Option complémentaire
Informatique
Langages formels et L-system

14 décembre 2019

Copyright

Avertissement

Table des matières

1 Introduction

Ce cours s'adresse à des élèves dont les connaissances en informatiques se limitent à l'utilisation de logiciels et à des connaissances de base (boucles et fonctions) en programmation.

Il ne s'adresse pas spécifiquement à des élèves non mathématiciens, car ceux-ci peuvent aussi y trouver de l'intérêt, mais surtout à des personnes qui ne se destinent ni à l'informatique, ni au mathématiques. C'est pourquoi, l'accent à été mis sur la plus grande clarté possible dans l'exposé des notions qui relèvent des mathématiques et sur les exemples. Il constitue à mon avis une approche adaptée pour des personnes intéressées par la biologie et par les langues.

Les objectifs sont multiples.

Le premier d'entre eux est de montrer aux élèves que l'analyse formelle est un préalable nécessaire à la compréhension des langages, en particulier informatiques.

Le deuxième est de présenter un exemple de la diversité des modes de représentation de ces langages.

Le troisième est d'illustrer le lien qui existe entre les langages et certains processus naturels, lien qui montre l'importance de l'étude des formalismes présentés.

Le dernier objectif est finalement de permettre à l'élève d'utiliser des connaissances très sommaires de programmation pour analyser les processus mécaniquement dans des proportion inaccessibles à l'homme.

2 Langage formel

Rendre compte de la notion de langage formel le plus simplement possible n'est pas chose facile. Il se trouve que c'est la rigueur du langage mathématique qui permet d'en exprimer au mieux l'idée. Nous allons donc l'utiliser le plus simplement possible étant donné les difficultés qu'il peut représenter. Il sera néanmoins bien souvent plus une aide qu'un obstacle.

Tout langage est au départ constitué d'un alphabet. Il faut considérer le terme d'alphabet au sens large du terme comme un simple ensemble de signes ou lettres, même non alphabétiques au sens commun du terme. Ainsi, les ensembles suivants notés Σ_i constituent des exemples d'alphabet :

$$\Sigma_1 = \{\text{€}, \text{✠}, \text{❦}, \checkmark, 1, 4, \textvisiblespace\}$$
$$\Sigma_2 = \{\neg, \$, \text{ⓐ}, \pounds\}$$
$$\Sigma_3 = \{\text{a}, \text{b}, \textvisiblespace, \epsilon\}$$

A partir de l'alphabet, on peut alors définir ce qu'est un mot. Il s'agit simplement d'une suite finie de lettres prises dans l'alphabet. Ainsi, on peut former les mots suivants sur l'alphabet Σ_1 : €€✠€€✓ ou ✓14✠✠ Pour l'alphabet Σ_3, on pourrait avoir : aababbab ou aabbbba mais pas aab¬

Si m est un mot sur un alphabet Σ, on écrira :

$$\mathrm{m} \in \Sigma^*$$

pour dire que le mot m appartient à l'ensemble Σ^* des mot possibles formé sur l'alphabet Σ. Finalement, on définit ϵ comme un mot vide. La raison pour laquelle on fait cela sera vue plus tard.

Maintenant qu'on a des mots, le problème est de pouvoir les utiliser. Si on considère les mots suivants de Σ_3 :

abaab␣ aba␣ aabaaba␣

il serait possible de former queque chose qui ressemble à des phrases en plaçant ces mots les uns à côté des autres. Cette opération se nomme la concaténation et on la note par un point. Ainsi, on va formellement définir la concaténation ainsi :

Soit m et n $\in \Sigma^*$ Alors, m.n=mn

On peut ainsi tenter de reproduire l'apparence formelle d'un langage par concaténation des mots ci-dessus (en s'imaginant un espace invisible) :

abaab␣.aba␣.aabaaba␣ = abaab aba aabaaba

Certes, le résultat ne correspond qu'en apparence à un langage. Mais, l'alphabet est très restreint.

De manière naturelle, on va donc définir la notion de langage comme un sous-ensemble de l'ensemble de tous les mots d'un alphabet donné. Un langage est donc un ensemble de mots. On peut donc faire l'union, l'intersection, ... de deux langages.

