LFSAB1402 2018 - Oz Player

This year's project is musical. You will have to write two Oz functions. The first one, `PartitionToTimedList`, will transform a musical partition into a list of notes and chords, each associated with a duration. The second one, `Mix`, will mix multiple entry formats (such as partitions, lists of timed notes/chords and raw audio data) in order to emit an audio file encoded in the WAV format.

General instructions

Re-read this before submitting!

- The project is to be carried out in groups of two students.

- The project is to be submitted on INGInious[1] for the **06 December at 18:00** (GMT+1) at the latest. The server may not survive a sudden influx of submissions — **do not take risks, start the project right away**. Feel free to submit intermediate versions even if your project is not fully completed.

- Your project must be submitted as an archive named `NOMA1-NOMA2.zip` containing at least three files: `code.oz` contains your solution, `example.dj.oz` contains an input example for your program and `report.pdf` contains your report. If you choose the creativity bonus, you must also include the files `creation.dj.oz` and `creation.wav`. **Please respect this format**.

- Your code must call the `Projet.run` function exactly once with the content of `example.dj.oz`.

- **Mention your first names, last names and student id in your <u>code</u> and in your <u>report</u>!** Forgetting it is a silly way to lose points....

- **Strictly** observe the specifications given for functions, transformations, etc. Part of the evaluation will be done by automated tests. Given the nature of the project, it is possible to finish with code that gives a pleasant sound but does not pass the tests ...

- We will not disclose the tests (no need to ask). However, you are encouraged to write your own tests, perhaps even before you start coding[2].
  To help you do this, we provide you with a test template in the `tests.oz` file. Use `\insert 'tests.oz'` to include this file in your code and call the `Test` function — refer to the template provided for `code.oz.`

  !! You must remove the `code.oz` tests before submitting your project.

- Read carefully the **specific instructions for the report** at the end of this document.

- We will be **uncompromising on plagiarism**. It is allowed to discuss the project between groups, not to exchange code bits. The easiest way to resist temptation is to not submit yourself to it...

1. https://inginious.info.ucl.ac.be/course/FSAB1402/project
2. https://en.wikipedia.org/wiki/Test-driven_development

Evaluation

You will be evaluated on the quality of your code and report. This means that your code must not only be correct and reasonably effective, but it must also be documented, structured, readable, understandable, reusable and editable. Your report must be clear, structured and in a correct English (or French). Consider using a spell checker.

The extensions that you can add to the project can make the difference between an 18 and a 20 but will not be able to save a bad project. Focus first on what is being asked.

Troubleshooting

In case of misunderstanding or doubts about the instructions, you are invited to send any question on the Moodle forum (or Slack channel) and to send me an email (nicolas.laurent@uclouvain.be) if there is no answer within a reasonable time.

If you have any technical questions, please also address your questions on Moodle or Slack first. Similarly, you can send me an email after a reasonable time **BUT**, for me to answer, it must imperatively contain:

- A link to the Moodle question.

- A clear formulation of the question. Do not assume that all the details of the project are fresh in my head, introduce some context.

- The list of things you have tried to solve and/or diagnose your project.
    - Think in particular of all the experiments you could perform yourself using Mozart to get answers to your questions.
    - What research in the course and on the internet have you done?

- If your problem concerns behaviour observed in your code, a **minimal working example**[2b] that demonstrates your problem.

We remind you that there are several hundred of students that follow the course and that it is impossible to help all of you with the details, so the questions are reserved for the real

problems, not those that you should be able to solve by yourselves with a bit of effort (this is also part of what the project is evaluating).

2b. https://en.wikipedia.org/wiki/Minimal_working_example

Specifications

We now give the specifications for the `PartitionToTimedList`, and `Mix` functions that you will need to implement.

Do not start the implementation without reading all the directives. It is not necessary to implement the project in the order used in this document (this may be a way to divide the work).

These specifications use grammars similar to those used to define the Oz syntax. In these, we will use as primitive elements ⟨*integer*⟩, ⟨*natural*⟩, and ⟨*boolean*⟩ (whose meaning should be obvious), as well as ⟨*duration*⟩ and ⟨*factor*⟩ — positive floats, and ⟨*sample*⟩ — a float within the interval [~1.0, 1.0].

