Projet : Improvisation musicale et *live coding*

Pierre Talbot (pierre.talbot@univ-nantes.fr)

Printemps 2019

Objectif(s)

- ✯ Maitriser les concepts de la programmation fonctionnelle.
- ✯ Implémenter un programme OCaml en utilisant les traits modulaires du langage.

Ce projet est inspiré de plusieurs exercices de Charlotte Truchet que je remercie vivement !

1 Rendu

— **Quand ?** : 29 avril 2019 à 23h59.
— **À qui ?** : Un unique mail avec comme destinataires `pierre.talbot@univ-nantes.fr`.
— **Quoi ?** : Une archive contenant votre projet compilable avec OCaml `4.07.1`, avec (au moins) les dossiers/fichiers suivants :
  — `src/` : Les sources du projet.
  — `tests/` : Les fichiers de tests unitaires (utiliser par exemple `alcotest`).
  — `README.md` : Fichier indiquant comment compiler et tester le projet.
  — `rapport.pdf` : Un rapport explicitant vos choix :
    — Description du travail produit : qu'avez-vous réalisé par rapport à l'énoncé, qu'est-ce qui marche (ou ne marche pas).
    — La description détaillée de vos choix techniques, les difficultés rencontrés.
    — Toutes autres informations pertinentes.
    — Soyez **précis** dans vos explications mais ne répéter pas le sujet.
    — Des **exemples** sont les bienvenus.
— **De plus...** : Le code source doit pouvoir être compilé et exécuté aisément grâce à un script bash :

```
$ ./run.sh
```

— **Notation** :
  — La clarté du code (pas de fonction de plus de 10 lignes, choix des noms de variables, ...).
  — La **décomposition en modules** du projet : attention ce n'est pas explicité dans les questions, c'est-à-vous de le choisir.
  — L'architecture du projet.
  — La clarté du rapport.
  — La clarté de la présentation.
  — ... (en gros : faites ça bien).

2 Échauffement

Exercice 1 – Hauteur de note

Le but de cet exercice est d'écrire une petite bibliothèque permettant de jouer des notes de musique. En informatique, les représentations musicales sont variées et très dépendantes de l'usage. Il est pratique d'utiliser un format de type MIDI (couramment utilisé pour la communication avec des synthétiseurs par exemple), ou bien une représentation plus structurée proche d'une partition (ou encore d'autres). Dans un premier temps, on s'intéresse aux hauteurs des notes de musique, sans tenir compte de leurs durées. On considère une représentation simplifiée de notes de musique, un peu restrictive mais assez proche de ce qui se fait en musique occidentale. On considèrera que la hauteur d'une note est définie par :

— Un nom, à choisir parmi Do, Ré, Mi, Fa, Sol, La, Si, qui indique la position dans la gamme [1].
— Une altération, à choisir parmi Becarre, Dièse, qui indique si la note est jouée "telle quelle" (Bécarre) ou "un peu augmentée" (Dièse).
— Une octave, qui indique la hauteur de la gamme (les gammes sont jouées cycliquement, pensez au clavier de piano dont les motifs se répètent).

A partir de ces informations, on peut calculer deux informations intéressantes sur la note : la hauteur MIDI et la fréquence. La hauteur MIDI est définie ainsi : par convention, le do du milieu du piano, qui avec notre notation s'appelle Do Becarre 3, a une hauteur MIDI de 60. Ensuite, pour une note quelconque, on ajoute 1 (resp. -1) à chaque fois qu'on ajoute (resp. enlève un demi-ton). Dit autrement, voici, données explicitement, les hauteurs MIDI sur une gamme, en notant la valeur "Dièse" avec un symbole ♯ :

Note	Do	Do♯	Ré	Ré♯	Mi	Fa	Fa♯	Sol	Sol♯	La	La♯	Si	Do	...
MIDI	60	61	62	63	64	65	66	67	68	69	70	71	72	...

La fréquence s'obtient avec une formule sur la hauteur MIDI. Pour une note ayant une hauteur MIDI de M, la fréquence f est :

$$f = 440 * 2^{(M-69)/12}$$

1. Écrire le type `note_name` qui représente les noms possibles d'une note, et `alteration` qui indique si la note est Bécarre ou Dièse. A partir de ces deux types, écrire un type enregistrement `note`, avec un champ pour le nom, un pour l'altération et un pour l'octave.
2. Déclarer les notes suivantes : do4, do dièse 4, la2, la3, la4 et mi3.
3. Écrire une fonction `note2midi` qui convertit une hauteur en sa hauteur MIDI. On peut la tester sur la2, la3 et la4 pour lesquelles les hauteurs MIDI sont 57, 69 et 81, puis pour do4 (72), do dièse 4 (73) et mi3 (64).
4. Écrire une fonction `midi2freq` qui convertit une hauteur MIDI en fréquence, et en déduire une fonction `note2freq` qui convertit une note (de type `note`) en fréquences. On pourra la tester sur les valeurs suivantes : la3 (440), la2 (220), la4 (880), do dièse 4 (554) et do 4 (523).
5. Écrivez une fonction `play_note : note -> unit` qui permet de jouer chaque note. Pour cela, vous utilisez la bibliothèque `Graphics` (voir https://caml.inria.fr/pub/docs/manual-ocaml/libgraph.html). Elle contient une fonction `sound : int -> int -> unit` permettant de jouer un son dont la fréquence est le premier argument et la durée le second.

