LINFO1104 Projet 2022 - MaestrOz

Vous avez toujours rêvé de devenir un DJ mondialement reconnu comme la Swedish Oz Mafia ? Alors le prOZjet de cette année est fait pour vous ! Pour ce projet, vous allez produire de la musique en appliquant les concepts vus en cours, et avec l'aide du meilleur langage de l'univers, aka Oz.
Vous allez devoir écrire deux fonctions en Oz. La première, `PartitionToTimedList`, permettra de transformer une partition musicale en une liste de notes et d'accords, chacun associé à une durée. La seconde, `Mix`, sera capable de mixer plusieurs formats d'entrée (dont les partitions, les listes de notes/accord chronometrés, et des données audio brutes) afin d'émettre un fichier audio au format WAV**.**

Consignes Générales

À relire avant de soumettre.

- Le projet est à réaliser par groupe de deux étudiants (pas nécessairement du même local). Si vous souhaitez le faire seul, prévoyez votre plus solide argumentation et faites-en nous part. Nous déciderons si votre justification semble justifiée ou non.

- Le projet est à remettre sur INGInious[1] pour le **25 avril à 18h00** au plus tard. Le serveur risque de ne pas survivre à un afflux soudain de soumissions — **ne prenez pas de risques, prenez de l'avance**. N'hésitez pas à soumettre des versions intermédiaires même si votre projet n'est pas complètement parachevé.

- Votre solution doit être remise sur INGInious[1]. La tâche contient trois questions **obligatoires** :
  - **Question 1: Project Code** contient votre solution.
  - **Question 2: Example of input for your program** doit contenir un exemple valide d'input pour votre programme.
  - **Question 3: Report** contient votre rapport dans le format **PDF**.

  Si vous optez pour le bonus de créativité, vous devez également inclure les fichiers `creation.dj.oz` et `creation.wav` dans les questions 4 et 5 prévues à cet effet.
  Merci de respecter ce format. Nous n'accepterons pas de soumissions envoyées par mail ou par tout autre moyen qu'INGInious.

- Votre code doit faire appel exactement une fois à la fonction `Projet.run` avec le contenu de `example.dj.oz`.

- **Mentionnez vos noms, prénoms et NOMA dans votre code et dans votre rapport!** C'est une façon idiote de perdre des points...

- Respectez **scrupuleusement** les spécifications données pour les fonctions, transformations, etc. Une partie de votre évaluation sera effectuée par des tests automatisés, hors étant donné la nature du projet, il est possible de finir avec du code qui donne un son plaisant mais ne passe pas les tests...

  Nous ne divulguerons pas les tests (pas la peine de demander). Vous êtes néanmoins encouragés à écrire vos propres tests, peut-être même avant de commencer à coder[2]. Pour vous aidez à cela, nous vous fournissons un squelette de tests dans le fichier `tests.oz`. Il faut utiliser `\insert 'tests.oz'` pour inclure ce fichier dans votre code et appeler la fonction `Test` — voyez le template fourni pour `code.oz`.

  !! Il faut supprimer les tests de `code.oz` avant de remettre votre projet.

- Lisez attentivement les **consignes spécifiques au rapport** qui sont à la fin de ce document.

- Il va sans dire que nous serons **intransigeants** sur le plagiat. Il est autorisé de discuter le projet entre groupes, pas de s'échanger des bouts de code. La façon la plus simple de résister à la tentation est de ne pas s'y soumettre...

1. https://inginious.info.ucl.ac.be/course/LINFO1104/djoz
2. https://fr.wikipedia.org/wiki/Test_driven_development

Évaluation

Vous serez évalués sur la qualité de votre code et de votre rapport. Ceci implique que votre code doit non-seulement être correct et raisonnablement efficace, mais il doit aussi être documenté, structuré, lisible, compréhensible, réutilisable et modifiable. Votre rapport doit être clair, structuré et dans un français (ou anglais) correct. Pensez à utiliser un correcteur orthographique.

Les extensions que vous pouvez apporter au projet peuvent faire la différence entre un 18 et un 20 mais ne pourront pas sauver un projet mal fait. Concentrez-vous d'abord sur ce que l'on vous demande.

Troubleshooting

En cas de non-compréhension ou de doute sur les consignes, vous êtes invités à adresser toutes questions sur le forum Moodle.