1. `PartitionToTimedList`

The signature of this function is
`{PartitionToTimedList ⟨partition⟩}` = ⟨*flat partition*⟩
This function takes a partition as argument and returns a list of notes and chords, each associated with a duration.

The format of the `Partition` argument is given by the following grammar:

/* <partition> is a list of <partition item> */
⟨*partition*⟩ : := nil | ⟨*partition item*⟩ '|' ⟨*partition*⟩

⟨*partition item*⟩ : :=
⟨*note*⟩
| ⟨*chord*⟩
| ⟨*extended note*⟩
| ⟨*extended chord*⟩
| ⟨*transformation*⟩

Notes are given in scientific notation[3]. These notes represent a raw frequency sound at maximum volume for exactly one second.

3. http://en.wikipedia.org/wiki/Scientific_pitch_notation

The note space is divided into octaves of 12 notes each. These are: C, C#, D, D, D#, E, F, F#, F#, G, G#, A, A# and B. The symbol '#' is called a hash. You will notice that there are no hash versions of E and B. C# is pronounced "C sharp" in English.

Each note is separated from the next note by a *semitone*. The B of an octave is separated from the C of the next octave by a semitone. All this implies that a note is separated by 12 semitones from its equivalents in the next octave and in the previous octave.
The syntax for designating notes in a partition is as follows. If the octave of a note is not indicated, it is assumed by default that it is the fourth octave.

⟨*note*⟩ ::=
    silence
    | ⟨*name*⟩
    | ⟨*name*⟩⟨*octave*⟩
    | ⟨*name*⟩#⟨*octave*⟩

⟨*name*⟩ ::= a | b | c | d | e | f | g
⟨*octave*⟩ ::= ⟨*natural*⟩

This notation is useful for defining a partition, but not very practical to manipulate with code. For example `a2` is an atom, but `a#2` is a tuple.

To help you, we provide you (in the code template) with the `NoteToExtended` function that allows you to switch from this simplified notation to an extended notation. You are free to change this function if necessary.

⟨*extended note*⟩ : :=
    silence(duration:⟨*duration*⟩)
    | note(
        name:⟨*name*⟩
        octave:⟨*octave*⟩
        sharp:⟨*boolean*⟩
        duration:⟨*duration*⟩
        instrument:⟨*atom*⟩)

Note that unless you choose to implement the extension on instruments, the *instrument* field will always be *none*.

A *chord* is simply a set of notes played simultaneously.
You will represent the chords by a list of notes, **whose duration must be identical**. Empty chords are allowed and are considered to have a null duration.

⟨*chord*⟩ ::= nil | ⟨n*ote*⟩ '|' ⟨*chord*⟩
⟨*extended chord*⟩ ::= nil | ⟨*extended note*⟩ '|' ⟨*extended chord*⟩

The function has to return a list of extended notes and chords composed by these notes.

⟨*extended sound*⟩ ::= ⟨*extended note*⟩ | ⟨*extended chord*⟩
⟨*flat partition*⟩ ::= nil | ⟨*extended sound*⟩ '|' ⟨*flat partition*⟩

Finally, each partition can also contain these transformations:

⟨*transformation*⟩ ::=
duration(seconds:⟨*duration*⟩ ⟨*partition*⟩)
| stretch(factor:⟨*factor*⟩ ⟨*partition*⟩)
| drone(note:⟨*note or chord*⟩ amount:⟨*natural*⟩)
| transpose(semitones:⟨*integer*⟩ ⟨*partition*⟩)

⟨*note or chord*⟩ ::= ⟨*note*⟩ | ⟨*chord*⟩ | ⟨*extended note*⟩ | ⟨*extended chord*⟩

duration(seconds:⟨*duration*⟩ ⟨*partition*⟩)

This transformation sets the duration of the partition to the specified number of seconds. It is therefore necessary to adapt the duration of each note and chord proportionally to their current duration so that the total duration becomes the one indicated in the transformation.

stretch(factor:⟨*factor*⟩ ⟨*partition*⟩)

This transformation stretches the duration of the partition by the specified factor. It is therefore necessary to extend the duration of each note and chord accordingly.

drone(note:⟨*note or chord*⟩ amount:⟨*natural*⟩)

A *drone* is a repetition of identical notes (or chords). The note or chord must be repeated as many times as the quantity indicated by `amount`.

