Option PF2, L2-Info&Math, Automne 2016

« Programmation Scheme Avancée »

TP4-audio

1. Jouer une liste de notes de piano

```
(require rackunit rsound rsound/piano-tones)
```

Rappel : les notes du piano (blanches et noires) sont espacées d'un *demi-ton*, ou *half-step*. Un octave comprend 12 demi-tons. Les demi-tons possèdent la propriété suivante :

(M) : On passe de la note MIDI numéro `m` à la note MIDI `m+1` située un demi-ton à droite en multipliant la fréquence de `m` par $\sqrt[12]{2}$. On passe donc de la note numéro `m` d'un octave à la note numéro `m+12` de l'octave suivant en multipliant sa fréquence par 2.

Exercice 4.1 a) En utilisant la fonction `(midi-note-num->pitch n)` renvoyant la fréquence de la note MIDI numéro `n`, calculez la fréquence de la note numéro 60 (le DO du 4ème octave, alias DO4, alias C4), puis celle du DO5.

b) Tirez un numéro `n` de note au hasard dans [60,72] et vérifiez la propriété (M) sur les notes `n` et `n` +1 en utilisant `check-=` du module `rackunit` (`check-equal?` n'est pas adapté aux nombres inexacts).

• Nous jouerons des notes de piano dans une *pstream* notée `ps` :

```
(define ps (make-pstream))
```

Il est possible d'envoyer un son `snd` directement dans `ps` pour qu'il soit joué, avec `pstream-play` :

```
(pstream-play ps (piano-tone 60))      ; piano-tone : [1,127] → rsound
```

mais ce n'est pas intéressant s'il s'agit d'envoyer des listes de notes. Si nous avons 20 notes à jouer, nous n'allons pas attendre d'avoir joué une note pour envoyer la suivante, mais nous allons les **enfiler**[1] dans `ps` : les mettre dans une file d'attente. Nous utiliserons pour cela la fonction `(pstream-queue ps snd t)` qui demande à la *pstream* `ps` de jouer le son `snd` au temps `t`. Le temps `t` est mesuré en nombres d'échantillons ou *frames* (donc 1 *frame* = 1/44100 seconde), le temps 0 étant celui de création de la *pstream*. Il est parfois commode de parler en secondes flottantes plutôt qu'en *frames* :

```
(define (sec n)       ; real → frames
   (inexact->exact (round (* n 44100))))      ; (sec 1.5) = 66150 frames
```

Pour structurer les notes en leur incorporant l'heure à laquelle elles seront jouées, nous définissons une structure `note` à 3 champs `num` (numéro de note MIDI), `time` (heure de la jouer) et `dur` (durée de la note) :

```
(define-struct note (num time dur) #:transparent)
```

Voici un exemple de *partition* (liste de notes structurées) qu'il s'agit de jouer au piano :

```
(define partition1
  (list (make-note 60 (sec 5) (sec 0.5))      ; DO4 joué à t=5 sec, durée 0.5 sec
        (make-note 64 (sec 5.5) (sec 0.15))   ; DO# joué juste après, durée 0.15 sec
        (make-note 65 (sec 5.65) (sec 0.15))  ; etc.
        (make-note 67 (sec 5.8) (sec 1))
        (make-note 67 (sec 6.8) (sec 0.15))
        (make-note 69 (sec 6.95) (sec 1))
        (make-note 65 (sec 7.95) (sec 0.2))
        (make-note 67 (sec 8.15) (sec 2))))   ; et on termine sur une note complète
```

[1] Ne confondez pas *enfiler* (file d'attente FIFO) et *empiler* (pile LIFO)...

Exercice 4.2 **a)** Programmez une fonction `(ps-play-note note)` prenant une structure `note` et enfilant cette note dans ps avec `pstream-queue`. Le résultat (sans trop d'importance) sera la `pstream ps`, la fonction étant intéressante pour son *effet*[2].
b) En déduire une fonction récursive `(ps-play-partition L)` enfilant une liste `L` de structures de type `note`. Le résultat sera là aussi `ps` et vous utiliserez la forme séquentielle `(begin e1 e2 ...)`. Faites jouer `partition1`.