En cas de questions techniques, adressez également vos questions sur Moodle. Si votre question nécessite absolument de montrer votre code, vous pouvez l'adresser par email à francois.dekeersmaeker@uclouvain.be ET thomas.wirtgen@uclouvain.be, **MAIS**, pour espérer

une réponse, celui-ci devra impérativement contenir:

- Une formulation claire de la question. Ne supposez pas que tous les détails du projet soient frais, introduisez du contexte.

- La liste des choses que vous avez tenté pour résoudre et/ou diagnostiquer votre projet.

    - Pensez en particulier à toute les expériences que vous pourriez vous-même entreprendre à l'aide de Mozart pour obtenir des réponses à vos questions.
    - Quelles recherches dans le cours et sur le net avez vous entrepris?

- Si votre problème concerne un comportement observé dans votre code, un **exemple minimal fonctionnel**[3] qui démontre votre problème.

On rappelle que vous êtes plusieurs centaines et qu'il est impossible de tous vous aider dans le détail, à ce titre, les questions sont réservées aux véritables problèmes, pas à ceux que vous devriez être capable de régler vous-même avec un poil d'huile de coude (ça fait aussi partie de ce que le projet évalue).

3. https://fr.wikipedia.org/wiki/Exemple_minimal_fonctionnel

Spécifications

Nous donnons maintenant les spécifications pour les fonctions `PartitionToTimedList`, et `Mix` que vous allez devoir implémenter.

Ne commencez pas l'implémentation sans avoir lu tout l'énoncé. Il n'est pas forcément nécessaire d'implémenter le projet dans l'ordre utilisé dans ce document (cela peut-être une façon de diviser le travail).

Ces spécifications utilisent des grammaires semblables à celles utilisées pour définir la syntaxe de Oz. Dans celles-ci, nous utiliserons comme éléments primitifs ⟨*integer*⟩, ⟨*natural*⟩, et ⟨*boolean*⟩ (dont la signification devrait être évidente), ainsi que ⟨*duration*⟩ et ⟨*factor*⟩, des flottants positifs, et ⟨*sample*⟩, un flottant compris dans l'intervalle [~1.0, 1.0].

1. `PartitionToTimedList`

La signature de cette fonction est
`{PartitionToTimedList ⟨partition⟩}` = `⟨flat partition⟩`
Cette fonction prend une partition en argument et renvoie la partition *aplatie (flattened)*, c'est-à-dire la partition avec toutes les notes et accords, chacun associé à une durée, listés explicitement les uns après les autres.

Le format de l'argument `Partition` est donné par la grammaire suivante:

/* <partition> est une liste de <partition item> */
⟨*partition*⟩ : := nil | ⟨*partition item*⟩ *'|'* ⟨*partition*⟩

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

Les notes sont données en notation scientifique[3]. Ces notes représentent un son de fréquence pure à volume maximal durant exactement une seconde.

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

L'espace des notes est décomposé en octaves de 12 notes chacune. Celles-ci sont: C, C#, D, D#, E, F, F#, G, G#, A, A# et B. On appelle le symbole '#' une dièse. Vous constaterez qu'il n'existe pas de versions dièse de E et de B. C# est prononcé "C sharp" en anglais.

La note C correspond à ce qu'on appelle communément *Do* en Français, et ainsi de suite dans l'ordre *Do Ré Mi Fa Sol La Si*: par exemple, A correspond à *La*.

Chaque note est séparée de la note suivante par un *demi-ton* (*semitone* en anglais). Le B d'une octave est séparé du C de l'octave suivante par un demi-ton. Tout ceci implique qu'une note est séparée de 12 demi-tons de ses équivalents dans l'octave suivante et dans l'octave précédente. La syntaxe pour désigner les notes dans une partition est la suivante. Si on n'indique pas l'octave d'une note, on suppose par défaut qu'il s'agit de la quatrième octave.

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

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

Cette notation est pratique pour définir une partition, mais peu pratique à manipuler par du code, car par exemple `a2` est un atome, mais `a#2` est un tuple.

Pour vous aider, nous vous fournissons (dans le template de code) la fonction `NoteToExtended` qui permet de passer de cette notation simplifiée à une notation étendue. Vous êtes libre de modifier cette fonction au besoin.

