# Génération automatique de figures

Lucas Michel

Simon Lemal

18 avril 2021

## Introduction

Ce document constitue le rapport du projet réalisé dans le cadre du cours *INFO0054-1 – Programmation fonctionnelle*. La première section décrit la représentation des *L-systèmes* utilisée dans notre projet et le choix d'implémentation pour la génération des chaînes associées. Dans la deuxième section, nous décrivons notre implémentation de l'interpréteur du *langage tortue*. La troisième section détaille le processus de traduction en *SVG*. Finalement, la quatrième section propose divers exemples de L-systèmes. En particulier, un L-système associé au *flocon de Koch* est présenté.

## 1 Implémentation de L-système

### 1.1 Représentations utilisées

Dans le fichier `lsystem.scm`, nous avons choisi les représentations suivantes :

— Un symbole S n'ayant pas de paramètre est représenté par le string `"S"`.

— Un symbole S ayant un ou plusieurs paramètres est représenté par une liste dont le `car` est le string `"S"` et dont le `cdr` est la liste des expressions arithmétiques associées aux paramètres. Rappelons qu'une expression arithmétique est soit un nombre, soit une variable (un symbole Scheme différent de `'+`, `'-`, `'*`, `'/` et `'()`), soit une liste dont le `car` est un des symboles `'+`, `'-`, `'*` ou `'/` et dont le `cdr` est une liste d'expression arithmétique.

— Une règle de la forme $x \to X$ qui n'est pas associée à une probabilité est représenté par une liste dont le `car` est la représentation du symbole $x$ et dont le `cdr` est une liste constituée des représentations des symboles de $X$.

— Une règle de la forme $x \to X$ associée à une probabilité $p \in ]0, 1[$ est représentée par une liste de trois éléments dont le premier est la représentation du symbole x, le second est la liste constituée des représentations des symboles de $X$ et le troisième est le nombre $p$.

Considérons un L-système $L = (N, \Sigma, A, P, T)$ où

— N est un ensemble de symboles non-terminaux, dont chaque symbole peut éventuellement être accompagné de paramètres,

— $\Sigma$ est un ensemble de symboles terminaux, dont chaque symbole peut éventuellement être accompagné de paramètres,

— $A$ est un axiome (ie : une suite finie non vide de $N \cup \Sigma$,
— $P$ est un ensemble de règles de production,
— $T$ est un ensemble de règles de terminaison.

Nous avons choisi de représenter le L-système $L$ par une liste de trois éléments

`(axiom p-rules t-rules)`

où

— `axiom` est la liste de la représentation des symboles de $A$,
— `p-rules` est la liste des représentations des règles de $P$,
— `t-rules` est la liste des représentations des règles de $T$.

Les informations relatives à $N$ et $\Sigma$ n'étant pas utile en pratique, nous avons décidé de ne pas les garder en mémoire. En guise d'exemple, le L-système relatif au *flocon de Koch* présenté dans la section 4 est représenté par la liste

```
(("T" "-" "-" "T" "-" "-" "T")
 (("T" ("T" "+" "T" "-" "-" "T" "+" "T")))
 ())
```

Pour illustrer notre représentation d'un L-système manipulant des paramètres, la représentation du L-système de l'exemple 2.3 de l'énoncé du projet est donné par la liste

```
(("A" 16)
 ((("A" x) (("A" (* 0.5 x)) "B"))
  ("B" ("G" "B") ))
 ((("A" x) (("F" x)))
  ("B" ("G"))))
```

Pour conclure, la représentation du L-système nommé *arbre* dans l'énoncé du projet est donné par la liste

```
(("T")
 (("T" ("T" "<" "+" "T" ">" "T" "<" "-" "T" ">" "T") 0.33)
  ("T" ("T" "<" "+" "T" ">" "T") 0.33)
  ("T" ("T" "<" "-" "T" ">" "T") 0.34))
 ())
```

et met en exergue un L-système mettant en jeu des probabilités dans son ensemble de règles de production.