Exercice 2 – Partition

1. On représente une séquence musicale par une liste de notes. Écrire un type `score` (ce qui veut dire partition en anglais) correspondant à une liste de notes, chacune étant de type `note`. Déclarer `my_score` une suite de notes de votre choix, de longueur au moins 4.
2. On souhaite pouvoir afficher chaque note. Écrire une fonction `print_note` qui affiche une note, et puis `print_score` qui affiche une partition.

1. Voir `https://fr.wikipedia.org/wiki/Note_de_musique#Symbole_sur_une_partition`

3. Écrivez une fonction `play_score` permettant d'écouter une partition de type `score`.
4. On souhaite maintenant, à partir d'une suite de notes, les transposer, c'est-à-dire les monter/descendre d'un intervalle fixé. Si l'on ne considère que les hauteurs MIDI, cette opération est une addition (ou soustraction) ; l'intervalle étant aussi donné en version MIDI. Écrire une fonction `transpose` qui permette d'appliquer une transposition à une liste de type `score`.
5. Voici quelques exemples de mélodies que vous pourrez tester :

```
let ode_to_the_joy =
[71; 71; 72; 74; 74; 72; 71; 69; 67; 67; 69; 71; 71; 69; 69 ; 71; 71; 72;
74; 74; 72; 71; 69; 67; 67; 69; 71; 69; 67; 67; 69; 69; 71; 67; 69; 71;
72; 71; 67; 69; 71; 72; 71; 69; 67; 69; 62 71; 71; 72; 74; 74; 72; 71;
69; 67; 67; 69; 71; 69 ;67; 67]

let game_of_thrones = [67; 60; 63; 65; 67; 60; 63; 65; 67; 60; 63; 65; 62;
65; 58; 63; 62; 65; 58;  63; 62; 60; 67; 60; 63; 65; 67; 60; 63; 65; 67; 60;
63; 65; 62; 65; 58; 63; 62; 65; 58;  63; 62; 60]

let itcrowd= [60;60;60;67;63;63;63;70;67;67;67;74;70;70;70;67;60;60;60;67;63;
63;63;70;67;67;67;74;75;75;75;67;60;60;60;67;63;63;63;70;67;67;67;74;70;70;70;
67;60;60;56;56;63;63;60;60;67;67;63;63;60;60;63;63]
```

Exercice 3 – Arpège

On s'intéresse à la génération d'arpèges. Un arpège [2] est un accord, c'est-à-dire un groupe de notes sonnant harmonieusement quand elles sont jouées ensemble, mais dont les notes sont jouées successivement. En général, les notes sont jouées dans un ordre croissant, décroissant, ou d'abord croissant puis décroissant. On considèrera qu'il y a un intervalle de temps constant entre chaque note jouée. On veut ici générer des arpèges en calculant les hauteurs MIDI des notes jouées successivement. Cela dépend de l'accord que l'on veut arpégier : dans la suite, cet accord sera donné par trois notes, encodées de la façon suivante :

— Une note basse (par exemple do 3, ou 60 en hauteur MIDI).
— L'intervalle entre la note basse et la deuxième note de l'accord (par exemple une tierce majeure, ou 4 en MIDI).
— L'intervalle entre la note basse et la troisième note de l'accord (par exemple une quinte juste, ou 7 en MIDI).

Cet encodage est assez proche d'un chiffrage d'accord, notion couramment utilisée aussi bien en musique occidentale classique qu'en musiques actuelles, et qui permet de déconnecter l'accord lui-même (décrit par les intervalles) de la hauteur à laquelle il est joué (donné par la note basse). Par exemple, voici quelques exemple d'accords donnés par un même chiffrage, celui d'accord parfait majeur :

Noms des notes	Hauteurs MIDI	Note basse	Intervalle
do3 - mi3 - sol3	60 64 67	60	4 7
do4 - mi4 - sol4	72 76 79	72	4 7
sol3 - si3 - re4	67 71 74	67	4 7

À partir d'une note basse, que l'on notera basse dans la suite, et de deux intervalles `itv1` et `itv2`, on jouera naturellement un arpège de quatre notes déterminées ainsi : `basse, basse + itv1, basse + itv2, basse + 12` (on finit en général par la note basse, jouée un octave plus haut). On peut répéter ce principe sur plusieurs octaves. On veut en outre enchaîner un arpège montant avec le même arpège descendant. Voici par exemple la suite de notes correspondant à `basse = 48, itv1 = 4, itv2 = 7` sur trois octaves :

48-52-55-60-64-67-72-76-79-84-84-79-76-72-67-64-60-55-52-48

Cet exercice peut être fait, selon votre aisance en programmation fonctionnelle, de deux façons : soit directement (excellent exercice de modélisation et récursion), en écrivant directement une fonction `arpegie` qui calcule les valeurs successives des notes, et les imprime à la volée, soit avec les indications suivantes :

2. Pour écouter, voir par exemple https://www.youtube.com/watch?v=VtEOxECkfEc ou http://www.six-cordes.com/cours-guitare-debutant/premiers-pas/arpeges-accords-la-maj/.

