables are recognized as ordinary variables and not as pattern variables, because they are not declared with a `@local` in the scope of the pattern.

Additional constraints on the matching can be specified through a *where* clause which specifies a logical expression which must be true for the matching to succeed:

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

```

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

```

specifies a sequence of two successive notes such that:

- their pitch is equal and this value in midicents is the value of the local variable `$h`;
- `$h` in midicents is higher than 6300;
- and the duration `:::atescofo $dur2` of the second note must be lower than the duration `:::atescofo $dur1` of the first note.

The logical expression after the `where` is an arbitrary expression. If it involves variables, the value of these variables is the value of the variable at the time of matching.

Pattern Causality

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

However, it is easy to postpone a `antescofo::: where` clause to an event where all pattern variables have been set. For example, writing:

```

@pattern_def pattern::R
{
  @local $h, $dur1, $dur2

  Note $h $dur1 where ($h > 6300) && ($dur2 > 0.5)
  Note $h $dur2 where $dur2 < $dur1
}

```

One can also write

```

@pattern_def pattern::R
{
  @local $h, $dur1, $dur2

  Note $h $dur1 where $h > 6300
  Note $h $dur2 where ($dur2 < $dur1) && ($dur2 > 0.5)
}

```

A Complete Example

The pattern


```

@pattern_def pattern::M
{
    @local $h, $dur

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

```

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

Event on Arbitrary Variables

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

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

```

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

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

```

The keyword `Event` is used here to specify that the event we are looking for is an *update* in the value of the variable `$S`¹. We say that `$S` is the *watched variable* of the pattern.

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

Four optional clauses can be used to constrain an `Event` pattern:

1. The `before` clause is used to specify a temporal scope for looking at the pattern.
2. The `value` clause is used to give a name or to constrain the value of the variable specified in the `at` matching time.

¹A variable may be updated while keeping the same value, as for instance when evaluating `let $S := $S`. Why `$S` is updated or what it represents does not matter here. For example, `$S` can be the result of some computation in `Antescofo` to record a rhythmic structure. Or `$S` is computed in the environment using a pitch detector or a gesture follower and its value is notified to `Antescofo` using a `setvar` message.

3. The `at` clause can be used to refer elsewhere to the time at which the pattern occurs.
4. The `where` clause can be used to specify additional logical constraint.

The `Before` clause must be given before the `Event` keyword. The last three clauses can be given in any order after the specification of the watched variable.

Contrary to the `Note` pattern, there is no “duration” clause because an event is point wise in time: it detects the update of a variable, which is instantaneous.

The `value` Clause

The `value` clause used in an `Event` is more general than that used in a `Note` pattern: it accepts a pattern variable or an arbitrary expression. An arbitrary expression constrains the value of the watched variable to be equal to the value of this expression². A pattern variable is bound to the value of the watched variable. This pattern variable can be used elsewhere in the pattern.

The `at` Clause.

An `at` clause is used to bind a local variable to the value of the `$NOW` variable when the match occurs. This variable can then be used in another clause, *e.g.* to assert some properties about the time elapsed between two events or in the body of the `whenever`.

Unlike in a `::atescofo` value clause, it is not possible to directly specify a value for the clause but this value can be tested in the clause:

```
@pattern_def pattern::S
{
    @local $s, $x, $y

    Event $S at $s where $s==5 ; cannot write directly: Event $S at $s
    Event $S at $x
    Event $S at $y where ($y - $x) < 2
}
```

Note that it is very unluckily that the matching time of a pattern is exactly “5”. Notice also that the date is expressed in absolute time.

The `where` Clause

As for patterns, a clause is used to constraint the parameters of an event (value and occurrence time). It can also be used to check any property that must hold at the time of matching. For example: in the clause:

```
@pattern_def pattern::S
{
```

²To achieve the same effect in a `Note` pattern, you need to use the `where` clause: a pattern variable is used to bind the pitchor the duration value and then the logical expression is checked in the `where` clause.

```

    Event $S where $ok
}

```

will match an update of `$S` only when `$ok` is true.

The **before** Clause

For a pattern q that follows a pattern p , the `before` clause can be used to relax the temporal scope on which q is looked for.

When *Antescofo* is waiting to match the pattern $q = \text{Event } \$X$, it starts to watch the variable right after the match of the previous pattern p . Then, at the **first value change** of $\$X$, *Antescofo* checks the various constraints on q . If the constraints are not met, the matching fails.

The `before` clause can be used to shrink or to extend the temporal interval on which the pattern is matched beyond the first value change. For instance, the pattern

```

@pattern_def pattern::twice
{
    @local $x
    Event $V value $x
    Before [3s] Event $V value $x
}

```

is looking for two updates of variable $\$V$ for the same value $\$x$ in less than 3 seconds. *Nota bene* that other updates may occur but $\$V$ must be updated for the same value before 3 seconds have elapsed for the pattern to match.

If we replace the temporal scope `[3s]` by a logical count `[3#]`, we are looking for an update for the same value that occurs in the next 3 updates of the watched variable. The temporal scope can also be specified in relative time `[3]`.

Notice that a `before` clause cannot be achieved using an `at` clause with a `where` clause; pattern *twice*

```

@pattern_def pattern::twice2[$x]
{
    @local $x, $s1 $s2
    Event $V value $x at $s1
    Event $V value $x at $s2 where ($s2 - $s1) <= 3
}

```

`pattern::twice2` does not match the same thing as `pattern::twice` because for `pattern::twice2` the two matched events are two *successive* updates of $\$V$.

When the temporal scope of a pattern is extended beyond the first value change, it is possible that several updates occurring within the temporal scope satisfy the various patterns' constraints³. *However*, the pattern matching stops looking for further occurrences in the same

³If there is no `before` clause, the temporal scope is “the first value change” which implies that there is at most one match.

temporal scope after having found the first one. This behavior is called the **single match** property.

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

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

Watching Multiple Variables Simultaneously

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

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

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

is a pattern looking for two successive updates of either $\$X$ or $\$Y$ in less than one second. Notice that when watching multiple variables, it is not possible to use a `value` clause.

A Complex Example

As mentioned, it is possible to freely mix and patterns, for example to watch some variables after the occurrence of a musical event:

```
@pattern_def pattern::T
{
    @local $d, $s1, $s2, $z

    Note D4 $d
    Before [2.5] Event $X, $Y at $s1
    Event $Z value $z at $s2 where ($z > $d) && $d > ($s2 - $s1)
}
```

Note that different variables are watched after the occurrence of a note D4 (6400 midicents). This pattern is waiting for an assignment to variable $\$X$ or $\$Y$ in an interval of 2.5 beats after a note D4, followed by a change in variable $\$Z$ for a value such that the duration of D4 is greater than the interval between the changes in $\$X$ or $\$Y$, and such that the value of $\$Z$ is also greater than this interval.

State Patterns

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

A Motivating Example

Suppose we want to trigger an action when a variable takes the value 0 for at least 2 beats. The following pattern:

```
Event $X value 0
```

does not work because the constraint “at least 2 beats” is not taken into account. The pattern matches every time `$X` takes the value 0.

The pattern sequence

```
@local $start, $stop
Event $X value 0 at $start
Event $X value 0 at $stop where ($stop - $start) >= 2
```

is no better: it matches two successive updates of `$X` that span over 2 seconds. It would not match three consecutive updates of for the same value 0, one at each beat, a configuration that should be recognized. In addition, converting the absolute duration into relative time is difficult because it would require tracking every tempo change in the interval.

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

```
State $X where $X == 0 during 2
```

matches an interval of 2 beats where the variable constantly has the value 0 (irrespectively of the variable updates).

Five optional clauses can be used to constrain a `state` pattern:

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

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

The first three clauses are similar to those described for an event pattern, except that the `at` is split into the `start` and the `stop` clauses because here the pattern is not instantaneous; it spans over an interval of time.

The initiation of a `state` Pattern

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

The `during` Clause

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

The figure below illustrates the behavior of the pattern

```
@Refractory r
State $X during d where $X > a
Before [s]
State $X where $X > b
```

The schema assumes that variable `$X` is sampling a continuous variation.

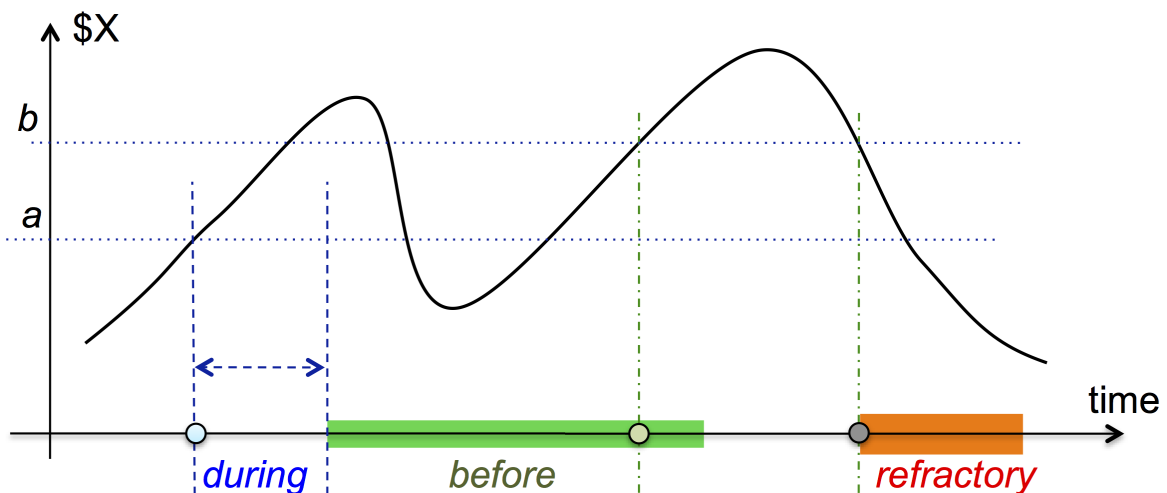


Figure 18.2: state pattern with `during`, `before` and `@refractory` clauses

The first pattern is looking for an interval of length `d` where `$X` is constantly greater than `a`.

The second pattern must start to match before s beats have elapsed since the end of the previous pattern (the allowed time zone is in green). The match starts as soon as it is greater than b .

The second pattern finishes its matching as soon as it becomes smaller than b because there is no specification of a duration.

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

Limiting the Number of Matches of a Pattern

The refractory period is defined for a pattern sequence, not for an atomic pattern. The clause must be specified at the beginning of the pattern sequence just before or after an eventual `@local` clause.

This clause specifies the period after a successful match (of the whole pattern) during which no other matches may occur. This period is given in absolute time and counted starting from the end of the successful match. The refractory period is represented in red in the above figure. The effect of a refractory period is to restrict the number of matching per time interval.

Pattern Compilation

Patterns are not a core feature of the language: internally they are compiled in a nest of `whenever`, conditionals and local variables. If verbosity is greater than zero, the `[printfwd]` command reveals the result of the pattern compilation in the printed score.

Two properties of the generated code must be kept in mind:

1. **Causality:** The pattern compiler assumes that the various constraints expressed in a pattern are free of side-effects and the pattern matching is achieved on-line, that is, sequentially in time and without assumptions about the future.
2. **Single match property:** When a pattern sequence occurs several times starting at the same time t , only one pattern occurrence is reported⁴.

Pattern semantics and pattern compilation are detailed in [Real-Time Matching of Antescofo Temporal Patterns](#).

⁴Alternative behaviors may be considered in the future.

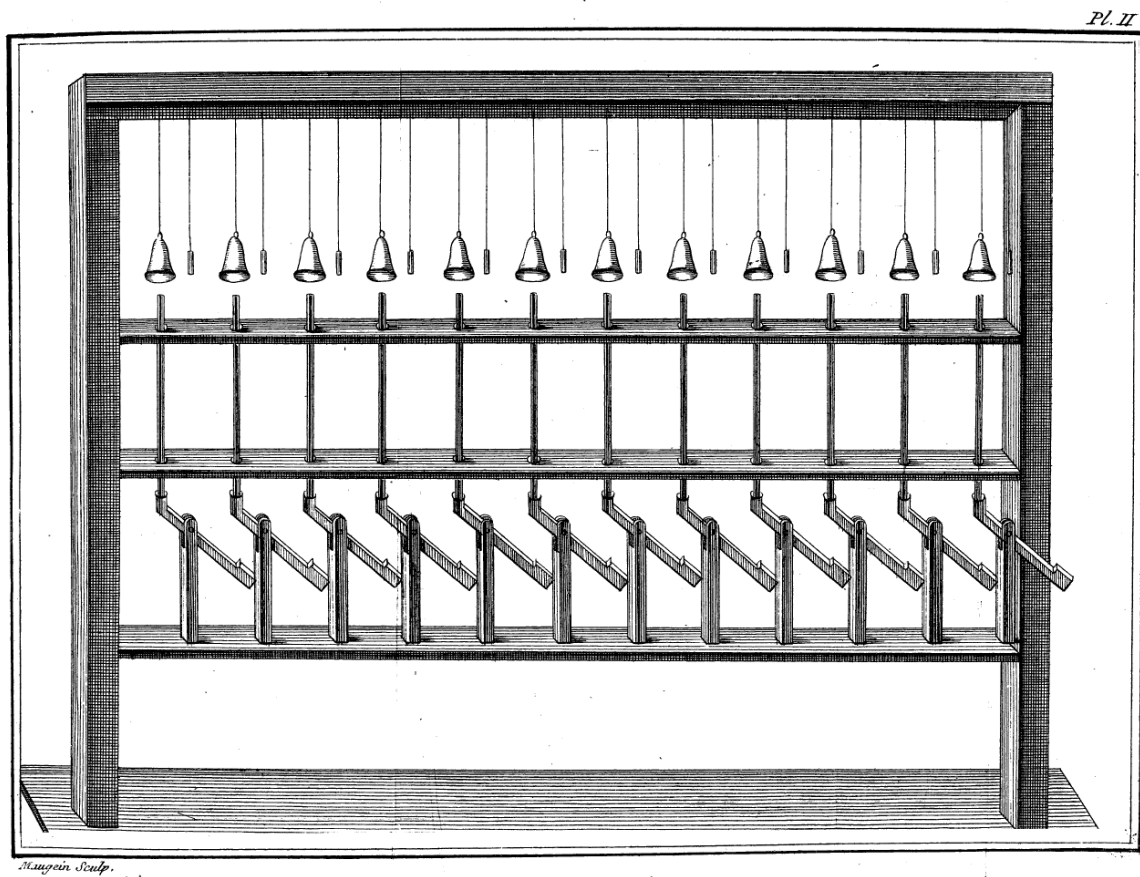


Figure 18.3: magnetic harpsichord

Chapter 19

Additional Elements

In this chapter, we present some additional elements of the *Antescofo* language that do not fit in the previous chapters:

- the [track mechanism](#) which allows the disabling of some messages during a program run;
- the [splitting](#) of a large score into several files;
- the [evaluation](#) that can be performed when a score is loaded.

Tracks

A *track* refers to all actions that have a label of some form and to the message whose head has some form. A track is defined using a statement:

```
@track_def track::T
{
    print, "synth.*"
}
```

refers to all actions that: (1) have a label or a label that matches (*i.e.* any name that starts with the prefix) *and* (2) all Max or PD messages whose receivers satisfy the same constraints *and* (3) the children of these actions (recursively).

More generally,

- a track definition is a list of tokens separated by a comma;
- a token is either a symbol (an identifier without double-quotes) or a string;
- a symbol refers to labels or receivers equal to this symbol;
- a string denotes a regular expressions¹ (without the double quote) used to match a label or a receiver name;

¹The syntax used to define the regular expression follows the posix extended syntax as defined in IEEE Std 1003.2, see for instance [regular expression on Wikipedia](#).

- an action belongs to a track if there is a symbol in the track equal to the label of the action or if there is a regular expression that matches the label;
- in addition, a Max or PD message belongs to the track if the receiver's name fulfills the same constraint;
- in addition, an action nested in a compound action belonging to the track also belongs to the track;
- an action may belong to several tracks (or none);
- there is a maximum of 32 definable tracks.

Tracks can be muted or unmuted:

```
antescofo::mute track::T
antescofo::unmute track::T
```

A string can be also used for the name of the track:

```
antescofo::mute "track::T"
antescofo::unmute "track::T"
```

which means that the track to mute or unmute can be computed dynamically:

```
$tracks := ["track::T", "track::Harmo", "track::Synthe", "track::Rever

whenever ($track_to_mute == $track_to_mute)
{
    antescofo::mute ($tracks[$track_to_mute])
}
```

Tracks are muted/unmuted independently. An action is muted if it belongs to a track that is muted, otherwise it is unmuted. A muted action has the same behavior as an unmuted action **except** for messages: their arguments are evaluated as usual but the final shipping to Max or PD is inhibited. It is important to note that muting/unmuting a track has no effect on the *Antescofo* internal computations, only in the sending of messages.

For example, to inhibit the sending of all messages, one can define the track:

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

and mute it:

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

File Structure of an Antescofo Score

Writing an augmented score through multiple files

A simple *Antescofo* augmented score is written in a single text file. However, for large scores, places where one wants to make underlying organization explicit, or places some procedures or score elements are reused in several pieces, an *Antescofo* augmented score may be specified in several scores.

There is always a *main* file from which the additional files will be loaded. The file loaded in Max/PD or used as the last argument in a standalone command task will be read as the *main* file.

From the *main*, other files are loaded using the `@insert` directive:

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

The `@insert` keyword can be capitalized (`@INSERT`) as can any other predefined `@`-keyword (`@local`, `@target`, *etc.*). Beware not to confuse the `@insert` directive used to include a text file at score loading time and the `[@insert]` function.

The `@insert` command is often used to store definitions and initializations of the main Antescofo score in alternate files. So several setups may coexist with the main score.

The `@insert_once` command is similar to `@insert` directive except that the file is included only once in the current score, when the directive is encountered the first time. This behavior makes possible to include a library of primitives in set of files without the burden to take care of the dependencies.

Load and Preload Command

A file is loaded through a `load` message sent to the Antescofo object (in Max or PD). The argument of this message is the path of the file to load. The effect of this command is to abort the current computations (if any) and to load the specified file which becomes the **current score**. The previous score, if any, is simply thrown away with all of its definitions: definitions are not shared between two score loads.

A file can also be preloaded using the command `preload file name`. The `file` argument is the path of a file while `name` is a simple identifier to refer to the score in future command.

Preloaded scores are used to defines functions, processes, actors, data structures: the definitions in a preloaded score are added to the currently known definitions. So they are in use for the next preloads and for the (final) load. A preload does not define a current score. So, the usual workflow is to preload a set of files with a final load.

Preloaded scores can also contain definitions of musical events to follow. During the program execution, one can switch from the current score to the score defined by a preloaded score using the `start name` command with `name` referring to a preloaded score. A `start` command without `name` simply starts the current score.

The `start name` command can be issued from the *Antescofo* program itself using the `antescofo::start` action. This will keep the program state (variables values, processes, *etc.*) but divert the listening machine to follow the new specified score.

Evaluation at Score Loading Time

When a score loads, some expressions can be evaluated and some actions can be performed. This makes it possible to pre-compute some data or to run some initialization before the real running of the program.

Constant Expressions

Constant expressions are expressions whose values do not depend on context and are independent of the date at which the expression is evaluated.

Determining if an expression is a constant expression is difficult. But *Antescofo* detects a large subset of constant expressions and evaluates them when the score is loaded. The idea is to speed up the actual program run as much as possible by doing some evaluations beforehand.

So for instance

```
let $x := 1 + @sin(3.1415)
```

is internally rewritten in

```
let $x := 1.00009
```

Expression with variables are not constant expression (even if there no assignment in scope, variables can be assigned externally using [setvar](#) and their value is always supposed unknown). The application of a user-defined functions is not a constant expression like the [impure predefined functions](#).

Constant expressions are detected in actions. However, it is also possible to write constant expressions in a BPM specification.

```
BPM (1.1*120)
```

This seems useless but it combines well with macro-definition:

```
@macro_def @BaseTempo { 120 }
; ...
BPM @BaseTempo
; ...
BPM (@BaseTempo + 10)
```

Which makes it possible to change the base tempo of a piece by changing only the macro-definition.

@eval_when_load Clause

A `@eval_when_load` clause specifies a list of *actions* that must be performed just after loading a file and before the run of the program. Several such clauses may exist in a file: they

are performed in the order of appearance right after having completed the parsing of the full score.

Such a clause can be used, for instance, to read some parameter saved in a file or to precompute some values. For example

```
@fun_def fib($x)
{
  if ($x < 2) { return 1 }
  else { return @fib($x-1) +@fib($x-2) }
}

@eval_when_load {
  $fib36 := @fib(36)
}

; ...

NOTE C4
  print $fib36
```

When this file is loaded, the clause is evaluated to compute `@fib(36)` which takes a noticeable amount of time because it uses a doubly recursive function. This value is then used when the program is started and the C4 event occurs, without requiring a costly computation. If not for evaluation at load time, the performance would be interrupted by complex computations like this one.

By using `[@insert]`, `[@insert_once]`, `[@eval_when_load]` and the *Antescofo* preload commands, together with functions `[@dumpvar]`, `[@loadvar]`, `[@loadvalue]` and `[@savevalue]`, one can manage a library of reusable functions and *reusable setups* mutualized between pieces.

