# L-systèmes

inspiré de ENS 2001

Jean-Baptiste Rouquier

## 1 Remarques sur les DM

- Si vous ne deviez retenir qu'une seule chose : mettez des commentaires et expliquez vos fonctions. J'ai cherché bien plus longtemps qu'un correcteur de concours à comprendre ce que font vos fonctions auxiliaires.
- Pour ajouter un élément `a` en tête d'une liste `list`, on écrit
  `a :: list`
  Ne jamais écrire `[a]@list` ni (pire) `list@[a]`. `@` sert uniquement à concaténer des longues listes, et il est souvent possible et préférable de s'en passer (i.e. d'écrire la fonction de manière à ne pas avoir besoin de concaténer des longues listes). De même pour ajoutet quelques éléments, on écrira `a::b::c::list`.
- Testez vos fonctions.
- Il faut lire le sujet et voir où il veut nous mener. Par exemple il peut demander une première version naïve d'une fonction, puis une version plus élaborée. On n'exigera pas dans la première version d'avoir une bonne complexité, mais seulement d'être simple. Au contraire, la deuxième version devra être meilleure (souvent sur la complexité, parfois seulement sur la quantité de mémoire utilisée). Ou bien on verra que le but du sujet est de créer des circuits, non d'évaluer leur valeur. Ou encore on devinera quelles questions utilisent les questions précédentes. Une réponse qui passe à côté de ce qu'attend l'auteur du sujet ne rapporte souvent rien.
- En caml, les fonctions sont curryfiées. Evitez désormais le style `f(x,y,z)`. Le seul point où le style décurryfié est utile, c'est pour grouper des arguments semblable, par exemple
  `let dessine_triangle (ax,ay) (bx,by) (cx,cy) = ...`
- On se fiche du type affiché par le compilateur, du moment qu'il est compatible avec celui demandé par le sujet. Les annotations de types (telles que `(foo:mon_nouveau_type)`) sont hors programme. Pour faire de vrais programmes, on crée un fichier .mli qui spécifie les types, et Caml vérifie que ce qu'il a inféré est compatible (mais c'est aussi hors programme).
- Le sujet suppose souvent que les fonctions ne seront appelées qu'avec des arguments valides, vous n'avez pas à les vérifier. En revanche, si vous êtes amenez à écrire un cas de filtrage qui normalement ne se produit pas, le mieux est d'utiliser `failwith` plutôt que de renvoyer une valeur arbitraire.
- N'hésitez pas à revenir à la ligne chaque fois que cela rend le code plus clair, même un peu. Je n'ai vu aucun retour à la ligne superflu, mais beaucoup de manquants. Indentez au même niveau les mots-clés qui se correspondent :

`for...done`, `if...then...else`, etc. Donc avec un retour à la ligne juste avant.
Préférez
```
let foo = ... in
let bar = ... in
let baz = ... in
...
```
à
```
let foo = ... and bar = ... and baz = ... in
...
```
dès que les définitions sont un peu longues.
L'indentation n'est pas que de la coqueterie, elle rend le code plus lisible, c'est l'équivalent de faire des phrases claires en français.

- Certaines parenthèses non nécessaires rendent le code plus clair au goût de certains, en regroupant des expressions d'environ une demi-ligne. Elles sont acceptées. Mais en général, les parenthèses inutiles surcharge le code et gênent réellement la lecture lorsqu'on lit beaucoup de code. Écrivez toujours les expressions suivantes sans parenthèses : `f x`, `!n`, `x := !x *2`,
```
match foo,bar with
  | 42,_ -> ...
  | _,42 -> ...
  | _,_ -> ...
```
Écrivez des fonction currifiées !
- Testez et commentez vos fonctions, on ne le répétera jamais assez.

# 2 L-systèmes

## 2.1 Fonctions auxiliaires

**Question 2.1.** Écrire `print_int_list`. `print_int_list [0;2;1;0;1;2;1;0]` affiche `02101210`. (bonus : 6 tokens).

**Question 2.2.** Écrire `list_max : 'a list -> 'a option` qui renvoie `None` sur une liste vide. `list_max` renvoie le maximum d'une liste. Écrire éventuellement une fonction auxiliaire qui n'utilise pas le type `option`. On rappelle que le type `option` est déjà défini en caml :

```
type 'a option =
  | None
  | Some of 'a
```

## 2.2 Retour sur le mot de Thue-Morse et mots infinis

Un morphisme est une fonction $A^* \to B^*$ telle que $\forall u, v \in A^* \quad f(uv) = f(u)f(v)$.

**Question 2.3.** Vérifier oralement qu'il suffit de donner l'image des lettres pour définir un morphisme : $f(a_1 \dots a_n) = f(a_1) \dots f(a_n)$.

