Xavier Lambein 54621300

**Rapport : DJ'Oz**

LFSAB1402

Victor Lecomte 65531300

# 1 Structure et Conception

## 1.1 Conception générale

Pour plus de clareté, nous avons choisi de diviser notre code source en quatre fichiers différents, correspondant à quatre fonctions ou aspects particuliers :

— `code.oz`: rassemble les autres fichers `.oz` et exécute `Project.run` ;

— `interprete.oz`: définit la fonction `Interprete`, servant à transformer une partition en liste d'échantillons ;

— `mix.oz`: définit la fonction `Mix`, servant à transformer une liste d'échantillons et d'effets en vecteur audio ;

— `envelopes.oz`: définit les différentes enveloppes que le programme propose d'appliquer aux échantillons (plus de détails sur les enveloppes dans la section 4.1).

Comme demandé par l'énoncé du projet, le code que nous avons écrit utilise exclusivement le paradigme fonctionnel. Aucune entorse n'a été nécessaire pour implémenter des fonctions qui nous semblaient à la fois simple et efficaces. Cependant, notre programme ne présente aucune gestion des erreurs : sans exceptions, celles-ci auraient diminué significativement la clareté de notre implémentation.

## 1.2 Fonction `Interprete`

Le coeur de la fonction `Interprete` est la sous-fonction `InterpreteRecursive`. Celle-ci analyse une partition et produit un vecteur d'échantillons. Son corps n'est rien d'autre qu'un grand `case` qui traite les suites de partitions (par récursion), les transformations et les notes.

Les paramètres de `InterpreteRecursive` sont les suivants :

— `Score` : une partition à analyser ;

— `Mod` : la composée des transformations à appliquer (voir ci-dessous) ;

— `Next` : une liste d'échantillons à ajouter à la fin de la liste produite par `Score` (détails ci-dessous).

Les transformations sont effectuées par *higher-order programming* : lorsqu'une certaine transformation est traitée par `InterpreteRecursive`, elle produit une fonction composable à partir des paramètres de la transformation via une fonction `Make`*`NomDeLaTransformation`*.

Ensuite, la sous-partition sur laquelle s'applique la transformation est à son tour traitée via `InterpreteRecursive` ; cependant, la fonction composable produite ci-dessus est passée dans l'argument `Mod` afin de « Diffuser » la transformation aux sous-partitions. Dans le cas où `Mod` aurait déjà été définie plus haut, la nouvelle transformation est composée aux précédentes.

Enfin, lorsque `InterpreteRecursive` en arrive à traiter une note, il la convertit en échantillon et applique sur celui-ci la fonction `Mod`, c'est-à-dire la composée des transformations.

Afin d'obtenir une liste unique de tous les échantillons produits par des appels récursifs successifs, nous avons recours au paramètre `Next` de `InterpreteRecursive`. Voilà comment nous aurions procédé sans : lorsqu'une liste est traitée dans le `case`, il est nécessaire de faire un appel récursive à la tête de liste, puis à la queue, puis d'utiliser `Append` sur les deux listes d'échantillons produites afin d'obtenir une unique liste continue ; cela revient à remplacer `nil` à la fin de la première liste par la seconde liste.

Cependant, étant donné que nous parcourons de toute façon ces listes au moins une fois pour créer les échantillons, plutôt que d'utiliser `Append` pour les combiner, nous passons, lors de l'appel à `InterpreteRecursive` sur la tête, un argument `Next` qui contient la liste d'échantillons produits par l'appel à `InterpreteRecursive` sur la queue. Ensuite, à l'intérieur de `InterpreteRecursive`, plutôt que de terminer une liste d'échantillons par `|nil`, nous la terminons par `|Next`. Ainsi, les listes sont combinées sans allonger le temps d'exécution par des appels à `Append`.

## 1.3 Fonction Mix

Bien que plus compliquée, la fonction `Mix` fonctionne sur le même principe que `Interprete` : il s'agit d'un `case` sur une musique/liste de morceaux (fonction `MusicToAV`). Chaque morceau est analysé dans un autre `case` et traité différemment selon qu'il s'agit d'une voix, d'une partition, d'un fichier wave, d'un merge ou d'un filtre (fonction `PieceToAV`). De nouveau, la plupart des sous-fonctions de `Mix` utilisent un paramètre Next, pour les mêmes raisons que dans `InterpreteRecursive`.

L'application des filtres, cependant, ne se fait absolument de la même manière que les transformations dans `InterpreteRecursive`. La raison est qu'un filtre nécessite généralement d'appliquer un traitement sur tout un vecteur audio à la fois, et non pas élément par élément. Nous avons donc une fonction définie pour chaque filtre, à laquelle nous fournissons un vecteur audio se terminant par `nil` et le vecteur suivant qui doit venir dans la musique (le paramètre `Next`). Le filtre applique son traitement sur le vecteur tout entier en le parcourant puis retourne un nouveau vecteur audio.