• L'inconvénient de la fonction précédente est l'obligation de calculer à la main les heures auxquelles il faut jouer les notes (5, 5.5, 5.65, ...). Mieux vaudrait donner seulement la liste des notes et leur durée, et laisser Scheme calculer en boucle l'heure à laquelle jouer chaque note !

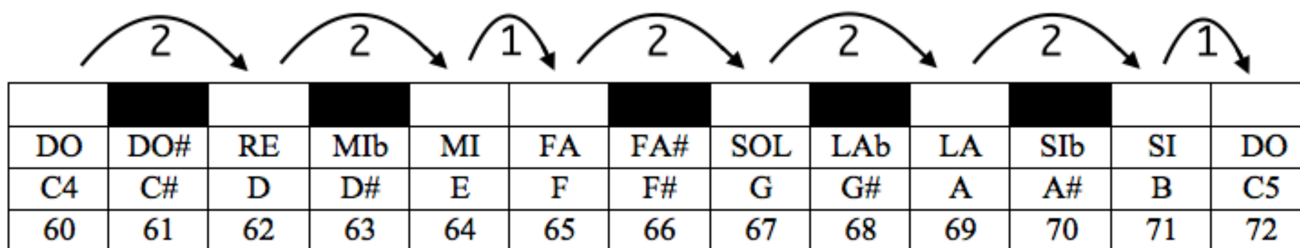
```
(define partition2 (build-partition (sec 2)       ; début au temps t=2sec
                              `((60 ,(sec 0.5))     ; DO4  durée 0.5 sec
                                (64 ,(sec 0.15))    ; DO#  juste après, durée 0.15 sec
                                (65 ,(sec 0.15))    ; etc.
                                (67 ,(sec 1))
                                (67 ,(sec 0.15))
                                (69 ,(sec 1))
                                (65 ,(sec 0.2))
                                (67 ,(sec 2)))))
```

Exercice 4.3 **a)** Programmez la fonction `(build-partition t0 L)` prenant une liste `L` de couples `(n d)` où `n` est un numéro de note MIDI et `d` sa durée en frames. Elle retourne la liste contenant les notes structurées correspondantes. L'argument `t0` est l'heure à laquelle on commence à jouer la première note :

```
 > (build-partition t0 (list '(60 22000) '(64 6600) ...))
 (list (make-note 60 t0 22000) (make-note 64 (+ t0 22000) 6600) ...)
```

b) Faites jouer la partition `partition2`.

• La **gamme de C/DO majeur** est bien connue : DO-RE-MI-FA-SOL-LA-SI-DO (à l'anglo-saxonne C-D-E-F-G-A-B-C). Les intervalles séparant les numéros de notes MIDI ne sont pas tous les mêmes, mais valent 1 ou 2 demi-tons. Par exemple on passe de C à D par un ton (2 demi-tons) mais de E à F par 1 seul demi-ton ! La suite des variations d'intervalles de la gamme majeure est donc : 2-2-1-2-2-2-1.



