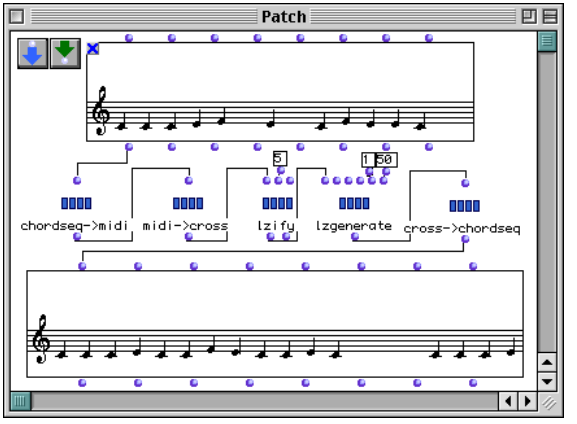
Ircam documentation

- *Research reports*
- *Musical works*
- *Software*

OpenMusic

LZ 2.2 Library

version 2



Third Edition, November 2001

ircam
Centre
Pompidou

This documentation was written by Gérard Assayag and Olivier Lartillot, and was produced under the editorial responsibility of Marc Battier, Marketing Office, Ircam.

OpenMusic was conceived and programmed by Gérard Assayag and Carlos Agon.

The LZ library was originally conceived and programmed by Gérard Assayag. Version1 by G. Assayag. Versions 2 2.x by O. Lartillo. Statistical Model and algorithm by S. Dubnov and G. Assayag. Improvements by O. Lartillot

Third edition of the manual, November 2001.

This documentation corresponds to version 2.2 of the LZ library, and to version 2.0 or higher of OpenMusic.

Ircam
1, place Igor-Stravinsky
F-75004 Paris
Tel. 01 44 78 49 62
Fax 01 44 78 15 40
E-mail ircam-doc@ircam.fr

Groupe d'utilisateurs Ircam

L'utilisation de ce programme et de sa documentation est strictement réservée aux membres des groupes d'utilisateurs de logiciels Ircam. Pour tout renseignement supplémentaire, contactez :

Département de la Valorisation
Ircam
1, Place Stravinsky
F-75004 Paris
France

Courrier électronique: bousac@ircam.fr

Veuillez faire parvenir tout commentaire ou suggestion à :

M. Battier
Département de la Valorisation
Ircam
1, Place Stravinsky, F-75004 Paris, France

Courrier électronique : bam@ircam.fr

http://www.ircam.fr/forumnet

To see the table of contents of this manual, click on the Bookmark Button located in the Viewing section of the Adobe Acrobat Reader toolbar.

To move between pages,
use Acrobat Reader navigations buttons
or your keyboard’s arrow keys

Content

1 LIBRARY PRESENTATION

1.1. Introduction

It is commonly admitted (see the long paper " Automatic Modeling of Musical Style" (ICMC.rtf)) that musical perception is guided by expectations based on the recent past context. Predictive theories are often related to stochastic models which estimate the probability for musical elements to appear in a given musical context, such as Markov chains, already used extensively in computer music. The main problem with these models is that the length of musical context (size of memory) is highly variable, ranging from short figurations to longer motifs. Taking a large fixed context makes the parameters difficult to estimate and the computational cost grows exponentially with the size of the context. We have designed a new model which builds a statistical representation of any polyphonic musical sequence, and then lets you generate variants of these sequences in the same style .

1.2. Dictionary-based prediction

We use a dictionary-based prediction method, which parses an existing musical text into a lexicon of phrases/patterns, called *motifs*, and provides an inference method for choosing the next musical object following a current past *context*. The parsing scheme must satisfy two conflicting constraints. On the one hand, one wants to maximally increase the dictionary to achieve better prediction, but on the other hand, enough evidence must be gathered before introducing a new phrase, so that a reliable estimate of the conditional probability is obtained. The secret of dictionary-based prediction (and compression) methods is that they cleverly sample the data so that most of the information is reliably represented by few selected phrases. This could be contrasted to better known Markov models that build large probability tables for the next symbol at every context entry. Although it might seem that the two methods operate in a different manner, it is helpful to understand that basically they employ similar statistical principles.