## 1.2 Choix de l'implémentation pour la génération de chaînes

Tout d'abord, notons que la représentation des symboles présentés dans la sous-section 1.1 est différente de celle demandée par le langage tortue. En effet, nous représentons le symbole T muni du paramètre 1 par la liste `("T" 1)` et non `"T[1]"`. Cela nous permet de manipuler plus facilement les paramètres via la manipulation des listes aisée en Scheme. La fonction `lstr->tstr`

se charge de cette conversion et sera appelé une fois pour toute lorsque la chaîne souhaitée est générée mais n'est pas encore interprétable par le langage tortue.

Afin de générer les chaînes d'ordre $K$ d'un L-système donné, nous avons procédé comme proposé par le pseudo code de la section 2.2 de l'énoncé du projet. Plus précisément, si on considère un naturel $K$ et un L-système qui génère un turtle string, alors la fonction `lsystem.develop-string` renvoie la liste des symboles correspondante à $K$ applications successives des règles de production du L-système sur les axiomes. Si $K = 0$, la représentation de l'axiome est renvoyée. La fonction `lsystem.terminate-string` se charge d'appliquer les règles de terminaison d'un L-système donné à une liste de représentation de symbole. Finalement, `lsystem.generate-string` renvoie la chaîne d'ordre $K$ du L-système interprétable par le langage tortue.

Il reste à expliquer comment nous avons implémenté l'application d'un ensemble de règles (typiquement, de production ou de terminaison) à un symbole dont les paramètres potentiels sont supposés être des nombres. Si `rules` est la représentation de l'ensemble de règles $R$, et si `x` est la représentation d'un symbole $x$, alors la fonction `apply-rules` renvoie

— la liste constituée uniquement de `x` si aucune règle de $R$ ne s'applique à $x$,
— la liste des représentation des symboles de $X$ si la règle $x \to X$ constitue l'unique règle de $R$ qui s'applique à $x$.
— la liste des représentation des symboles de $X_i$ avec une probabilité $p_i$ si les règles

$$x \to X_1 \text{ (prob. } p_1), ..., x \to X_n \text{ (prob. } p_n)$$

sont toutes les règles de $R$ qui s'appliquent à $x$ et sont telles que $\sum_i p_i = 1$.

Nous sommes donc réduit à expliciter l'application d'une unique règle $x \to X$ représentée par `(x ls)` au symbole $x$ représenté par `x`. La fonction `apply-rule` a cette responsabilité. Si $x$ n'admet pas de paramètre, alors la liste `ls` est renvoyée. Si $x$ admet un ou plusieurs paramètres supposés être des nombres, alors $x$ est représenté par `("x" y1 ... yn)` et la liste résultante de la règle prend donc la forme d'une liste dont les éléments sont de la forme `"li"` ou `(("li" a1 ... ak)` où `a1`, ..., `ak` sont des expressions arithmétiques qui dépendent de `y1`, ..., `yn`. La fonction `eval-symbol` se charge de remplacer, pour chaque élément `("li" a1 ... ak)` (ou `"li"`) les symboles `'y1`, ..., `'yn` apparaissant dans les expressions `a1`, ..., `ak` par leur valeur telle que spécifiée dans les paramètres de `x`, et d'évaluer ces expressions arithmétiques.

Pour illustrer ce qui a été dit, observons le résultat de l'application de la règle

$$A[y_1, y_2] \to T[0.5y_1] - T[y_1 + y_2]$$

représentée par la liste

```
(("A" y1 y2) (("T" (* 0.5 y1)) "-" ("T" (+ y1 y2)))
```

au symbole $x$ représenté par `("A" 2 1)`. Grâce à la fonction `eval-symbol` appliquée à chaque élément de la liste résultante de la règle, on obtient cette même liste à laquelle on a remplacé respectivement chaque occurrence des paramètre `y1` et `y2` par 2 et 1. Cela donne la liste

```
(("T" 1) "-" ("T" 3))
```

qui correspond bien à $T[0.5 * 2] - T[2 + 1]$.