DO	DO#	RE	MIb	MI	FA	FA#	SOL	LAb	LA	SIb	SI	DO
C4	C#	D	D#	E	F	F#	G	G#	A	A#	B	C5
60	61	62	63	64	65	66	67	68	69	70	71	72

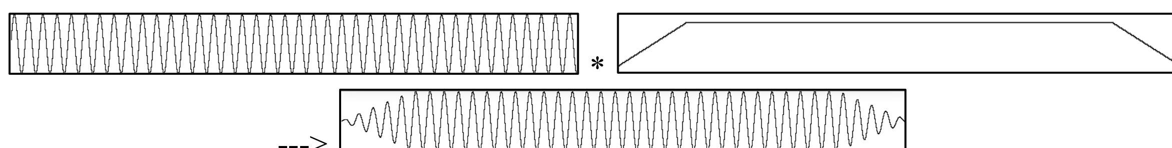
Exercice 4.4 **a)** Avec `ps-play-partition`, faites jouer la gamme en partant de C4.
b) Faites jouer la suite D-E-F-G-A-B-C-D. On ne reconnaît pas la même mélodie[3].
c) En respectant les mêmes variations d'intervalles 2-2-1-2-2-2-1, faites jouer 8 notes en partant de C#. Constatez qu'on retrouve la même mélodie que la gamme usuelle.
d) Programmez une fonction `(gamme t0 n dt)` faisant jouer à l'heure `t0` de la pstream la gamme à partir de la note numéro `n`. L'argument `dt` est la durée commune de chaque note. Par exemple on fait jouer la gamme débutant à FA# à la 32ème seconde de la pstream, chaque note durant 0.5 seconde :

```
(gamme (sec 32) 66 (sec 0.5))   ; gamme de FA#
```

[2] La musique à cet égard sort ici du cadre strict de la programmation fonctionnelle, le temps se mettant de la partie [*begin*].
[3] C'est le mode *dorien* de la gamme de DO.

Exercice 4.5 Le ***fading***. Il s'agit d'atténuer un son en un temps très court, plutôt que de le couper brutalement (les variations brutales de fréquences entraînent souvent des *clics* désagréables dans les sons). Idem au début pour le lancer. En fait il s'agit simplement d'une **enveloppe** consistant à placer une rampe montante au début et une rampe descendante à la fin, toutes deux du même nombre de frames. Pour que le son naisse et meure en douceur :-)

a) En utilisant `build-sound`, programmez la fonction `(rampe len r-in r-out)` retournant un son de longueur `len`, valant 1 partout sauf sur les `r-in` premières *frames* montantes de 0 à 1, et sur les `r-out` dernières *frames* descendantes de 1 à 0. Lequel son sera bien entendu inaudible, son seul but est d'être multiplié par un autre son pour en modifier l'enveloppe ! Avec `rs-draw`, faites dessiner cette rampe comme ci-dessous.



b) En déduire la fonction `(fade-in-out snd r-in r-out)` retournant une version du son `snd` atténuée aux deux extrémités, avec des rampes de longueur `r-in` et `r-out`. Testez sur un son LA440 de longueur 4000, avec r = 500. *Toujours insérer dans une animation un son avec fading aux deux extrémités !*

Signaux et réseaux

Les références livresques tournent autour de tout ce qui a trait au thème "Signaux et Systèmes" qui est un grand chapitre (difficile) de la physique du signal (audio, vidéo, etc). Ces "systèmes" (networks) sont vus en RSound sous un angle "dataflow", d'une manière encore un peu expérimentale...

Exercice 4.6 Construisez un signal sinusoïdal `sig` ayant une fréquence de 39.1 Kz. Ecoutez-le 10 sec. *Quoi ? 39.1 Kz ? Mais je... euh... il me semble que...* **Avez-vous une remarque intéressante à faire ?**

Exercice 4.7 En construisant chaque fois un petit **réseau** à l'aide de la primitive `network`, définissez les **signaux** suivants, et écoutez-les bien entendu :

a) `bip1` qui envoie une onde sinusoïdale de fréquence 880 Hz.

b) `bip2` qui envoie toutes les 2 secondes un bip de 880 Hz d'une durée de 1 seconde. Vous utiliserez pour cela un signal carré `square-wave`. Faites dessiner avec `rs-draw` un son de 10 secondes.

c) `bip3`, qui envoie 2 fois par seconde un bip de 880 Hz d'une durée de 0.1 seconde. Vous utiliserez cette fois un signal impulsionnel `pulse-wave`.
d) Sauvez dans un fichier `bip.wav` un son de 10 secondes contenant la superposition (la somme) des signaux `bip2` et `bip3`.