1.2.1. Incremental Parsing

We chose to use an incremental parsing (IP) algorithm suggested by Lempel and Ziv [LZ78]. IP builds a dictionary of distinct motifs by sequentially adding every new phrase that differs by a single next character from the longest match that already exists in the dictionary. For instance, given a text {ababaaabaabbabba...}, IP parses it into {a, b, ab, aa, aba, abb, abba...} where motifs are separated by commas. The dictionary may be represented as a tree, each node being one particular symbol, and each branch being one characteristic motif. For our example, we get :

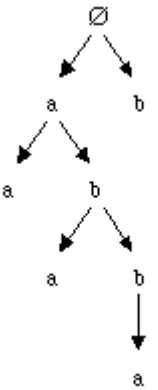


Figure 1 : the tree resulting from the LZ analysis of the sequence {ababaaabaabbabba...}

1.2.2. Aleatory Generation

The dictionary, which is the result of the "style" analysis by Incremental Parsing, may be used to generate new instances of this "style". At each step of the generation phase, we try to identify as many last symbols as possible - that have just been generated - with one subbranch of the tree that is not a complete branch. The final node of this subbranch will be called the *context* of this step of the generation phase. For example, if we have already generated the sequence "aabbbaababbab" and if we would like to add another symbol, following the style modelized by the tree displayed fig. 1, we see that the subbranch {b} contains the last generated sequence {b}. The trouble is, this branch is complete so we have to choose another subbranch. We see also that the subbranch {ab} contains the last generated sequence {ab} and is not a complete branch. And we also remark that we cannot find a longer subbranch that contains a part of the last generated sequence. So the context is the node "b" of the subbranch {ab}.

This context contains one or several children. The choice of the next generated symbol is simply the stochastic choice of one of these children, each of them having a probability proportional to the size of the subtree under this child, that is to say the number of nodes in the subtree whose root is this child. In our example, the two candidate children are "a" whose subtree size is 1, and "b" chose subtree size is 2. So the choice of the next symbol is the result of the stochastic choice between "a" with probability 1/3 and "b" with probability 2/3.

1.2.3. Relation to Markov models

An interesting relation between Lempel-Ziv and Markov models was discovered by [WIL91] when considering the length of the context used for prediction. In IP every prediction is done in the context of earlier prediction, thus resulting in a sawtooth behavior of the context length. For every new phrase the first character has no context, the second has context of length one, and so on. In contrast, the Markov algorithm makes predictions using a totally flat context line determined by the order of the model. Thus, while a Markov algorithm makes all of its prediction based on 3- or 4-character contexts, the IP algorithm will make some of the predictions from lower depth, but very quickly it will exceed the Markov constant depth and use a better context. Our experiments show that this IP scheme, along with the appropriate linear representation of music, provides with patterns and inferences that successfully match musical expectation.

Another important feature of the dictionary-based methods is that they are "universal". If the model of the data sequence was known ahead of time, an optimum prediction could be achieved at all times. The difficulty with most real situations is that the probability model for the data is unknown. Therefore one must use a predictor that works well no matter what the data model is. This idea is called "universal prediction" and it is contrasted to Markov predictors that assume a given order of the data model. Universal prediction algorithms make minimal assumptions on the underlying stochastic sources of musical sequences. Thus, they can be used in a great variety of musical and stylistic situations. Our IP based predictor is one such example of universal predictor. This differs also from knowledge-based systems, where specific knowledge about a particular style has to be first understood and implemented [COP96].

1.3. The Library

The library itself is extremely simple to use. It is based upon 4 main functions : *Lzify*, PSTify *Lzgenerate* and PSTgenerate. *Lzify* and PSTify will take a Midifile and build the motif dictionary, while *Lzgenerate* and PSTgenerate will take that dictionary as input and generate a new sequence that behaves statistically like the original (a *variant*).