## 2 Interprétation du langage tortue

Pour implémenter la fonction `cons-init-turtle-store`, nous commençons par implémenter une fonction `cons-turtle-store`, qui étant donné un « drawing » et un angle, va construire et renvoyer un « turtle f-store » encapsulant le « drawing » et l'angle.

Le « turtle f-store » renvoyé n'est rien d'autre qu'une fonction d'un paramètre `t-symb` (qui est supposé être une chaîne de caractères). Si la fonction est appelée avec pour argument la chaîne vide, elle renvoie le « drawing » dans sa clôture. Si elle est appelée avec un argument qui n'est pas un symbole tortue, une erreur est renvoyée. Finalement, si le « f-store » est appelé avec pour argument un symbole tortue, la fonction commence par créer un nouveau « drawing » à partir de l'ancien (qui est stocké dans sa cloture) et en fonction du symbole fourni (`"T"`, `"F"`, `"<"`, `">"`, `"+"` ou `"-"`, éventuellement suivi d'un paramètre `"[xx.x]"`).

Si le symbole est `"F"`, il suffit de rajouter une nouvelle polyligne contenant un nouveau point, ce qui est fait grâce à la fonction `drawing.peek-new-polyline`. Si le symbole est `"T"`, il suffit d'ajouter un nouveau point à la dernière polyligne du dessin, ce qui se fait grâce à la fonction `drawing.update-polyline`, qui est très similaire à `drawing.peek-new-polyline`. La nouvelle position est calculée à partir de la dernière position du dessin, translaté de `defx` unités dans la direction du dessin, où `defx` vaut 1 si le symbole n'a pas de paramètre et est égal au paramètre sinon. Que le symbole soit `"F"` ou `"T"`, il faut également mettre à jour la « bounding box » pour que celle-ci contienne la nouvelle position, ce qui est fait par la fonction `drawing.update-bounding-box`.

Si le symbole est `"<"`, il suffit de créer un « d&p » à partir de la direction et de la dernière position du dessin, et de l'empiler sur la pile du dessin. Si le symbole est `">"`, il faut dépiler la dernière position sauvée (s'il y en a une), mettre à jour la direction du nouveau dessin et initialiser une nouvelle polyligne en la position dépilée (à nouveau, la fonction `drawing.peek-new-polyline` fait le travail). Par contre, il n'est pas nécessaire de mettre à jour la « bounding box » car la position dépilée à forcément dû être empilée plus tôt, donc appartient déjà à la « bounding box ».

Si le symbole est `"+"` ou `"-"`, la direction du dessin est mise à jour en y ajoutant ou retirant `defx` fois l'angle de rotation stocké par le « f-store ».

Une fois que toutes les composantes du dessin sont mises à jour, on peut créer un nouveau dessin avec `cons-drawing` puis utiliser ce nouveau dessin pour créer un nouveau « f-store », en utilisant comme angle l'angle stocké dans la clôture du « f-store » qui s'évalue.

Pour gérer facilement les symboles ayant des paramètres, nous utilisons des expressions régulières. L'expression `#px"^[TF+-](\\[\\d+(.\\d+)?\\])?$"` correspond à un des symboles `"T"`, `"F"`, `"+"` ou `"-"`, éventuellement suivi d'un paramètre entre crochets, ce paramètre étant un entier ou un réel (en notation standard). Si `t-symb` correspond à cette expression régulière, on définit `s` comme étant le caractère de tête, qui correspond à `#px"^[TF+-]"`, et `x` comme étant le nombre correspondant à la chaîne de caractères entre les crochets. Pour cela, on utilise la fonction `string->number`. Cette fonction renvoie `#f` si la chaîne ne correspond pas à un nombre, ce qui dans notre cas n'arrive que si `t-symb` n'a pas de paramètre. Ainsi, en définissant `defx` via l'expression `(if x x 1)`, `defx` vaut 1 si le symbole n'a pas de paramètre et est égal à la valeur du paramètre sinon.