Exercice 4.8 Nous travaillons ici sur les deux dernières pages du cours, avec le début (Beethoven) du morceau `5th.wav`.

a) Simplifiez encore un peu le signal `looper` du cours page 22, en utilisant la primitive :

```
(loop-ctr len skip)
```

dont vous lirez la doc.

b) Faites un `reverse-looper` qui joue en boucle le morceau à l'envers (votre chaîne hifi sait faire ça ?). Mais n'inversez pas le son d'abord, faites plutôt fonctionner le `looper` à l'envers... Le résultat est une boucle syncopée assez mystérieuse du *DJ Ludwigvan* !

Exercice 4.9 Lisez la doc sur la fonction `(build-sound n f)` et construisez divers sons avec des formules trigonométriques tordues... Voici un générateur `f` pour obtenir un son avec **modulation de fréquence** (la fréquence 440 Hz est ici modulée par un signal basse fréquence à 1 Hz) :

```
(lambda (t) (let ((t (/ frame FRAME-RATE))           ; FRAME-RATE = 44100
                  (sin (* 2 pi 440 (sin (* 2 pi 1 t)))))))
```

Exercice 4.10 ***Construire un oscillateur.*** Nous avons vu que `sine-wave` était un **réseau (*network*) à une entrée** : la fréquence (cours p. 8). Comment programmerions-nous ce réseau s'il n'était pas prédéfini, en utilisant un sinus mathématique ? Il suffit d'envoyer en sortie de ce réseau les 44100 échantillons par seconde. Prenons `freq = 440` Hz : le signal se reproduit 440 fois par seconde. Les échantillons s'obtiennent donc en interpolant la sinusoïde sur 44100 points le long de 440 périodes chacune de longueur 2π, d'accord ?

a) Quel est en fonction de `freq` l'intervalle `dt` séparant deux échantillons consécutifs ?

b) Complétez le code ci-dessous (il vous suffira de définir `angle+` qui additionne deux angles) :

```
(define (angle+ t1 t2)            ; [0,2π] x [0,2π] --> [0,2π]
  ...)

define $sine-wave
  (network (freq)                 ; freq est un nombre en Hz, par exemple 440
    (angle = (prev suivant 0))    ; prev est expliqué plus bas (extrait de la doc)
    (suivant = (angle+ angle dt)) ; pour dt, cf question a)
    (out = (sin angle))))

(define sig (network ()
              (s <= $sine-wave 440)
              (out = (* 0.1 s))))

(signal-play sig)                 ; (stop) pour stopper...
```

The special `(prev node-label init-val)` *form may be used to refer to the previous value of the corresponding node. It's fine to have "forward" references to clauses that haven't been evaluated yet. The* `init-val` *value is used as the previous value the first time the network is used.*

Exercice 4.11 Téléchargez le fichier `freq-mod.rkt`. Il contient en première partie le programme `katy-sliders` de John Clements, avec son copyright que vous conserverez. Il s'agit d'une panneau de cinq sliders dont la valeur varie entre 0 et 1. Ils sont placés au début à 0.2 (avec une indication de 20 sur une échelle de 0 à 99). Vers la fin du fichier se trouve le programme de l'utilisateur, ici une démo de la *modulation de fréquence* sur un signal de fréquence `FREQ` avec un modulateur basse fréquence `LFO`. Vous pouvez remplacer ce programme par le vôtre, sans toucher si possible au code de John. La dernière ligne lance l'animation (dont le monde est celui des 5 sliders).

Si John Clements a construit lui-même ses sliders au lieu d'utiliser ceux des GUI, c'est pour des raisons à la fois didactiques (fabriquer ses propres widgets) mais aussi pour des raisons liées aux GUI. Comparez avec le fichier `freq-mod-gui.rkt`