There are 2 way to use this : with a single midifile or with a set of midifile concatenated. In the last case, the dictionary will be considered as a compressed representation of the *motivic* space set up by these midifiles. Thus *Lzgenerate* and PSTgenerate in that case will build a kind of interpolated path into that space (see Patch Example 3).

The *Lzify* and PSTify functions analyze strings of symbols. So a chord-seq or a midi-file, in order to be analyzed by this function, has to be translated into a string of symbols by the function *midi->cross*.

Similarly, once the *Lzgenerate* or *PSTgenerate* function has generated a sequence of symbols, it has to be translated into a chord-seq by the *cross->chordseq* function in order to be displayed by a chord-seq object.

1.4. New features of the 2.2 version.

The LZ algorithm has been significantly improved. Now the generation is significantly better. You don't need to use *MinPast* and *MinComplex* to get good musical resuts any more !

A new set of operatos, called PST, that behave like LZ but in a more precise way. The trouble is, it is slower too. The new objects are*PSTify*, *PSTgenerate* and *PSTprint*.

1.5. New features of the 2.1 version.

Each time a patch is loaded, the *lzify* boxes are automatically unlocked. Indeed, it is more convenient and faster to compute their information again than loading them. In the same way, every duplicated *lzify* box is automatically unlocked.

A bug has been fixed in the duration quantization algorithm taken from *OMKant* library. And this algorithm is not called any more when you do not need it.

1.6. New features of the 2.0 version.

In this new version, it is possible to analyze not only monodies or superposition of monodies, but also complex polyphonies, containing chords, melodies, or anything else.

This new *midi->cross* function also features a toolbox of functions that can simplify the original text in order to ease the analysis phase.

This new version introduces the concept of *analysis information* and *synthesis information*. Each symbol of the original sequence can be splitted into a couple of two symbols : an *analysis symbol* that will be used by the Incremental Parsing algorithm, and a *synthesis symbol* that contains other information that

will be added to the generated analysis symbols. For example, you can analyse only the pitch of the notes (this will improve this analyse because there will be much more redondance), but keep in mind the duration of the notes, in order to put it again into the generated symbols.

It is possible to constrain the generation phase to avoid *failures*. Indeed, sometimes, at some steps of the generation phase, there may be no context at all (the last generated symbols do not belong to any branch of the tree). In this case, the generation algorithm is able to decide to go back to a previous step of the generation and try another symbol, until it succeeds.

Is is also possible to escape from looping states during the generation phase. Indeed, the *Lzgenerate* may sometimes generate infinite loops, for example infinite trills. The algorithm is now able to detect this kind of loops and dec

ide to escape from it by taking into account, when choosing the context, not only the longest past but also shorter pasts.

The generated sequence can begin by a predefined sequence (an *incipit*).

It is also possible to define any personal constraint that has to be true at each step of the generation phase.

The new version analyzes not only MIDI files, but also chord-seqs.

The tree can be explicitly displayed in the Listener by the *lzprint* function. It is also possible to get particular information like the number of nodes of the tree (*lzsize*), the size of the longest branch (*lzlength*) of the tree or to display one by one all the branches of the tree (*lzuntree*).

(see the long paper " Automatic Modeling of Musical Style ", ICMC.pdf)

Arguments inside brackets are optional.

LZify

LZify *text niter (type)*

Builds a pattern dictionary containing a statistical model.