# 3 Traduction en SVG

Pour traduire un « drawing » en SVG, nous commençons par utiliser la « bounding box » du « drawing » comme `viewBox`. Cela permet d'éviter les calculs du au changement d'échelle et de laisser l'interpréteur les gérer.

La fonction `format` permet de gérer facilement la substitution des arguments dans les différentes chaînes de caractères.

Nous avons naturellement choisi d'utiliser la balise `polyline` pour représenter une polyligne. Étant donné une polyligne (une liste de la forme `((x1 . y1) (x2 . y2) ...)`) et une épaisseur de trait, la fonction `polyline->svg` renvoie la chaîne de caractères représentant cette polyligne en SVG (c'est à dire une chaîne de la forme `<polyline points="x1, y1 x2, y2 ..." fill="none" stroke="black" stroke-width="~a"/>` où `~a` est remplacé par l'épaisseur de trait spécifiée.

Pour représenter le « drawing » en entier, la fonction `drawing->svg` commence par formatter l'entête du fichier SVG à produire en y substituant les arguments `height` et `width` donné ainsi que les dimensions de la « bounding box ». Ensuite, il suffit d'appliquer la fonction `polyline->svg` sur chaque élément de la liste de polyligne du « drawing » et de concaténer toutes les chaînes de caractères. Nous utilisons comme épaisseur empirique de trait un centième de la racine carré de la dimension la plus grande de la « bounding box ». Ainsi, le trait n'est ni trop fin ni trop épais pour toutes les figures considérées jusqu'à présent.

# 4 Exemples de L-systèmes

Dans cette section sont présentés divers L-systèmes, dont un associé au *flocon de Koch*. Tous sont implémentés dans le fichier `lsystem.scm`. On utilise les notations de l'énoncé du projet.

**Flocon de Koch**
- $N = \emptyset$
- $\Sigma = \{T, +, -\}$
- Axiome : $T--T--T$
- Règle de production :
  $T \to T+T--T+T$
- Règle de terminaison : /
- Angle : 60 degrés

**Courbe de Koch** ($90°$)
- $N = \emptyset$
- $\Sigma = \{T, +, -\}$
- Axiome : $T$
- Règle de production : $T \to T-T+T+T-T$
- Règle de terminaison : /
- Angle : 90 degrés

**Arbre binaire**
- $N = \{A, B\}$
- $\Sigma = \{T, +, -, <, >\}$
- Axiome : $A$
- Règles de production :
  $A \to B<+A>-A$ (prob. 0,75)
  $A \to A$ (prob. 0,75)
  $B \to BB$
- Règle de terminaison :
  $A \to T$
  $B \to T$
- Angle : 45 degrés

**Branche**

— $N = \{X\}$
— $\Sigma = \{T, +, -, <, >\}$
— Axiome : $A$
— Règles de production :
$X \rightarrow T- << X > +X > +T < +T\,X > -X$
$T \rightarrow T\,T$
— Règle de terminaison : /
— Angle : 25 degrés

**Sierpinski arrowhead**

— $N = \{A, B\}$
— $\Sigma = \{T, +, -\}$
— Axiome : $A$
— Règles de production :
$A \rightarrow B - A - B$
$B \rightarrow A + B + A$
— Règle de terminaison :
$A \rightarrow T$
$B \rightarrow T$
— Angle : 60 degrés

# Références


[1] *L-system – Wikipedia.* URL : `https://en.wikipedia.org/wiki/L-system` (visité le 13/04/2021).

[2] *List of fractals by Hausdorff dimension – Wikipedia.* URL : `https://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimension` (visité le 18/04/2021).

[3] *Space-filling curve – Wikipedia.* URL : `https://en.wikipedia.org/wiki/Space-filling_curve` (visité le 18/04/2021).