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

Notez que sauf si vous optez pour l'implémentation de l'extension sur les instruments, le champ *instrument* vaudra toujours *none*.

Un *accord*, ou *chord* en anglais est simplement un ensemble de notes jouées simultanément. Vous représenterez les accord par une liste de notes, **dont la durée doit être identique**. Les accords vides sont autorisés et on considère qu'ils ont une durée nulle.

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

La fonction doit retourner une liste de notes étendues et d'accord composés de ces notes.

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

Finalement, chaque partition peut également contenir des 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*⟩)

Cette transformation fixe la durée de la partition au nombre de secondes indiqué. Il faut donc adapter la durée de chaque note et accord proportionnellement à leur durée actuelle pour que la durée totale devienne celle indiquée dans la transformation.
stretch(factor:⟨*factor*⟩ ⟨*partition*⟩)

Cette transformation étire la durée de la partition par le facteur indiqué. Il faut donc étirer la durée de chaque note et accord en conséquence.

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

Un *bourdon* (*drone* en anglais) est une répétition de notes (ou d'accords) identiques. Il faut répéter la note ou l'accord autant de fois que la quantité indiquée par `amount`.

Cette transformation peut sembler inutile, car elle semble identique à une note ou un accord plus long. Mais nous verrons plus tard que les notes sont transformées en signal. Un bourdon introduit une discontinuité dans le signal qui n'est pas présent dans une note maintenue. De plus, les notes qui composent le bourdon sont susceptibles à être transformée individuellement par certaines des extensions, en particulier les enveloppes sonores.

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

Cette transformation transpose la partition d'un certain nombre de demi-tons vers le haut (nombre positif) ou vers le bas (nombre négatif). Référez vous à la section sur les notes ci-dessus pour plus de détails concernant les distances en demi-tons entre les notes. Par exemple, transposer A4 de 4 demi-tons vers le haut donne C#5 (par intervalle d'un demi-ton: A4, A#4, B4, C5, C#5).

Pour transposer un accord, il suffit de transposer chacune de ses notes constituantes.

2. `Mix`

La signature de cette fonction est `{Mix P2T Music}` = ⟨*samples*⟩

L'argument `P2T` correspond à une fonction qui satisfait la spécification de `PartitionToTimedList`. En pratique, vous passerez toujours `PartitionToTimedList` pour ce paramètre. La raison de la présence de cet argument est qu'il nous permet de tester votre fonction `Mix` indépendamment de votre fonction `PartitionToTimedList`.

Le but de la fonction est d'interpréter l'argument `Music` et de retourner une liste d'échantillons (`samples` en anglais). Un son est un signal (une sinusoïde en ce qui concerne les notes simples), mais il est en pratique difficile de décrire une musique en terme de combinaisons de signaux. On choisira plutôt d'approximer le signal en enregistrant les valeurs de celui-ci à intervalles réguliers, un processus nommé échantillonnage. Chaque échantillon a une valeur comprise entre ~1.0 et 1.0.

On échantillonne la musique à une fréquence de 44100 Hz (c'est à dire que l'on prend 44100 échantillons par seconde). Pour obtenir un échantillon à partir d'une note, il faut d'abord obtenir

la *hauteur* de la note. La hauteur d'une note est le nombre de demi-tons qui la sépare de A4 (le *La* de référence à 440 Hz). Si la note est en dessous de A4 (par exemple D4 ou A3), la hauteur sera négative. Étant donné une hauteur h, la fréquence correspondante à la note s'obtient par la formule (1). On obtient ensuite une suite d'échantillons a_i en utilisant la formule (2). Pour chaque note, il faut produire une suite d'échantillons qui correspond à sa longueur en tenant compte du taux d'échantillonnage de 44100 Hz. Une note maintenue une seconde doit donc donner 44100 échantillons. Les échantillons qui résultent d'un silence valent 0.

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

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

La liste d'échantillons retournée par `Mix` sera utilisée pour générer un fichier WAV (`.wav`) lisible par n'importe quel lecteur audio.