L'exemple classique est le langage des expressions arithmétiques. Il est défini à partir de l'alphabet suivant :

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$$

Le langage des expressions arithmétiques s'écrit alors :

$$L(\text{expr. arithm.}) = \{\ldots, 342, 34 + 4, 45 + 34, 2 = 4, \ldots\}$$

Relevez qu'il ne s'agit pas de l'ensemble de tous les mots formé sur l'alphabet, mais d'un sous ensemble qui ne contient pas l'expression 34+++43==, par exemple. Évidemment, il faut spécifier comment on réalise le choix de chaque expression.

Exercice 1 Sur l'alphabet des chiffres binaires (que vous noterez précisément), donnez le langage L des nombres à trois chiffres.

Exercice 2 Pour la molécule d'ADN, définissez un alphabet de ses composants et déterminez quelques éléments du langage des *codons*, acide aminé constitué de triplets de nucléotides. Combien peut-il y en avoir si leur arrangement tient compte de l'ordre et qu'il peut y avoir des répétitions.

Indication : les nucléotides sont les briques de base de l'ADN.

La génération ou la vérification des expressions valables pour un langage donné peut se faire de différentes manières. Nous allons présenter dans la section suivante les moyens qui nous permettrons de décrire simplement des langages particuliers. On peut utiliser pour cela soit un automate soit une expression régulière.

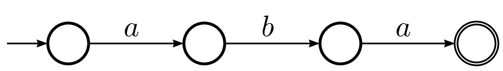
3 Langage rationnel

3.1 Les automates

Une manière particulièrement compréhensible de tester l'appartenance d'une expression à un langage est d'utiliser un automate. Il s'agit d'une sorte de machine symbolique capable de générer un ensemble de mots. On la suit dans la construction de ceux-ci selon des flèches qui mènent d'un état à l'autre par la concaténation successive de certaines mots de l'alphabet. Pour des raison de compréhension, on va commencer par donner un exemple d'automate à travers une représentation graphique qui fait correspondre des ronds aux états et des flèches aux éléments de l'alphabet. Un état initial est un rond sur lequel arrive une flèche vide et un état final est constitué d'un double cercle. Ainsi, on représente un automate reconnaissant le mot :

$$aba \text{ sur l'alphabet } \Sigma = \{a, b\}$$

par la figure suivante :

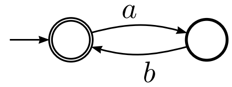


Relevons qu'il est possible de considérer l'unique mot construit par cet automate comme un langage qu'on peut définir mathématiquement très simplement par :

$$L = \{m \in \Sigma^* \,|\, m = aba\}$$

où la barre verticale | signifie "tel que".

Évidemment, l'objectif est de permettre la construction de langages plus étendus. On peut par exemple construire le langage constitué d'une séquence de ab ainsi :

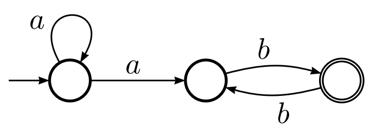


Cet exemple permet d'engendrer une infinité de mots du type ab, abab, ababab, ...

Par ailleurs, la présence d'un état initial, simultanément final, implique l'existence dans le langage d'un mot vide noté ϵ. On écrira donc mathématiquement le langage ainsi :

$$L = \{m \in \Sigma^* \,|\, m = \epsilon, ab, abab, ababab, \ldots\}$$

Un dernier exemple montre qu'il est possible de générer des langages bien plus complexes. Sans dire immédiatement ce dont il s'agit, voici l'automate :



Le langage reconnu est constitué de mots comprenant en premier lieu au moins un a, suivi par un nombre impair de b. On peut aussi écrire :

$$L = \{m \in \Sigma^* \,|\, m = a^n b^m,\ n \in \mathbb{N}^* \text{ et } m = 2 \cdot k + 1,\ k \in \mathbb{N}\}$$

où $\mathbb{N}^*$ est l'ensemble des nombres naturels (entiers positifs) sans le zéro et $\mathbb{N}$ celui avec le zéro.