This transformation may seem unnecessary, as it seems identical to a note or a longer chord. But we will see later that the notes are transformed into a signal. A drone introduces a discontinuity in the signal that is not present in a held note. In addition, the notes that make up the drone are likely to be individually transformed by some of the extensions, particularly the sound envelopes.

transpose(semitones:⟨*integer*⟩ ⟨*partition*⟩)

This transformation transposes the partition by a certain number of semitones upwards (positive number) or downwards (negative number). Refer to the section on notes above for more details regarding the distances in semitones between notes. For example, transposing A4 by 4 semitones upwards gives C#5 (by interval of one semitone: A4, A#4, B4, C5, C#5).

To transpose a chord, each of its constituent notes needs to be transposed.

2. `Mix`

The signature of this function is `{Mix P2T Music}` = ⟨*samples*⟩

The `P2T` argument corresponds to a function that meets the `PartitionToTimedList` specification. In practice, you will always pass `PartitionToTimedList` for this parameter. The reason for the presence of this argument is that it allows us to test your `Mix` function independently of your `PartitionToTimedList` function.

The purpose of the function is to interpret the `Music` argument and return a list of samples. A sound is a signal (a sinusoid for simple notes), but in practice it is difficult to describe a music in terms of signal combinations. Instead, the signal should be approximated by recording its values at regular intervals, a process called sampling. Each sample has a value between ~1.0 and 1.0.

The music is sampled at a frequency of 44100 Hz (i.e. 44100 samples per second are taken). To obtain a sample from a note, you must first obtain the height of the note. The height of a note is the number of semitones that separate it from A4 (the reference A at 440 Hz). If the note is below A4 (e. g. D4 or A3), the height will be negative. Given a height h, the frequency corresponding to the note is obtained by the formula (1). A series of samples a_i is then obtained using formula (2). For each note, a sequence of samples must be produced that corresponds to its length, taking into account the sampling rate of 44100 Hz. A note maintained for one second must therefore give 44100 samples. Samples resulting from a silence have a value of 0.

(1) $f = 2^{h/12} * 440\ Hz$

(2) $a_i = ½ \sin(2\pi * f * i / 44100)$

The sample list returned by `Mix` will be used to generate a WAV file (`.wav`) that can be read by any audio player.

Be careful: The final sample list that will be obtained by `Project.run` must contain only samples in the interval [~1.0, 1.0] (otherwise the function will simply refuse to process your request). However, it is acceptable for lists of intermediate samples to contain samples outside this range. The `{Mix P2T Music}` = ⟨*samples*⟩ specification is therefore too strict if you recursively use `Mix`.

The format of the `Music` argument is given by the following grammar:

⟨*music*⟩ ::= nil | ⟨*part*⟩ '|' ⟨*music*⟩

⟨*part*⟩ ::=
    samples(⟨*samples*⟩)
    | partition(⟨*partition*⟩)

| wave(⟨*file name*⟩)
| merge(⟨*musics with intensities*⟩)
| ⟨*filter*⟩

⟨*samples*⟩ ::= nil | ⟨*sample*⟩ '|' ⟨*samples*⟩

⟨*musics with intensities*⟩ ::= nil | (⟨*factor*⟩ # ⟨*music*⟩) '|' ⟨*musics with intensities*⟩

A music is a list of *parts*, which can be:

samples(⟨*samples*⟩)

Where ⟨*samples*⟩ is a list of samples such as `Mix` is supposed to return. The main advantage of this type of part is that it is easier to test your filters (see below).

partition(⟨*partition*⟩)

A partition as defined above, to be interpreted using the `P2T` argument (**do not call `PartitionToTimedList` directly!!!**) and then to be sampled.

wave(⟨*file name*⟩)

Where ⟨*file name*⟩ is the **relative** path to a WAV file.

You must use the `Project.load` function to read this file. This function will return a list of samples similar to what your own `Mix` function should return.

This format being quite complex, we only support PCM encoding at 8, 16, 24 or 32 bits. The sampling frequency should always be 44100 Hz.

Among the files provided, the `wave/animals` folder contains several audio files which you can use for tests.

merge(⟨*musics with intensities*⟩)

This operation allows you to play several parts at the same time. Each music (**not a part!** a music is a list!) is associated with an intensity between 0 and 1. The sample list that results from the interpretation of each music must be multiplied by the corresponding intensity and all these lists must be added member to member (like an addition of vectors). Lists shorter than the longest one are supplemented by silence.