Attention: La liste d'échantillon finale qui sera obtenue par `Project.run` ne doit contenir que des échantillons dans l'intervalle [~1.0, 1.0] (sinon la fonction refusera purement et simplement de traiter votre demande). Par contre, il est acceptable que des listes d'échantillons intermédiaires contiennent des échantillons en dehors de cet interval. La spécification `{Mix P2T Music}` = ⟨*samples*⟩ est donc trop stricte si vous faites récursivement appel à `Mix`.

Le format de l'argument `Music` est donné par la grammaire suivante:

⟨*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*⟩

Une musique est une liste de morceaux (*parts*), qui peuvent être:

samples(⟨*samples*⟩)

Où ⟨*samples*⟩ est une liste d'échantillons tel que `Mix` est supposé retourner. Le principal intérêt de ce type de morceau est de pouvoir plus facilement tester vos filtres (voir plus loin).

partition(⟨*partition*⟩)

Une partition tel que définie précédemment, à interpréter à l'aide de l'argument `P2T` (**n'appelez pas `PartitionToTimedList` directement!!**) puis à échantillonner.

wave(⟨*file name*⟩)

Où ⟨*file name*⟩ est le chemin **relatif** vers un fichier au format WAV.

Vous devez utiliser la fonction `Project.load` pour lire ce fichier. Cette fonction vous retournera une liste d'échantillons similaire à ce que votre propre fonction `Mix` doit retourner.

Ce format étant assez complexe, nous ne supportons que l'encodage PCM à 8, 16, 24 ou 32 bits. La fréquence d'échantillonnage doit toujours être de 44100 Hz.

Parmi les fichiers fournis, le dossier `wave/animals` contient plusieurs fichiers audio avec lesquels vous pouvez faire des tests.

merge(⟨*musics with intensities*⟩)

Cette opération permet de jouer plusieurs morceaux en même temps. Chaque musique (**pas un morceau**! une musique est une liste) est associé à une intensité comprise entre 0 et 1. La liste d'échantillon qui résulte de l'interprétation de chaque musique doit être multipliée par l'intensité, et toutes ces listes doivent être additionnées membre à membre (à la façon d'une addition de vecteurs). Les listes plus courtes que la plus longue sont complétées par du silence.

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

⟨*filter*⟩

Un morceau peut aussi correspondre à l'application de l'un des filtres suivants.

Certains filtres imposent des conditions supplémentaires sur leurs options (par exemple *low* < *high* pour le filtre *clip*). Vous ne devez pas vérifier ces conditions, mais vous devez faire en sorte que vos exemples ne fassent pas appel à des filtres avec des options invalides.

⟨*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*⟩)

Consiste à jouer la musique à l'envers (inverser la liste des échantillons). Cette technique hautement avancée permet aux spécialistes de découvrir ou de cacher des messages sataniques.[4] À utiliser avec modération...

4. https://fr.wikipedia.org/wiki/Backmasking#Messages_sataniques

repeat(amount:⟨*natural*⟩ ⟨*music*⟩)

Consiste à répéter la musique le nombre de fois indiqué.

loop(seconds:⟨*duration*⟩ ⟨*music*⟩)

Consiste à jouer la musique en boucle pour le nombre de secondes indiqué. On tronque la dernière répétition de la musique pour ne pas dépasser la durée indiquée.

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

Consiste à contraindre les échantillons à respecter une valeur plancher (*low*) et une valeur plafond (*high*). Les échantillons en dehors de ces bornes sont augmentés ou diminués à la valeur de la borne. La condition *low* < *high* doit être respectée.

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

Consiste à introduire de l'écho dans la musique. Cela revient au même que d'effectuer un *merge* entre la musique et une copie de celle-ci précédé d'un silence dont la durée est donnée par *delay*. L'intensité de l'écho est donnée par *decay* (un facteur multiplicatif à appliquer aux échantillons de la musique originale).

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

Le fondu (fade en anglais) est une technique qui vise à adoucir les transitions entre les morceaux de musique. L'option *start* donne la durée en secondes à partir du début de la musique pendant laquelle l'intensité de la musique va croître linéairement entre 0.0 et 1.0. L'option *out* est analogue à *start* pour la fin de la musique, et représente la durée en secondes pendant laquelle l'intensité de la musique va décroître linéairement entre 1.0 et 0.0 à la fin de la musique. La durée de la musique doit être supérieure à la somme de *start* et *out*.

