MPIL(LI332)
Année 2015/2016
Mini-projet de MPIL : partie 1

Objectifs

1. Structure d'arbre
2. Lecture et écriture de fichiers MIDI
3. Module graphique
4. Stratégies de génération

Le chant *polyphonique* (*i.e.* à plusieurs voix) apparaît vers le IX^e siècle dans la musique religieuse (chant grégoriens). L'une des pratiques les plus anciennes consiste à improviser une deuxième voix, le *contre-chant*, par dessus une ligne mélodique simple et dépourvue de rythme appelée *cantus firmus* (figure 1).

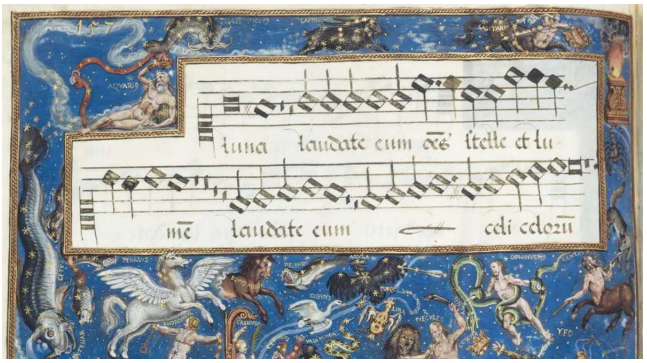

FIGURE 1 – Exemple de *cantus firmus*. *La carte du ciel étoilé* peinte par Hans Mielich en 1570 pour le duc Albert V de Bavière en illustration du motet Laudate Dominum d'Orlando de Lassus.

Dans le cadre de ce projet, nous souhaitons créer un algorithme de génération automatique de contre-chant. Pour un cantus firmus donné, l'ensemble des contre-chants possibles peut être représenté dans une structure d'arbre de listes proche de celle d'un arbre lexical. Il suffit ensuite de parcourir l'une de ces branches pour générer une mélodie, de la même manière que l'on parcoure les branches d'un arbre lexical pour former des mots.

Cette première partie porte sur l'implémentation de deux modules :

- un module d'import des données depuis un fichier *csv*
- une structure d'arbre représentant l'ensemble des contre-chants possibles

Un module principal sera construit à l'aide d'un foncteur prenant ces deux modules en argument (voir figure 2). L'ensemble du projet, qui sera présenté dans une seconde partie, comportera également un module de lecture et écriture de fichiers au format MIDI et un module d'édition de partition.

Exercice 1 : Structure d'arbre

Une voix est une succession de notes que l'on représente par une liste d'entiers. Chaque entier est associé à une hauteur de note, le Ré central étant fixé à 0 et les altérations ignorées. Ainsi, la liste $[|0; 1; 2|]$ représente la mélodie Ré, Mi, Fa. Dans la suite du projet on ne manipulera les listes de notes que sous leur forme numérique, et aucune connaissance musicale n'est indispensable.

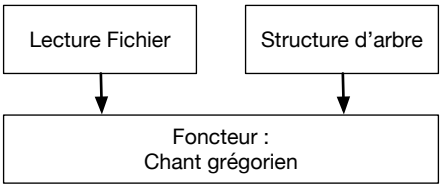

FIGURE 2 – Schéma de fonctionnement des différents modules du système. Le module principale est généré par un foncteur prenant en argument le module de lecture d'un fichier et le module de structure d'arbre

Q.1.1 Créer un module de lecture de fichiers qui comporte une exception *NonValidFile* en cas de fichier invalide et une fonction de lecture *lire_partition* qui prend en entrée un nom de fichier et retourne une liste d'entier. On écrira d'abord une signature *READSCORE*, puis une implémentation adaptée à la lecture de fichiers *CSV*. Un fichier d'exemple *cantus.csv* est disponible.

Q.1.2 Pour un cantus firmus donné, les règles strictes qui régissent l'écriture d'un contre-chant se traduisent par des contraintes mathématiques simples lorsque l'on représente les mélodies par des listes d'entiers. Il est alors possible de parcourir l'ensemble des listes d'entiers respectant ces contraintes, c'est-à-dire de parcourir de manière exhaustive l'ensemble des contre-chants possibles à un cantus firmus donné. Dans la suite de l'énoncé, $f(1), ..., f(n), ...f(N)$ fait référence à la liste d'entiers représentant le cantus firmus et $c(1), ..., c(n), ...c(N)$ à la liste d'entier représentant un contre-chant possible.