Inputs

text	list of anything.
niter	integer >=1.
type	a list containing : a function selecting the analysis information inside a symbol. a function selecting the synthesis information inside a symbol. a function that reconstructs the symbol from analysis and synthesis information (will be used by *LZGenerate*).

output

a LZ continuation tree.

a LZ pattern tree.

description

LZify takes a list of anything considered as an ordered sequence. It then builds a pattern dictionary that encodes patterns of various lengths discovered over this sequence, as well as the conditional probabilities that a certain pattern be followed by certain elements. If *niter* is greater than 1, the analysis of the sequence is iterated *niter* times, each time skipping the next element on the left of the sequence. *niter*>1 increases the number of patterns discovered. It is equivalent to analyzing a longer sequence, thus increasing the statistical properties (redundancy). Empirical experience shows that *niter* = 4 is good value for such data as bach-like counterpoint or jazz chorus.

Then *LZify* builds the continuation tree. This tree is another representation of the pattern dictionary, suited to generation features : the branch are reversed, in order to ease the search for the maximum context, and the continuations of each context are explicitly represented, linked with their corresponding probabilities.

In the 2.0 version, if you lock this object, saving, loading or copying this required a significant amount of time, because all the trees would be saved or duplicated. That is why, in the 2.1 version, this object is automatically unlocked each time the patch is loaded or when the box is duplicated.

LZgenerate

LZgenerate *dict maxpast length (mostprobable minpast mincomplex incipit1 incipit2 reconstr strategy constraints equiv1 equiv2)*

Generates a new sequence following the model of a given LZ continuation tree (generated by the *LZify* function).

Inputs

dict	a LZ continuation tree generated by the *LZify* function.
maxPast	*nil* or integer. maximum length of the context. If *nil* : no maximum past constraint.
Length	integer, >=1. length of the sequence to be generated.
mostprobable	if not true, inverse all the probability distributions (the less probable one becomes the most probable one).
minPast	integer, >=0. minimum length of the context.
minComplex	integer, >=0. minimum size of the subLZtree of each context.
incipit1	list. a sequence of analysis symboles that will be the beginning of the generated analysis sequence.
incipit2	list. a sequence of synthesis symboles that will be the beginning of the generated synthesis sequence.
reconstr	a function that reconstructs the symbol from analysis and synthesis information. If nil, the predefined (in *LZify*) fonction will be used.
strategy	function that chooses new synthesis information according to its last evolution.
constraints	constraint function of the last generated analysis information and last generated sequence, both reversed.
equiv1	function that compares a symbole at the root of the tree with the last generated one.
equiv2	function that compares a symbole with any in the tree.

output

a list of events in the same alphabet as the analyzed sequence.

description

After building a pattern dictionary using *LZify*, *LZGenerate* may be used to generate a new sequence that imitates the statistical behaviour encoded into the dictionary. If a list of <something> had been analyzed by LZify, the result will be a new list of <something>.

At a every point of the generation, LZGenerate looks at the longest sequence of last generated elements that belongs to the LZ tree (even those that do not start exactly from the root).

It then checks the conditional probabilities associated with that pattern (or context) , then generates a new element with regard to the probability. It then adds this element to the right of the generated sequence, and iterates.

If *maxPast* in an integer, the length of the context must not exceed *maxPast*. That is to say, this limits the size of the memory of what LZGenerate has previously generated.

If, at a certain point of the generation, the length of the context is lower than *minPast*, then *LZGenerate* goes back one step before and generates another symbols until it respects the *minPast* constraint. It may go back as far as necessary. If no sequence can respect the constraint, *LZGenerate* returns *nil*.

Thanks to the *minPast* parameter, you can prevent *LZGenerate* from generating with no or little context. *minPast* = 1 is generally high enough. For *minPast* > 1, the constraint mat be too high : there may be no more possible result or only stationnary results.

If, at a certain point of the generation, the number of all the nodes of the tree that can be reached from now on (which is called the subtree generated by the present context) is lower than *minComplex*, then *LZGenerate* takes into account not only the continuation of the longest context, but also those of more little context, this in order to get out of this subtree. Thanks to the *minContext* parameter, you can prevent *LZGenerate* from getting stuck inside a little subtree.