En terme d'implémentation, pour une croissance linéaire, le premier échantillon devrait être multiplié par une intensité de 0.0, le dernier échantillon par une intensité de presque 1.0, et vice-versa pour une décroissance. Par exemple, si *start* est de durée correspondante à 5 échantillons, les intensités à appliquer aux 6 premiers échantillons seraient de [0.0 0.2 0.4 0.6 0.8 1.0] (en supposant que le fade-out ne démarre pas dès l'échantillon suivant le fade-in). Si

out est de durée équivalente à 5 échantillons, les intensités à appliquer aux 5 derniers échantillons seront: [0.8 0.6 0.4 0.2 0.0].

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

Consiste à récupérer une portion de la musique située entre le temps *start* et *finish*. Si l'intervalle excède la taille de la musique, il faut la compléter avec du silence.

Extensions

Les extensions visent à améliorer, étendre et mettre en œuvre des aspects spécifiques de votre implémentation. Elles ne sont pas obligatoires et ne permettent pas de sauver un projet mal réalisé. Faites d'abord ce qu'on vous demande. Combinées avec un bon projet, une extension peut apporter un bonus de deux points, et deux extensions (ou plus) un bonus de trois points.

Important: Les extensions doivent être désactivées dans le code que vous nous remettez (le code doit être présent, mais ne doit pas être appelé). Autrement, vous risqueriez de ne pas passer nos tests automatisés. Précisez dans votre rapport comment l'on peut activer vos extensions (on suggère fortement une variable contenant un booléen).

Lissage

Comme vous le remarquerez sûrement lors de l'implémentation, la synthèse de sons sur base de notes crée des bruits désagréables entre chaque note. Pour palier ce problème, il faut adoucir le début et la fin des notes. Si vous choisissez cette extension, vous allez modéliser une enveloppe sonore[5] semblable à celles utilisées par les synthétiseurs que l'on trouve dans les instruments électriques.

L'enveloppe la plus simple est un trapèze. Le son augmente doucement jusqu'à sa puissance maximale, continue aussi longtemps que désiré, puis diminue doucement jusqu'à disparaître. Cette enveloppe revient à utiliser le filtre de fondu sur toutes les notes. Vous êtes libres d'utiliser la durée de fondu qui vous convient, ou d'utiliser une autre enveloppe pour un son plus réaliste (voir le lien sur l'enveloppe sonore).

5. http://fr.wikipedia.org/wiki/Enveloppe_sonore

Instruments

Cette extension met à profit le champ instrument des notes étendues. Pour cette extension, on étend la grammaire des transformations de partitions avec la transformation *instrument*:

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

Cette transformation change l'instrument utilisé pour jouer la partition. L'instrument par défaut est *none*. Dans le cas ou plusieurs instruments sont spécifiés, c'est toujours l'instrument situé au plus près de la note qui est utilisé. Par exemple, nous avons l'égalité suivante:

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

Lors de la synthèse d'une note dont l'instrument n'est pas *none*, il ne faut plus générer vous même le vecteur audio mais utiliser les sons de référence fournis dans le dossier `wave/instruments`. Tous les fichiers d'instrument ont le format `<nom>_<note>.wav`. Pour générer le nom de fichier à lire en fonction de la note, allez lire la documentation de Mozart/Oz, et plus spécialement la fonction `VirtualString.toAtom`[6].

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

À vous de définir ce qui se passe quand la durée de l'échantillon est différente de celle du fichier. Si vous deviez couper le vecteur audio, considérez la possibilité de lisser les coupures par un fondu.

Créativité

Le but de cette extension est de mettre en œuvre les fonctionnalités du projet. Vous pouvez transcrire une partition à plusieurs voix, ou mixer quelques extraits en boucle à la Daft Punk. Usez de votre créativité et produisez une musique originale d'une taille raisonnable (ni trop courte ou trop simple, ni de taille à faire tomber INGInious).

La qualité de votre musique sera jugée sur base du fichier final et sur le nombre de transformations. Autrement dit, il faut que votre musique comporte un certain nombre de transformations et produise un résultat qui ne déchire pas les oreilles.
Vous pouvez bien entendu utiliser d'autres fichiers audio que ceux fournis avec le projet. Dans ce cas, n'oubliez pas de les remettre dans l'archive finale.