Auto-Delimited Expressions

{!BNF_DIAGRAMS/closed_expr.html!}

Expressions appear everywhere to parameterize actions (and actions may appear in expressions, cf. [Action As Expression](#)). This may cause some syntax problems. For example, when writing:

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

there is a possible ambiguity: it can be interpreted as the message with two arguments (the function `@f` and the integer 1) or it can be the message with only one argument (the result of the application of function `@f` 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 there is no separator between the arguments of a messages. So we don't know where the expression starting by `@f` finishes.

The example here shows a more general problem which leads us to distinguish a subset of expressions: *auto-delimited expressions* are meaningful expressions that cannot be "extended" with what follows. Integers, for example, are auto-delimited expressions. We can write

```
print 1 (2)
```

without ambiguity: this is the message with two arguments because there is no other possible interpretation. Variables are another example of auto-delimited expressions.

Being auto-delimited is a sophisticated property involving the type of the actual value of the expressions. So, the *Antescofo* approach is to accept a simple syntactic subset of expressions to avoid possible ambiguities in the places where this is needed. This subset is defined by the syntax diagrams given above.

For example:

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

To disambiguate our first example, we can also use parentheses:

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

The second form is interpreted as only one argument, because a functional constant is useless as an argument in a message sent to the environment. But the third form shows how to force the alternative interpretation.

[Returns to chapter Expression](#)

Simple Expressions

Simple expressions are divided into several categories

Constant Values

Boolean Constants

{!BNF_DIAGRAMS/bool_expr.html!}

Numeric Constants

Integers, decimals and scientific notation can be used to define a numeric constant.

{!BNF_DIAGRAMS/constant_expr.html!}

Data Structure Definition

Tab Definition by Enumeration

{!BNF_DIAGRAMS/tabdef_expr.html!}

Tab Definition by Comprehension

{!BNF_DIAGRAMS/tabcomprehension_expr.html!}

Map Definition

{!BNF_DIAGRAMS/map_expr.html!}

Nim Definition

cexp means [closed expression](#)

Continuous NIM

{!BNF_DIAGRAMS/nim_expr.html!}

Discontinuous NIM

{!BNF_DIAGRAMS/dinim_expr.html!}

Tab Access

{!BNF_DIAGRAMS/tab_expr.html!}

Variables and Variables Management

{!BNF_DIAGRAMS/var_expr.html!}

Infix Unary Expressions

{!BNF_DIAGRAMS/unary_expr.html!}

Infix Binary Expressions

Arithmetic, relational and logical binary operators.

{!BNF_DIAGRAMS/binary_expr.html!}

Conditional

{!BNF_DIAGRAMS/conditional_expr.html!}

Infix Predicates

{!BNF_DIAGRAMS/predicate_expr.html!}

Function Application and Process Call

{!BNF_DIAGRAMS/apply_expr.html!}

Action As Expression

{!BNF_DIAGRAMS/action_expr.html!}

[Returns to chapter Expression](#)

Macro *vs.* Function *vs.* Process

How do you chose between macros, functions and processes?

These three mechanisms can be used to abstract a code fragment and to parameterize it by argument provided when the code fragment is reused (at a call point). However, these three mechanisms don't have the same benefits, flexibilities or shortcomings.

The following figure compares the three mechanisms from several points of view (click for access the table as a PDF document). See also the paragraph [What to Choose Between Macro, Functions and Processes](#) for additional advice.

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

Argument evaluation strategies

Programming languages through the years have introduced several strategies to determine when and how to evaluate the argument of a routine call (by a *routine*, we mean a sequence

of code that can be called with [parameters](#) like functions, procedures, methods, processes, coroutines, *etc.*). When a routine is called, its code is run with the parameters substituted by the actual arguments provided during the call. What differs is the exact nature of the substitution.

Call-by-value replaces the parameters in the routine body with the *value* of the arguments. In other words, the expressions in arguments are evaluated and this value is bound to the routine's parameters seen as local variables of the routine. Thus, the callee cannot modify a value referred to by the caller through its argument. This strategy is the most common strategy, used in C, Scheme, *etc.*

Call-by-reference replaces the parameters by a *reference* to the value of the arguments. The reference is handled transparently in the routine body, where parameters are used like local variables. This strategy allows a routine to alter a value referred by the caller. This strategy is available in Pascal, Fortran, C++ (using a reference for the type of the parameter), *etc.*

Call-by-name replaces the parameters in the routine body with the expressions given as actual arguments. This strategy is not very common in programming languages (Algol60 introduced it), but corresponds to the macro mechanism, with the exception of the time of evaluation (calling-by-name is interleaved with evaluation, while macros are textual substitutions done before any evaluation, which may lead to differences when a routine definition can be the product of the evaluation).

This categorization is blurred by the nature of the programming languages: declarative (*e.g.* purely functional) or imperative, the type system (if any; consider the C++ approach where arguments are passed by values, but a reference type exists), the representation of values (if all values are “boxed” and implemented as pointers to boxes, the call by value cannot be distinguished from the call by reference), and additional mechanisms (like [lazy evaluation](#), [memoization](#) or [futures and promises](#)), *etc.*

Antescofo Evaluation Strategy

Antescofo implements call-by-value for functions, process and objects; and *Antescofo* macros implement textual substitution, which loosely corresponds to call-by-name.

However, *nota bene* that *Antescofo* [data structures](#), *i.e.* tab, map, nim and string, are implemented through a reference to an underlying memory area. So, for instance, a tab may be shared between the caller and the callee even if the call is by value. This ends up with a behavior like call-by-reference for data structures. This behavior “call-by-value where the value is a reference” is common: this is for instance the behavior in C, C++ or Java.

The following example illustrates this point. Consider the following process and variable definitions:

```
@proc_def ::P($a, $b)
{
    let $b := $b + 1
    let $a[0] := $b
}

$t := [3, 1, 2]
$x := 10
```

the corresponding data layout:

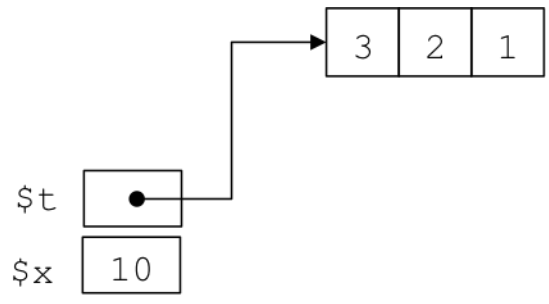


Figure 19.1: data layout

Then, the call

```

::P($t, $x)
print $t
print $x
    
```

will produce:

```

[11, 2, 1]
10
    
```

The evaluation is sketched in the following diagram

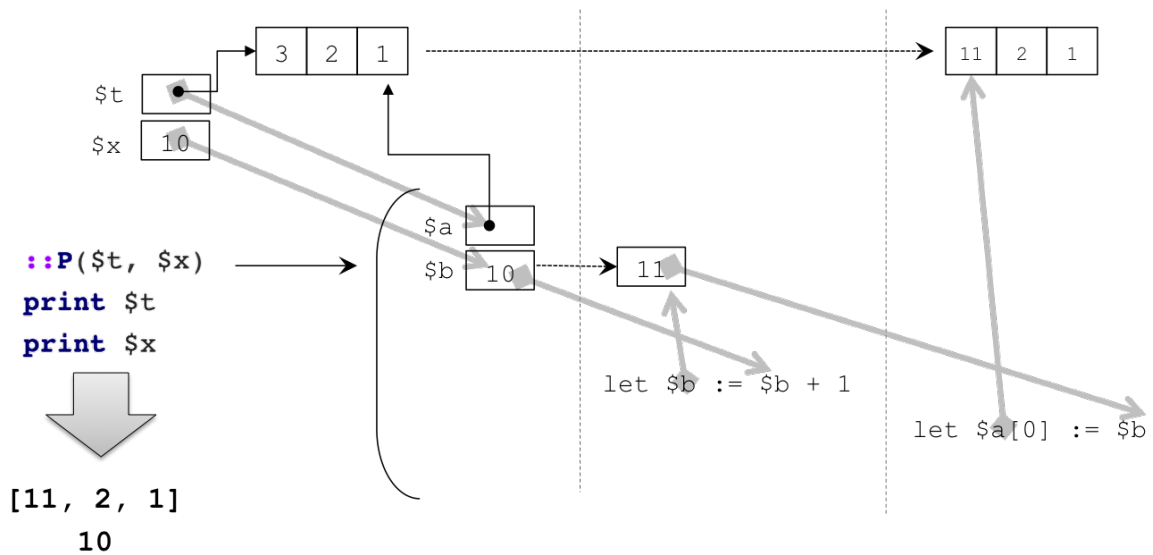


Figure 19.2: call by value 1

The value of `$t` is copied into the parameter `$a` but this value is a reference to an underlying memory zone where the tab is actually stored. So, when the first element of the tab is updated in the process, this is visible in the value referenced by `$t`. On the other hand, `$x` still refers

to the value 10 because this value is fully copied into parameter `$b` and a subsequent change does not affect the value referred by `$x`.

Consider now the call:

```
::P($t + [10, 20, 30], $x+1)
print $t
print $x
```

it will produce:

```
[3, 2, 1]
10
```

The values referred to by `$t` and `$x` are untouched by the evaluation of `::P` which is sketched in the following diagram:

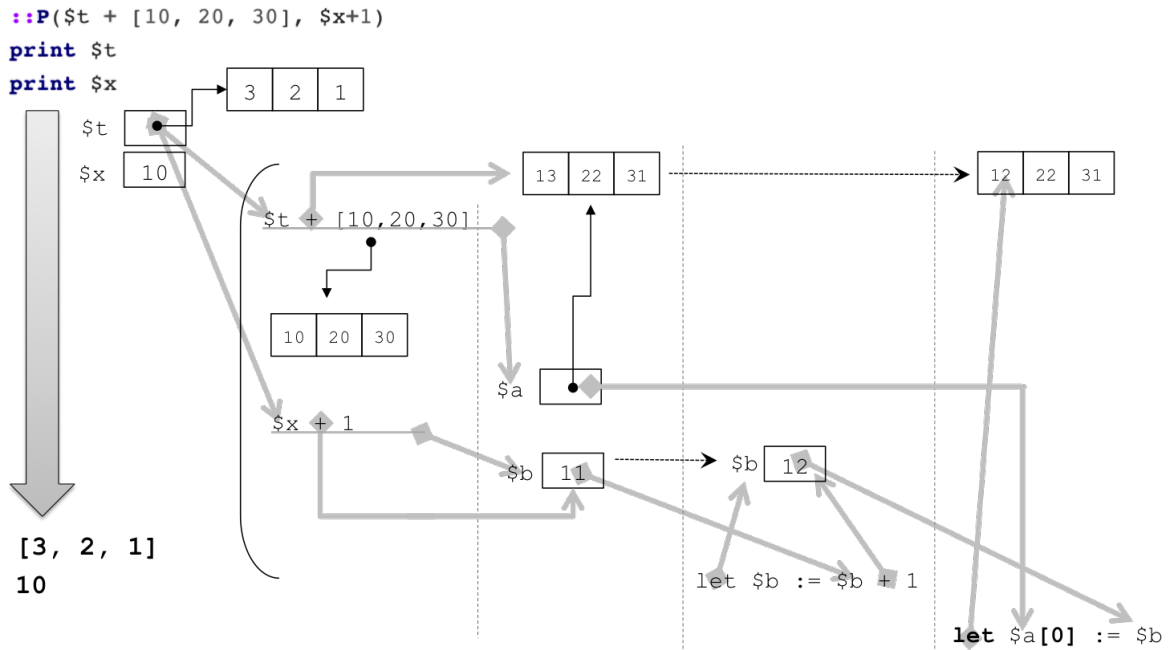


Figure 19.3: call by value 2

This time, the values in the caller are unaffected by the execution of `::P`. As a matter of fact, expression `$t + [10, 20, 30]` creates a new tab and it is this new tab that is mutated by the element assignment in the process. The consequence is that the value referred by `$t` remains unaltered.

As a final example, examine the following expression:

```
::P(@sort($t), $x)
print $t
print $x
```

it will produce:

```
[11, 2, 3]
10
```

This time, the argument `@sort($t)` modifies the tab referred by `$t` in place and returns its argument. So the tab processed by `::P` is also the tab referred by `$t`.

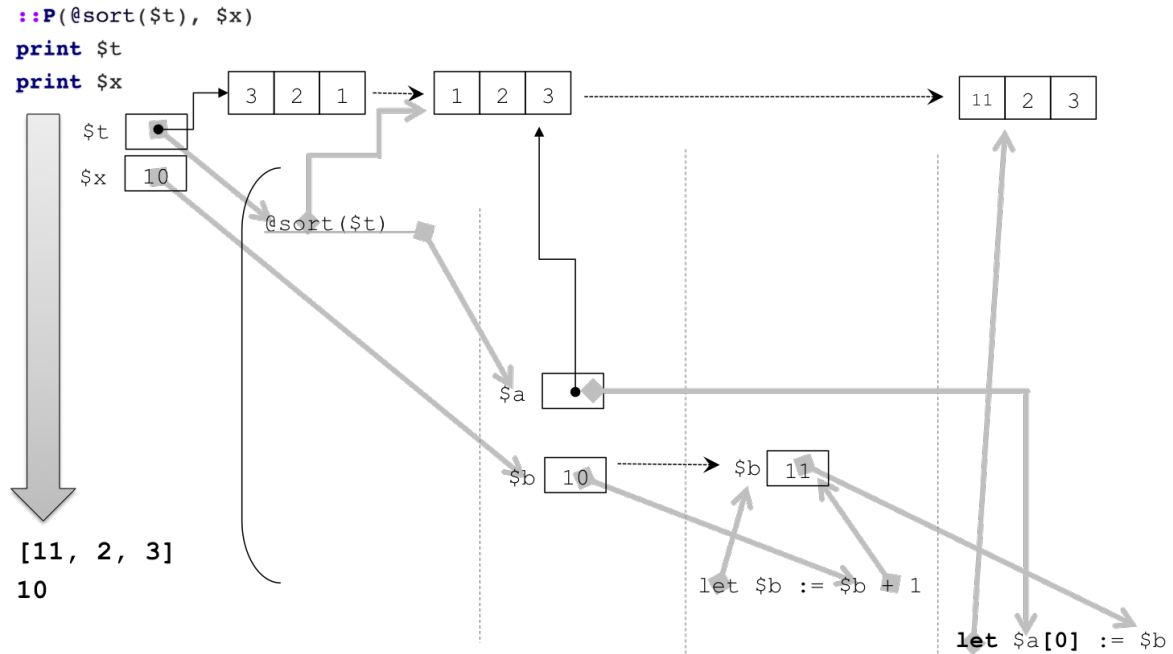


Figure 19.4: call by value 3

Notice that the previous examples involve processes but the same behavior is achieved with functions. Nota bene that scalars are never modified through a function or a process call.

Grammar of object definitions

{!BNF_DIAGRAMS/object_def.html!}

Antescofo Workflow

Antescofo programs strongly interact with the environment through the listening machine, but also through messages send through the Max/PD patch. These interactions happen during the performance, but also at previous phases of the *Antescofo* workflow, when editing or debugging the score and during rehearsals. Messages understood by *Antescofo* are described in section [internal commands](#). The User's guide also contains three useful sections on:

- [Editing the score](#)

- [Interacting with MAX/PureData](#)
- and [Preparing the Performance, Rehearsals.](#)

Chapter 20

Acknowledgements and credits

The *Antescofo* documentation was written by [Jean-Louis Giavitto](#) with the help of [Arshia Cont](#), [Julia Blondeau](#) and [José Echeveste](#). Sam Wiseman revised the first version of the manual during an internship.

Antescofo was born out of a collaboration between a researcher (Arshia Cont), a composer (Marco Stroppa), and a saxophonist (Claude Delangle) for the world premier of *... of Silence* in late 2007. *Antescofo* is particularly grateful to composer Marco Stroppa, the main motivation behind its existence and his continuous and generous intellectual support. Since 2007, many composers and computer musicians have joined the active camp to whom *Antescofo* is always grateful: Pierre Boulez, Philippe Manoury, Gilbert Nouno, Serge Lemouton, Larry Nelson, José Miguel Fernandez, Julia Blondeau, Yann Marez, Jason Freeman, Christopher Trapani and others...

Antescofo is also in debt to and heartily acknowledges the patience of RIMs at IRCAM and elsewhere that have been in the front line in the use of *Antescofo* in actual performances: Greg Beller, José Echeveste, José Miguel Fernandez, Thomas Goepfer, Carlo Laurenzi, Serge Lemouton, Grégoire Lorieux, Augustin Muller, Gilbert Nouno and others...

Over the years, the *Antescofo* system has been developed by Arshia Cont, Philippe Cuvillier, José Echeveste and Jean-Louis Giavitto. Additional help on Ascograph by Grig Burloiu, Thomas Coffy and Robert Piechaud.

The development of *Antescofo* has been made possible by support from [Ircam](#), [CNRS umr STMS 9912](#), [Inria project MuTant](#), [University of Paris 6](#) Pierre et Marie Curie, and [ANR project INEDIT](#).

abort action antescofo argument assignment attributes audio body breakpoints called case clause command compound computation condition current curve date def definition delay duration elements end evaluated event example expression figure file following form function given group history icmd identifier instance integer interpolation iteration label launched list local loop macro map max message name nim note number numeric object operator order pattern performed position predicate print process ref refers result returns score sec size specification specified

start string

synchronization systems target tempo temporal tight

trace true type updated used value var variable whenever



Figure 20.1: Final image: experimental music

Chapter 21

Index

A

[@abort]

 [@abs]

 [@acos]

 [@active]

 [@add_pair]

 [@aggregate]

 [@align_breakpoints]

 [@ante]

 [@approx]

 [@arch_darwin]

 [@arch_linux]

 [@arch_windows]

 [@asin]

 [@atan]

[a strongly timed language](#)

[abort](#)

[abort handler](#)

[action](#)

[action as expression](#)

[action label](#)

[action priority](#)

[action specification](#)

[actor](#)

[alive](#)

[antescofo cookbook](#)

antescofo distribution
antescofo workflow
antescofo::actions
antescofo::add_completion_string
antescofo::analysis
antescofo::asco_trace
antescofo::ascographheight_set
antescofo::ascographwidth_set
antescofo::ascographxy_set
antescofo::before_nextlabel
antescofo::bpmtolerance
antescofo::calibrate
antescofo::clear
antescofo::decodewindow
antescofo::filewatchset
antescofo::gamma
antescofo::get_current_score
antescofo::get_patch_receivers
antescofo::getlabels
antescofo::gotobeat
antescofo::gotolabel
antescofo::harmlist
antescofo::info
antescofo::killall
antescofo::mode
antescofo::mute
antescofo::nextaction
antescofo::nextevent
antescofo::nextlabel
antescofo::nextlabeltempo
antescofo::nofharm
antescofo::normin
antescofo::obsexp
antescofo::pedal
antescofo::pedalcoeff
antescofo::pedaltime
antescofo::piano
antescofo::play

antescofo::playfrombeat
antescofo::playfromlabel
antescofo::playstring
antescofo::playstring_append
antescofo::playtobeat
antescofo::playtolabel
antescofo::preload
antescofo::preventzigzag
antescofo::previousevent
antescofo::previouslabel
antescofo::printfwd
antescofo::printscore
antescofo::read
antescofo::report
antescofo::score
antescofo::scrubtobeat
antescofo::scrubtolabel
antescofo::setvar
antescofo::start
antescofo::startfrombeat
antescofo::startfromlabel
antescofo::static_analysis
antescofo::stop
antescofo::suivi
antescofo::tempo
antescofo::tempoinit
antescofo::temposmoothness
antescofo::tune
antescofo::unmute
antescofo::variance
antescofo::verbosity
antescofo::version
argument evaluation strategies
argument passing strategies
articulating time
ascograph
assignment
atomic action

- [atomicActionInExpression](#)
- [attribute](#)
- [auto delimited expressions](#)
- [auto-delimited expression](#)

B

- [\[@between\]](#)
 - [\[@bounded_integrate\]](#)
 - [\[@bounded_integrate_inv\]](#)
 - [\[@broadcast\]](#)
 - [boolean](#)

C

- [\[@car\]](#)
 - [\[@cdr\]](#)
 - [\[@ceil\]](#)
 - [\[@clear\]](#)
 - [\[@concat\]](#)
 - [\[@cons\]](#)
 - [\[@conservative\]](#)
 - [\[@copy\]](#)
 - [\[@cos\]](#)
 - [\[@cosh\]](#)
 - [\[@count\]](#)
 - [chuck](#)
 - [closed expression](#)
 - [commands](#)
 - [compound action](#)
 - [conditional action](#)
 - [conditional expression](#)
 - [constant bpm expression](#)
 - [continuation](#)
 - [continuation operator](#)
 - [coroutine](#)
 - [\[curryfied functions\]](#)
 - [curve](#)

curve interpolation methods
curve playing a nim

D

[@dim]

 [@div]

 [@domain]

 [@drop]

 [@dump]

 [@dumpvar]

data structures

dead

dealing with errors

delay

dot notation

during

E

[@empty]

 [@exclusive]

 [@exp]

 [@explode]

end clause

error

error handling strategy

error strategies

eval when load

evaluation

evaluation at load time

event specification

events

exe

exec

exec value

expression

extended expressions

[extensional function](#)
[external assignment](#)