Warning : if minPast value is set to 0 and if you specify either a positive minComplex value or a constraint, the generation algorithm will have to remember, at each generation step, a lot of information. In this case, the algorithm may need too much memory.

PSTify

PSTify *text pmin a ymin r l (type)*

Builds a pattern suffix tree modeling the text.

Inputs

text	list of anything.
Pmin	min frequency.
a	
ymin	min probability.
r	quotient threshold.
L	memory length.
type	a list containing :

a function selecting the analysis information inside a symbol.

a function selecting the synthesis information inside a symbol.

a function that reconstructs the symbol from analysis and synthesis information (will be used by *LZGenerate*).

output

a PST (Prediction Suffix Tree)

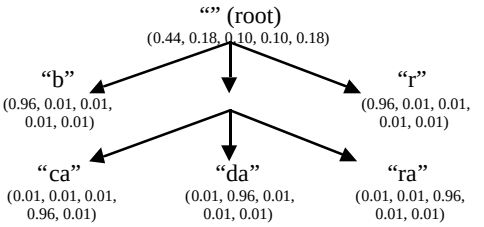
description

First, we define L to be the memory length of the PST, i.e. the maximal length of a possible string in the tree. We work out gradually through the space of all possible subsequences of length 1 through L, starting at single letter subsequences, and abstaining from further extending a subsequence whenever its empirical probability has gone below a certain threshold (Pmin), or on having reached the maximal L length boundary. The Pmin cutoff avoids an exponentially large (in L) search space. At the beginning of the search we hold a PST consisting of a single root node. Then, for each subsequence we decide to examine, we check whether there is some symbol in the alphabet for which the empirical probability of observing that symbol right after the given subsequence is non negligible, and is also significantly different (i.e. the quotient exceeds a certain threshold r) from the empirical probability of observing that same symbol right after the string obtained from deleting the leftmost letter from our subsequence. This string corresponds to the label of the direct father of the node we are currently examining (note that the father node has not necessarily been added itself to the PST at this time). Whenever these two conditions hold, the subsequence, and all necessary nodes on its path, are added to our PST.

The reason for the two step pruning (first defining all nodes to be examined, then going over each and every one of them) stems from the nature of PSTs. A leaf in a PST is deemed useless if its prediction function is identical (or almost identical) to its parent node. However, this in itself is no reason not to examine its sons further while searching for significant patterns. Therefore, it may, and does happen that consecutive inner PST nodes are almost identical.

Finally, the node prediction functions are added to the resulting PST skeleton, using the appropriate conditional empirical probability, and then these probabilities are smoothed using a standard technique so that no single symbol is absolutely impossible right after any given subsequence (even though the empirical counts may attest differently).

Here is the PST analysis of “abracadabra”, with Pmin = 0.1, r = 2, L = 10 and a minimum smoothed probability of 0.01. For each node is associated the list of probabilities that the continuation be, respectively, a, b, c, d and r.



PSTgenerate

PSTgenerate *dict length (minpast incipit1 incipit2 reconstr strategy constraints equiv1 equiv2)*

Generates a new sequence following the model of a given PST (generated by the *PSTify* function).

Inputs

dict	a PST generated by the *PSTify* function.
Length	integer, >=1. length of the sequence to be generated.
minPast	integer, >=0. minimum length of the context.
incipit1	list. a sequence of analysis symboles that will be the beginning of the generated analysis sequence.
incipit2	list. a sequence of synthesis symboles that will be the beginning of the generated synthesis sequence.
reconstr	a function that reconstructs the symbol from analysis and synthesis information. If nil, the predefined (in *PSTify*) fonction will be used.
strategy	function that chooses new synthesis information according to its last evolution.
constraints	constraint function of the last generated analysis information and last generated sequence, both reversed.
equiv1	function that compares a symbole at the root of the tree with the last generated one.
equiv2	function that compares a symbole with any in the tree.

output

a list of events in the same alphabet as the analyzed sequence.

description:

After building a pattern dictionary using PSTify, PSTGenerate may be used to generate a new sequence that imitates the statistical behaviour encoded into the dictionary.