Vous devez remettre un fichier `creation.dj.oz` qui contient votre création et un fichier `creation.wav` qui contient le résultat fourni par votre programme.

Pour vous inspirer, cette page démontre ce que certains étudiants ont produit par le passé lors d'un projet similaire: https://perso.uclouvain.be/guillaume.maudoux/dewplayer/

Notez qu'il faut activer le plugin Flash pour pouvoir jouer les musiques. Si vous consultez la source de la page, vous pourrez aussi retrouver le `mp3` et le fichier `.dj.oz` correspondant dans le répertoire https://perso.uclouvain.be/guillaume.maudoux/mp3/ (attention, le format est différent, mais relativement similaire).

On précise bien que le projet était légèrement différent cette année là, il est donc contre-productif d'essayer de retrouver ces solutions pour s'en inspirer. On rappelle que le plagiat est sévèrement puni, et que nous avons ces anciennes solutions aussi…

Vous pourriez toutefois décider de porter une de ces créations pour votre exemple, mais cela ne peut pas compter pour le bonus de créativité. Si vous le faites, merci de le préciser dans votre rapport.

Pour ceux qui voudraient accomplir quelque chose de similaire, on vous suggère de regarder du côté des nombreux sites qui agrègent des partitions et des tablatures de guitare sur le net!

Effets Complexes

Cette extension consiste à ajouter des filtres à la fonction `Mix`. Il vous faut ajouter trois nouveau filtres. Nous suggérons par exemple:

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

> Un filtre d'écho qui permet d'enchainer plusieurs écho. Chaque nouvel écho est un écho du précédent, c'est à dire que les délais s'additionnent et le decay se multiplie. Par exemple, si repeat vaut *2*, decay *0.9* et delay 1.0, un second écho apparaîtra deux secondes après le début, avec une intensité multipliée par 0.81.

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

> Ce filtre superpose un fondu sortant (fade-out) sur la première musique et un fondu entrant (fade-in) sur la seconde musique. C'est une technique très utilisée par les DJ pour transitionner d'une musique vers une autre.

Vous n'êtes pas obligé d'implémenter ces deux filtres en particulier, et vous devrez de toute façon en imaginer un troisième. Vous pourriez par exemple opter pour des filtre passe-bas, passe-haut, de la modulation d'amplitude, ou si vous êtes motivés, du phaser, de la distorsion, du fuzz ou du wah-wah!

Exemple

Votre code doit être accompagné d'un exemple exécutable dans un fichier nommé `example.dj.oz`. Imitez le fichier `joy.dj.oz` pour le contenu.

Cet exemple est important: il permet de nous assurer que votre code fonctionne, et cela même si certains de nos tests automatisés ne fonctionnent pas.

C'est l'occasion de démontrer que les filtres / transformations que vous avez implémenté fonctionnent réellement. Pas besoin de tout démontrer (passez plutôt votre temps à tester votre code!).

Si vous avez choisi le bonus de créativité, vous devez tout de même fournir un exemple. Votre création originale doit quand à elle être inclue dans le fichier `creation.dj.oz`.

Pas contre, il est impératif de démontrer vos extensions. Si c'était plus opportun, vous pouvez inclure ces démonstrations dans un autre fichier avec l'extension `.dj.oz`. **Précisez le dans votre rapport.**

Rapport

Votre rapport doit mentionner vos noms, prénoms et noma.

En outre il doit contenir les sections suivantes, numérotées tel quel:

1. Listez les limitations et problèmes connus de votre programme. Les problèmes non répertoriés seront jugés plus sévèrement que les problèmes répertoriés. Des "limites" qui découlent des spécification que nous vous donnons n'en sont pas vraiment, pas la peine de les lister! (Il est donc possible que cette section soit vide.)

2. Justifiez toute les constructions non-déclaratives que vous utilisez (par exemple les cellules). Comment pourrait-on s'en passer et en quoi votre programme serait-il pire? Si vous n'utilisez pas de constructions non-déclaratives, mentionnez le en une phrase.

3. Discutez vos choix d'implémentation qui vous paraissent surprenant. Cette section est destiné à nous faire comprendre des choses qui pourraient de prime abord nous surprendre. Si vous estimez que votre code ne risque pas de nous surprendre, il est parfaitement acceptable de laisser cette section vide. Si elle est vide et que le code n'est de fait pas surprenant, on sera même très satisfait.