1. Déterminer le type de la fonction `arpegie`.
2. Déterminer le(s) cas de base de la fonction `arpegie` : cette fonction a deux régimes, arpège montant et arpège descendant. Comment gérer cela ? Quels sont les cas d'arrêt de chacun des régimes ?
3. À un instant donné, on suppose que la fonction `arpegie` joue dans l'arpège montant une note courante. Déterminer le calcul qui permet de calculer la note suivante, en distinguant bien les cas.
4. Écrire une fonction `arpegie_up` qui calcule les notes de l'arpège montant (cette fonction pourra ensuite être locale à `arpegie`).
5. Déterminer comment utiliser la même récursion pour les arpèges descendantes.
6. Écrire la fonction `arpegie`.

3 Improvisation musicale

Exercice 4 – Algorithme de compression LZW

Le but de l'exercice est de construire des arbres représentant de façon compacte une suite de symboles. Dans la suite, les symboles seront des lettres de l'alphabet, mais le programme de construction de l'arbre doit être polymorphe. On construit l'arbre avec une méthode très inspirée de l'algorithme de Lempel-Ziv-Welch, utilisé dans la compression sans perte de fichiers (par exemple dans le format d'images GIF). La principale différence est qu'ici, on compresse une séquence en un arbre et non un dictionnaire. On utilisera des arbres n-aires avec le type suivant (que vous pouvez adapter/changer) :

```
type 'a tree = 'a * int * ('a tree list)
```

où l'entier sert à stocker l'ordre dans lequel les symboles sont insérés dans l'arbre (on peut l'ignorer dans un premier temps). La construction prend en entrée un arbre, vide au départ, et une liste de symboles `data`, on prendra comme exemple dans la suite

```
let data = ["a";"b";"c";"a";"c";"d";"a";"c";"b";"d";"e"]
```

On regarde le premier symbole de la liste `symb`. S'il n'est pas présent parmi les enfants du nœud racine, alors on l'ajoute à la liste de ces enfants et on relance la construction sur la queue de la liste. S'il est présent, c'est à dire si un enfant du nœud racine est étiqueté par `symb`, alors on reprend la construction sur l'enfant et la queue de la liste. Cet algorithme sera illustré au tableau en cours.

1. Préparer la construction récursive : identifier le(s) cas d'arrêt, le(s) cas récursifs, valeur(s) de retour. Quels paramètres est-il judicieux de choisir ?
2. Écrire une fonction `add_symbol` qui ajoute dans un arbre la plus longue suite de symboles possible d'une liste donnée.
3. Écrire une fonction `compress` qui construit l'arbre des préfixes à partir d'une liste quelconque.
4. Écrire une fonction `decompress` qui donne la liste initiale à partir de l'arbre.
5. Si les symboles sont des notes, alors on peut parcourir l'arbre en jouant les notes. Avec une certaine probabilité (faible), vous permettrez de continuer l'exploration sur une note voisine, ce qui permettra de jouer une mélodie différente. Cela permet d'improviser sur une mélodie connue avec des suite de notes cohérentes mais différentes de la musique original. Vous pouvez boucler sur cet algorithme pour parcourir plusieurs fois l'arbre et obtenir des musiques différentes.

4 Live coding

Exercice 5 – Live Coding

Vous avez maintenant plusieurs fonctions permettant de jouer de la musique interactivement. Nous allons les regrouper pour en faire un outil pour un artiste. Cet exercice est libre et les questions suivantes sont indicatives et *non suffisantes* (c'est pour vous donner de l'inspiration).

1. Proposer un programme permettant à l'utilisateur de tester des partitions interactivement. Vous pouvez créer une interface graphique.
2. Augmenter le programme précédent pour permettre au compositeur de répéter des notes, par exemple si il note :

   ```
   repeat 5 (DO 2)
   ```

   Ça répète 5 fois la note *DO* dans l'octave 2.

   ```
   repeat (DO 2) every 2 beats
   ```

   Répète indéfiniment la note (DO 2) tous les 2 temps.
3. Permettre l'utilisation de l'algorithme de compression précédent.
4. Étendez ce langage musical dans la direction que vous voulez. Voir *Live coding* (https://en.wikipedia.org/wiki/Live˙coding).
5. Permettre la visualisation de la partition qui est en train d'être jouée. La visualisation peut être artistique, soyez créatif !