1. Limitations du programme

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.)

- Silence dans accord:
    - Lorsqu'il y a un silence(duration: D) dans un accord (Chord), le programme va crasher. En effet, comme expliqué dans les choix surprenants, nous ne faisons pas appel à la fonction qui sample une note pour chaque note de l'accord. Cette-dernière fonction traite toutefois les silences.
- Les autres limites ne nous sont pas connues

2. Justifications des constuctions non-déclaratives

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.

Merge

La fonction Merge (ci-dessus) utilise une cellule, qui est un élément non déclaratif. Pour justifier ce choix, réfléchissons à comment notre implémentation de Merge fonctionne.
La fonction Merge est appelée comme suit:
merge[A#M1 B#M2 C#M3 …]
L'objectif est de sommer M1, M2, M3, Mn, multipliés par leur facteur respectif. Pour ce faire, nous avons défini la fonction {SumList L1 I1 L2 I2}. Cette fonction fait l'addition de deux listes L1 et L2, élément par élémént, multipliés par leur facteur respectif: I1 et I2.

Nous fonctionnons donc comme suit:
Merge crée une cellule nommée First, cette cellulle contient une liste vide.
On parcourt la liste [A1#M1 A2#M2 A3#M3 …], et fait l'opération {Sumlist @First 1.0 Mn An}. Le résultat de cette opération donne l'addition de @First et l'élément suivant de la liste. Ce résultat est stocké dans la cellule First.

Voici ce que contient la cellule First lors des 3 premières étapes:
@First = []
@First = [A1M1] -- A1M1 contient chaque élémént de M1 multiplié par A1
@First = [A1M1 + A2M2] (on considère ici la somme vectorielle)

@First = [A1M1 + A2M2 + A3M3]

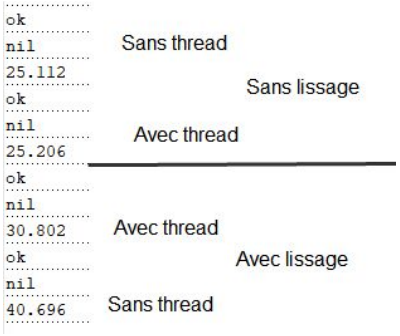
Le choix de cette implémentation facilite grandement l'écriture du code. Nous aurions pu utiliser un accumulateur qui contiendrait la somme de tous les éléments précédents. Toutefois, nous ne voyons pas d'avantage particulier à ce fonctionnement, et cela ne ferait qu'alourdir le code.

Normaliser

La fonction normaliser sert à linéairement borner un ensemble de sample entre les valeurs -1 et 1. Pour ce faire, il faut: dans un premier temps, parcourir toute la liste (FindHigh) afin de trouver l'élément donc la valeur est la plus grande, et dans un second temps, diviser chaque élément de la liste initiale par ce facteur si sa valeur absolue est supérieure à 1.
L'utilisation de cellules se fait dans la première partie: FindHigh.
Nous avons décidé, lorsque nous parcourons la liste, de stocker la plus grosse, et plus petite valeur que l'on trouve dans une cellule. Ainsi, en parcourant la liste, nous comparons chaque élément à Highest et Smallest, les plus gros et plus petits éléments trouvés jusqu'alors, et les modifions si nécessaire.
Une fois de plus, un argument contenant cette valeur aurait très facilement pu éviter cette implémentation de cellules. Toutefois, nous voulons faire usage de Threads en normalisant: un thread se charge de trouver la plus grosse valeur, pendant que l'autre se charge de diviser chaque élément par cette valeur, pas encore assignée. Dans notre cas, pour une raison qui nous est incomprise, le thread ne fonctionne pas lorsque nous faisons la seconde méthode. Nous avons donc opté pour une implémentation non-déclarative.

Threads

Pour le lissage, ainsi que pour la fonction Normaliser, qui borne linéairement toute liste d'échantillons entre -1 et 1, nous avons décidé de faire usage de threads. Ce choix a été fait pour rendre le code plus rapide. Ne pas utiliser de thread ne change absolument rien à l'éxecution du code, si ce n'est la vitesse d'éxécution.
L'éxecution de thread <s> end revient à juste exécuter <s> dans le même thread, seulement, nous avons remarqué que c'est plus rapide.
Pour justifier nos propos, nous avons implémenté le lissage sous deux formes différentes: une où pour chaque note on appelle {Fade 0.1 0.1 Note}, et un autre où pour chaque note on appelle thread {Fade 0.1 0.1 Note} end. La différence est flagrante: le lissage s'effecue littéralement deux fois plus vite.



3. Choix d'implémentation surprenants

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.

Un choix surprenant est l'existence de nos fonctions {IsNote}, {IsChord}, {IsTrans},... qui retournent un booléen si l'argument qui leur est donné est de la nature concernée. l'existence de ces fonctions peut paraître inutile, en effet, dans PartitionToTimedList, chaque élément parcourt une suite de if… elseif.. elseif… qui associe l'élément à son type. Pour ensuite seulement agir sur l'élément. Ce choix paraît le plus surprenant pour la fonction {IsTrans}, qui fait un pattern matching avec chaque type de record associé à une transformation et renvoie true si le pattern a matché. Lorsque IsTrans à renvoyé true, PartitionToTimedList appelle la fonction {Transform }, qui à son tour fait le même pattern matching, mais effectue la transformation plutôt que de renvoyer un booléen.
Nous opérons ainsi pour faciliter la clarté du code, quitte à effectuer un pattern matching plusieurs fois sur le même élément.
Un second choix surprenant peut-être vu dans notre fonction SampledChord. En effet nous n'utilisons pas SempledNote sur chaque note pour ensuite faire la moyenne des échantillons de chaque note comme cela pourrait être attendu. Au lieu de cela nous faisons la moyenne des sinus de chaque note.

4. Complexités dans le code

Etablissez et expliquez les complexités suivantes le plus précisément possible, pour votre propre code:
a. La transformation duration, en supposant qu'elle s'applique sur une partition composée uniquement de notes simples.

Dans la fonction Duration, non faisons appel à deux autres fonctions, GetDuration et Stretch. Examinons donc la complexité de ces deux fonctions.

GetDuration : nous parcourons la liste de notes, la complexité est donc O(n) avec n la taille de la partition donnée en argument.

Stretch :Dans le cas où nous avons que des notes, nous parcourons également la liste, et à chaque parcours nous rappelons la fonction. La compexité est donc O(n)
Nous avons alors une complexité de O(n) pour Duration puisque les deux fonctions qu'elle utilise ont une complexité de O(n).

b. Le merge de plusieurs musiques, en supposant que ces musiques ne contiennent que des morceaux de type samples.

La complexité de Merge dépend de Sumlist et de Mix.

Sumlist fait appel à Recurs qui est de complexité O(n) avec n la taille de la plus grande liste donné en argument. Merge fait ensuite m appel à la fonction sumliste, avec m la taille de la liste donnée à Merge.

L'appel à la fonction Mix est ce cas-ci de complexité O(1) care les morceaux sont composés uniquement de samples.

La complexité de Merge est donc $O(n^2)$.

c. Le filtre loop, en supposant qu'il s'applique sur une musique ne contenant que des morceaux de type samples.

La complexité de Loop x

a. Duration
b. Merge
c. Loop

5. Extentions apportées au projet