If a list of <something> had been analyzed by PSTify, the result will be a new list of <something>.

At a every point of the generation, PSTGenerate looks at the longest sequence of last generated elements

that belongs to the PST.

It then checks the conditional probabilities associated with that pattern (or context) , then generates a new element with regard to the probability. It then adds this element to the right of the generated sequence, and iterates.

If, at a certain point of the generation, the length of the context is lower than minPast, then PSTGenerate goes back one step before and generates another symbols until it respects the minPast constraint. It may go back as far as necessary. If no sequence can respect the constraint, PSTGenerate returns nil.

Thanks to the minPast parameter, you can prevent PSTGenerate from generating with no or little context.

midi->cross

midi->cross *midi-info (legatime arpegtime releastime staccatime toltime)*

Transforms the output of a MidiFile into a pitch/duration Cross-Alphabet sequence.

Inputs

midi-info	a *MidiFile* object from a *midifile* box, or a list from a *mf-info* box.
legatime	an integer or *nil* : maximum *legato* time, in ms. if *nil* : no time constraints for *legato* filter.
arpegtime	an integer : minimum arpegio time, in ms.
releastime	an integer or *nil* : maximum release synchro time, in ms. if *nil* : infinite release-synchro.
staccatime	an integer or *nil* : maximum *staccato* time, in ms. if *nil* : no time constraints for *staccato* filter.
toltime	an integer : time tolerance of quantization, in percent.

output

a list of polyphonic slices.

description

The cross-alphabet sequence is a list containing sublists of the form : (... ((*c1 c2 ... cn*) *d*) ...)

where the *ci* are either 0 (empty canal) or a list of the form (*p1 p2 ... pm*)

where the *pi* are pitches in midicents

and *d* is a duration in ms.

Each of this sublist encodes a polyphonic slice, containing a set of canals - which are sets of superposed pitches - and lasting for a certain duration. The concatenation of all these slices is musically equal to the original midi sequence.

This representation captures the essentials of the polyphonic/rythmic structure of the midifile. It is thus convenient to be given as input to the *LZify* function if you want to build a statistical model of a polyphonic piece from a midifile.

If you use then *LZgenerate* or its variants to generate an improvisation of the piece, you'll get again a pitch/duration cross-alphabet sequence. Then you'll need a function such as *cross->chordseq* to put your data back into a musical form.

Filters are designed to simplify the alphabet of symbols and the dictionnary of patterns :

Each time a new note is activated, the *legato* filter inspects the continuation of the sequence until the release of this note or until *legatime* msec. Any note that has been activated before the new note and that is released during the inspected part will then be released just when the new note is activated. Therefore, this filter discards intermediate states where two juxtaposed notes are superimposed during a very short time.

Each time a new note is activated, the arpeggio filter inspects the continuation of the sequence until *arpegtime* msec. Any note that was due to be activated during the inspected part will in fact be activated just when the former note is activated. Thus, this filter synchronizes arpeggios.

Each time a note is released, the release synchro filter synchronizes all the following releases - until the activation of a new note or until *releastime* msec - at the date of the first release. Thus, this filter synchronizes note releases.

The *staccato* filter removes each release period, between two played periods, that lasts less than *staccatime* msec.

Finally, the duration of each symbol is quantified with the help of an OMKant library function : *makeregular*. This function accepts one particular parameter, here *toltime*, which specifies the percentage of error that is tolerated during the quantization.

listmidi->cross

listmidi->cross *midi-infos (legatime arpegtime releastime staccatime toltime)*

A version of *midi->cross* dedicated to a list of several midi files or midi informations instead of only one midi file or midi information.

cross->chordseq

cross->chordseq *cross time-coef*

Transforms a Cross-Alphabet sequence into a chord-seq.