F

[\[@filter_max_t\]](#)
[\[@filter_median_t\]](#)
[\[@filter_min_t\]](#)
[\[@find\]](#)
[\[@flatten\]](#)
[\[@floor\]](#)
[\[@fun_def\]](#)
[files layout](#)
[\[fig:Excerpt1\]](#)
[\[fig:Tesla\]](#)
[ForAll](#)
[ForumUser](#)
[function](#)
[function as value](#)
[functions library](#)

G

[\[@global\]](#)
[\[@gnuplot\]](#)
[\[@gshift_map\]](#)
[grammar of object definition](#)
[Group](#)

H

[\[@history_length\]](#)
[history of a variable](#)

I

[\[@immediate\]](#)
[\[@init\]](#)

[\[@insert\]](#)
[\[@integrate\]](#)
[\[@iota\]](#)
[\[@is_bool\]](#)
[\[@is_defined\]](#)
[\[@is_exec\]](#)
[\[@is_fct\]](#)
[\[@is_float\]](#)
[\[@is_function\]](#)
[\[@is_int\]](#)
[\[@is_integer_indexed\]](#)
[\[@is_interpolatedmap\]](#)
[\[@is_list\]](#)
[\[@is_map\]](#)
[\[@is_nim\]](#)
[\[@is_numeric\]](#)
[\[@is_obj\]](#)
[\[@is_obj_xxx\]](#)
[\[@is_prefix\]](#)
[\[@is_proc\]](#)
[\[@is_string\]](#)
[\[@is_subsequence\]](#)
[\[@is_suffix\]](#)
[\[@is_symbol\]](#)
[\[@is_tab\]](#)
[\[@is_undef\]](#)
[\[@is_vector\]](#)

[if](#)

[impure predefined functions](#)

[infix Notation for Function Calls](#)

[int](#)

[intentional function](#)

[internal command](#)

[internal commands](#)

J

[jump](#)

K**L**

[@lace]

 [@last]

 [@latency]

 [@linearize]

 [@listify]

 [@loadvalue]

 [@loadvar]

 [@log]

 [@log10]

 [@log2]

 [@local]

 [@loose]

 lexical elements

 library Functions

 library

 local tempo

 logical instant

 loop

M

[@macro_def]

 [@make_bpm_map]

 [@make_bpm_tab]

 [@make_duration_map]

 [@make_duration_tab]

 [@make_label_bpm]

 [@make_label_duration]

 [@make_label_pitches]

 [@make_label_pos]

 [@make_pitch_tab]

 [@make_score_map]

 [@map]

 [@map_compose]

[\[@map_concat\]](#)
[\[@map_history\]](#)
[\[@map_history_date\]](#)
[\[@map_history_rdate\]](#)
[\[@map_normalize\]](#)
[\[@map_reverse\]](#)
[\[@mapval\]](#)
[\[@max\]](#)
[\[@max_key\]](#)
[\[@max_val\]](#)
[\[@median\]](#)
[\[@member\]](#)
[\[@merge\]](#)
[\[@midi_getChannel\]](#)
[\[@midi_getCommand\]](#)
[\[@midi_getCommandByte\]](#)
[\[@midi_getMetaType\]](#)
[\[@midi_isAftertouch\]](#)
[\[@midi_isController\]](#)
[\[@midi_isEndOfTrack\]](#)
[\[@midi_isMeta\]](#)
[\[@midi_isNote\]](#)
[\[@midi_isNoteOff\]](#)
[\[@midi_isNoteOn\]](#)
[\[@midi_isPatchChange\]](#)
[\[@midi_isPitchbend\]](#)
[\[@midi_isPressure\]](#)
[\[@midi_isTempo\]](#)
[\[@midi_read\]](#)
[\[@midi_track2ascii\]](#)
[\[@min\]](#)
[\[@min_key\]](#)
[\[@min_val\]](#)

[macro versus Function versus Process](#)

[macro, Function and Processus](#)

[macro](#)

[management of Time](#)

[map](#)

- [max](#)
- [message](#)
- [methods](#)
- [mutating a tab element](#)

N

- [\[@norec\]](#)
 - [\[@normalize\]](#)
 - [\[@number_active\]](#)
- [Nim](#)

O

- [\[@obj_def\]](#)
 - [\[@occurs\]](#)
- [obj](#)
- [objects](#)
- [object instantiation](#)
- [open Scores and Dynamic jumps](#)
- [OSC message](#)
- [OSC messages](#)
- [OSC protocol](#)
- [OSCRECEIVE](#)

P

- [\[@pattern_def\]](#)
 - [\[@permute\]](#)
 - [\[@plot\]](#)
 - [\[@post\]](#)
 - [\[@pow\]](#)
 - [\[@proc_def\]](#)
 - [\[@progressive\]](#)
 - [\[@projection\]](#)
 - [\[@push_back\]](#)
 - [\[@push_front\]](#)
- [pattern](#)

- [patterns](#)
- [pd](#)
- [priority](#)
- [proc](#)
- [proc value](#)
- [proc values](#)
- [process call](#)
- [process](#)
- [processes](#)
- [processus](#)
- [procVariable](#)
- [program Structure](#)

Q

R

[@rand]

[@rand_int]

[@random]

[@range]

[@reduce]

[@remove]

[@remove_duplicate]

[@replace]

[@reshape]

[@resize]

[@reverse]

[@rnd_bernoulli]

[@rnd_binomial]

[@rnd_exponential]

[@rnd_gamma]

[@rnd_geometric]

[@rnd_normal]

[@rnd_uniform_float]

[@rnd_uniform_int]

[@rotate]

[@round]

[\[@rplot\]](#)
[Reference Manual](#)
[reserved @-identifier](#)
[reserved keywords](#)
[return](#)

S

[\[@sample\]](#)
[\[@savevalue\]](#)
[\[@scan\]](#)
[scoped variable](#)
[\[@score_tempi\]](#)
[\[@scramble\]](#)
[\[@select_map\]](#)
[\[@set_osc_handling_tab\]](#)
[\[@shape\]](#)
[\[@shift_map\]](#)
[\[@simplify_lang_v\]](#)
[\[@simplify_radial_distance_t\]](#)
[\[@simplify_radial_distance_v\]](#)
[\[@sin\]](#)
[\[@sinh\]](#)
[\[@size\]](#)
[\[@slice\]](#)
[\[@sort\]](#)
[\[@sputter\]](#)
[\[@sqrt\]](#)
[\[@string2fun\]](#)
[\[@string2obj\]](#)
[\[@string2proc\]](#)
[\[@stutter\]](#)
[\[@sync\]](#)
[\[@system\]](#)
[scalar values](#)
[Scheduling priorities](#)
[score import](#)
[setvar](#)

- side effect
- simple expressions
- splitting
- string
- superdense time
- switch action
- Switch
- synchronization
- synchrony hypothesis
- Synchronization Strategies
- system variables

T

- [@tab_history]
- [@tab_history_date]
- [@tab_history_rdate]
- [@take]
- [@tan]
- [@target]
- [@tempo]
- [@tempovar]
- [@tight]
- [@Tracing]
- [@track_def]
- Tab
- Temporal Pattern
- Temporal Patterns
- temporal scope
- temporal variable
- tempovar
- the fabric of time
- the manufacturing of time
- three kinds of expressions
- Tracing
- Tracks

U[\[@UnTracing\]](#)[undef](#)[until](#)[user Guide](#)**V**[value](#)[variable](#)[variable declaration](#)[variables and notifications](#)[vectorial curve](#)**W**[\[@whenever\]](#)[\[@window_filter_t\]](#)[what to Choose Between Macro, Functions and Processes](#)[Whenever](#)[while](#)**X****Y****Z****Miscellaneous**[@!=](#)[@%](#)[@&&](#)[\[@*\]](#)[@+](#)[@-](#)[@<](#)[@<=](#)

@==

@>

@>=

@||

@/

Antescofo Library of Predefined Functions

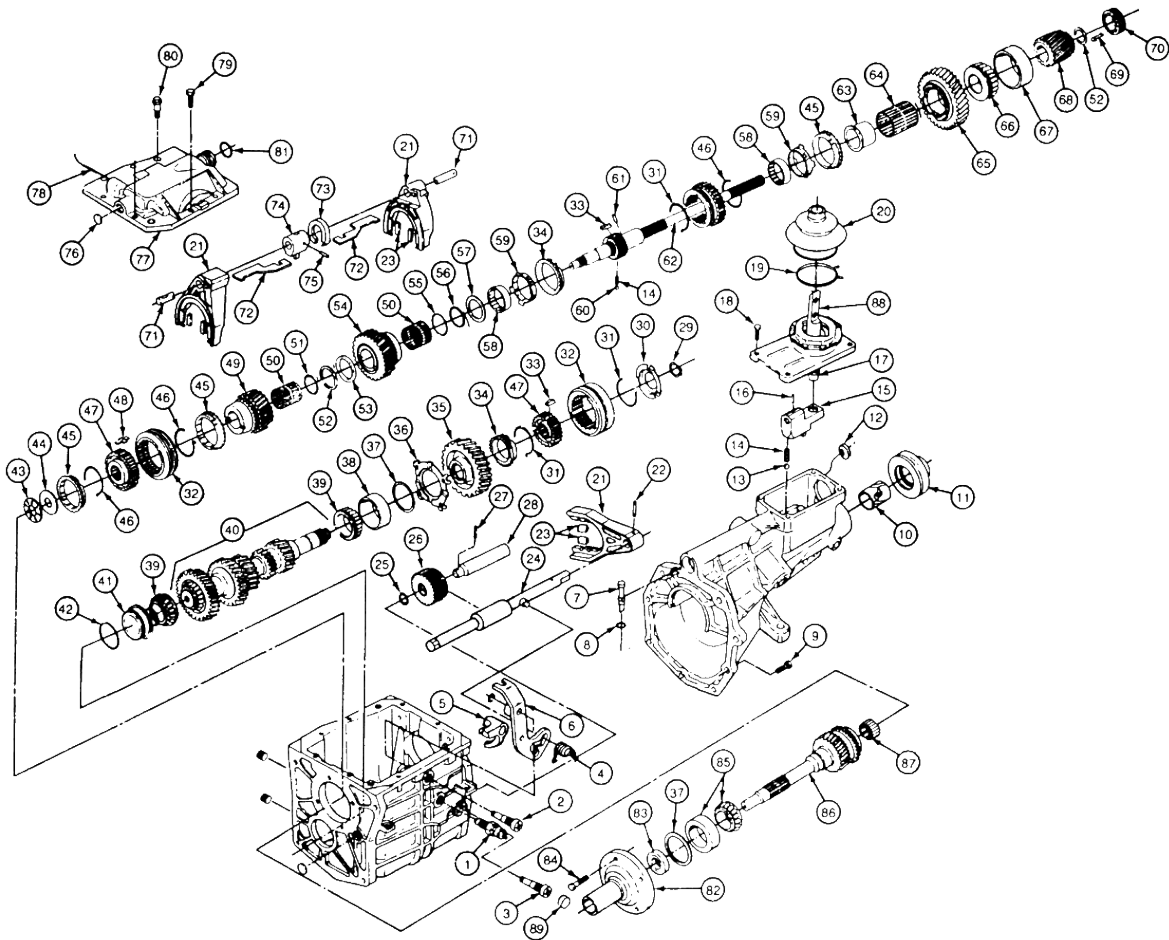


Figure 21.1: header figure

Antescofo includes a set of predefined functions. They are described mostly in the chapters [Expressions](#), [Scalar values](#) and [Data structures](#). For the reader's convenience, we give here a list of these functions.

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

Listable Functions and Listable Predicates

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

For example, `@abs` is a **listable function** on `numeric`, so it can be applied to a tab of numerics. The result is the tab of the absolute value of the elements of the tab argument.

Another example: the function `@approx` is a **listable predicate** and `@approx(u, v)` returns true if `@approx(u[i], v[i])` returns true for all elements `i` of the tabs `u` and `v`.

Overloaded functions

Some functions accept different kinds of values for the same argument. For example `[@insert]` accepts `tab` or `map` as its first argument. Such functions are said **overloaded**: they gather under the same name several specialized version of the function.

A user-defined function can be overloaded: it requires to check the type of the argument value to dispatch to the specialized versions (written elsewhere).

Overloading methods is more simple: the same identifier can be used to name a method in different objects.

Side-Effect

Most functions are **pure functions**, that is, they do not modify their arguments and build a new value for the result.

In some cases, the function works by a side-effect, like `@push_back` which modifies its argument *in place*. Such functions are marked as **impure**. We also qualify functions that may return different values when called with the same arguments as impure, even if they do not produce a side-effect (for example, functions that return a random number).

Special forms

Special forms are syntactic constructs similar to function calls but that are subject to some restriction or that behave differently of a function call.

For example, boolean predicates are a special form of function application because they do not always evaluate all their arguments (they are *lazy*). For instance, the conjunction `&&` evaluates its second argument only if the first is not `false`.

Another example is `[@plot]`, `[@history_length]` or `[@dumpvar]` that only accepts a variable as an argument, not a general expression.

Special forms cannot be curried nor passed as an argument: they are not ordinary functional values. Note, however that they can be wrapped in a user-defined function which is an ordinary functional value.

Function call in infix form

A function call is usually written in **prefix form**:

```
@drop($t, 1)
@scramble($t)
```

It is possible to write function calls in **infix form**, as follows:

```
$t.@drop(1)
$t.@scramble()
```

The @ character is optional when naming a function in infix call, so we can also write:

```
$t.drop(1)
$t.scramble()
```

This syntax is the same as for a *method call*. The general form is:

```
arg1 . @fct (arg2, arg3, ...)    ; or more simply
arg1 . fct (arg2, arg3, ...)
```

The arg_i are expressions. Notice that the infix call, with or without the @ in the function name, is not ambiguous with the notation used to refer to a variable local $\$x$ in a compound action from the execution of this action, $exe.\$x$, because $\$x$ cannot be the name of a function.

The infix notation is less general than the prefix notation. In the prefix notation, the function can be given by an expression. For example, functions can be stored into an array and then called following the result of an expression:

```
$t := [@f, @g]
; ...
($t[exp])()
```

will call @f or @g following the value returned by the evaluation of exp . Only function name (with or without @) are accepted in the infix notation.

Listing by categories

```
{!Library/Functions/math_functions.list!}
```

```
{!Library/Functions/random_functions.list!}
```

```
{!Library/Functions/tab_functions.list!}
```

in addition, see *listable* functions.

```
{!Library/Functions/listable_functions.list!}
```

```
{!Library/Functions/nim_functions.list!}
```

```
{!Library/Functions/map_functions.list!}
```

```
{!Library/Functions/string_functions.list!}
```

```
{!Library/Functions/predicates_functions.list!}
```

```
{!Library/Functions/score_functions.list!}
```

```
{!Library/Functions/midi_functions.list!}
```

```
{!Library/Functions/system_functions.list!}
```

Alphabetical Listing of *Antescofo* Predefined Functions

```
{!Library/Functions/functions.list!}
```

```
@!=(value, value) ; listable
```

Prefix form of the infix relational operator. Same remarks as for `@<`.

See also `@==`, `@>`, `@>=`, `@<`, `@<=`

```
@==(value, value) ; listable
```

prefix form of the infix relational operator. Same remarks as for `@<`. So, beware that

```
@==(1, 1.0)
```

which is equivalent to `1 == 1.0` evaluates to `true`.

See also `@!=`, `@>`, `@>=`, `@<`, `@<=`

```
@<(value, value) ; listable
```

Prefix form of the infix relational operator `<`.

This is a **total order**: values of different type can be compared and the order between unrelated type is *ad hoc*. Notice however that coercion between numeric applies if needed. So

```
0 < 0.0
```

returns `false`

See also `@==`, `@!=`, `@>`, `@>=`, `@<=`

```
@<=(value, value) ; listable
```

Prefix form of the infix relational operator `<=`. Same remarks as for `@<`.

See also `@==`, `@!=`, `@>`, `@>=`, `@<`

`@>=(value, value) ; listable`

Prefix form of the infix relational operator `>=`. Same remarks as for `@<`.

See also `@==`, `@!=`, `@>`, `@<`, `@<=`

`@>=(value, value) ; listable`

Prefix form of the infix relational operator `>=`. Same remarks as for `@<`.

See also `@==`, `@!=`, `@>`, `@<`, `@<=`

`@||(value, value) ; listable`

functional prefix form of the infix logical disjunction. Contrary to the `||` operator, the functional form is *not lazy*, cf. sect. [lazy logical operator]: so `@||(a, b)` evaluates `b` irrespectively of the value of `a`.

See also `@&&`.

`@&&(value, value) ; listable`

functional prefix form of the infix logical conjunction. Contrary to the `&&` operator, the functional form is *not lazy*, cf. sect. [lazy logical operator]: so `@&&(a, b)` evaluates `b` irrespectively of the value of `a`.

See also `@||`.

`@+(value, value) ; listable`

`@+` is the prefix form of the infix binary operator `+`

The functional form of the operator is useful as an argument of a high-order function as in `@reduce(@+, v)` which sums up all the elements of the tab `v`.

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

The addition of two `string` corresponds to the concatenation of the arguments.

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

See also `@-`, `@*`, `@/`, `@%`

`@-(numeric, numeric) ; listable`

prefix form of the infix arithmetic operator `-` (substraction). Coercions between numeric apply when needed.

See also `@+`, `@*`, `@/`, `@%`

@%(numeric, numeric) ; listable

prefix form of the infix binary operator %. Coercions between numerics apply when needed.

See also @+, @-, [@*], @/

@!=(value, value) ; listable

Prefix form of the infix relational operator. Same remarks as for @<.

See also @==, @>, @>=, @<, @<=

@%(numeric, numeric) ; listable

prefix form of the infix binary operator %. Coercions between numerics apply when needed.

See also @+, @-, [@*], @/

@&&(value, value) ; listable

functional prefix form of the infix logical conjunction. Contrary to the && operator, the functional form is *not lazy*, cf. sect. [lazy logical operator]: so @&&(a, b) evaluates b irrespectively of the value of a.

See also @||.

@*(numeric, numeric) ; listable

prefix form of the infix arithmetic operator * (multiplication). Coercions between numeric apply when needed.

See also [@+-], [@-*], @/, @%

@+(value, value) ; listable

@+ is the prefix form of the infix binary operator +

The functional form of the operator is useful as an argument of a high-order function as in @reduce(@+, v) which sums up all the elements of the tab v.

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

The addition of two string corresponds to the concatenation of the arguments.

The addition of a string and any other value convert this value into its string representation before the concatenation.

See also @-, [@*], @/, @%

@-(numeric, numeric) ; listable

prefix form of the infix arithmetic operator - (substraction). Coercions between numeric apply when needed.

See also @+, [@*], @/, @%

@<(value, value) ; listable

Prefix form of the infix relational operator <.

This is a **total order**: values of different type can be compared and the order between unrelated type is *ad hoc*. Notice however that coercion between numeric applies if needed. So

```
0 < 0.0
```

returns false

See also @==, @!=, @>, @>=, @<=

@<=(value, value) ; listable

Prefix form of the infix relational operator <=. Same remarks as for @<.

See also @==, @!=, @>, @>=, @<

(,), : prefix form of the infix relational operator. Same remarks as for . So, beware that evaluates to .

@==(value, value) ; listable

prefix form of the infix relational operator. Same remarks as for @<. So, beware that

```
@==(1, 1.0)
```

which is equivalent to 1 == 1.0 evaluates to true.

See also @!=, @>, @>=, @<, @<=

@>(value, value) ; listable

Prefix form of the infix relational operator >. Same remarks as for @<.

See also @==, @!=, @>=, @<, @<=

@>=(value, value) ; listable

Prefix form of the infix relational operator >=. Same remarks as for @<.

See also @==, @!=, @>, @<, @<=

@/(numeric, numeric) ; listable

prefix form of the infix arithmetic operator /. Coercions between numeric apply when needed.

See also @+, @-, [@*], @%

@||(value, value) ; listable

functional prefix form of the infix logical disjunction. Contrary to the `||` operator, the functional form is *not lazy*, cf. sect. [lazy logical operator]: so `@|| (a, b)` evaluates `b` irrespectively of the value of `a`.

See also [@&&](#).

`@/(numeric, numeric) ; listable`

prefix form of the infix arithmetic operator `/`. Coercions between numeric apply when needed.

See also [@+](#), [@-](#), [\[@*\]](#), [@%](#)

`@abs(numeric) ; listable`

absolute value

See also `{!Library/Functions/math_functions.list!}`

`@acos(numeric) ; listable`

arc cosine

See also `{!Library/Functions/math_functions.list!}`

`@active(string)`

`@active(proc)`

`@active()`

returns in a `tab` the active (alive) `exe` of the instances of a process or an object specified through its name (a string with or without the prefix `::` or `obj::`) or through its `proc` value. With no argument, return all active `exe` in a `tab`.

See also [\[@number_active\]](#)

`@add_pair(dico:map, key:value, val:value)`

add a new entry to a dictionary. If an entry with key `key` already exists, value `val` replaces the old value. The dictionary `dico` is updated *in place*. The function returns its first argument.

See also [\[@insert\]](#)

`@aggregate(n1:nim, n2:nim, ...)`

aggregates the arguments into a vectorial `nim`. The number of components of the result is the sum of the number of components of each arguments. This function admits a variable number of argument and cannot be curried.

See also [\[@projection\]](#)

See also `{!Library/Functions/nim_functions.list!}`

`@align_breakpoints(nim)`

builds a new `nim` with linear interpolation type whose *homogeneous* breakpoints are the breakpoints of all components of the arguments.