En conclusion, on peut définir un automate de manière générale comme un ensemble constitué

— d'un alphabet Σ,
— d'un ensemble fini d'état, généralement noté E,
— d'un ensemble d'état initiaux I tel que $I \subseteq E$,
— d'un ensemble d'état finaux F tel que $F \subseteq E$ et
— d'une fonction de transition

$$\delta : E \times \Sigma \longrightarrow E$$

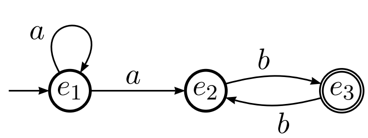
Où le signe $\subseteq$ signifie "appartient à".

Le dernier point de cette définition mérite des explications. La représentation graphique des automates donnée ci-dessus met clairement en évidence les transitions (flèches) entre états. Celles-ci doivent aussi trouver une expression mathématique. C'est la fonction de transition.

Reprenons le dernier exemple d'automate donné ci-dessus. Considérons chacun des points permettant de le définir mathématiquement.

L'alphabet tout d'abord est clairement donné par $\Sigma = \{a, b\}$.

L'ensemble des états est $E = \{e_1, e_2, e_3\}$ et on peut en donner une représentation légèrement plus complète ainsi :



L'ensemble des états initiaux se résume à $I = \{e_1\}$ et celui des états finaux à $F = \{e_3\}$. La fonction de transition δ est alors donnée par :

$$\begin{array}{ll}
\delta(e_1, a) = e_1 \,|\, e_2 & \delta(e_1, b) = \perp \\
\delta(e_2, a) = \perp & \delta(e_2, b) = e_3 \\
\delta(e_3, a) = \perp & \delta(e_3, b) = e_2
\end{array}$$

où $|$ signifie “ou” et $\perp$ signifie “indéfini”.

Ce dernier exemple est aussi intéressant du point de vue du déterminisme du résultat. La fonction de transition $\delta(e_1, a)$ donne deux solution possibles : e_1 ou e_2. Cela signifie qu'à partir de l'état e_1, par la transition a, un choix non déterminé peut avoir lieu. L'automate est alors dit *non déterministe*. Cela peut constituer un problème et il est possible de le rendre déterministe. Mais nous n'aborderons pas ce point dans ce cours. Il convenait cependant de le mentionner.

Nous avons vu dans ce paragraphe une manière de *reconnaître* mécaniquement un langage simplement formé de mots issus d'un alphabet à l'aide d'un automate. Nous avons vu deux manières de représenter celui-ci, par un diagramme et mathématiquement. Chacune de ces deux expressions constitue un mode de représentation qui a des avantages et des inconvénients. Les automates soulignent clairement le caractère mécanique du processus de reconnaissance. L'expression mathématique exprime quant à elle mieux les propriétés ensemblistes et relationnelles de la notion d'automate.

Exercice 3 Construisez un automate sur l'alphabet $\Sigma = \{a, b\}$ qui reconnaît les mots suivants : aab, abb et bbb.

Exercice 4 Construisez un automate qui reconnaisse la succession des feux de circulation rouge-jaune-vert, allumé successivement et indéfiniment dans cet ordre.

Exercice 5 Construisez un automate qui reconnaisse les codons de l'exercice 2.

3.2 Les expressions rationnelles

Voyons maintenant une autre manière de représenter les langages reconnus par les automates : les *expressions rationnelles* [a]. Elles trouvent ici leur place pour souligner qu'il existe différentes approches dans la reconnaissance des langages (dits rationnels).

Premièrement, une *expression rationelle* est un mot. L'alphabet permettant de construire ce mot est particulier, puisque s'y trouve obligatoirement les symboles suivants : ( ) + . $*$ $\emptyset$ Ainsi, on peut voir l'alphabet sur lequel on construit les expressions rationelles comme composé de deux ensemble : un alphabet Σ choisi selon les besoins et l'ensemble des symboles donné ci-dessus. L'alphabet des expressions rationelles est donc formellement :

$$\Sigma^{\text{expressions rationnelles}} = \Sigma \cup \{(,),+,.,^*,\varepsilon\}$$

a. On parle aussi d'expressions régulières en raison de l'anglais *regular expression*.

où ε est le caractère vide.