Inputs

cross	a pitch/duration cross-alphabet sequence generated by the function *midi->cross*.
time-coeff	float number > 0.0

output

a <chord-seq> object.

see *midi->cross*.

midi->chordseq

midi->chordseq *midifile*

Converts the output of a *midifile* box, or the output of a *mf-info* box to a single *chord-seq* object. The tracks will be merged, and events occuring quasi-simultaneously will be grouped into chords with regards to the approximation parameter 'delta value' (ms) available in the preferences of OpenMusic.

chordseq->midi *chordseq*

Transforms a *chord-seq* into a *mf-info*.

input

a Chord-Seq.

output

a mf-info.

lzprint

lzprint *tree*

Print a given pattern dictionnary.

input :

tree a LZ pattern tree or a LZ continuation tree generated by the *LZify* function.

output

nil.

description

Each line in the display of the tree features one node of the tree. The indentation of the line is proportional to the deepness of the node in the tree. For a continuation tree, each line features the analysis symbol, and between brackets, the sequence of synthesis symbols associated to the branch from the root of the tree to the node.

Let's consider again the previous exemple {a,b,ab,aa,aba,abb,abba...}, and let's add for each analysis symbol a synthesis symbol. Suppose the synthesis sequence is {1, 2, 3 1, 2 3, 1 2 3, 1 2 3, 1 2 3 1...}. If we add the synthesis symbols in the tree, we get :

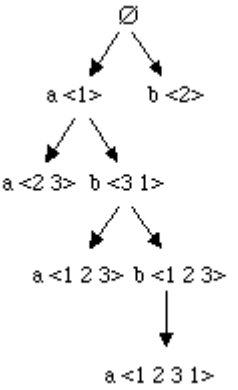


Figure 2 : The pattern tree of an analysis/synthesis example.

So the result of *Lzprint* is :

0 <nil>

a <1>

a <2 3>

b <3 1>

a <1 2 3>

b <1 2 3>

a <1 2 3 1>

b <2>

The continuation tree is an optimized representation of the pattern tree : each branch is reversed, and for each node is featured all the possible continuations with corresponding number of iterations. For each continuation is presented, inside double brackets, a table associating each possible synthesis sequence with its possible continuations. The continuation tree of our previous example is :

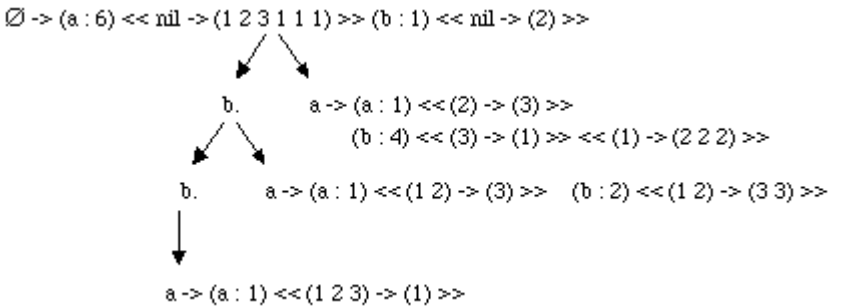


Figure 3 : The continuation tree of previous example.

Its display in the Listener by *Lzprint* is :

0 -> (a : 6) << nil -> (1 2 3 1 1 1) >> (b : 1) << nil -> (2) >>

a -> (a : 1) << (2) -> (3) >> (b : 4) << (3) -> (1) >> << (1) -> (2 2 2) >>

b.

a -> (a : 1) << (1 2) -> (3) >> (b : 2) << (1 2) -> (3 3) >>

b.

a -> (a : 1) << (1 2 3) -> (1) >>

Then *Lzprint* writes the name of the default reconstruction function that has been defined in *Lzify* and that will be used by *Lzgenerate*.

pstprint tree

Print a given PST.

input

tree — a PST generated by the *PSTify* function.

output

nil.

description

Cf. *lzprint*.

lzprintreconstr

lzprintreconstr *tree*

Indicates the *reconstr* function used by the continuation tree.