A vectorial `nim` is **homogeneous** if all its component have the same breakpoints, *i.e.* breakpoints with the same abscisses.

See also `@align_breakpoints`, `@sample` and the `nim` simplification functions: `@simplify_radial_distance_t`, `@simplify_radial_distance_v`, `@simplify_lang_v`, `@filter_median_t`, `@filter_min_t`, `@filter_max_t`, `@window_filter_t`

In the figure below, the diagram at the top left shows a vectorial `nim` with two components:

- the effect of `@sample` is pictured at top right,
- the effect of `@align_breakpoints` is sketched at bottom left,
- and the effect of `@linearize` is illustrated at bottom right.

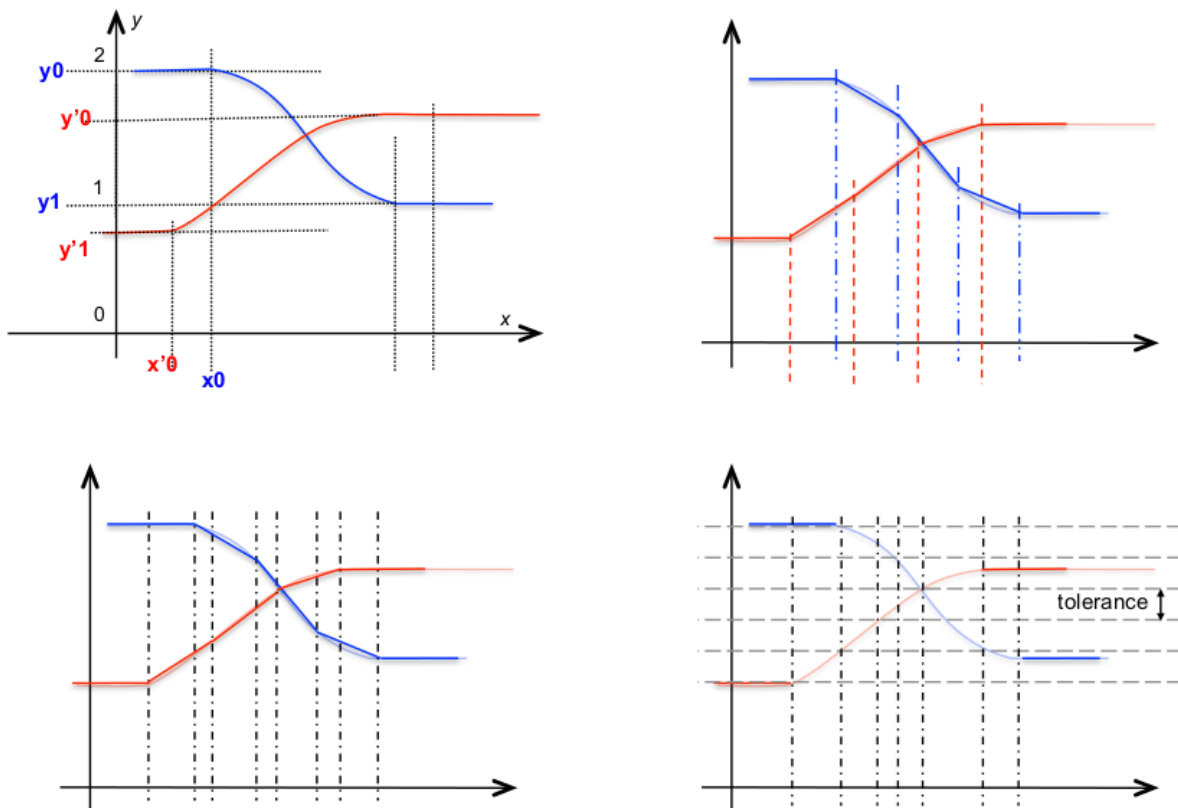


Figure 21.2: the effect of `@sample`, `@align_breakpoints` and `@linearize` on a `nim`

`@approx(x:numeric, y:numeric)` ; listable

The function call can also be written with the special syntax

```
(x ~ y)
```

note that the parenthesis are mandatory.

This predicate returns true if

```
abs((x - y)/max(x, y)) < $APPROX_RATIO
```

The predefined variable `$APPROX_RATIO` is initialized to `0.1` so `(x ~ y)` means `x` and `y` differ by less than 10%. By changing the value of the variable, one changes the level of approximation for the following calls to `@approx`.

Notice that using this function to check if a number is near zero is a bad idea: `(x ~ 0)` results in the comparison of `1` or `-1` (as a result of `abs(x)/x`) even if `x` is zero.

If one argument is a tab and the other is a scalar `u`, the scalar argument is extended to tab if (all elements of the extension are equal to `u`) and the predicate returns true if it hold pointwise for all elements of the tabs. For example

```
(tab[1, 2] ~ 1.02)
```

returns false because we don't have `(2 ~ 1.02)`.

```
@arch_darwin()
```

this predicate returns true if the underlying host is Mac OSX and false elsewhere.

See also `[@arch_linux]` and `[@arch_windows]`

```
@arch_linux()
```

this predicate returns true if the underlying host is a Linux system and false elsewhere.

See also `[@arch_darwin]` and `[@arch_windows]`

```
@arch_windows()
```

this predicate returns true if the underlying host is a Windows system and false elsewhere.

See also `[@arch_darwin]` and `[@arch_linux]`

```
@asin(numeric) ; listable
```

arc sine

```
{!Library/Functions/math_functions.list!}
```

```
@atan(numeric) ; listable
```

arc tangente

```
{!Library/Functions/math_functions.list!}
```



```
@atan2(x:numeric, y:numeric) ; listable
```

The `atan2()` function computes the principal value of the arc tangent of y/x , using the signs of both arguments to determine the quadrant of the return value.

The `atan2()` function is used mostly to convert from rectangular (x, y) to polar (r, θ) coordinates that must satisfy $x = r \cos(\theta)$ and $y = r \sin(\theta)$. In general, conversions to polar coordinates should be computed thus:

```
$r := @sqrt($x*$x + $y*$y)
$theta := @atan2($y, $x).
```

```
{!Library/Functions/math_functions.list!}
```

```
@between(a:numeric, x:numeric, b:numeric) ; listable
```

This function admits an infix special syntax and can be written

```
(x in a .. b)
```

the parenthesis are *mandatory*. This predicate is true if

```
(a < x) && (x < b)
```

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

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

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

```
{!Library/Functions/math_functions.list!}
```

```
@bounded_integrate() ; deprecated
```

this function is deprecated

```
@bounded_integrate_inv() ; deprecated
```

this function is deprecated

```
@car(tab)
```

returns the first element of `tab` if is not empty, else an empty `tab`.

Some functions handle `tab` as *lisp lists*: `[@car]`, `[@cdr]`, `[@concat]`, `[@cons]`, `[@empty]`, `[@drop]`, `[@take]`.

See also `{!Library/Functions/tab_functions.list!}`

`@cdr (tab)`

if the argument is not empty, it returns a copy of it but deprived of its first element, else it returns an empty `tab`.

Some functions handle `tab` as *lisp lists*: `[@car]`, `[@cdr]`, `[@concat]`, `[@cons]`, `[@empty]`, `[@drop]`, `[@take]`.

See also `{!Library/Functions/tab_functions.list!}`

`@ceil (numeric)`

This function returns the smallest integral value greater than or equal to its argument.

See also `{!Library/Functions/math_functions.list!}`

`@clear (tab)`

`@clear (map)`

clear all elements in the `tab` or in the `map` argument, resulting in a vector or a dictionary of size zero.

See also `{!Library/Functions/tab_functions.list!}`

See also `{!Library/Functions/map_functions.list!}`

`@concat (tab, tab)`

returns a new `tab` made by the concatenation of the two `tab` arguments.

See also `{!Library/Functions/tab_functions.list!}`

`@cons (v:value, t:tab)`

build a new `tab` by prepending `v` to `t`.

See also `{!Library/Functions/tab_functions.list!}`

`@copy (value)`

returns a fresh copy of the argument. For data structure like `map` or `tab`, the copy is a deep copy: elements of the data structure are also copied.

`@cos (numeric)`

computes the cosine of its argument (measured in radians).

See also `{!Library/Functions/math_functions.list!}`

`@cosh (numeric)`

computes the hyperbolic cosine of its argument.

See also `{!Library/Functions/math_functions.list!}`

```
@count(tab, value)
@count(map, value)
@count(string, value)
```

computes the number of times the second argument appears in the first argument. For a map, the second argument refers to a value stored in the dictionary. For a string, the second argument must be a string and the call returns the number of occurrences of the first character in the first string:

```
@count("abacaa", "a")
```

returns 4.

See also [[@find](#)], [[@member](#)] and [[@occurs](#)].

```
@dim(t:tab)
@dim(n:nim)
```

if the argument is a tab t , the call returns the dimension of t , *i.e.* the maximal number of nested tabs. If the argument is a nim n , the call returns the number of components of the nim, *i.e.* the number of elements in the tab returned by the application of the nim.

In either case, the returned value is an integer strictly greater than 0. If the argument is not a tab nor a nim, the dimension is 0.

```
@domain(m:map)
```

returns a tab containing all the keys present in the map m .

See also [[@range](#)].

See also `{!Library/Functions/map_functions.list!}`

```
@drop(t:tab, n:numeric)
@drop(t:tab, x:tab)
```

`@drop(t:tab, n:numeric)` build a new tab which is t with its first n elements dropped if $n > 0$, and with its last n elements dropped if $n < 0$.

`@drop(t:tab, x:tab)` returns the tab formed by the elements of t whose indices are not element of x .

See also *lisp like* functions: [[@car](#)], [[@cdr](#)], [[@concat](#)], [[@cons](#)], [[@empty](#)], [[@drop](#)], [[@take](#)].

See also `{!Library/Functions/tab_functions.list!}`

```
@dump(file:string, variable_1, ... variable_p)
```

is a special form: the arguments `variable_i` are restricted to be variables. Calling this special form store the values of the variables in the file whose path is `file`. This function is a special form, so it cannot be curryfied.

The stored value can be restored using the function `[@loadvar]`. The dump file produced by `[@dump]` is in a human readable format and corresponds to a fragment of the *Antescofo* grammar.

The returned value is `true` if the value of the variable have been saved, `false` elsewhere (e.g. if `file` cannot be created). The process of saving the values is done asynchronously in a dedicated thread, so the run-time computation are not perturbed.

The dump file can be produced during one program execution and can be read in another program execution. This mechanism can be used to manage **presets**.

Note that this special form expands into the ordinary function `[@dumpvar]`. The same comments apply.

See also `[@dumpvar]`, `[@savevalue]` and `[@loadvalue]`.

```
@dump(file:string, id_1:string, v_1:value, ..., id_p:string, v_p:value)
```

save in `file` the value `v_i` under the name `id_i`. Then `file` can be used by the function `[@loadvar]` to define and set or to reset the variable with identifier `id_i`. The format used in `file` is a text format corresponding to the *Antescofo* grammar.

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

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

See also `[@dump]`, `[@savevalue]` and `[@loadvalue]`.

```
@empty(tab)
```

```
@empty(map)
```

returns true for an empty tab or an empty dictionary and false elsewhere.

See also `{!Library/Functions/tab_functions.list!}`

See also `{!Library/Functions/map_functions.list!}`

```
@exp(x:numeric)
```

the base *e* exponential of *x*.

See also `{!Library/Functions/math_functions.list!}`

```
@explode(s:string)
```

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

```

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

```

See also `{!Library/Functions/string_functions.list!}`

```
@filter_max_t(nim, n:numeric)
```

build a new *smoother* nim from the nim argument. The result is build by replacing every image y_0 of a breakpoint (x_0, y_0) by the **maximum** value of the y in a sequence of $2n + 1$ breakpoints centered on (x_0, y_0) . The first n breakpoints and the last n breakpoints are leaved untouched. The resulting nim has the same number of breakpoint as the argument with the same x.

See also `[@filter_median_t]`, `[@filter_min_t]`, `[@window_filter_t]`.

```
@filter_median_t(nim, n:numeric)
```

build a new *smoother* nim from the nim argument. The result is build by replacing every image y_0 of a breakpoint (x_0, y_0) by the **median** value of the y in a sequence of $2n + 1$ breakpoints centered on (x_0, y_0) . The first n breakpoints and the last n breakpoints are leaved untouched. The resulting nim has the same number of breakpoint as the argument with the same x.

See also `[@filter_max_t]`, `[@filter_min_t]`, `[@window_filter_t]`.

```
@filter_min_t(nim, n:numeric)
```

build a new *smoother* nim from the nim argument. The result is build by replacing every image y_0 of a breakpoint (x_0, y_0) by the **minimum** value of the y in a sequence of $2n + 1$ breakpoints centered on (x_0, y_0) . The first n breakpoints and the last n breakpoints are leaved untouched. The resulting nim has the same number of breakpoint as the argument with the same x.

See also `[@filter_max_t]`, `[@filter_median_t]`, `[@window_filter_t]`.

```
@find(t:tab, f:function)
@find(m:map, f:function)
@find(s:string, f:function)
```

returns the index of the first element of t, m or s that satisfies the predicate f.

The predicate is a binary function taking the index or the key as the first argument and the associated value for the second argument.

The undef value (for maps) or the integer -1 (for tab and string) is returned if there is no pair satisfying the predicate.

See also `[@count]`, `[@member]` and `[@occurs]`.

Example:

```
@fun_def p(i, v) { return (i > 1) && (v % 2 == 0) }
@find([1, 2, 3, 4], @p) ; returns 4
```

`[@find]` returns the first index greather than one such that the corresponding element is a multiple of two.

```
@fun_def p(i, v) { return (i > 1) && (v >= "b") }
@find("abcdefg", @p) ; returns 2
```

the index 2 is returned because the "c" is the first character greater or equal to "b" such that its position is strictly greater than one (character numbering starts at zero).

```
@flatten(t:tab)
@flatten(t:tab, n:numeric)
```

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

```
@flatten([[1, 2], [3], [[]], [4, 5]]) -> [1, 2, 3, 4, 5]
```

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

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

See also some other *lisp like functions*: [`@car`], [`@cdr`], [`@concat`], [`@cons`], [`@empty`], [`@drop`], [`@take`].

See also `{!Library/Functions/tab_functions.list!}`

```
@floor(x:numeric)
```

returns the largest integral value less than or equal to `x`

See also `{!Library/Functions/math_functions.list!}`

```
@gnuplot(data:tab)
@gplot(title:string, data:tab)
@gplot(time:tab, data:tab)
@gplot(title:string, time:tab, data:tab)
@gplot(title1:string, time1:tab, data1:tab, title2: string, time2:tab,
data2:tab, ...)
```

The function [`@gnuplot`] has a variable number of arguments and cannot be curried. It admits five forms. See also [`@plot`] and [`@rplot`].

```
@gnuplot(data:tab)
```

The function `[@gnuplot]` plots the elements in the `data` tab as a time series. If `data` is a tab of numeric values, a simple curve is plotted: an element `e` of index `i` gives a point of coordinate (i, e) . If `data` is a tab of tab (of `p` numeric values), `p` curves are plotted on the same window.

Each `[@gnuplot]` invocation lead to a new window. The invocation is done asynchronously: when the function returns, the plot process is still running for its completion.

Function `[@gnuplot]` returns `true` if the plot succeeded, and `false` elsewhere.

To work, the `gnuplot` program must be installed on the system Cf. <http://www.gnuplot.info> and must be visible from the *Antescofo* object (on a Mac system, it can be installed through *fink*, *macport* or *brew*; on linux, it can be installed through the package management system). They are several ways to make this command visible and the search of a `gnuplot` executable is done in this order:

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

The command is launched on a shell with the option `-persistent` and the absolute path of the `gnuplot` command file.

The data are tabulated in a file `/tmp/tmp.antescofo.data.n` (where `n` is an integer) in a format suitable for `gnuplot`. The `gnuplot` commands used to plot the data are in the file `/tmp/tmp.antescofo.gnuplot.n`. These two files persists between two *Antescofo* session and can then be used to plot with other option.

The variable `$gnuplot_linestyle` can be used to change the style of the lines used to connect the data points. The value must be as string giving the `gnuplot` name of the chosen style. The default value is `"linespoints"`. Other possible values are: `"dots"`, `"points"`, `"linespoints"`, `"impulses"`, `"steps"`, `"fsteps"`, `"histeps"`, see [Gnuplot Plot style](#).

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

```
@gnuplot(title:string, data:tab)
```

same as the previous form, but the first argument is used as the label of the plotted curve. If `data` is a tab of tab, (*e.g.* the history of a tab valued variable), then the label of each curve takes the form `title_i`.

```
@gnuplot(time:tab, data:tab)
```

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

```
@gnuplot(title:string, time:tab, data:tab)
```

Same as the previous entry but the first argument is used as the label of the curve(s).

```
@gnuplot(title1:string, time1:tab, data1:tab, title2:string, time2:tab, data2:tab, ...)
```

In this variant, several curves are plotted in the same window. One curve is specified by 2 or 3 consecutive arguments. Three arguments are used if the first considered argument is a string: in this case, this argument is the label of the curve. The following argument is used as the `x` coordinates and the next one as the `y` coordinates of the plotted point. The tab arguments must be tab of numeric values (they cannot be tab of tab).

```
@gshift_map(a:map, f:function)
```

(where `f` can be a map, a nim or an intentional function), builds a new map `b` such that

$$b(f(x)) = a(x)$$

See also `{!Library/Functions/map_functions.list!}`

```
@history_length(variable)
```

This is a special form: the argument must be a variable (it cannot be a general expression). It returns the maximal length of the history of the variable, *i.e.* the number of update that are recorded.

```
@hz2midi(numeric)
```

convert a frequency (expressed in Hz) into a midi note. For example, 440 is converted in 69. See [midi tuning](#) and functions `[@midicent2hz]`, `[@hz2midicent]`, `[@hz2symb]` and `[@symb2midicent]`.

See also `{!Library/Functions/midi_functions.list!}`

```
@hz2midicent(numeric)
```


convert a frequency (expressed in Hz) into a midi note expressed in midicent. For example, 440 is converted in 6900. See [midi tuning](#) and functions [[@midicent2hz](#)], [[@hz2midi](#)], [[@hz2symb](#)] and [[@symb2midicent](#)].

See also `{!Library/Functions/midi_functions.list!}`

`@hz2symb(numeric)`

convert a frequency (expressed in Hz) into a string in [Scientific pitch notation](#) representing the pitch. The *scientific pitch notation* is the notation used in the specification of [pitch in events](#). For example, 440 is converted in "A4".

NOTE: the microtonal alteration, as in `:::atescofo A#4+50` are not currently supported.

See [midi tuning](#) and functions [[@midicent2hz](#)], [[@hz2midi](#)], [[@hz2symb](#)] and [[@symb2midicent](#)].

See also `{!Library/Functions/midi_functions.list!}`

`@insert(t:tab, i:numeric, v:value)`

`@insert(m:map, k:val, v:value)`

[[@insert](#)] is an impure overloaded function.

`@insert(t, i, v)` inserts “in place” the value `v` into the tab `t` after the index `i` (tab’s elements are indexed starting with 0). If `i` is negative, the insertion take place in front of the tab. If `i >= @size(t)` the insertion takes place at the end of the tab.

`@insert(m, k, v)` inserts “in place” a new entry with value `v` under key `k` in the map `m`. If the entry already exists, the current value is replaced by `v`. See [[@add_pair](#)]

Notice that the form is also used to include a file at parsing time. See section [file structure](#).
not yet documented

`@iota(n:numeric)`

returns [`$x | $x in n`] that is, a tab listing the integers from to 0 to `n` excluded.

`@is_bool(value)`

the predicate returns true if its argument is a boolean value.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_defined(dico:map, k:value)`

the predicate returns true if `k` is a key present in `dico` Do not mismatch with the negation of the predicate [[@is_undef](#)].

See also `{!Library/Functions/predicates_functions.list!}`

`@is_exec(value)`

the predicate returns true if the argument represents an *exec*, that is, the instance of a compound action (*e.g.*, the result of a process call or an object instantiation).

The predicate returns true even if the compound action has finished its computation. However, the *exec* itself used in a boolean condition evaluates to true if the compound action is still running and false elsewhere.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_fct (value)`

the predicate returns true if its argument is an intentional function, *i.e.* a predefined function, or a function defined using `@fun_def`, or a partial application of such functions.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_float (value)`

the predicate returns true if its argument is a floating point value (a decimal number). Floating point values correspond to *IEEE-754 double precision values* (the C type double).

See also `{!Library/Functions/predicates_functions.list!}`

`@is_function (value)`

the predicate returns true if its argument is a map, a nim, an intentional function (defined using `@fun_def`, a method, a signal or a partial application of such functions).

If a value satisfies this predicates, it can be applied to (zero or more) arguments to achieve a function call.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_int (value)`

the predicate returns true if its argument is a signed integer. Integeres are represented using C type long. The number of bits depends of the *Antescofo* variants and may differs between the 32 and the 64 bits version.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_integer_indexed (value)`

the predicate returns true if its argument is a map whose domain is a set of integers.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_interpolatedmap (value)` ; deprecated

deprecated

`@is_list (value)`

the predicate returns true if its argument is a map whose domain is the integers $[0 \dots n]$ for some n .

See also `{!Library/Functions/predicates_functions.list!}`

`@is_map(value)`

the predicate returns true if its argument is a map.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_nim(value)`

the predicate returns true if its argument is a nim.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_numeric(value)`

the predicate returns true if its argument is an integer or a floating point value.