Contrairement aux transformations, les fonctions de filtre sont généralement très différentes les unes des autres, mais elles fonctionnent toutes selon un même sque-

lette : lors d'un appel récursif au filtre effectué sur tout le vecteur, une transformation est appliquée à chaque élément du vecteur. Cette transformation dépend de paramètres déterminés par le vecteur dans son entièreté, ou simplement par les éléments précédents ; c'est la raison pour laquelle les filtres ne fonctionnent pas comme les transformations. Enfin, chaque élement traité est ajouté ajouté à une nouvelle liste, terminant non plus par `nil` mais par `Next`.

Pour terminer, abordons la partie plus « physique » de `Mix` : la génération d'un vecteur audio à partir d'échantillons (fonction `SampleToAV`). La fonction agit de trois façons différentes selon que l'échantillon est un silence, une note pure ou une note d'un instrument. Dans le cas d'un silence, la elle génère simplement un vecteur de zéros de la longueur voulue. Dans le cas d'une note pure, une sinusoïde de la fréquence de l'échantillon est produite, puis une enveloppe est appliquée dessus (voir sections 1.4 et 4.1 pour plus de détails sur les enveloppes). Enfin, si un instrument est donné dans l'échantillon, plutôt que de générer une sinusoïde, le programme utilise le nom de l'instrument et la hauteur de la note pour récupérer le fichier wav correspondant dans `./wave/instruments/` et le convertir en vecteur audio ; ensuite, le vecteur est tronqué et un fondu en fermeture est appliqué dessus.

### 1.4 Implémentation des enveloppes

## 2 Limitations

De loin la plus grande limitation du programme est la lenteure due au code sur lequel nous n'avons pas de contrôle : l'implémentation de Mozart 2.

## 3 Complexité

## 4 Extensions

Comme extensions, nous avons choisi d'ajouter des enveloppes sonores aux échantillons, la gestion des instruments, et une composition personnelle.

### 4.1 Enveloppes sonores

Afin de d'éviter des sauts désagréables entre les notes, et d'obtenir une qualité sonore plus lisse, nous avons décidé d'utiliser des enveloppes sonores, qui adoucissent le début et la fin des échantillons.

Dans le code, nous avons défini ces enveloppes comme des fonctions qui renvoient un facteur de volume pour chacune des positions de l'échantillon. Plus de détails sur l'implémentation se trouvent dans la section 1.

Nous avons essayé plusieurs types d'enveloppes :

— **Enveloppe en trapèze :** cette enveloppe prend deux paramètres : *attack* et *release*.

Au début de l'échantillon, le volume va augmenter linéairement pendant un temps *attack* avant d'atteindre son niveau de régime, puis à la fin il va diminuer linéairement pendant un temps *release* jusqu'à un volume nul.

Cette enveloppe est simple et permet de se débarasser du bruit de coupure, mais elle est assez peu paramétrable. Elle est implémentée par la fonction `EnvTrapezoid`.

— **Enveloppe ADSR :** analogue à la méthode du trapèze, elle accepte les quatre paramètres *attack*, *decay*, *sustain* et *release*.

Comme pour le trapèze, au début de l'échantillon, le volume augmente linéairement pendant un temps *attack* jusqu'à atteindre un volume maximal. Mais après, le volume décroit de nouveau linéairement, pendant un temps *decay*, avant d'attendre le niveau de régime *sustain*, inférieur au niveau maximal. À la fin, le volume va diminuer en un temps *release* comme pour le trapèze.

Cette enveloppe a plus de liberté dans l'attaque initiale, ce qui permet d'imiter avec plus de précision des instruments de musique. Elle est implémentée par la fonction `EnvADSR`.

— **Enveloppe en hyperbole :** Cette méthode est notre invention. Nous trouvions que les autres méthodes n'étaient pas assez souples : en effet, pour qu'elles puissent gérer les échantillons courts sans saut, il faut que les paramètres *attack* et *release* soient petits, ce qui implique un son assez dur, même sur les échantillons plus longs.

Pour éviter cela, il nous fallait donc une méthode qui s'adapte à la taille de l'échantillon, tout en gardant une dureté équivalente quelle qu'en soit la taille. Nous avons donc décidé de modéliser l'enveloppe comme un produit de deux hyperboles :

- l'une croissante, fixée à 0 au début et tendant vers 1 à la fin,
- l'autre décroissante, fixée à 0 à la fin et tendant vers 1 au début.

Elle s'exprime donc comme :

$$\text{Volume} = \frac{x}{x - \text{att}} \times \frac{L - x}{L - x + \text{att}}$$

où $x$ est la position, $L$ la longueur de l'échantillon et att un temps d'attaque.

Le grand avantage de cette enveloppe est que le domaine n'est pas séparé en sous-intervalles, ce qui évite des sauts non-intentionnels.

Un effet secondaire de cette méthode a été de diminuer le volume des courts échantillons quand leur longueur est trop proche du paramètre att. Nous avons résolu cela en multipliant l'expression par un facteur qui assure que l'amplitude au milieu de l'échantillon vaille 1.

C'est l'enveloppe que nous avons utilisé pour adoucir nos propres sons. Elle est implémentée par la fonction `EnvHyperbola`.

## 4.2 Instruments

## 4.3 Composition