input

tree a LZ continuation tree generated by the *LZify* function.

output

nil.

lzsize

lzsize *tree*

Finds the size (number of nodes) of the tree.

input

tree a LZ continuation tree or pattern tree generated by the *LZify* function.

output

nil.

lzlength

lzlength *tree*

Finds the lengths of the longest branch of the tree.

input

tree a LZ continuation tree or pattern tree generated by the *LZify* function.

output

nil.

lzuntree

lzuntree *tree (delay)*

Appends all the patters of a given pattern dictionnary, separating one from each other by a delay.

input

tree	a LZ continuation tree or pattern tree generated by the *LZify* function.
delay	delay time between branches.

output

A sequence of patterns.

crop

crop *midi-info begin end*

Crops the Midi Info from time <begin> to <end>

Inputs

midi-info	a list given by the mf-info box
begin, end	integer in ms

output

a list in the same format than given by mf-info box (or the output of a MidiFile)

Crops the Midi data from time *begin* to *end*, and delivers a list in the same format than given by mf-info box : (midicents, onset-time(ms), duration(ms), velocity, channel)

This list can serve as input to the box *midi->cross*.

timescaler

timescaler *midi-info scaler*

Time scales the Midi info by <scaler>

Inputs

midi-info	a list given by the mf-info box
scaler	a float number > 0.0

output

a list in the same format than given by *mf-info* box (or the output of a MidiFile)

Time scales the Midi data by coefficient *scaler* and delivers a list in the same format than given by mf-info box : (midicents, onset-time(ms), duration(ms), velocity, channel)

This list can serve as input to the box *midi->cross*.

transposer

transposer *midi-info offset*

Transposes the Midi information (given by mf-info) by *offset* semitones

Inputs

midi-info	a list given by the mf-info box (or the output of a MidiFile)
offset	an integer

output

a list in the same format than given by mf-info box.

Transposes the Midi data by offsett <offset> in semitones and delivers a list in the same format than given by mf-info box : (midicents, onset-time(ms), duration(ms), velocity, channel)

This list can serve as input to the box *midi->cross*.

3 LIBRARY TUTORIAL

Three sample patches are provided with the LZ library. They will be found on the workspace window of OpenMusic, in the folder « LZ examples ». If they are not found there, please go to the finder and open the folder

OM x.x : User Library : LZ x.x :

then drag the folder « LZ examples » from there to the WorkSpace window.

Note that all the patches use MidiFiles. The first time a patch is open, OpenMusic will issue a dialog box and ask you to locate the MidiFile. The sample MidiFiles are all located in the folder

OM x.x : User Library : LZ x.x : LZ examples

Nearly all these example patches take one or several Midifiles, compute a pattern dictionary using Lzify, then compute a variant of the Midifile using Lzgenerate, and send it to a chordseq. When using several MidiFiles, the variant is really a structural interpolation between them.

Example 1 : Basic of LZ.

Example 2 : Improvement of example 1.

Example 3 : Avoiding excessive repetitions.

Example 4 : Analysis of a set of several midi files.

Example 5 : The analysis / synthesis information.

Example 6 : Information about the result of the LZ analysis.

Example 7 : The options of Lzgenerate.

Example 8 : Some examples of constraints.

new ! Pst : Basic of PST.

Pst vs Lz : Comparison between PST and LZ.

Virtual_Corea : LZ example with Corea improvisation.

4 ANNEX

See the ICMC 1999 short paper:
"Guessing the Composer's Mind: Applying Universal Prediction to Musical Style"
Gérard Assayag (Ircam) , Shlomo Dubnov (Ben Gurion University), Olivier Delerue (Ircam)
Proceedings ICMC 99, Beijing, China
File : ICMC 99 Short.pdf

See also the long paper :
"Automatic Modeling of Musical Style"
O. Lartillot, S. Dubnov, G. Assayag, G. Bejerano
File : ICMC.pdf