Exemple of merge: `merge([0.5#Music1 0.2#Music2 0.3#Music3])`

⟨*filter*⟩

A song can also correspond to the application of one of the following filters.

Some filters impose additional conditions on their options (e.g. *low* < *high* for the *clip* filter). You should not check these conditions, but you should ensure that your examples do not use filters with invalid options.

⟨*filter*⟩ : :=
reverse(⟨*music*⟩)
| repeat(amount:⟨*integer*⟩ ⟨*music*⟩)
| loop(duration:⟨*duration*⟩ ⟨*music*⟩)
| clip(low:⟨*sample*⟩ high:⟨*sample*⟩ ⟨*music*⟩)
| echo(delay:⟨*duration*⟩ ⟨*music*⟩)
| fade(in:⟨*duration*⟩ out:⟨*duration*⟩ ⟨*music*⟩)
| cut(start:⟨*duration*⟩ end:⟨*duration*⟩ ⟨*music*⟩)

reverse(⟨*music*⟩)

Consists in playing the music backwards (inverting the list of samples). This highly advanced technique allows specialists to discover or hide satanic messages[6]. Use with moderation...

6. https://en.wikipedia.org/wiki/Backmasking#Satanic_backmasking

repeat(amount:⟨*natural*⟩ ⟨*music*⟩)

Consists in repeating the music the number of times indicated.

loop(seconds:⟨*duration*⟩ ⟨*music*⟩)

Consists in playing the music in a loop for the specified number of seconds. The last repetition of the music is truncated in order to not to exceed the indicated time.

clip(low:⟨*sample*⟩ high:⟨*sample*⟩ ⟨*music*⟩)

Consists in forcing the samples to respect a floor value (*low*) and a ceiling value (*high*). Samples outside these terminals are increased or decreased to the value of the bound. The *low* < *high* condition must be respected.

echo(delay:⟨*duration*⟩ decay:⟨*factor*⟩ ⟨*music*⟩)

Consists in introducing echo into the music. It is the same as making a *merge* between the music and a copy of it preceded by a silence whose duration is given by *delay*. The intensity of the echo is given by *decay* (a multiplicative factor to be applied to the samples of the original music).

fade(start:⟨*duration*⟩ out:⟨*duration*⟩ ⟨*music*⟩)

The fade is a technique that aims to smooth transitions between pieces of music. The *start* option gives the duration in seconds from the beginning of the song during which the intensity of the music will increase linearly between 0.0 and 1.0. The *out* option forces a decreasing intensity at the end of the music. The duration of the music must be longer than the sum of *start* and *out*.

In terms of implementation, the first sample should be multiplied by an intensity of 0.0, the last sample by an intensity of almost 1.0. For example, if *start* is a duration corresponding to 5 samples, the intensities to apply to the first 6 samples would be [0.0 0.2 0.4 0.6 0.8 1.0] (assuming that the fade-out does not start from the sample following the fade-in). If *out* is of duration equivalent to 5 samples, the intensities to be applied to the last 5 samples will be: [0.8 0.6 0.4 0.2 0.0].

cut(start:⟨*duration*⟩ finish:⟨*duration*⟩ ⟨*music*⟩)

Consists in recovering a portion of the music between the *start* and *finish* time. If the interval exceeds the size of the music, it should be completed with silence.

Extensions

Extensions are designed to improve, extend and demonstrate your implementation. They are not mandatory and do not save a poorly executed project. Do the mandatory part first. Combined with a good project, one extension can bring a two-point bonus, and two (or more) extensions can bring a three-point bonus.

Important: The extensions must be disabled in the code you submit (the code must be present, but must not be called). Otherwise, you may not pass our automated tests. Specify in your report how your extensions can be activated (we strongly suggest a variable containing a boolean).

Lissage

As you will probably notice during implementation, note-based sound synthesis creates unpleasant noises between each note. To overcome this problem, the beginning and end of the notes must be softened.

In this extension, you will model a sound envelope[9] similar to those used by synthesizers found in electrical instruments.

The simplest envelope is a trapeze. The sound increases slowly until it reaches its maximum power, continues for as long as desired, then decreases slowly until it disappears.