See also `{!Library/Functions/predicates_functions.list!}`

`@is_obj(value)`

the predicate returns true if its argument is an object.

Objects are implemented using processes so if a value x satisfies `@is_obj`, then it satisfies `@is_exec` (but all execs are not objs).

See also `{!Library/Functions/predicates_functions.list!}`

`@is_obj_xxx(value)`

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

See also `{!Library/Functions/predicates_functions.list!}`

`@is_prefix(s1:string, s2:string)`

`@is_prefix(s1:string, s2:string, cmp:fct)`

`@is_prefix(t1:tab, t2:tab)`

`@is_prefix(t1:tab, t2:tab, cmp:fct)`

`[@is_prefix]` is an overloaded function. See also `[@is_suffix]` and `[@is_subsequence]`.

`@is_prefix(s1:string, s2:string)`

returns true if string $s1$ is a prefix of $s2$.

```
@is_prefix(s1:string, s2:string, cmp:fct)
```

the predicate returns true if `s1` is a prefix of `s2` where the characters are compared with the function `cmp` (taking two arguments). The characters are passed to the function `cmp` as strings of length one.

```
@is_prefix(t1:tab, t2:tab)
```

the predicate returns true if `t1` is a prefix of `t2`, that is, if the elements of `t1` are the initial elements of `t2`.

```
@is_prefix(t1:tab, t2:tab, cmp:fct)
```

same as the previous version but the function is used to test the equality between the elements, instead of the usual comparison between values. For example:

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

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

See also `{!Library/Functions/predicates_functions.list!}`

```
@is_proc(value)
```

the predicate returns true if its argument is a process definition. Notice that a `proc` refers to a process definition and not to a running instance of this definition: it is not an `exec`

See also `{!Library/Functions/predicates_functions.list!}`

```
@is_string(value)
```

the predicate returns true if its argument is a string.

See also `{!Library/Functions/predicates_functions.list!}`

```
@is_subsequence(s1:string, s2:string)
@is_subsequence(s1:string, s2:string, cmp:fct)
@is_subsequence(t1:tab, t2:tab)
@is_subsequence(t1:tab, t2:tab, cmp:fct)
```

`[@is_subsequence]` is an overloaded function. See also the functions `[@is_suffix]` and `[@is_prefix]`.

```
antescofo @is_subsequence(s1:string,
s2:string) the function returns the index of the
first occurrence of string s1 in string s2. A
negative value is returned if s1 does not occurs in
s2.
```

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

```
@is_subsequence(t1:tab, t2:tab)
```

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

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

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

See also `{!Library/Functions/predicates_functions.list!}`

```
@is_suffix(s1:string, s2:string)
@is_suffix(s1:string, s2:string, cmp:fct)
@is_suffix(t1:tab, t2:tab)
@is_suffix(t1:tab, t2:tab, cmp:fct)
```

`[@is_suffix]` is an overloaded function. See also the functions `[@is_subsequence]` and `[@is_prefix]`.

```
antescofo @is_suffix(s1:string,
s2:string) the predicate returns true if string
s1 is a suffix of string s2.
```

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

```
@is_suffix(t1:tab, t2:tab)
```

the predicate returns true if the sequence of elements of `t1` is a suffix of the sequence of element of `t2`.

```
@is_suffix(t1:tab, t2:tab, cmp:fct)
```

same as the previous version but the function `cmp` is used to test the equality between the elements, instead of the usual comparison between values. For example:

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

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

See also `{!Library/Functions/predicates_functions.list!}`

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

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

is returned because and hold.

```
@is_symbol(value)
```

the predicate returns true if its argument is a symbol. Symbol appears as Max/PD identifiers.

See also `{!Library/Functions/predicates_functions.list!}`

```
@is_tab(value)
```

the predicate returns true if its argument is a tab (an indexed sequence of values).

See also `{!Library/Functions/predicates_functions.list!}`

```
@is_undef(value)
```

the predicate returns true if its argument is the undefined value. Do not mismatch with the negation of the predicate `[@is_defined]`.

See also `{!Library/Functions/predicates_functions.list!}`

```
@is_vector(value) ; deprecated
```

deprecated

```
@lace(t:tab, n:numeric)
```

builds a new tab whose elements are interlaced sequences of the elements of the `t` subcollections, up to size `n`.

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

returns

```
[1, 6, "foo", 2, 6, "bar", 3, 6, "foo"]
```

the first elements is taken in the first element of `t`, the second in the second element of `t`, *etc.*, in a cyclic way, until 9 elements have been acquired. Scalar elements are extended to tab of constants and element in a subcollection are taken modulo the size of the subcollection.

See also `{!Library/Functions/tab_functions.list!}`

```
@last(tab)
```

returns the last element of a tab, or undef.

Some other functions handle `tab` as *lisp lists*: `[@car]`, `[@cdr]`, `[@concat]`, `[@cons]`, `[@empty]`, `[@drop]`, `[@take]`.

See also `{!Library/Functions/tab_functions.list!}`

```
@linearize(nim, tol:numeric)
```

build a new `nim` that approximates the argument. The new `nim` uses only linear interpolation and homogeneous breakpoints. An adaptive sampling step achieves an approximation within `tol` (*i.e.* for any point in the domain, the images of the `nim` argument and the image of the resulting `nim` are withing `tol`).

The result is a linear homogeneous `nim`.

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

See also `@align_breakpoints`, `@sample` and the nim simplification functions: `@simplify_radial_distance_t`, `@simplify_radial_distance_v`, `@simplify_lang_v`, `@filter_median_t`, `@filter_min_t`, `@filter_max_t`, `@window_filter_t`

In the figure below, the diagram at the top left shows a vectorial nim with two components:

- the effect of `@sample` is pictured at top right,
- the effect of `@align_breakpoints` is sketched at bottom left,
- and the effect of `@linearize` is illustrated at bottom right.

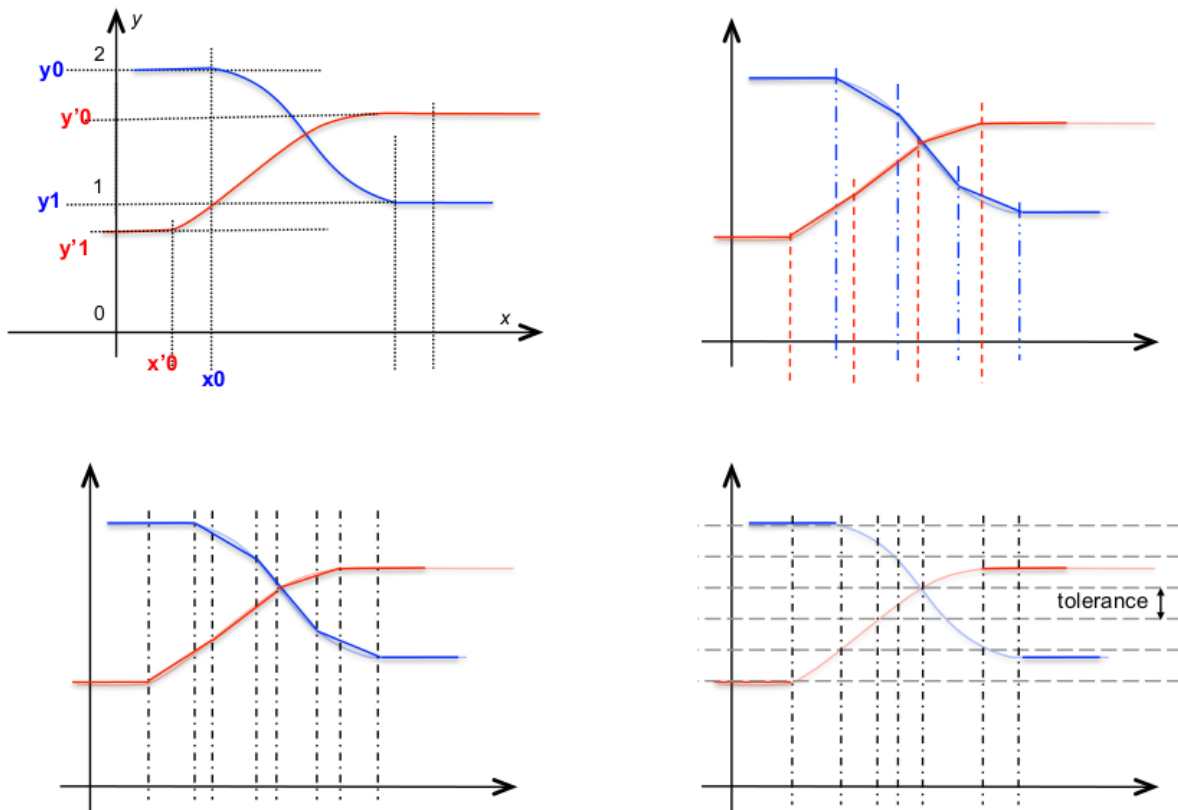


Figure 21.3: the effect of `@sample`, `@align_breakpoints` and `@linearize` on a nim

`@listify(map)`

returns the range of its argument as a list, *i.e.* the returned map is obtained by replacing the keys in the arguments by consecutive integers starting from 1.

See also `{!Library/Functions/map_functions.list!}`

`@loadvalue(*file*:string)`

read a file produced by a call to the function `[@savevalue]` and returns the value that was saved. If something goes wrong, an undefined value is returned. See also function `[@loadvar]` for an example and the related functions `[@dumpvar]` and `[@dump]`.

A call to this function may take a noticeable time depending on the size of the values to store in the dump file. While this time is usually negligible, loading a tab of 10000 integers represents a file of size about 60Kb and takes between 2ms and 3ms. This computational cost may have a negative impact on the audio processing in heavy cases. However, the intended use of and functions is to restore a “preset” at isolated places like the beginning of the score (see `eval_when_load` clauses) or between musical sequences, a usage where this cost should have no impact. Notice that saving a value or variables is done *asynchronously* and does not disturb the “main” computation. See the remarks of function `[@dump]`.

See also `{!Library/Functions/system_functions.list!}`

```
@loadvar(*file*:string)
```

read a file produced by a call to `[@dump]` (or `[@dumpvar]`) and set the value of the corresponding variables.

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

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

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

Beware that `[@loadvar]` does not trigger the `whenever`.

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

Here is an example:

```
@global $a, $b, $c
$a := 1
$b := 2
$c := 3
$ret := @dump("/tmp/dump1", $a, $b, $c)
$a := 0
$b := 0
$c := 0
Group G1
{
  @local $b
  $b := 22
```

```

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

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

```

In this example, the values of the global variables `$a`, `$b` and `$c` are saved by the `[@dump]` special form in file `/tmp/dump1`. The `[@loadvar]` is done in a context where a local variables `$band $c` hidde the global ones and these local variable (with global variable `$a`) will be affected by the `[@loadvar]` command.

See also `{!Library/Functions/system_functions.list!}`

`@log(numeric)`

computes the value of the natural logarithm of its argument.

See also `{!Library/Functions/math_functions.list!}`

`@log10(numeric)`

computes the value of the logarithm of its argument to base 10.

See also `{!Library/Functions/math_functions.list!}`

`@log2(numeric)`

computes the value of the logarithm of its argument to base 2.

See also `{!Library/Functions/math_functions.list!}`

`@make_bpm_map()`

`@make_bpm_map(start:numeric)`

`@make_bpm_map(start:numeric, stop:numeric)`

returns a map where the BPM of the i th event of the score, is associated to i (the keys of the map are the **rank**s of the events). Called with no arguments, the events considered are all the events in the score.

With `start`, only the events whose position *in beat* is greater than `start` are considered.

If a `stop` is specified, all events must have a position *in beat* between `start` and `stop`.

Nota Bene: The numbering of musical events starts at 1. Grace notes, *i.e.* musical event with a bpm of 0, do not appear in the map and does not count in the ranking.

For example

```
NOTE D6 1      event1
NOTE C7 0      event2
NOTE D6 1/2    event3
BPM 30
NOTE C7 1/5
CHORD (D1 A7 Eb7) 1/8 event5
trill (CC6 D7 A7) 1/8 event6
```

with this score, @make_bpm_map() will return:

```
MAP{ (1, 60.0), (2, 60.0), (3, 30.0), (4, 30.0), (5, 30.0) }
```

Notice the grace note C7 which does not appear in the map.

See also {!Library/Functions/score_functions.list!}

```
@make_bpm_tab()
@make_bpm_tab(start:numeric)
@make_bpm_tab(start:numeric, stop:numeric)
```

returns a tab whose *i*th element is the BPM of the *i*th musical event in the score.

Called with no arguments, the events considered are all the events in the score. With a *start*, only the events whose position in beats is greater than *start* are considered. If a *stop* is specified, all events must have a position in beats between *start* and *stop*. Grace events do not appear in the tab.

Examples:

```
NOTE D6 1
NOTE C7 0
NOTE D6 1/2
BPM 30
NOTE C7 1/5
CHORD (D1 A7 Eb7) 1/8
Trill (CC6 D7 A7) 1/8 event6
```

With this score, @make_bpm_tab returns:

```
TAB[60.0, 60.0, 30.0, 30.0, 30.0]
```

Function [!@make_duration_tab!] can be used to complete the bpm information with the duration information for an event.

See also {!Library/Functions/score_functions.list!}

```
@make_duration_map()
@make_duration_map(start:numeric)
@make_duration_map(start:numeric, stop:numeric)
```

returns a map where the duration (in beat) of the i th event of the score, is associated to i (the keys of the map are the **ranks** of the events). Called with no arguments, the events considered are all the events in the score.

With `start`, only the events whose position *in beat* is greater than `start` are considered.

If a `stop` is specified, all events must have a position *in beat* between `start` and `stop`.

Nota Bene: The numbering of musical events starts at 1. Grace notes, *i.e.* musical event with a duration of 0, do not appear in the map and does not count in the ranking.

For example

```
NOTE C7 0 mes2
NOTE D6 1/2
NOTE C7 1/5
NOTE Eb7 2/5
NOTE G#6 1/2
CHORD (D1 A7) 1/8 mes2_2
NOTE C2 1/8
```

with this score, `@make_duration_map()` will return:

```
MAP{ (1, 0.5), (2, 0.2), (3, 0.4), (4, 0.5), (5, 0.125), (6, 0.125) }
```

Notice the grace note C4 which does not appear in the map.

See also `{!Library/Functions/score_functions.list!}`

```
@make_duration_tab()
@make_duration_tab(start:numeric)
@make_duration_tab(start:numeric, stop:numeric)
```

returns a tab whose i th element is the duration in beats the i th musical event in the score.

Called with no arguments, the events considered are all the events in the score. With a `start`, only the events whose position in beats is greater than `start` are considered. If a `stop` is specified, all events must have a position in beats between `start` and `stop`. Grace events do not appear in the tab.

Examples:

```
NOTE D6 1
NOTE C7 0
NOTE D6 1/2
NOTE C7 1/5
CHORD (D1 A7 Eb7) 1/8
Trill (CC6 D7 A7) 1/8 event6
```

With this score, @make_duration_tab returns:

```
TAB[1.0, 0.5, 0.2, 0.125, 0.125]
```

Function [@make_bpm_tab] can be used to get the information necessary to translate the relative duration in absolute duration.

See also {!Library/Functions/score_functions.list!}

```
@make_label_bpm()
@make_label_bpm(start:numeric)
@make_label_bpm(start:numeric, stop:numeric)
```

returns a map associating the event labels to the BPM at this point in the score. Events with no label or with a zero duration (grace note) do not appear in the map.

Called with no arguments, the events considered are all the events in the score. With start, only the events whose position (in beats) is greater than start are considered. If a stop is also specified, all events must have a position between start and stop.

For example, with this score:

```
NOTE D6 1    event1
NOTE C7 0    event2
BPM 120
NOTE D6 2    event3
NOTE C7 2
CHORD (D1 A7 Eb7) 4 event5
BPM 30
trill (CC6 D7 A7) 2 event6
```

the call @make_label_bpm() returns

```
MAP{ ("event1", 1.0), ("event3", 0.5), ("event5", 0.5), ("event6", 2.0) }
```

BEWARE: Contrary to the functions [@make_bpm_tab] and [@make_bpm_tab] and despite the function name, the value associated to the key in the returned map is not in BPM but is **second per beat**, *i.e.* $\frac{60}{\text{bpm}}$. *This is expected to change.*

See also {!Library/Functions/score_functions.list!}

```
@make_label_duration()
@make_label_duration(start:numeric)
@make_label_duration(start:numeric, stop:numeric)
```

returns a map associating to the label of an event, the duration of this event. Events with no label do not appear in the map.

Called with no arguments, the events considered are all the events in the score. With a start, only the events whose position in beats is greater than start are considered. If a stop is specified, all events must have a position in beats between start and stop.

See also {!Library/Functions/score_functions.list!}

```
@make_label_pitches()
@make_label_pitches(start:numeric)
@make_label_pitches(start:numeric, stop:numeric)
```

returns a map associating a vector of pitches to the label of an event. A NOTE corresponds to a tab of size 1, and a CHORDS with n pitches to a tab of size n . Events with no label do not appear in the map, as well as grace notes.

Called with no arguments, the events considered are all the events in the score. With a *start*, only the events whose position in beats is greater than *start* are considered. If a *stop* is specified, all events must have a position in beats between *start* and *stop*.

Examples:

```
NOTE D6 1      event1
NOTE C7 0      event2
NOTE D6 1/2    event3
NOTE C7 1/5
CHORD (D1 A7 Eb7) 1/8 event5
TRILL (CC6 D7 A7) 1/8 event6
```

with this score, @make_label_pitches returns:

```
MAP{ ("event1", TAB[8600.0]),
      ("event3", TAB[8600.0]),
      ("event5", TAB[2600.0, 10500.0, 9900.0]),
      ("event6", TAB[1206.0, 9800.0, 10500.0]) }
```

See also `{!Library/Functions/score_functions.list!}`

```
@make_label_pos()
@make_label_pos(start:numeric)
@make_label_pos(start:numeric, stop:numeric)
```

this function returns a map whose keys are the labels of the events and the value, the position *in beats* of the events. Events with no label do not appear in the map, as well as grace notes.

Called with no arguments, the events considered are all the events in the score. With a *start*, only the events whose position in beats is greater than *start* are considered. If a *stop* is specified, all events must have a position in beats between *start* and *stop*.

Examples:

```
NOTE D6 1      event1
NOTE C7 0      event2
NOTE D6 1/2    event3
NOTE C7 1/5
CHORD (D1 A7 Eb7) 1/8 event5
TRILL (CC6 D7 A7) 1/8 event6
```

with this score, @make_label_pos () returns:

```
MAP{ ("event1", 0.0), ("event3", 1.0), ("event5", 1.7), ("event6", 1.825)
```

See also {!Library/Functions/score_functions.list!}

```
@make_pitch_tab()
@make_pitch_tab(start:numeric)
@make_pitch_tab(start:numeric, stop:numeric)
```

returns a tab whose i th element is the vector of pitches to the i th musical event in the score. A NOTE corresponds to a tab of size 1, and a CHORDS with n pitches to a tab of size n . Grace notes are not counted.

Called with no arguments, the events considered are all the events in the score. With a *start*, only the events whose position in beats is greater than *start* are considered. If a *stop* is specified, all events must have a position in beats between *start* and *stop*.

Grace notes do not appear in the tab.

Examples:

```
NOTE D6 1
NOTE C7 0
NOTE D6 1/2
NOTE C7 1/5
CHORD (D1 A7 Eb7) 1/8
TRILL (CC6 D7 A7) 1/8 event6
```

With this score, @make_pitch_tab returns:

```
TAB[ TAB[8600.0],
      TAB[8600.0],
      TAB[9600.0],
      TAB[2600.0, 0500.0, 9900.0],
      TAB[1206.0, 9800.0, 10500.0] ]
```

See also {!Library/Functions/score_functions.list!}

```
@make_score_map()
@make_score_map(start:numeric)
@make_score_map(start:numeric, stop:numeric)
```

returns a map where the keys are the rank i of the musical events and the value, the *position in beat* of the i th event. Called with no arguments, the events considered are all the events in the score. With *start*, only the events whose position in beats is greater than *start* are considered. If a *stop* is specified, all events must have a position between *start* and *stop*.

See also {!Library/Functions/score_functions.list!}

```
@map(f:function, t:tab)
```

returns a `tab` such that element i is the result of applying f to element $t[i]$. Note that the computation is equivalent to

$$[f(\$x) \mid \$x \text{ in } t]$$

See also `@scan` and `@reduce` for other `tab`-morphisms.

See also `{!Library/Functions/tab_functions.list!}`

Note that function `@map` is related to `tab`, not to `map`: see `@map_compose` and `@map_concat` for `map` related functions.

`@map_compose(a:map, b:map)`

returns the composition of functions a and c , that is, a `map` c such that

$$c(x) == b(a(x))$$

See also function `@mapval` to compose a `map` and a function and `@merge`.

See also `{!Library/Functions/map_functions.list!}`

`@map_concat(a:map, b:map) ; deprecated`

deprecated

See also `{!Library/Functions/map_functions.list!}`

`@map_history(variable)`

This is a special form: the argument must be a variable identifier. It returns a `map` where the keys are integers and the values are the successive values assigned to the variable. Integer 0 corresponds to the current value, 1 to the previous value, *etc.* Variable's history has a bounded length that can be specified using a `@local` or `@global` declaration.

See also `@tab_history`, `@map_history_date` and `@map_history_rdate`.