4. Listez et décrivez toutes les extensions que vous avez apporté au projet. En particulier, nous voulons d'abord une **spécification claire** de vos extensions (prenez le présent énoncé comme référence). Que font-elles, exactement? Si vous avez rencontré des difficultés d'implémentation particulières en implémentant celles-ci, vous pouvez également les discuter.

   Précisez où vos extensions sont démontrées. Dans `example.dj.oz`? A quel moment? Dans un fichier séparé?

   Précisez comment l'on peut activer vos extensions.

Limitez votre rapport à une taille raisonnable. Répondez aux questions de façon claire et compréhensible, mais sans raconter autre chose! Nous imposons une limite stricte de 4 pages. C'est une **limite supérieure**, il est fort possible que votre rapport soit beaucoup plus court.

Trucs & Conseils

- Un squelette de code est disponible dans l'archive. Il démontre le format attendu pour la soumission.

- Un script Python est fourni pour vous aider à visualiser le contenu d'un fichier WAV. Vous pouvez l'appeler avec la commande suivante:
  ```
  python3 plot_wave.py nom_du_fichier.wav
  ```
  (**Attention**, ce script nécessite les packages Python `scipy` et `matplotlib`, que vous pouvez installer à l'aide de `pip`.)

- Commencez par de petits exemples. Implémentez une seule transformation à la fois et vérifiez à chaque étape si vos résultats sont valides. Faites du test-driven development.

- Pour débugger votre projet, vous pouvez utiliser la fonction `Show` en outre de `Browse`, qui est des fois un petit peu facétieux. Aussi bien l'un que l'autre limite la quantité de données qu'ils affichent. Pour permettre l'affichage de plus de données, vous pouvez utiliser les appels suivants:

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

  Pensez également à utiliser la syntaxe des records pour "annoter" vos browse et show: `{Browse partition(Partition)}`

- Pensez à utiliser les fonctions prédéfinies de Mozart, et en particulier les fonctions sur les listes[7]. Vous pouvez faire des recherches dans la documentation via Google en préfixant votre requête par "mozart oz" ou "site:https://mozart.github.io". Pensez aussi à consulter l'index[8].
- Oz ne supporte pas l'arithmétique "mixte" entre entiers et flottants. Pensez à spécifier la partie décimale dans vos litéraux flottants (0.0 plutôt que 0). Vous pouvez utiliser les fonctions `IntToFloat` et `FloatToInt` pour convertir entre les deux types.

- Utilisez la fonction `Time` importée dans le template que l'on vous fournit pour mesurer le temps d'exécution pris par des parties de vos codes. N'oubliez pas de supprimer ces mesures avant de soumettre. Est-ce que les mesures sont consistantes avec votre estimation de complexité?

- N'oubliez pas de gérer les accords partout où ils peuvent apparaître!

- Le compilateur Oz est très lent. Nous en sommes profondément désolés, mais cela signifie aussi que votre temps sera mieux investi à raisonner sur votre code plutôt qu'à attendre que le compilateur vous dise là ou vous vous êtes trompés.

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

Fichiers & Fonctions Fournis

- `Project2022.ozf`: une librairie qui fournit les fonctions suivantes, à conserver dans le même dossier que votre code, mais à ne pas soumettre avec votre 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(...)` (normalement inutile)
    - `{Project.load ⟨file name⟩}` = La valeur oz contenue dans le fichier chargé (normalement une ⟨*music*⟩).
- `code.oz`: un template pour votre propre solution. Le seul des fichiers fourni à remettre (modifié avec votre solution, bien sur…).
- `test.oz`: un template pour vos tests qui propose déjà des noms de tests à effectuer (ce sont les mêmes fonctions que nous utiliserons pour tester vos projets).
- `joy.dj.oz`: un exemple de musique pour tester votre solution.
- `wave/`: un dossier contenant d'une part des sons d'animaux (afin de tester le mixage de musique issue d'un fichier WAV), d'autre part des instruments (pour ceux qui réalisent l'extension associée).
- `plot_wave.py`: un script Python pour afficher le contenu d'un fichier WAV.