This envelope is like using the fade filter on all notes. You are free to use the fade time that suits you, or use another envelope for a more realistic sound (see the link on the sound envelope).

9. https://en.wikipedia.org/wiki/Synthesizer#ADSR_envelope

Instruments

This extension takes advantage of the instrument field of extended notes. For this extension, we extend the grammar of partition transformations with the *instrument* transformation:

⟨*transformation*⟩ ::= …
| instrument(name:⟨*instrument*⟩ ⟨*partition*⟩)

This transformation changes the instrument used to play the score. The default instrument is *none*. It is always the instrument closest to the note that is used. In other words

```
{PartitionToTimedNotes instrument(name:guitar instrument(name:piano a4))}
    == {PartitionToTimedNotes instrument(name:piano a4)}
```

When synthesizing a note whose instrument is not *none*, you must no longer generate the audio vector yourself but use the reference sounds provided in the `wave/instruments` folder. All instrument files have the format `<nom>_<note>.wav`. To generate the file name to read according to the note, read the Mozart/Oz documentation, especially the `VirtualString.toAtom`[10] function.

10. https://mozart.github.io/mozart-v1/doc-1.4.0/base/virtualstring.html

It is up to you to define what happens when the duration of the sample is different from that of the file. If you were to cut the audio vector, consider smoothing the cuts by fading them out.

Creativity

The purpose of this extension is to use the project's functionalities. You can transcribe a partition with several voices, or mix a few extracts in a Daft Punk style loop. Use your creativity and produce original music of a reasonable size (neither too short or too simple nor too big for INGInious).

The quality of your music will be judged on the basis of the final file and the number of transformations. In other words, your music must undergo a certain number of transformations and produce a result that does not make our ears bleed.
You can of course use other audio files than those provided with the project. In this case, do not forget to include them in the final archive.

You must submit a `creation.dj.oz` file that contains your creation and a `creation.wav` file that contains the result provided by your program.

To inspire you, this page shows what some students have produced in the past in a similar project: https://perso.uclouvain.be/guillaume.maudoux/dewplayer/

Note that you must activate the Flash plugin to be able to play the music. If you consult the page source, you can also find the `mp3` and the corresponding `.dj.oz` file in the https://perso.uclouvain.be/guillaume.maudoux/mp3/ directory (warning, the format is different, but relatively similar).

We precise that the project was slightly different that year, so it is counterproductive to try to find its solutions to be copy them. We remind you that plagiarism is severely punished, and that we have these old solutions too...

You may want to remake one of these creations for your example, but this cannot count towards the creativity bonus. If you do, please specify this in your report.

For those who would like to accomplish something similar, we suggest you look at the many sites that aggregate partitions and guitar tablatures on the net!

Complex Effects

This extension consists of adding new filters to the `Mix` function. You need to add three new filters. We suggest for example:

echo(delay:⟨*duration*⟩ decay:⟨*factor*⟩ repeat:⟨*natural*⟩ ⟨*music*⟩)

> An echo filter that allows you to chain several echoes. Each new echo is an echo of the previous one, i.e. the delays add up and the decay multiplies. For example, if repeat is 2, decay 0.9 and delay 1.0, a second echo will appear two seconds after the beginning, with an intensity multiplied by 0.81.

crossfade(seconds:⟨*duration*⟩ ⟨*music*⟩$_1$ ⟨*music*⟩$_2$)

> This filter overlays a fade-out on the first music and a fade-in on the second music. This is a technique widely used by DJs to transition from one music to another.

You are not required to implement these two filters in particular, and you will have to imagine a third one anyway. For example, you could opt for low-pass, high-pass, amplitude modulation, or if you are motivated, phaser, distortion, fuzz or wah-wah!

Example

Your code must be accompanied by an executable example in a file named `example.dj.oz`. Look at the `joy.dj.oz` file for the content.

This example is important: it ensures that your code works, even if some of our automated tests do not work.

This is an opportunity to demonstrate that the filters / transformations you have implemented really work. No need to demonstrate everything (rather spend time testing your code!).

If you have chosen the creativity bonus, you must still provide an example. Your original creation must be included in the `creation.dj.oz`.

However, it is imperative to demonstrate your extensions. If it is more appropriate, you can include these demonstrations in another file with the extension `.dj.oz`. **Specify this in your report**.

Report

Your report must include your full names and student ids.