See also `{!Library/Functions/system_functions.list!}`

`@map_history_date(variable)`

This is a special form: the argument must be a variable identifier. It returns a `map` where the keys are integers and the values are the date in physical times of the successive assignments to the variable. Integer 0 corresponds to the current value, 1 to the previous value, *etc.* Variable's history has a bounded length that can be specified using a `@local` or `@global` declaration.

See also `@tab_history`, `@map_history` and `@map_history_rdate`.

See also `{!Library/Functions/system_functions.list!}`

`@map_history_rdate(variable)`

This is a special form: the argument must be a variable identifier. It returns a map where the keys are integers and the values are the date in relative time (*beats*) of the successive assignments to the variable. Integer 0 corresponds to the current value, 1 to the previous value, *etc.* Variable's history has a bounded length that can be specified using a `@local` or `@global` declaration.

See also `[@tab_history]`, `[@map_history]` and `[@map_history_date]`.

See also `{!Library/Functions/system_functions.list!}`

`@map_normalize(map)` ; deprecated

deprecated

See also `{!Library/Functions/map_functions.list!}`

`@map_reverse(map)` ; deprecated

deprecated

See also `{!Library/Functions/map_functions.list!}`

`@mapval(a:map, b:function)`

returns the composition of function `a` and `c`, that is, a map `c` such that

$$c(x) == b(a(x))$$

See also `[@map_compose]` and `[@merge]`.

See also `{!Library/Functions/map_functions.list!}`

`@max(value, value)`

return the maximum of its two arguments.

Values in *Antescofo* are totally ordered. The order between two elements of different types is implementation dependent. However, the order on numeric is as expected (numeric ordering: the integers are embedded in the decimals). For two argument of the same type, the ordering is as expected (lexicographic ordering for string, and tab, *etc.*).

See `[@min]`, `[@min_key]`, `[@max_key]`, `[@min_val]`, `[@max_val]`, `[@sort]`.

`@max_key(nim)`

returns the coordinate `x_n` of the last breakpoint of the `nim`. This coordinate is the sum of `x_0` (the coordinate of the first breakpoint of the `nim`, and of all intervals `d_i` of the breakpoints `i`. If the `nim` is vectorial, `x_n` is a tab.

See also `[@min_key]`, `[@min_val]` and `[@max_val]`.

See also `{!Library/Functions/nim_functions.list!}`

```
@max_val(tab)
@max_val(map)
@max_val(nim)
```

This overloaded functions returns the maximal element in the `tab` if it is a `tab`, and the maximal element in the range if the argument is a `map` or a `nim`.

If the argument is empty, a `undef` value is returned.

See `[@max]`, `[@range]`.

See also `{!Library/Functions/tab_functions.list!}`

See also `{!Library/Functions/map_functions.list!}`

See also `{!Library/Functions/nim_functions.list!}`

```
@median(tab)
```

computes the *median* of the `tab`'s element. The median of a list of numbers can be found by arranging all the numbers from lowest value to highest value and picking the middle one. For example, the median of `[11, 5, 3, 3, 9]` is 5.

See also `{!Library/Functions/tab_functions.list!}`

```
@member(tab, value)
@member(map, value)
@member(string, value)
```

returns true if the second argument is an element of the first. For a `map`, the second arguments refers to a value stored in the dictionary. For `string`, the value must be a character, *i.e.* a string of size 1.

See also `[@is_prefix]`, `[@is_suffix]`, `[@is_subsequence]`, `[@find]` and `[@occurs]`.

See also `{!Library/Functions/predicates_functions.list!}`

```
@merge(map, map)
```

returns a new `map` which is the asymmetric merge of the two argument `maps`.

The result of `@merge(a, b)` is a `map c` such that $c(x) = a(x)$ if $a(x)$ is defined, and $b(x)$ elsewhere.

Notice that $a(x)$ is defined if x is a key in a but the value $a(x)$ may be the `undef` value.

See also `[@map_compose]` and `[@mapval]`.

See also `{!Library/Functions/map_functions.list!}`

```
@midi_getChannel(tab)
```

The argument is a `tab` representing a midi message, see function `[@midi_read]`. The `[@midi_getChannel]` extracts from this `tab` the channel of the message if the message corresponds to a channel specific command (*i.e.*, `noteOn`, `noteOff`, `Aftertouch`, `Controller`, `PatchChange`, `Pressure` and `Pitchbend`, see `[@midi_read]`).

On a non-channel specific command, the result is undeterminate.

See also `{!Library/Functions/midi_functions.list!}`

@midi_getCommand(tab)

The argument is a tab representing a midi message, see function [midi_read]. The [midi_getCommand] extracts from this tab the command of the message (a number between 0x80 and 0x8f), or returns -1 if the message does not corresponds to a command.

See also {Library/Functions/midi_functions.list!}

@midi_getCommandByte(tab)

The argument is a tab representing a midi message, see function [midi_read]. The [midi_getCommandByte] extracts the first element of the tab. For the seven channel related midi commands, this corresponds to the encoding of a command plus a channel, cf. [midi_read]

See also {Library/Functions/midi_functions.list!}

@midi_getMetaType(tab)

returns the meta-message type for the MidiMessage. If the message is not a meta message, then returns -1. See the [standard midi file format](#).

The first byte of a meta message is a characteristic flag. The second byte specify the kind of meta message. So this function simply returns the second elemnt of the tab, if the first is the meta flag.

See also {Library/Functions/midi_functions.list!}

@midi_isAftertouch(tab)

the predicate returns true if the tab represents a Aftertouch midi message. See [midi_read].

See also {Library/Functions/midi_functions.list!}

@midi_isController(tab)

the predicate returns true if the tab represents a Continuous Controller midi message. See [midi_read].

See also {Library/Functions/midi_functions.list!}

@midi_isEndOfTrack(tab)

the predicate returns true if if message is a meta message for end-of-track (meta message type 0x2f). See [midi_read].

See also {Library/Functions/midi_functions.list!}

@midi_isMeta(tab)

the predicate returns true if the tab represents a is a Meta message (when the command byte is 0xff). See [midi_read].

See also {Library/Functions/midi_functions.list!}

`@midi_isNote(tab)`

the predicate returns true if the tab represents either a note-on or a note-off message. See `[@midi_read]`.

See also `{!Library/Functions/midi_functions.list!}`

`@midi_isNoteOff(tab)`

the predicate returns true if the command is a note off (0x80) or if the command is a note on 0 velocity (a 0 velocity means silence). See `[@midi_read]`.

See also `{!Library/Functions/midi_functions.list!}`

`@midi_isNoteOn(tab)`

the predicate returns true if the command is a note on with a non-zero velocity. See `[@midi_read]`.

See also `{!Library/Functions/midi_functions.list!}`

`@midi_isPatchChange(tab)`

the predicate returns true if the tab represents a PatchChange command. See `[@midi_read]`.

See also `{!Library/Functions/midi_functions.list!}`

`@midi_isPitchbend(tab)`

the predicate returns true if the tab represents a Pitchbend command. See `[@midi_read]`.

See also `{!Library/Functions/midi_functions.list!}`

`@midi_isPressure(tab)`

the predicate returns true if the tab represents a Pressure command. See `[@midi_read]`.

See also `{!Library/Functions/midi_functions.list!}`

`@midi_isMeta(tab)`

the predicate returns true if message is a meta message describing tempo (meta message type 0x51). See `[@midi_read]`.

See also `{!Library/Functions/midi_functions.list!}`

`@midi_read(string)`

`@midi_read(string, map)`

This function read a midi file whose pathname is given by the first argument (a [string](#)), and returns a [tab](#) representing a list of tracks. Each tracks is a list of timestamped midi messages. And a midi message si a tab of integer, each integer representing the corresponding byte of the encoded midi message:

```
@midi_read(string) --> [ track\ensuremath{_0}, track\ensuremath{_1}, ...]
track\ensuremath{_i} = [ [timestamp\ensuremath{_0}, [byte\ensuremath{_0}\ens
                        [timestamp\ensuremath{_1}, [byte\ensuremath{_1}\ensuremath{_0}, b
                        [timestamp\ensuremath{_2}, [byte\ensuremath{_2}\ensuremath{_0}, b
                        ....
                    ]
```

If there is only one track, the returned tab is directly the list of timestamped midi messages.

Timestamps interpretation

The timestamps are expressed in seconds and represent a duration starting from the previous midi message (delta times in midi). The timestamps are computed from the data in the midifile, taking into account the changes of tempo.

This is true irrespectively of the time representation used in the original midi file (tick or seconds, relative or absolute).

Parameterizing the track extraction

The second optional argument of the function is a [map](#) that can be used to specify several options by associating a value to a key characterizing the option:

- the entry "jointracks" defines a boolean to true if all the tracks of the midi file must be joined before producing the resulting tab. If this option is specified, there is only one track and the returned value is directly the list of timestamped midi messages.
- the entry "tracks" defines a tab enumerating the tracks to extract from the midi file. This option can be used independently of the previous option (then the results is a list of tracks, except if there is only one extracted track).
- the entry "noteonly" defines a boolean specifying if only NoteOn and NoteOff midi messages are extracted.
- the entry "start" is used to specify a date (in second, relatively to the start of the midi file) from which the midi event must be extracted. If not specified the extract start with the first midi message in the midi file.
- the entry "end" is used to specify a date (in second, relatively to the start of the midi file) from which the midi event should not be considered anymore. If not specified, the extraction goes until the last midi message in the midi file.

Example of the specification of some options:

```
@midi_read("bolero.mid", MAP{ ("jointracks", true),
                              ("tracks", [0, 2]),
                              ("end", 10) })
```

will extract the midi messages of the first and third tracks of the first 10 seconds of the midi file ‘bolero.mid’).

Encoding of Midi Message in Antescofo

Midi messages are encoded as a tab of integers. Element i in this tab encodes the value of the byte i in the binary representation of the message (*i.e.* a positive value between 0 and 256). This encoding makes easy to send the relevant data to a MAX `midioobject`, cf. the example given below.

Midi messages have a variable length. This link provide a [short but complete description of the Standard midi file format](#):

- The first byte in the MIDI message is expected to be a command byte, which is a byte in the range from 0x80 to 0xff (128-255 decimal):
 - There are seven midi commands from 0x80 to 0xe0 command that specify a midi channel (the command is specified by the top four bits of the byte. The bottom four bits of the byte indicates the midi channel involved by the command. Midi channels range from 0x0 to 0xf hex (0 to 15 decimal).
 - There are 16 miscellaneous commands starting with a 0xf0 nibble that don't refer to midi channels in the bottom bits.
- Each command has an expected number of parameter bytes after it, which are in the range from 0x00 to 0x7f hexadecimal (0 to 127 decimal). Here is a table summarizing the seven main midi commands and their required parameter count:

Command	Command name	#param	Parameter meaning
0x80	Note Off	2	key, off velocity
0x90	Note On	2	key, on velocity
0xA0	Aftertouch	2	key, pressure
0xB0	Controller	2	controller number, controller value
0xC0	Patch change	1	instument number
0xD0	Channel Pressure	2	key, off velocity
0xE0	Pitch-bend	2	LSB, MSB

Various predicate and observers takes the tab encoding a midi message and extract relevant information, see `{!Library/Functions/midi_functions.list!}`

The function `[@midi_track2ascii]` takes a list of timestamped midi messages, as returned by `[@midi_read]`, and produces a tab where the seven previous commands are given in a human readable way, with the channel is uncoupled from the command name. This function can be used for debugging purposes.

A simple midi player

The following code fragment implement a very simple midi player, by parsing a midifile with `[@midi_read]` and using the results to send the data to a `midio` max object (through the receiver `maxmidi`). The sending of the midi messages are implement with a loop that uses the timestamp as periods:

```

$filename := "/Users/giavitto/UTopIa/Antescofo/Work/TEST/Bolero-1.mid"
$option := MAP{ ("end", 35) }
$midi := @midi_read($filename, $op)

// player
$index := 0
$period := $midi[$index, 0]

Loop $period
{
    print "midi message: " ($midi[$index, 1])
    ForAll $e in $midi[$index, 1]
    {
        maxmidi $e
    }
    $index := $index + 1
    if ($index < @size($midi)) { $period := $midi[$index, 0] }
} while ($index < @size($midi))

```

Because timestamps are interpreted here as relative time (the period specifies a relative time, not an absolute time, even if the computed timestamps corresponds initially to seconds in the midfile), it is possible to modulate the tempo of the playback, using a computed tempo [`@tempo`] or to synchronize the playback with the play of the musician. Note that the playback at the original speed is achieved with a tempo of 60.

Nota Bene: because the way timestamps are computed by Antescofo, there is no further need to send the meta message relatively to the tempo. So the midi event sent to the midiout object in MAX must be restricted to note-on and note-off event, which can be achieved using the `noteonly` option when reading the midi file.

`@midi_track2ascii(tab)`

This function read a list of timestamped midi messages, as returned by [`@midi_read`], and produces a tab where the seven previous commands are given in a human readable way, with the channel is uncoupled from the command name. This function can be used for debugging purposes.

For example, suppose that

```
$t := @midi_read("bolero.mid")
```

returns the tab

```

TAB[ [0.0,          TAB[255, 3, 6, 66, 111, 108, 101, 114, 111]],
      [0.0,          TAB[255, 81, 3, 11, 188, 206]],
      [0.0,          TAB[255, 88, 4, 4, 2, 24, 8]],
      [1.45032,      TAB[240, 65, 16, 66, 18, 64, 0, 127, 0, 65, 247]],
      [0.717147,    TAB[176, 91, 127]],
      [0.020032,    TAB[177, 91, 127]],

```

```

[0.0400641, TAB[178, 91, 127]],
[0.0400641, TAB[179, 91, 127]],
[0.0400641, TAB[180, 91, 127]],
[0.00400641, TAB[201, 48]],
[0.0, TAB[153, 86, 64]],
[0.00400641, TAB[137, 86, 64]],
[0.0, TAB[200, 32]],
[0.0, TAB[184, 10, 65]],
[0.0, TAB[152, 36, 52]],
[0.0320512, TAB[181, 91, 127]],
[0.00801281, TAB[193, 60]],
[0.0, TAB[145, 67, 98]],
[0.0320512, TAB[182, 91, 127]],
[0.0400641, TAB[183, 91, 127]],
[0.0360577, TAB[129, 67, 98]],
[0.00400641, TAB[184, 91, 127]],
[0.0400641, TAB[185, 91, 127]],
[0.0400641, TAB[186, 91, 127]],
[0.0400641, TAB[187, 91, 127]],
[0.0400641, TAB[188, 91, 127]],
[0.0400641, TAB[189, 91, 127]],
[0.0120192, TAB[145, 67, 98]],
[0.0120192, TAB[153, 86, 31]],
[0.0, TAB[190, 91, 127]],
[0.0160256, TAB[191, 91, 127]],
[0.0520833, TAB[129, 67, 98]],
[0.0160256, TAB[137, 86, 31]],
[0.0360577, TAB[145, 67, 98]],
[0.0120192, TAB[153, 86, 31]],
[0.0240384, TAB[136, 36, 52]],
[0.020032, TAB[129, 67, 98]],
[0.0160256, TAB[137, 86, 31]],
[0.0360577, TAB[145, 67, 98]],
[0.0280448, TAB[153, 86, 31]],
[0.0440705, TAB[129, 67, 98]]

```

```
]
```

Then

```
@midi_tarck2ascii($t)
```

returns

```

TAB[ TAB[<<undef>>],
      TAB[<<undef>>],
      TAB[<<undef>>],
      TAB[<<undef>>],
      TAB["Controller", 0, 91, 127],
      TAB["Controller", 1, 91, 127],

```



```

TAB["Controller", 2, 91, 127],
TAB["Controller", 3, 91, 127],
TAB["Controller", 4, 91, 127],
TAB["PatchChange", 9, 48],
TAB["NoteOn", 9, "D6", 64],
TAB["NoteOff", 9, "D6"],
TAB["PatchChange", 8, 32],
TAB["Controller", 8, 10, 65],
TAB["NoteOn", 8, "C2", 52],
TAB["Controller", 5, 91, 127],
TAB["PatchChange", 1, 60],
TAB["NoteOn", 1, "G4", 98],
TAB["Controller", 6, 91, 127],
TAB["Controller", 7, 91, 127],
TAB["NoteOff", 1, "G4"],
TAB["Controller", 8, 91, 127],
TAB["Controller", 9, 91, 127],
TAB["Controller", 10, 91, 127],
TAB["Controller", 11, 91, 127],
TAB["Controller", 12, 91, 127],
TAB["Controller", 13, 91, 127],
TAB["NoteOn", 1, "G4", 98],
TAB["NoteOn", 9, "D6", 31],
TAB["Controller", 14, 91, 127],
TAB["Controller", 15, 91, 127],
TAB["NoteOff", 1, "G4"],
TAB["NoteOff", 9, "D6"],
TAB["NoteOn", 1, "G4", 98],
TAB["NoteOn", 9, "D6", 31],
TAB["NoteOff", 8, "C2"],
TAB["NoteOff", 1, "G4"],
TAB["NoteOff", 9, "D6"],
TAB["NoteOn", 1, "G4", 98],
TAB["NoteOn", 9, "D6", 31],
TAB["NoteOff", 1, "G4"]
]

```

The `TAB[<<undef>>]` is a tab with only one element, `<<undef>>`. This corresponds to midi header associated with the file and the track. The seven channel related commands (see `[@midiread]`) are described with a tab whose first element is the command name (as a string), the second element is the channel number, *etc.* Cf. the description of the parameters of a command at `[@midi_read]`.

The `[@midi_tracks2ascii]` is mainly used for printing partial information in a human readable format.

See also `{!Library/Functions/midi_functions.list!}`

`@midi2hz(m:numeric)`

convert a midi note number into a frequency. For example, 69 (the coding of the standard pitch A440) is converted in 440Hz. See [midi tuning](#) and functions [`@midicent2hz`], [`@hz2midi`], [`@hz2symb`] and [`@symb2midicent`].

See also `{!Library/Functions/midi_functions.list!}`

`@midicent2hz (numeric)`

convert a midi note expressed in *midicent* into a frequency. For example, 6900 (the coding in midicent of the standard pitch A440) is converted in 440Hz. See [midi tuning](#) and function [`@midicent2hz`], [`@hz2midi`], [`@hz2symb`] and [`@symb2midicent`].

See also `{!Library/Functions/midi_functions.list!}`

`@min (value, value)`

return the minimal value of its two arguments.

Values in *Antescofo* are totally ordered. The order between two elements of different types is implementation dependent. However, the order on numeric is as expected (numeric ordering: the integers are embedded in the decimals). For two argument of the same type, the ordering is as expected (lexicographic ordering for string, and tab, *etc.*).

See [`@max`], [`@min_key`], [`@max_key`], [`@min_val`], [`@max_val`], [`@sort`].

`@min_key (nim)`

returns the coordinate `x_0` of the initial breakpoint of the nim argument. If the nim is vectorial, `x_0` is a tab.

See also [`@max_key`], [`@min_val`] and [`@max_val`].

[`@min_key`], [`@min_val`] and [`@max_val`].

See also `{!Library/Functions/nim_functions.list!}`

`@min_val (tab)`

`@min_val (map)`

`@min_val (nim)`

This overloaded functions returns the minimal element in the tab if it is a tab, and the minimal element in the range if the argument is a map or a nim.

If the argument is empty, a `undef` value is returned.

See [`@min`] and [`@range`].

See also `{!Library/Functions/tab_functions.list!}`

See also `{!Library/Functions/map_functions.list!}`

See also `{!Library/Functions/nim_functions.list!}`

`@normalize (tab)`

`@normalize (tab, min:numeric, max:numeric)`

returns a new tab with the elements normalized between min and max. If they are omitted, they are assumed to be 0 and 1.

If an element of the tab is not a numeric, undef is returned.

See also `{!Library/Functions/tab_functions.list!}`

```
@number_active(string)
@number_active(proc)
@number_active()
```

returns the number of active (alive) instances of a process or an object specified through its name (a string with or without the prefix `::` or `obj::`) or through its `proc` value. With no argument returns the number of concurrent running processes.

See also `[@active]`

```
@occurs(tab, value)
@occurs(map, value)
@occurs(string, value)
```

returns the first index or the first key whose value equals the second argument. For example

```
@occurs(["a", "b", "c", "a", "b"], "b") -> 1
@occurs("xyz", "z") -> 2
@occurs(map{ ("zero", 0), ("null", 0), ("void", 0) }, 0) -> "null"
```

In the last example, the answer "null" is returned because `{"null" < "void" < "zero"}`.

See also `[@count]`, `[@find]`, `[@is_prefix]`, `[@is_subsequence]`, `[@is_suffix]` and `[@member]`.

See also `{!Library/Functions/predicates_functions.list!}`

```
@permute(t:tab, n:numeric)
```

returns a new tab which contains the n th permutations of the elements of `t`. They are factorial s permutations, where s is the size of `t`.

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

For example:

```
$t := [1, 2, 3]
@permute($t, 0) == [1, 2, 3]
@permute($t, 1) == [1, 3, 2]
```

```
@permute($t, 2) == [2, 1, 3]
@permute($t, 3) == [2, 3, 1]
@permute($t, 4) == [3, 1, 2]
@permute($t, 5) == [3, 2, 1]
```

See also [[@sort](#)].

See also `{!Library/Functions/tab_functions.list!}`

```
@plot(variable_1, ..., variable_p)
```

is a special form (the arguments are restricted to be variables).

