# LFSAB1402 - Informatique 2
# Rapport de projet : Oz Player

Théophile Schwaiger - 66401700
theophile.schwaiger@student.uclouvain.be
Lionel Boils - 32421700
lionel.boils@student.uclouvain.be

Louvain-La-Neuve
Le 6 décembre 2018

# 1 Rapport de projet : Oz Player

## 1.1 Limitations et problèmes connus de notre programme

Dans la première partie du programme (la fonction `PartitionToTimedList`), les 3 premières transformations ne nous ont pas posé de problèmes et nous pensons qu'elles répondent aux spécifications. Nous devons cependant admettre que ces 3 fonctions ne sont pas les plus rapides possibles. En effet, l'algorithme que nous avons codé peut sans doutes être amélioré pour devenir plus performant.
La 4 ème transformation (`transpose`), répond aux spécification mais est très lente. Nous lui admettons donc une limite qui est son temps d'exécution. Cela en fait un énorme problème. Il nous a dés lors été très difficile de faire des tests avec cette fonction puisque le temps d'attente devenait très long dés lors que nous faisions plusieurs tests à la suite.

Dans la deuxième partie du programme (la fonction `Mix`), nous avons connu un problème pour la fonction `filter`. Sur les 7 filtres, nous sommes parvenu à n'en coder que 5. Pour `reverse et repeat`, nous avons utilisé la récursion et nous ne pensons pas que notre programme pourrait être plus rapide. Pour le filtre `loop`, nous pensons que le programme pourrait être plus rapide et nous admettons surtout qu'il est limité à certains cas "faciles" puisque certains de nos tests utilisant des cas aux limites n'ont pas fonctionné, comme par exemple le cas d'une liste nulle passée en argument.
Le filtre `cut` fonctionnait lui aussi mais pas pour tous les cas, nottament les cas plus difficiles que nous avons essayé comme le fait de couper avec un point de départ se trouvant après le point de fin. Notre dernier filtre est `clip`, il répond aux spécifications et nous ne lui avons pas trouvé de problèmes. Nous n'avons cependant pas eu le temps de faire beaucoup de tests sur cette fonction.

## 1.2 Constructions non-déclaratives

Dans notre programme, nous utilisons des constructions non-déclaratives, nottament les cellules. Bien que nous les ayons utilisé en majeure partie dans la première fonction `PartitionToTimedList`, le fait de pas les utiliser aurait rendu notre programme beaucoup plus complexe à coder. En effet, les cellules nous ont permis de rendre notre programme plus dynamique et d'économiser énormément de temps. Prenons l'exemple de la fonction `transpose` pour laquelle nous avons utilisé des cellules. Ces dernières nous ont permis de transposer demi-ton par demi-ton tout en gardant en mémoire la note déja transposée en partie. Cela n'a été possible que grâce aux cellules qui ont rajouté un caractère dynamique à la fonction.
Si nous avions du écrire cette fonction sans constructions non-déclaratives, notre programme aurait eu un temps d'exécution bien moins long et il aurait surtout été bien plus difficile de le coder. Le programme aurait du recréer une nouvelle variables à chaque fois que nous avons modofié la valeur d'une cellule (par exemple). Les cellules coutent beaucoup plus chères en mémoire et en temps d'exécution mais elles sont très pratiques lors de l'implémentation.
En conclusion, si le programme n'avait pas été codé en utilisant des constructions non-déclaratives, il nous aurait été extrêmement difficile d'en implémenter les fonctions.

## 1.3 Implémentations surprenantes

L'implémentation de notre fonction `transpose` aura sans doutes du vous surprendre, elle est en effet assez longue et utilise de nombreuses sous-fonctions pour aboutir au résultat final. Nous ne voyons pas, malgré de longues réflexions, comment implémenter autrement cette fonction. Il est aussi important de remarquer que nous utilisons une cellule, cela peut être surprenant puisque nous cherchons à avoir un programme performant, celu fut cependant plus facile à coder de la sorte.
La fonction `loop` utilise un accumulateur dans sa sous-fonction afin de créer une boucle qui s'arretera quand ce dernier sera plus grand que le nombre d'éléments dans la musisque à répéter.
Il est aussi surprenant que vous ne voyez pas les 7 filtres. Comme expliqué plus haut, nous ne sommes pas parvenu à écrire les deux filtres : `echo et fade`.
La fonction `partition et merge` ne sont pas non plus complètes. Nous les avions écrites mais n'aboutissant pas au résultats voulu, nous avons préféré laisser leur implémentation vide. Par conséquent, lorsque nous codons la structure finale de la partie `Mix`, le code ne compile pas. Nous avons donc retiré le corps de la fonction générale `Mix` pour que le code soit accepté et puisse être testé.

## 1.4 Complexités

### 1.4.1 Transformation `duration`

Dans la fonction `duration` nous avons une première sous-fonction nommée `HowLong` qui additionne la durée de chaque élément de la liste ( une note ou un silence). Ensuite, nous utilisons le résultat `HowLong` et nous utilisons la transformation `stretch` en lui passant comme argument la durée que nous souhaitons obtenir divisée par la durée rendue par `HowLong`. Cela nous renvoie le facteur que l'on utilise dans `stretch`.
Puisque nous ne faisons que de lire des données dans une liste et que nous appellons simplement une fonction avec ces données récoltées, le nombre d'étapes est linéaire par rapport à la taille de la liste. La complexité est donc égale à O(n).

### 1.4.2 Fonction `merge`

/

### 1.4.3 Filtre `loop`

Cette fonction utilise une sous-fonction basée sur une condition qui est celle énoncée dans la spécification (On tronque la dernière répétition de la musique pour ne pas dépasser la durée indiquée). Si cette condition n'est pas encore remplie, on va alors créer notre nouvelle liste de répétition de la musique passée en argument de la fonction `loop`.
La fonction est linéaire, on n'appelle qu'une seule fonction à chaque récursion. Cela nous amène donc à une complexité égale à O(n).

## 1.5 Extensions

Notre code n'étant pas fonctionel totalement (En effet la fonction Mix ne compile pas puisqu'il lui manque quelques parties), nous n'avons pas trouvé d'intérêt à tenter de créer des extensions sans pouvoir les tester et sans qu'elles ne servent à quelques choses.

## 1.6 Conclusion

Pour conclure, nous aimerions ajouter que ce projet fut très difficile pour nous. Nous avons longtemps stagné sur la première partie avant de bien comprendre ce qui nous était demandé. C'est donc déja avec difficulté que nous sommes parvenu à coder la première partie. Nous nous sommes ensuite attaquer à la fonction `Mix` qui nous a apporter de nombreux problèmes elle aussi. Nous avons du limiter à seulement 5 filtres sur 7 car le temps et surtout la compréhension nous ont fait défaut.
Cependant, ce projet fut fort enrichissant. C'était la première fois que nous obtenions quelque-chose de concrêt avec le langage Oz et nous avons du travailler pour la première fois en groupe sur un même code.
Le 3 ème fichier, l'exemple est celui fourni dans l'énoncé. En effet, il devait être modifié par nos soins afin de prouver le fonctionnement de notre fonction `Mix`. Cette dernière ne fonctionnant pas, nous avons préféré vous remettre l'exemple de base pour utiliser notre temps à tenter de résoudre nos problèmes d'immplémentation.