Les éléments $f(n)$ sont compris entre 0 et 7 (Ré et Ré de l'octave supérieure). Les contraintes mathématiques sur l'écriture du contre-chant sont les suivantes :

contrainte tessiture les notes du contre-chant $c(n)$ ne peuvent prendre des valeurs comprises qu'entre 4 et 11.

contrainte harmonique la différence $|f(n) - c(n)|$ doit être comprise dans la liste $[0; 2; 4; 5; 7; 9; 11]$

contrainte mélodique la différence $|c(n-1) - c(n)|$ doit être comprise dans la liste $[0; 1; 2; 3; 4; 5; 7]$

Le module de structure d'arbre est décrit par la signature suivante

```
module type TREESTRUCTURE = sig
    type holy_tree
    val init_tree : holy_tree
    val contruire_arbre : int list -> holy_tree
    val parcours_arbre : holy_tree -> int list
end;;
```

On se propose dans les questions suivantes d'écrire une implémentation de la signature *TREESTRUCTURE*. Vous pourrez vous aider de l'exemple d'arbre donné dans le fichier *exemple.ml* pour tester vos fonctions, et du schéma représentant la construction d'un tel arbre (voir figure 3).

Les solutions possibles pour le cantus firmus sont représentées dans une structure d'arbre lexical. On fera en sorte que chaque noeud de l'arbre contienne un couple d'entier composé du contre-chant et du cantus firmus et la liste des noeuds sui-

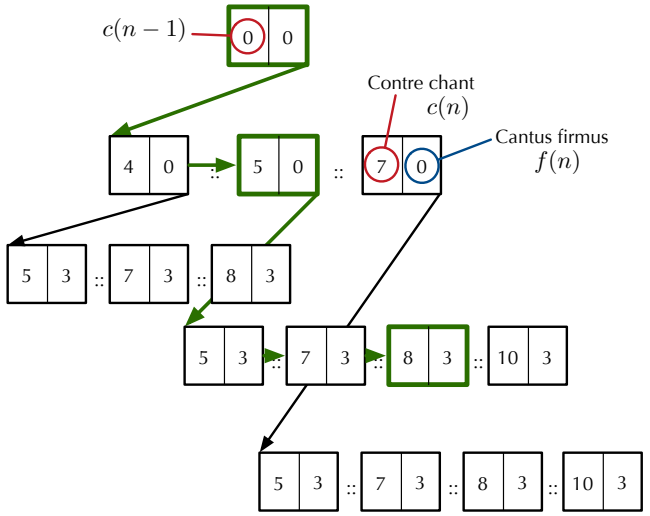

FIGURE 3 – Structure représentant les contre-chants possibles pour le cantus firmus (0,3). Chaque nœud contient un couple d'entiers et une liste de nœuds. La liste de nœuds correspond aux potentielles notes suivantes. En vert est tracé un parcours possible de l'arbre.

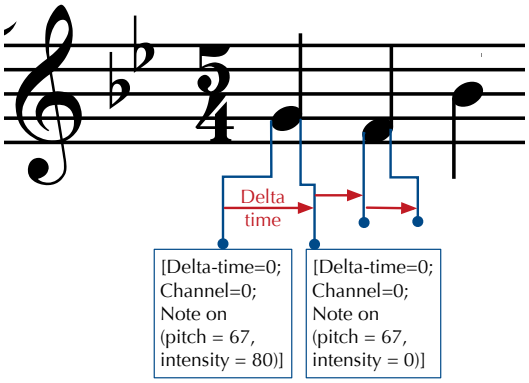

FIGURE 4 – Messages MIDI. Une note est représenté par un message note on et note off (très fréquemment un message note on d'intensité 0, comme sur la figure). Le temps d'arrivé d'un message est relatif est au temps d'arrivé du message précédent.

vants. écrire le type *holy_tree* qui représente une telle structure.

Contrairement à l'arbre lexical, il n'y a pas de booléen indiquant la fin d'un mot et toute séquence partant de la racine et mesurant la même longueur que le cantus firmus constitue un contre-chant possible.