Calling this special form plots the values stored in the history of the variables as time series in **absolute time** using the [[@gnuplot](#)] function.

See also [[@rplot](#)].

```
@pow(x:numeric, y:numeric) ; listable
```

computes x raised to the power y .

See also `{!Library/Functions/math_functions.list!}`

```
@projection(nim, p:numeric)
```

extracts the p th component of a vectorial `nim`.

See also [[@aggregate](#)].

See also `{!Library/Functions/nim_functions.list!}`

```
@push_back(tab, value)
```

```
@push_back(nim:NIM, d:numeric, y1:numeric)
```

```
@push_back(nim:NIM, d:numeric, y1:numeric, it:string)
```

```
@push_back(nim:NIM, y0:numeric, d:numeric, y1:numeric)
```

```
@push_back(nim:NIM, y0:numeric, d:numeric, y1:numeric, it:string)
```

[[@push_back](#)] is an impure overloaded function. See also [[@push_front](#)].

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

Usually, [[@push_front](#)] is slightly more efficient than [[@push_back](#)].

```
@push_back(nim:NIM, d:numeric, y1:numeric, it:string)
```

add a breakpoint as the last breakpoint of `nim`. The first argument, modified by side-effect, is the `nim`, which is also the returned value.

The argument `d` specifies the length of the interpolation since the previous breakpoint, `y1` is the final value attained at the end of the breakpoint, and `it` is the interpolation type. The

interpolation type can be omitted: in this case, the interpolation is linear. The initial value `y0` of the breakpoint is the `y1` value of the previous breakpoint.

```
antescofo @push_back(nim:NIM, y0:numeric, d:numeric,
y1:numeric, it:string) similar to the previous function but y0 is
explicitly given, making possible to specify discontinuous nim.
```

See also `{!Library/Functions/tab_functions.list!}`

See also `{!Library/Functions/nim_functions.list!}`

`@push_front(tab, value)`

add the second argument at the beginning of its first argument. The first argument, modified by side-effect, is the returned value.

Usually, `[@push_front]` is slightly more efficient than `[@push_back]`.

See also `{!Library/Functions/tab_functions.list!}`

`@rand()`

`[@rand]` is an impure function returning a random number between 0 and 1 (included).

`[@rand]` is similar to `[@random]` but rely on a different algorithm to generate the random numbers.

See also `{!Library/Functions/random_functions.list!}`

`@rand_int(int)`

returns a random integer between 0 and its argument (excluded). This is not a pure function because two calls with the same argument are likely to return different results.

See also `{!Library/Functions/random_functions.list!}`

`@random()`

`[@random]` is an impure function returning a random floating point number between 0 and 1 (included). This is not a pure function because two successive calls are likely to return different results.

The resolution of this random number generator is $1/(2^{31} - 1)$, which means that the minimal step between two numbers in the images of this function is $1/(2^{31} - 1)$.

`[@random]` is similar to `[@rand]` but relies on a different algorithm to generate the random numbers. See also `[@rand_int]`.

See also `{!Library/Functions/random_functions.list!}`

`@range(m:map)`

returns the tab of the values present in the map. The order in the tab is irrelevant.

See also `[@domain]`

See also `{!Library/Functions/map_functions.list!}`

```
@reduce(f:function, t:tab)
```

If `t` is empty, an undefined value is returned.

If `t` has only one element, this element is returned.

In the other case, the binary operation is used to combine all the elements in `t` into a single value

```
f(... f(f(t[0], t[1]), t[2]), ... t[n])
```

For example, if `t` is a vector of booleans, `@reduce(@||, t)` returns the logical disjunction of `t`'s elements.

```
@remove(t:tab, n:numeric)
```

```
@remove(m:map, k:value)
```

`[@remove]` is an impure overloaded function. See also `[@insert]`.

```
@remove(t:tab, n:numeric)
```

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

Note that building a new `tab` by removing elements satisfying some property `P` is easy with a comprehension:

```
[ $x | $x in t, P ]
```

which builds a new `tab`, leaving `t` untouched.

```
antescofo @remove(m:map, k:value)
removes the entry k in map (m is modified in
place). Does nothing if the key k is not
present in map .
```

See also `{!Library/Functions/tab_functions.list!}`

See also `{!Library/Functions/map_functions.list!}`

```
@remove_duplicate(t:tab)
```

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

See also `{!Library/Functions/tab_functions.list!}`

```
@replace(t:tab, find:value, rep:value)
```

returns a new tab in which a number of elements have been replaced by another.

The argument `find` represents a sub-sequence to be replaced: if it is not a tab, then all the occurrences of this value at the top-level of `t` are replaced by `rep`:

```
$t := [1, 2, 3, [2]]
@replace($t, 2, 0) -> [1, 0, 3, [2]]
```

If `find` is a tab, then the replacement is done on sub-sequence of `t`:

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

Note that the replacement is done eagerly: the first occurrence found is replaced and the replacement continue on the rest of the tab. Thus, there is no ambiguity in case of overlapping sub-sequences, only the first is replaced:

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

If the argument `rep` is a tab, then it represents a sub-sequence to be inserted in place of the occurrences of `find`. So, if the replacement is a tab, it must be wrapped into a tab:

```
@replace([1, 2, 3], 2, [4,5]) -> [1, 4, 5, 3]
@replace([1, 2, 3], 2, [[4, 5]]) -> [1, [4, 5], 3]
```

See also `{!Library/Functions/tab_functions.list!}`

```
@reshape(t:tab, s:tab)
```

builds a tab of shape `s` with the element of tab `t`. These elements are taken circularly one after the other. For instance

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

the result has shape 3×2 and the elements are the elements taken in `[1, 2, 3, 4, 5, 6]`.

See also `{!Library/Functions/tab_functions.list!}`

```
@resize(t:tab, int)
```

resizes its first argument and returns the results. The argument is modified. If the second argument is smaller than the size of the first argument, it effectively shrinks the first argument. If it is greater, undefined values are used to extend the tab.

See also `{!Library/Functions/tab_functions.list!}`

```
@reverse(tab)
@reverse(string)
```

returns a new tab or a new string where the elements (characters) are given in the reverse order.

See also `{!Library/Functions/tab_functions.list!}`

See also `{!Library/Functions/string_functions.list!}`

`@rnd_bernoulli(p:float)`

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

```
$bernoulli60 := @rnd_bernoulli(0.6)
$t := [ $bernoulli60() | (1000) ]
```

produces a tab of 1000 random boolean values with a probability of 0.6 to be true.

Random Generators

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

See also `{!Library/Functions/random_functions.list!}`

`@rnd_binomial(t:int, p:float)`

returns a random generator that produces integers according to a *binomial discrete distribution* P of parameters `t` and `p`:

$$P(i, | t, p) = \binom{t}{i} \cdot p^i \cdot (1 - p)^{(t-i)}, \quad i \ensuremath{\geq} 0.$$

See `[@rnd_bernoulli]` for a description of *random generators*.

See also `{!Library/Functions/random_functions.list!}`

`@rnd_exponential(\ensuremath{\lambda}:float)`

returns a random generator that produces floats x according to an *exponential distribution* P of parameter λ :

$$P(x | \ensuremath{\lambda}) = \ensuremath{\lambda} e^{(-\ensuremath{\lambda}x)}$$

See `[@rnd_bernoulli]` for a description of *random generators*.

See also `{!Library/Functions/random_functions.list!}`

`@rnd_gamma(\ensuremath{\alpha}:float)`

returns a random generator that produces floating-point values according to a *gamma distribution* P:

$$P(x \mid \alpha) = x^{\alpha-1} / \Gamma(\alpha)$$

See [rnd_bernoulli] for a description of *random generators*.

See also {Library/Functions/random_functions.list!}

@rnd_geometric(p:int)

returns a random generator that produces integers following a *geometric discrete distribution*:

$$P(i \mid p) = p (1 - p)^i, \quad i \geq 0.$$

See [rnd_bernoulli] for a description of *random generators*.

See also {Library/Functions/random_functions.list!}

@rnd_normal(mu:float, sigma:float)

returns a random generator that produces floating-point values according to a *normal distribution* P:

$$P(x \mid \mu, \sigma) = 1/(\sigma \sqrt{2\pi}) \exp(-x^2/(2\sigma^2))$$

See [rnd_bernoulli] for a description of *random generators*.

See also {Library/Functions/random_functions.list!}

@rnd_uniform_float(a:float, b:float)

returns a generator giving float values according to a *uniform distribution* P:

$$P(x \mid a, b) = 1/(b - a), \quad a \leq x < b$$

See [rnd_bernoulli] for a description of *random generators*.

See also {Library/Functions/random_functions.list!}

@rnd_uniform_float(a:int, b:int)

returns a generator giving int values according to a *uniform distribution* P:

$$P(x \mid a, b) = 1/(b - a + 1), \quad a \leq x \leq b$$

See [rnd_bernoulli] for a description of *random generators*.

See also {Library/Functions/random_functions.list!}

```
@rotate(t:tab, n:int)
```

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

```
@rotate([1, 2, 3, 4], -1) == [2, 3, 4, 1]
@rotate([1, 2, 3, 4], 1) == [4, 1, 2, 3]
```

See also `{!Library/Functions/tab_functions.list!}`

```
@round(numeric)
```

returns the integral value nearest to its argument rounding half-way cases away from zero, regardless of the current rounding direction.

See also `{!Library/Functions/math_functions.list!}`

```
@plot(variable_1, ..., variable_p)
```

is a special form (the arguments are restricted to be variables).

Calling this special form plots the values stored in the history of the variables as time series in **relative time** using the `[@gnuplot]` function.

See also `[@plot]`.

```
@sample(nim, n:integer)
```

build a new nim with linear interpolation type by sampling each component with n points equally spaced between the first and the last breakpoint.

The result is a nim with linear interpolation type for all component, irrespectively of the interpolation type of the nim argument. The resulting nim can be seen as a linear approximation of the original curve.

The sampling is made component by components, so the results is not necessarily *homogeneous*. The function `[@align_breakpoints]` can be used on a nim with linear interpolation type to obtain an equivalent nim with homogeneous breakpoints

The approximation made with sampling on n points is not always satisfactory because the variation of the nim can differ greatly between two intervals. The function `[@linearize]` uses an adaptive sampling step to linearize the nim, to align the breakpoints and to achieve an approximation within a given tolerance .

See also `[@align_breakpoints]`, `[@sample]` and the nim simplification functions: `[@simplify_radial_distance_t]`, `[@simplify_radial_distance_v]`, `[@simplify_lang_v]`, `[@filter_median_t]`, `[@filter_min_t]`, `[@filter_max_t]`, `[@window_filter_t]`

In the figure below, the diagram at the top left shows a vectorial nim with two components:

- the effect of `@sample` is pictured at top right,
- the effect of `@align_breakpoints` is sketched at bottom left,

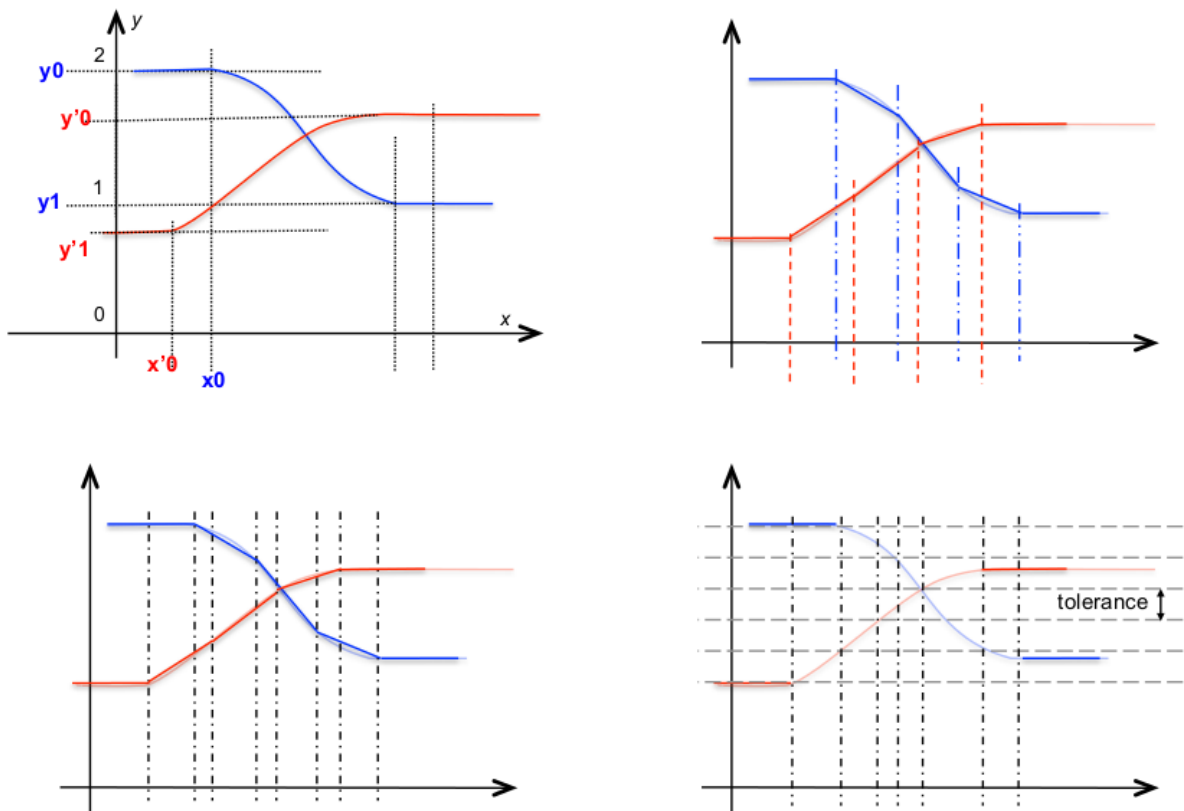


Figure 21.4: the effect of @sample, @align_breakpoints and @linearize on a nim

- and the effect of `@linearize` is illustrated at bottom right.

`@savevalue(s:string, value)`

argument `s` is interpreted as the path of a file where the value of the second argument is saved. The format of the file is textual and corresponds to the *Antescofo* grammar. This value can then be read using the function `[@loadvalue]`.

See also functions `[@dump]`, `[@dumpvar]` and `[@loadvar]`.

See also `{!Library/Functions/system_functions.list!}`

`@scan(f:function, t:tab)`

returns the tab of the partial reduction by `f` of the elements of `t`:

```
[ t[0], f(t[0],t[1]), f(f(t[0], t[1]),t[2]), ...]
```

For example, the tab of the partial sums of the integers between 0 (included) and 10 (excluded) is computed by the expression:

```
@scan(@+, [$x | $x in (10)]) -> [0,1,3,6,10,15,21,28,36,45]
```

If `t` is empty, the results in `undef`.

See also `[@reduce]` and `[@map]`.

See also `{!Library/Functions/tab_functions.list!}`

`@score_tempi()`

returns a tab of couples (*i.e.* tab of size 2). The *i*th elements correspond to the *i*th BPM changes in the score. The value of its element is the tab [position, BPM].

For example

```
NOTE D6 1    event1
NOTE C7 0    event2
BPM 120
NOTE D6 2    event3
NOTE C7 2
CHORD (D1 A7 Eb7) 4 event5
BPM 30
trill (CC6 D7 A7) 2 event6
```

with this score, `@make_bpm_map()` will return:

```
TAB[ TAB[0.0, 60.0],
      TAB[1.0, 120.0],
      TAB[1.5, 30.0] ]
```

See also `{!Library/Functions/score_functions.list!}`

```
@scramble(t:tab)
```

builds a new tab where the elements `t` of have been scrambled (the element are rearranged in a random order). The function is impure: two calls to the same tab does not produce the same result. The argument is unchanged.

See also `{!Library/Functions/tab_functions.list!}`

```
@select_map(m:map, f:funct)
```

build a new map containing the entries of `m` which satisfy predicate `f`. This predicate takes two arguments: the key and the value. For instance

```
@fun_def pred($key, $value) { return ($key > "b") && ($value < 4) }
$m := MAP{ ("a", 1), ("b", 2), ("c", 3), ("d", 4) }
$mm := @select_map($m, @pred)
print $mm
```

```
displays MAP{ ("c", 3) }
```

See also `{!Library/Functions/map_functions.list!}`

```
@set_osc_handling_double(bool)
```

changes the handling of floating point values in OSC messages. *OSC float* are only required to be 32 bits. Nevertheless, several implementations proposes 64 bits representation (aka *IEEE double*).

Antescofo default behavior is to send floating point values as 32 bits floats. However, *Antescofo* floating point values are represented as double. So the default behavior may result in silent overflow when sending integers with the 64 bits object.

A call to `@set_osc_handling_double(true)` switches the behavior to send double. The drawback is that this feature is not implemented in all OSC packages.

When receiving a message, floats and double are always correctly handled if the sender correctly handle them. Notice that receiving a double never result in an overflow.

Beware that a message as a limited size that depends both on the sender and receiver implementation limitation. The current *Antescofo* raw buffer size is 5096. The use of 64 bits integers increase the size of a message.

See the [Osc Messages](#) section in the reference manual and functions `[@set_osc_handling_tab]` and `[@set_osc_handling_int64]`.

See also `{!Library/Functions/system_functions.list!}`

```
@set_osc_handling_int64(bool)
```

changes the handling of integers in OSC messages. *OSC integers* are only required to be 32 bits integers in OSC. Nevertheless, several implementations proposes 64 bits integers.

By default *Antescofo* sends integers as 32 bits integers. However, Antescofo integers representation are 32 bits integers in the 32 bits version of the `antescofo~` objects, and 64 bits integers in the 64 bits version. So the default behavior may result in silent overflow when sending integers with the 64 bits object.

A call to `@set_osc_handling_int64(true)` switches the behavior to send 64 bits integers (irrespectively of the object version). The drawback is that this feature is not implemented in all OSC packages.

When receiving a message, 32 or 64 bits integers are always correctly handled if the sender correctly handle 32 and 64 bits integers. Notice that receiving a 64 bit integers with a 32 bits object may result in a silent overflow.

Beware that a message as a limited size that depends both on the sender and receiver implementation limitation. The current *Antescofo* raw buffer size is 5096. The use of 64 bits integers increase the size of a message.

See the [Osc Messages](#) section in the reference manual and functions `[@set_osc_handling_double]` and `[@set_osc_handling_tab]`.

See also `{!Library/Functions/system_functions.list!}`

`@set_osc_handling_tab(b:bool)`

changes the handling of tabs in OSC messages. *Arrays* are present only in OSC *v1.1* and not present in the initial protocol *v1.0*. Several implementation ignore the array construction.

By default *Antescofo* sends the elements of a tab as the successive arguments of a message, without using the array facilities. A call to `@set_osc_handling_tab(true)` switches the behavior to rely on the array feature present in *v1.1*: the `[` and `]` markers are used to wrap the sending of the tab elements. Calling the function with a false value enables to switch to the *v1.0* policy.

When receiving a message, array markers are always interpreted as tab delimiters.

Beware that a message as a limited size that depends both on the sender and receiver implementation limitation. The current *Antescofo* raw buffer size is 5096. Sending a tab or receiving an array must fit within one message.

See the [Osc Messages](#) section in the reference manual and functions `[@set_osc_handling_double]` and `[@set_osc_handling_int64]`.

See also `{!Library/Functions/system_functions.list!}`

`@shape(t:value)`

returns 0 if is not an array, and else returns a tab of integers each corresponding to the size of one of the dimensions of `t`. Notice that the elements of an array are homogeneous, *i.e.* they have all exactly the same dimension and the same shape.

See also `{!Library/Functions/tab_functions.list!}`

`@shift_map(m:map, n:numeric)`

build a new map containing the entries of `m` that have a integer key with `n` added. For example:

```
$m := MAP{ (1, 1), ("a", 1), (1.0, 1)
           (2, 2), ("b", 2), (2.0, 2) }
print (@shift_map($m, 11))
```

displays MAP{ (12, 1), (13, 2) }.

See also {!Library/Functions/map_functions.list!}

```
@simplify_lang_v(nim, tol:numeric, n:numeric)
```

build a new simpler nim by aggregating breakpoints using a *Lang polyline simplification algorithm* with tolerance `tol` and sequence size `n`.

In the simplification process, a breakpoint is assimilated to a d dimensional point for a nim with dimension d (in other words, the x part of the breakpoint is not taken into account). This is the case for all nim simplification functions that end with a `*_t*`.

The Lang simplification algorithm examines a sequence of such points of a fixed length n (*i.e.*, n successive breakpoints). The first and last points of that sequence specify a segment. This segment is used to calculate the perpendicular distance to each intermediate breakpoints. If any calculated distance is larger than the specified tolerance, the search region will be shrunk by excluding its last point. This process will continue until all calculated distances fall below the specified tolerance, or when there are no more intermediate breakpoints. All intermediate brakpoints are removed and a new search sequence is defined starting at the last point from old search region. This process is illustrated below.

See also [`@align_breakpoints`], [`@sample`] and the nim simplification functions: [`@simplify_radial_distance_t`], [`@simplify_radial_distance_v`], [`@simplify_lang_v`], [`@filter_median_t`], [`@filter_min_t`], [`@filter_max_t`], [`@window_filter_t`]

See also {!Library/Functions/nim_functions.list!}

```
@simplify_radial_distance_t(nim, tol:numeric)
```

builds a new simpler nim by aggregating consecutive breakpoints whose images are within a distance of `tol` (for each component independently)

In the simplification process, a breakpoint is assimilated to a d dimensional point for a nim with dimension d (in other words, the x part of the breakpoint is not taken into account). This is the case for all nim simplification functions that end with a `*_t*`.