Un ***L-système*** (en fait un $D0L$-système) sur un alphabet $A$ est un couple $(m_0, f)$ où $m_0 \in A^+$ et $f$ est un morphisme. C'est-à-dire

```
type 'a l_systeme = {
  axiome: 'a list;
  f: 'a -> 'a list
}
```

On prendra par exemple pour alphabet l'ensemble des 256 caractères, i.e. `char` pour le type `'a`. Mais ce n'est pas obligatoire. La dynamique est la suivante : on part de $m_0$, on pose $m_{i+1} := f(m_i)$ et on itère.

**Question 2.4.** Implémenter `l_systeme` qui prend $n \in \mathbb{N}$ et un L-système, et renvoie $m_n$ (utiliser éventuellement `flat_map`).

**Question 2.5.** Tester sur le L-système suivant :$(0, \begin{cases} 0 \to 01 \\ 1 \to 10 \end{cases})$ et vérifier expérimentalement que l'on obtient le mot de Thue-Morse pour les 32 premiers caractères (cf. TP sur les automates).

**Question 2.6.** (La réponse est dans la question suivante, ne pas la lire tout de suite.) Trouver une condition nécessaire et suffisante pour que pour tout $i$, $m_i$ soit un préfixe de $m_{i+1}$.

**Question 2.7.** Écrire `mot_infini: l_systeme -> bool` qui teste si $m_0$ est préfixe de $m_1$.

Si de plus la suite $(|m_i|)_{i\in\mathbb{N}}$ est non bornée, tous les $m_i$ sont préfixes d'un unique mot infini, qui est donc défini par le L-système. C'est le cas pour le mot de Thue-Morse.

Si on ne suppose pas $f : A \to A^+$, certaines lettres peuvent avoir pour image $\varepsilon$. On n'est plus sûr d'avoir des mots de plus en plus grands. Une lettre $\alpha$ est dite ***mortelle*** si $\exists n, f^n(\alpha) = \varepsilon$.

**Question 2.8 (bonus).** Écrire `taille_alphabet: int l_systeme -> int` qui renvoie la taille de l'alphabet utilisé par $(m_i)_{i\in\mathbb{N}}$. On supposera le morphisme $\mathbb{N} \to \mathbb{N}$. On supposera de plus que la réponse est de la forme $[\![0, n]\!]$. Utiliser `list_max`. Tester avec le code fourni.

**Question 2.9 (bonus).** Déterminer les lettres mortelles par l'algorithme suivant. On prendra un `int l_systeme`. On stocke dans un tableau l'image de chacune des lettres. Les lettres dont l'image est $\varepsilon$ sont mortelles, on les ajoute dans une liste `a_eliminer`. Tant que cette liste est non vide, on retire une lettre de cette liste, on la met dans `resultat`, et on la retire de l'image de toutes les lettres (1). Si l'image d'une lettre devient $\varepsilon$, on l'ajoute à `a_eliminer` (2).

**Question 2.10 (bonus).** On améliore l'opération (1) ainsi : une matrice `nb` stocke dans la case $(\alpha, \beta)$ le nombre d'occurences de $\alpha$ dans l'image de $\beta$. On améliore l'opération (2) ainsi : un tableau `longeur` stocke la longueur de chacune des images (qui diminue quand on supprime des lettres). Refaire la question précédente sans stocker l'image des lettres.

## 2.3 Dessins de L-systèmes

Les L-systèmes on été introduits par Lindemayer en 1968 pour modéliser la croissance des plantes.

L'idée est de piloter une tortue qui peut avancer en traçant un trait ou changer de direction. Elle lit un mot (généré par le L-système) sur l'alphabet $\{Avancer, Tournerdroite, Tournergauche\}$ et exécute les lettres une à une.

Un état de la tortue est représenté par

```
type etat = {x:float; y:float; dir:float};;
```

Les lettres qu'elle peut lire sont

```
type lettre =
  | A (* Avance *)
  | G (* Gauche *)
  | D (* Droite *)
;;
```

**Question 2.11.** Écrire `trace : float -> float -> etat -> lettre list -> etat` qui prend en argument le pas (distance à parcourir en lisant A), l'angle (de combien faut-il tourner en lisant D ou G), l'état initial et le mot à tracer.

**Question 2.12.** Tester les fonctions avec le code fourni puis définir le L-système qui trace le flocon de Von Koch.

**Question 2.13.** On ajoute les lettres suivantes : voler (avancer sans tracer), sauver l'état courant, restaurer le dernier état sauvé.

```
type lettre =
  | A (* Avance *)
  | V (* Vole *)
  | G (* Gauche *)
  | D (* Droite *)
  | S (* Sauvegarde *)
  | R (* Restauration *);;
```

Modifier `trace` en comprenant l'aide fournie (stack signifie pile).

**Question 2.14.** Tester avec le code fourni puis définir un arbre où chaque branche est remplacée par un Y.

**Question 2.15.** On ajoute deux lettres : B (bourgeon, lieu de futur branchements multiples) et T (tige, lieu de croissance). Ce sont les seules lettres modifiées par le morphisme. Modifier `trace`, tester avec le code fourni, définir vos propres plantes.