Q.1.3 Définir le type t du module d'arbre, composé de deux champs mutables : une liste d'entiers (le cantus firmus) et un holy_tree.

Q.1.4 écrire une fonction *noeud_possibles* qui prend en entrée une note du cantus firmus $f(n)$ et une note du contre chant $c(n-1)$ et qui donne en sortie une *holy_tree list* contenant les notes possibles pour le contre chant. On tient compte pour cela des 3 règles, mélodique, harmonique et absolue, définies précédemment.

Q.1.5 A l'aide de la fonction *noeud_possible* précédemment définie, écrire une fonction *construire_arbre* qui prend en argument une liste d'entiers correspondant au cantus firmus et retourne le *holy tree* des contre-chants possibles. Un exemple de construction d'arbre est donné dans *exemple.ml*.

Q.1.6 Un chemin partant de la racine de l'arbre est allant jusqu'à une feuille définit un contre-chant potentiel. Il est possible que lors de la construction de l'arbre, certains chemins aboutissent à une impasse (aucune contre-chant ne permet de satisfaire les contraintes à l'étape suivante). On souhaite élaguer l'arbre avant sa phase de parcours, c'est-à-dire supprimer les chemins impasse. Écrire une fonction *nettoyer_arbre* qui prend en entrée un *holy_tree* de profondeur n (chemin le plus long depuis la racine jusqu'à une feuille) et retourne un *holy_tree* dont les chemins de longueur strictement inférieure à n ont été supprimés. On modifiera la fonction *construire_arbre* précédemment définie pour qu'elle retourne systématiquement un arbre élagué.

Q.1.7 La dernière étape est d'écrire une fonction qui prend en entrée un *holy_tree* supposé élagué, effectue de manière aléatoire un parcours dans cet arbre depuis la racine jusqu'à une feuille, et retourne ce parcours sous la forme d'une liste d'entiers qui représente alors un contre-chant possible au cantus firmus de l'arbre.

Q.1.8 Définir le foncteur FCantusFirmus prenant en argument deux modules de signatures READSCORE et TREESTRUCTURE, et permettant de construire un module contenant :

- un enregistrement t composé de deux champs mutables : une liste d'entiers (le cantus firmus) et un *holy_tree*.
- les fonctions *lire_partition*, *construire_arbre* et *parcours_arbre*

Exercice 2 : Lecture et écriture de fichiers MIDI

MIDI est un protocole de communication entre des logiciels et des instruments de musique électronique. La norme MIDI peut également être utilisée pour l'écriture de partition numériques, et a longtemps était considérée comme le format de référence en informatique musicale. Ainsi, même pour du répertoire très spécialisé comme le chant grégorien, on peut trouver de larges bases de données MIDI librement accessibles.

On souhaite écrire un module de lecture de fichiers MIDI respectant la signature *READSCORE* et une signature/module d'écriture de fichiers MIDI pour pouvoir écouter le résultat de notre algorithme de génération.

Une partition écrite dans le format *MIDI* est une succession de *messages* : notes, silences, nuance, changement d'instruments... Dans le cadre très restreint du chant grégorien, on ne s'intéressera qu'aux messages concernant les notes. Un message *note On* représente le début d'une note, sa fin étant indiquée par un message *note Off*. Les messages note On et note Off contiennent deux informations : le *pitch* (hauteur de la note, entier entre 0 et 127) et l'*intensité* (entre 0 et 127). Il est très fréquent d'envoyer un message note on d'intensité 0 à la place d'un message note off.

On s'aidera d'une toolbox MIDI (lien) fournie dans le fichier *midi.ml*. Dans l'implémentation proposée, un message midi est représenté par le type midievent. Afin de savoir dans quel ordre doivent être jouées les notes, il est important de pouvoir situer les différents messages dans le temps. L'information

temporelle est relative en MIDI, c'est-à-dire décrite par la différence de temps (*delta time*) entre le message actuel et le message précédent. Ainsi l'ordre de lecture des messages conditionne l'ordre des notes, et le delta-time le rythme. Il est possible d'écrire des événements simultanés en utilisant un delta time nul. Une partition est donc une liste de messages :

```
(int * int * midievent) list
```

où le premier entier est le delta time, le deuxième entier un numéro de canal que nous ignorerons, et midievent est un événement midi de type note on ou note off. Référez-vous au fichier *midi.ml* pour plus de détails sur le type midievent et les fonctions de lecture et écriture.