It reduces successive vertices that are clustered too closely to a single vertex, called a key. The resulting keys form the simplified polyline. This process is illustrated below:

See also [`@align_breakpoints`], [`@sample`] and the nim simplification functions: [`@simplify_radial_distance_t`], [`@simplify_radial_distance_v`], [`@simplify_lang_v`], [`@filter_median_t`], [`@filter_min_t`], [`@filter_max_t`], [`@window_filter_t`]

See also {!Library/Functions/nim_functions.list!}

```
@simplify_radial_distance_v(nim, tol:numeric)
```

builds a new simpler nim by aggregating consecutive breakpoints whose images are within a distance of `tol`.

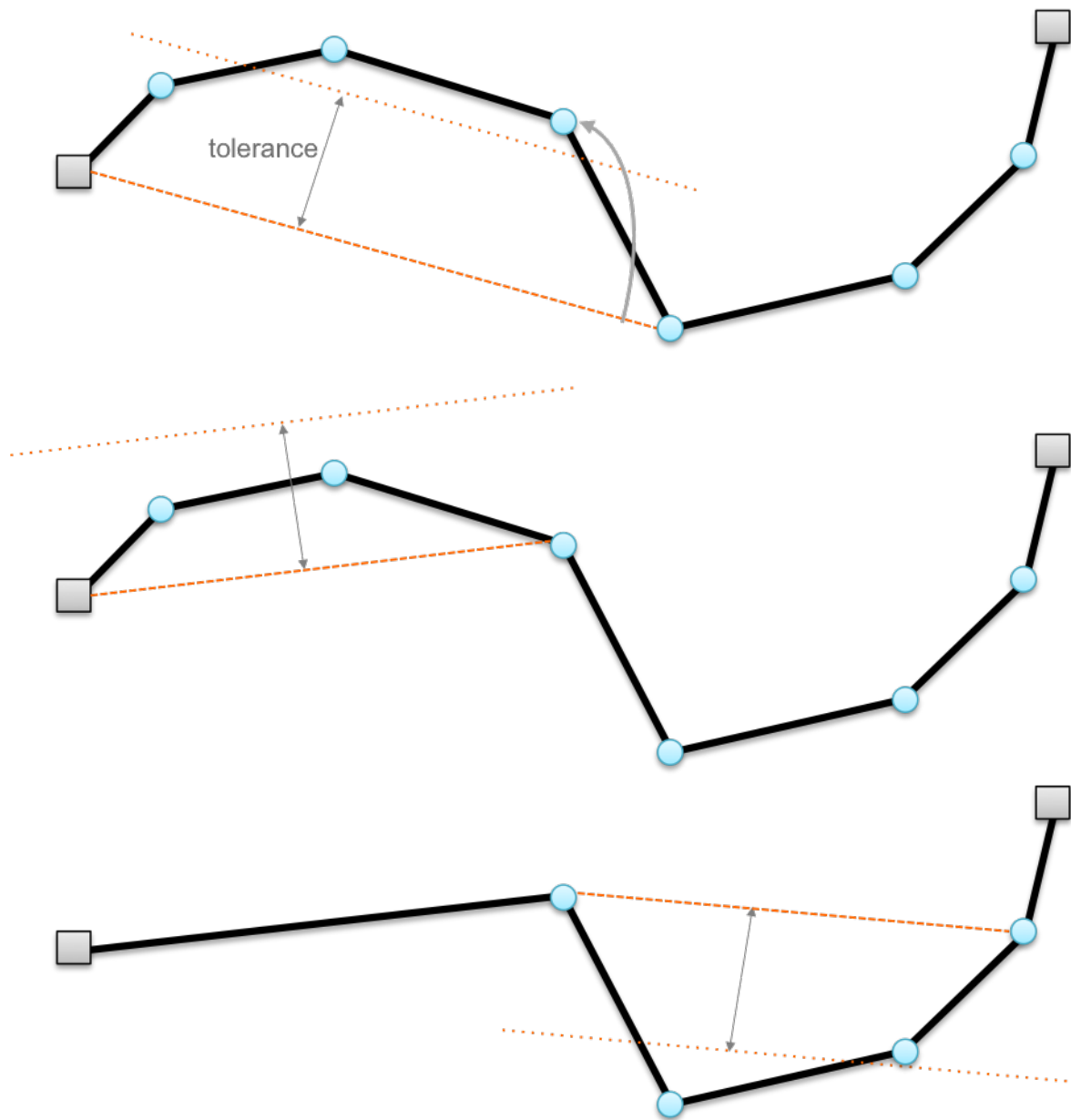


Figure 21.5: the Lang polyline simplification algorithm on a non-intersecting polygon

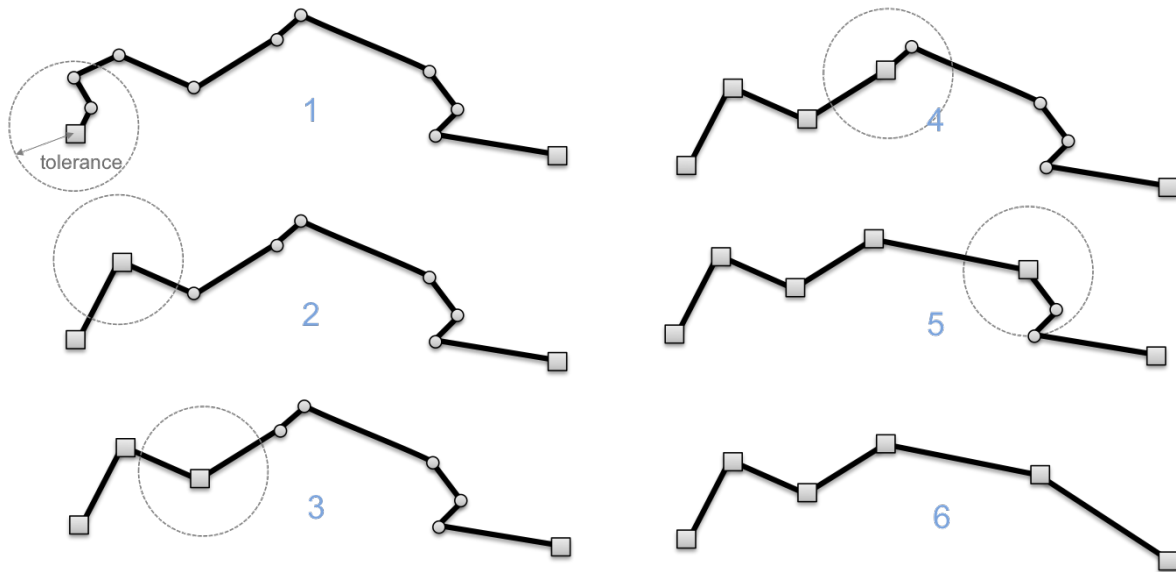


Figure 21.6: the radial distance simplification algorithm on a nim

In the simplification process, each component is handled separately and a breakpoint is assimilated to a 2-dimensional point with coordinate (x, y) . This is the case for all nim simplification functions that end with a `*_v*`.

It reduces successive vertices that are clustered too closely to a single vertex, called a key. The resulting keys form the simplified polyline. This process is illustrated below:

See also `[@align_breakpoints]`, `[@sample]` and the nim simplification functions: `[@simplify_radial_distance_t]`, `[@simplify_radial_distance_v]`, `[@simplify_lang_v]`, `[@filter_median_t]`, `[@filter_min_t]`, `[@filter_max_t]`, `[@window_filter_t]`

See also `{!Library/Functions/nim_functions.list!}`

`@sin(numeric)`

computes the sine of its argument (measured in radians).

See also `{!Library/Functions/math_functions.list!}`

`@sinh(numeric)`

computes the hyperbolic sine of its argument.

See also `{!Library/Functions/math_functions.list!}`

`@size(x:value)`

If `x` is a scalar value, the function returns a strictly negative integer related to the type of the argument (that is, two scalar values of the same type gives the same result).

If `x` is a map or a tab, the function returns the number of elements in its argument (which is a positive integer).

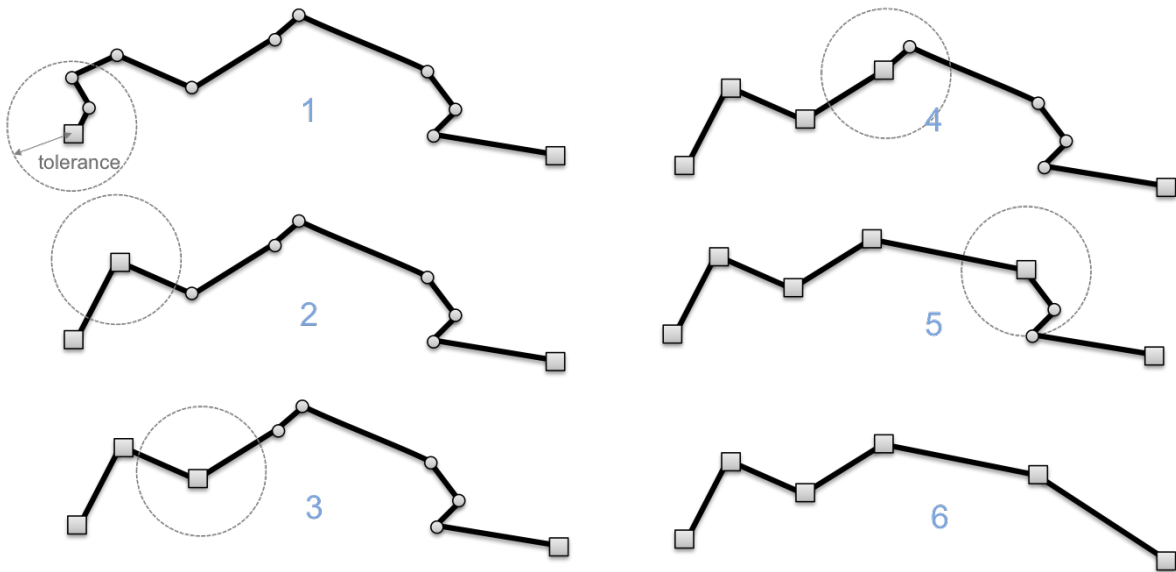


Figure 21.7: the radial distance simplification algorithm on a nim

If it is a nim, it returns the number of breakpoints of the nim (which is not the dimension of the nim). Note that a nim with zero breakpoints is the result of a wrong definition.

See also `@shape` and `{!Library/Functions/predicates_functions.list!}`

```
@slice(t:tab, n:numeric, m:numeric)
```

gives the elements of `t` of indices between `n` included up to `m` excluded. If `n > m` the element are given in reverse order. So

```
@slice(t, @size(t), 0)
```

is equivalent to

```
@reverse(t)
```

See also functions `@car`, `@cdr`, `@drop` and `@take`.

See also `{!Library/Functions/tab_functions.list!}`

```
@sort(t:tab)
@sort(t:tab, cmp:fct)
```

The `@sort` function is an impure overloaded function, with a variable number of arguments (so it cannot be curried).

See also `@permute`

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

```
@sort(t:tab, cmp:fct)
```

sorts in-place the elements into ascending order.

The elements are compared using the function `cmp`. This function must accept two elements of the `tab` as arguments, and returns a value converted to a `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second.

- if p' it is lower than p , then c becomes the value of e and the element next e is processed.
- If p' it is greater than p , then c takes the value of the next element in τ , this new value becomes the value of e and the element next e is processed.

Not that this function is impure as it returns a different result for each invocation. For example:

```
@sputter([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 0.5, 16)
```

can return

```
--> [ 1, 1, 1, 1, 1, 2, 3, 4, 5, 6, 6, 6, 7, 8, 8, 9 ]
--> [ 1, 2, 3, 3, 4, 5, 6, 7, 8, 8, 9, 9, 9, 9, 10 ]
--> [ 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5 ]
```

See also `{!Library/Functions/tab_functions.list!}`

```
@sqrt(x:numeric)
```

computes the non-negative square root of x .

See also `{!Library/Functions/math_functions.list!}`

```
@string2fun(s:string)
```

returns the function whose name is given in argument. The initial `@` in the function name can be omitted. This is useful to convert directly a string received through OSC or through messages into a function that can be applied.

See also `[@string2obj]`, `[@string2proc]` and `{!Library/Functions/system_functions.list!}`

```
@string2obj(s:string)
```

returns the object definition whose name is given in argument. The initial `obj::` in the `obj` name can be omitted. This is useful to convert directly a string received through OSC or through messages into an object that can be instantiated.

See also `[@string2fun]`, `[@string2proc]` and `{!Library/Functions/system_functions.list!}`

```
@string2proc(s:string)
```

returns the proc whose name is given in argument. The initial `::` in the proc identifier can be omitted. Useful to convert directly a string received through OSC or through a message into a process that can be instantiated.

For example, assuming that is set in the Max environment to a tab of two elements, the first being a process identifier and the second an integer, then the code

```
whenever($channel)
{
    :: (@string2proc($channel[0])) ($channel[1])
}
```

will react to the assignment to `$channel` by calling the corresponding processes with the specified integer.

See also `[@string2fun]`, `[@string2obj]` and `{!Library/Functions/system_functions.list!}`

`@stutter(t:tab, n:numeric)`

returns a new tab whose elements are repeated `n` times.

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

See also `{!Library/Functions/tab_functions.list!}`

`@symb2midicent(string)`

convert a string expressing a pitch in [Scientific pitch notation](#) into its equivalent midicent encoding. The *scientific pitch notation* is the notation used in the specification of [pitch in events](#). For example, "A4" is converted into '6900'.

NOTE: the microtonal alteration, as in `:::atescofo A#4+50`, are supported.

See [midi tuning](#) and functions `[@midicent2hz]`, `[@hz2midi]`, `[@hz2symb]` and `[@hz2symb]`.

See also `{!Library/Functions/midi_functions.list!}`

`@system(cmd:string)`

This impure function hands the argument `command` to the command interpreter `sh`. The calling process waits for the shell to finish executing the command, ignoring SIGINT and SIGQUIT, and blocking SIGCHLD. A false boolean value is returned if an error occurred (in this case an error message is issued).

See also `{!Library/Functions/system_functions.list!}`

`@tab_history(variable)`

This is a special form. It returns a tab of the values of the variable in argument.

See also `[@map_history]`, `[@tab_history_date]` and `[@tab_history_rdate]`.

`@tab_history_date(variable)`

This is a special form. It returns a tab of the date in *physical time* of the updates of the variable in argument.

See also `[@map_history]`, `[@tab_history]` and `[@tab_history_rdate]`.

(variable): This is a special form. It returns a tab See. sect. `[sec:mapvariable]` page and the functions.

```
@tab_history_rdate(variable)
```

This is a special form. It returns a tab of the date in *relative time* of the updates of the variable in argument.

See also `@map_history`, `@tab_history` and `@tab_history_date`.

(variable): This is a special form. It returns a tab See. sect. `[sec:mapvariable]` page and the functions.

```
@take(t:tab, n:numeric)
```

```
@take(t:tab, x:tab)
```

is a pure overloaded function. See also functions `@cdr`, `@drop` and `@slice`.

```
antescofo @take(t:tab, n:numeric)
builds a new tab with the n first elements of
t if n > 0 and the last -n elements of t if n
is negative.
```

```
@take(t:tab, x:tab)
```

gives the tab of elements whose indices are in tab `x`. This is equivalent to

```
[t[x[$i]] | $i in @size(x)]
```

See also `{!Library/Functions/tab_functions.list!}`

```
@tan(x:numeric)
```

computes the tangent of `x` (measured in radians).

See also `{!Library/Functions/math_functions.list!}`

```
@Tracing()
```

```
@Tracing(x:function, ...)
```

```
@Tracing(x:string, ...)
```

```
@Tracing(x:tab, ...)
```

Calling `@Tracing` starts to trace the calls to all functions specified by the arguments. Functions to trace are given by their name (as a string) or by their value (through their identifier) or by a tab containing such values.

With no argument, all user-defined function are traced.

See also `@UnTracing` and `{!Library/Functions/system_functions.list!}`

```
@UnTracing()
@UnTracing(x:function, ...)
@UnTracing(x:string, ...)
@UnTracing(x:tab, ...)
```

Calling `[@UnTracing]` stops the trace the calls of the functions specified by the arguments. If specified functions are not traced, there is no effect.

Without arguments, all traced functions stop to be traced.

See also `[@Tracing]` and `{!Library/Functions/system_functions.list!}`

```
@window_filter(nim, coef:tab, pos:numeric)
```

build a new `nim` by processing each component independently.

Each component is the result of a smoothing process of the breakpoints. Each breakpoint of the new `nim` is computed by the dot product of `coef` with a sequence of `y` of the same length as `coef` where position `pos` corresponds to the current breakpoint.

For example,

```
@window_filter(nim, [2], 0)
```

build a `nim` by scaling the image of `nim` by 2.

```
@window_filter(nim, [0.1, 0.2, 0.4, 0.2, 0.1], 2)
```

is a moving weighted average with symmetric weights around `y`.

See also `[@filter_max_t]`, `[@filter_median_t]` and `[@filter_min_t]`.

Chapter 22

The rest of the story is yet to be written... *by you*

We want to complete the *Antescofo* documentation with your contributions.

Please, send your comments, typos, bugs reports, hints and suggestions on the Antescofo [ForumUser](#) web pages. It will help us to improve the documentation and the *Antescofo* system.

The documentation will evolve to include snippets of code, tutorials, and howtos. Do not hesitate to send *your examples*. Ideally, your contribution must include a working patch with the needed audio files and a description of your example in the form of a README.md file written in markdown format. The goal is to have an autonomous demo with all the explanations needed to run the demo by the *Antescofo* users.

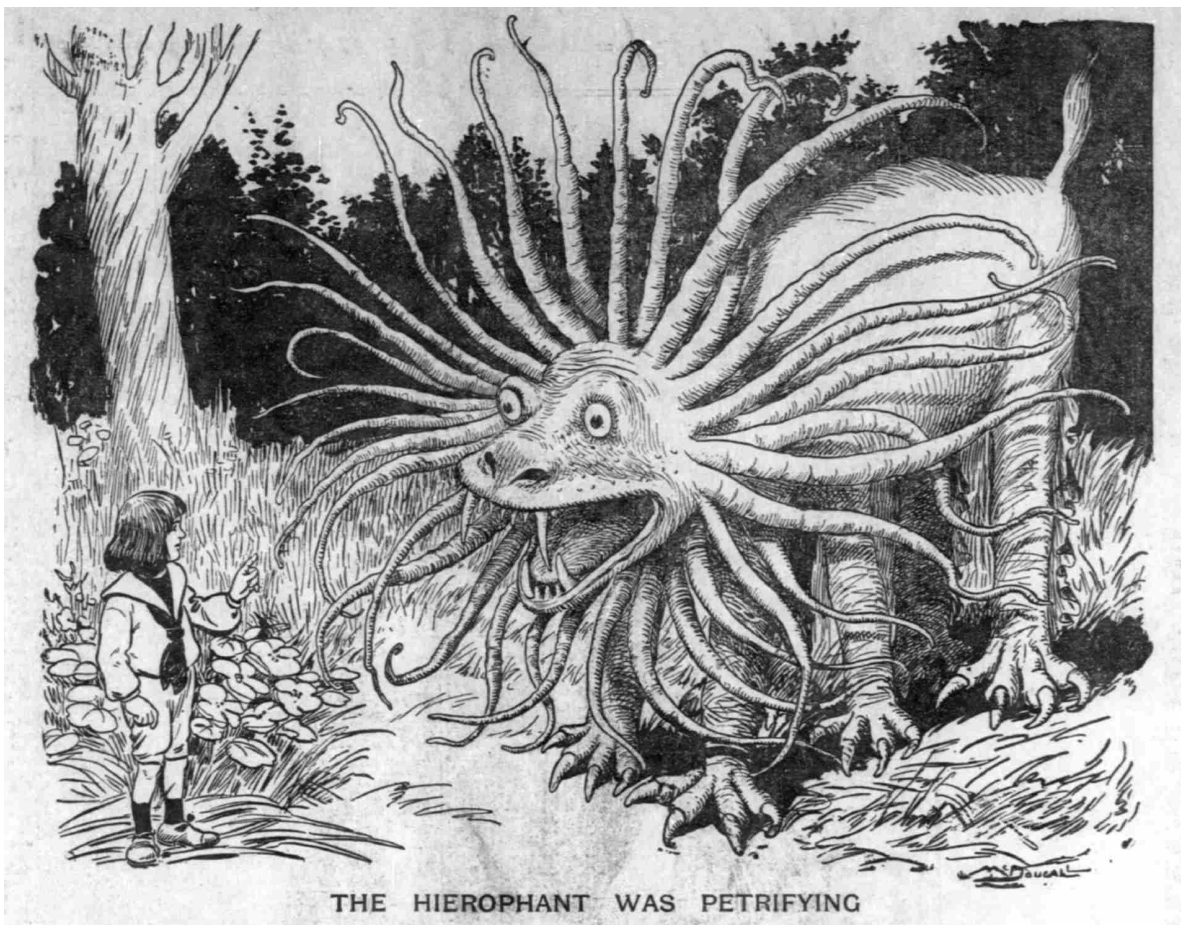


Figure 22.1: The hierophant was petrifying