In addition, it must contain the following sections, numbered as-is:

1. List the known limitations and problems of your program. Problems that are not mentioned will be judged more severely than problems that are mentioned. "Limitations" that result from the specifications we give you are not really limits, so don't bother to list them! (It is therefore possible to leave this section empty.)

2. Justify all non-declarative constructs you use (e. g. cells). How could we do without them and how would your program be worse off? If you do not use non-declarative constructs, mention it in one sentence.

3. Discuss your implementation choices that seem unusual. This section is intended to help us understand things that may at first glance surprise us. If you feel that your code will not surprise us, it is perfectly acceptable to leave this section blank. If it is empty and the code is not surprising, we will even be very satisfied.

4. Compute and explain the following complexities as precisely as possible, for your own code:
    a. The *duration* transformation, assuming that it applies to a score composed only of simple notes.
    b. The *merge* of several musics, assuming that these musics only contain *sample* parts.
    c. The *loop* filter, assuming it applies to a music containing only *sample* parts.

    This is an **important** part of the report, so do your best!

5. List and describe all the extensions you have made to the project. In particular, we want a clear **specification** of your extensions (take this statement as a reference).

What exactly do they do? If you have encountered particular difficulties when implementing these, you can also discuss them.

Specify where your extensions are demonstrated. In `example.dj.oz`? At what time? In a separate file?

Specify how your extensions can be activated.

Limit your report to a reasonable size. Answer the questions in a clear and understandable way without giving unrelated information! We impose a strict limit of 4 pages. This is an **upper limit**, it is quite possible that your report will be much shorter.

Tips & Tricks

- A code template is available in the archive. It showcases the expected format for the submission.

- Matlab and/or Octave can help to visualize the content of a WAV file. Simply use the command: `plot(wavread('nom_du_fichier.wav'));`.

- Start with small examples. Implement only one transformation at a time and check at each step whether your results are valid. Do test-driven development.

- To debug your project, you can use the `Show` function in addition to `Browse`, which is sometimes a little facetious. Both limit the amount of data they display. To allow more data to be displayed, you can use the following calls:

  ```
  {Property.put print print(width:1000)}
  {Property.put print print(depth:1000)}
  ```

  Also remember to use the record syntax to "annotate" your uses of `Browse` and `Show`: `{Browse partition(Partition)}`

- Remember to use Mozart's predefined functions, and in particular the functions on lists[11].

  You can search the documentation via Google by prefixing your request by "mozart oz" or "site:https://mozart.github.io". Also, use the index[12].

- Oz does not support "mixed" arithmetic between integers and floats. Remember to specify the decimal part in your floating literals (0.0 instead of 0). You can use the `IntToFloat` and `FloatToInt` functions to convert between the two types.

- Use the `Time` function imported in the template to measure the execution time taken by parts of your codes. Remember to remove these measures before submitting. Are the measurements consistent with your complexity estimations?

- Do not forget to manage chords wherever they may be present!

- The Oz compiler is very slow. We're deeply sorry about this, but it also means your time will be much better spent by carefully reasoning about your code than by waiting on the compiler to tell you what you did wrong.

11. https://mozart.github.io/mozart-v1/doc-1.4.0/base/list.html#section.records.lists
12. https://mozart.github.io/mozart-v1/doc-1.4.0/idx/index.html

Files & Functions Provided

- `Project2018.ozf`: a library that provides the following functions, to be kept in the same folder as your code, but not to be submitted with your solution.
    - `{Project.run Mix P2T Music ⟨file name⟩}` = `ok` OR `error(...)`
    - `{Project.readFile ⟨file name⟩}` = ⟨*samples*⟩ OR `error(...)`
    - `{Project.writeFile ⟨file name⟩ ⟨samples⟩}` = `ok` OR `error(...)` (normally not useful)
    - `{Project.load ⟨file name⟩}` = The oz value contained in the loaded file (normally a ⟨*music*⟩).
- `code.oz`: a template for your own solution. The only one of the provided files to be delivered (modified with your solution, of course...).
- `test.oz`: a template for your tests that already proposes some named tests (these are the same functions we will use to test your project).
- `joy.dj.oz`: an example of music to test your solution.
- `wave/`: a folder containing animal sounds (to test the loading of music from a WAV file) and instruments (for those who will implement the associated extension).

Good work :)