Q.2.1 Écrire un module ReadMidi respectant la signature READSCORE que vous avez implémentée dans la question 1. Il comportera notamment une fonction lire_partition prenant en argument un nom de fichier et renvoyant une liste d'entiers. On utilisera la fonction read du module Midi fournit.

- Pour faire le lien entre notre représentation des notes (entiers entre 0 et 11) et le pitch midi lu dans le fichier, on utilisera la fonction map_from_midi décrite ci-dessous. Une exception est soulevée lorsque le fichiers contient des notes qui sortent de l'ambitus classique du chant grégoriens.
- On se souviendra qu'une note off peut être une note on d'intensité 0.
- Il faudra repérer le cas où le fichier d'entrée est polyphonique et soulever une exception dans ce cas là. Un fichier est polyphonique lorsque deux notes ou plus sont jouées à un instant dans la partition.

```
let map_from_mid c =
  match c with
        | 62 -> 0
        | 64 -> 1
        | 65 -> 2
        | 67 -> 3
        | 69 -> 4
        | 71 -> 5
        | 72 -> 6
        | 74 -> 7
        | 76 -> 8
        | 77 -> 9
        | 79 -> 10
        | 81 -> 11
        | _ -> raise (AmbitusWrong c)
```

Q.2.2 Écrire un module WriteMidi respectant la signature suivante :

```
module type WRITESCORE = sig
 exception ListeIncorrecte of string
 val ecrire_partition : int list -> int list
       -> string -> unit
end;;
```

Les deux listes en argument de la fonction *ecrire_partition* sont respectivement la liste lue en entrée dans le fichier midi (le cantus firmus) et la liste produite après parcours de l'arbre (contre-chant). On s'assurera notamment que ces deux listes ont bien la même longueur.

Vous utiliserez la fonction write du module Midi. Pour produire un fichier midi, il faut donner à la fonction write une structure respectant celle décrite dans le fichier midi.ml, notamment en précisant une division que l'on prendra égale à 100. Vous pourrez choisir une intensité identique pour toutes les notes et égale à 80. Pour obtenir un pitch MIDI, vous déduirez la fonction map_to_mid qui est l'inverse de map_from_mid. De plus, on choisira une durée des notes égales à deux ou trois fois la division, soit entre 200 et 300. Souvenez-vous bien que la gestion du temps est relative entre les messages (delta time).

Q.2.3 Finalement, modifier le foncteur FCantusFirmus pour intégrer un module de signature WRITESCORE et créer un module avec ce foncteur qui permette de lire et écrire des fichiers Midi.

Exercice 3 : Annexe

```
let aaa = [Noeud(0,0,
            [Noeud(1,0,
                [Noeud(3,0,[]);
                Noeud(4,0,[]);
                Noeud(5,0,[])]);
            Noeud(2,0,[]);
            ])
          ]
let a = clean_tree (aaa,3) in
affiche a;;
(*  a vaut :
013
014
015
*)

let aaa =  contruire_arbre [0;3;5] in
affiche aaa;;
(*  aaa vaut :
0,4,5,5
0,4,5,7
0,4,5,9
0,4,5,10
0,4,7,5
0,4,7,7
0,4,7,9
0,4,7,10
0,4,8,5
0,4,8,7
0,4,8,9
0,4,8,10
0,5,5,5
0,5,5,7
0,5,5,9
0,5,5,10
0,5,7,5
0,5,7,7
0,5,7,9
0,5,7,10
0,5,8,5
0,5,8,7
0,5,8,9
0,5,8,10
0,5,10,5
0,5,10,7
0,5,10,9
0,5,10,10
0,7,5,5
0,7,5,7
0,7,5,9
0,7,5,10
0,7,7,5
0,7,7,7
0,7,7,9
0,7,7,10
0,7,8,5
0,7,8,7
0,7,8,9
0,7,8,10
0,7,10,5
0,7,10,7
0,7,10,9
0,7,10,10 *)
```