Secondement, une expression rationnelle doit obéir à une *syntaxe*. Une manière de construire un langage, en tant que partie de l'ensemble des mots de Σ, est de le faire à partir d'un automate. Une autre manière est de le faire à partir d'un ensemble d'opération qui permettent de le définir complètement. On appelle ces opérations des règles de *syntaxe*. En d'autre termes, ces règles autorisent ou interdisent la production de mots particuliers sans référence au sens des mots produits. Ainsi, si e et r sont des expressions rationnelle, alors il faut ajouter que les règles de syntaxe des expressions rationnelles autorisent les trois expressions (e+r), (e.r) et (e*) comme expressions rationnelles. On dit ici simplement que les trois suites de caractères précédentes sont correctement formées pour qu'il s'agisse d'expressions rationnelles. Le sens des symboles +, . et * n'est pas précisé car il ne s'agit que de règles de syntaxe.

En définitive donc, on peut définir l'ensemble des expressions rationnelles par une règle de base (B) et par un règle récursive (R) :

B. ε et e avec e $\in \Sigma$ sont des mots sur $\Sigma \cup \{(,),+,.,^*,\varepsilon\}$

R. si e et r sont des expressions rationnelles, alors (e+r), (e.r) et (e*) le sont aussi.

Voilà l'ensemble des expressions rationnelles correctement défini. Il s'agit d'une définitions complexe sur laquelle nous ne reviendrons pas dans le cadre de ce cours. Il faut cependant retenir l'idée de syntaxe comme règles de production d'expressions. Elle nous mènera plus tard à celle de grammaire.

Jusqu'à présent nous n'avons pas parlé de sémantique. La définition rigoureuse du sens des symboles utilisés précédemment pour les expressions rationnelles $(+,.,*)$ passe par la construction d'une relation de l'ensemble des expressions rationnelles sur une partie de l'ensemble des mots. Ces symboles prennent alors une signification particulière : le + devient un ou ($\cup$ ou $|$), le . une concaténation de deux expressions rationnelles et l'étoile devient la répétition de zéro ou plusieurs fois le caractère précédent.

Ainsi, en imaginant la reconnaissance du code postal d'une ville suisse constitués de quatre chiffres, on peut écrire une expression rationnelle qui exprime cette structure de la manière suivante :

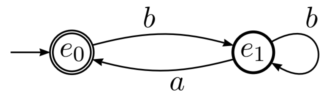
$$(0|1|2|3|4|5|6|7|8|9).(0|1|2|3|4|5|6|7|8|9).(0|1|2|3|4|5|6|7|8|9).(0|1|2|3|4|5|6|7|8|9)$$

Évidemment, cette notation n'est pas très pratique et en réalité on définit d'autres symboles qui permettent d'alléger l'écriture. Par exemple, de manière évidente :

$$[0..9]\{4\}$$

On trouvera avec Desgraupes (2008) un ouvrage très complet sur la syntaxe et la signification des symboles les plus utilisés.

Exercice 6 Exprimez l'automate suivant en français et transformez-le en expression rationnelle.



Exercice 7 Construisez l'expression régulière correspondant aux numéros de plaques minéralogiques suisses : deux lettres, suivies de cinq chiffres décimaux.

Exercice 8 Construisez l'expression régulière correspondant à l'automate de l'exercice 3, page 8.

4 Langage de Lindenmayer

Avant de nous intéresser aux langages de Lindenmayer, dit L-system, il faut pousser un peu plus avant l'étude des langages formels.

On a vu plusieurs manières de formaliser un langage simple. La formalisation des expressions rationnelles, par exemple, a fait apparaître des règles de syntaxe qui évoquent une *grammaire*. Il est intéressant ici de préciser cette notion pour pouvoir comprendre au mieux les L-system.

4.1 Grammaire

Une *grammaire* est un ensemble G d'objets tel que :

$G = \{T, N, R, S\}$ où :
- T est l'alphabet des lettres terminales,
- N est l'alphabet des lettres non terminales
- $S \in N$ est l'axiome
- $R \subseteq (T \cup N)^* \times (T \cup N)^*$ est un ensemble fini de règles de production.

Une convention pratique veut que les symboles non-terminaux s'écrivent en majuscule et les terminaux en minuscule.

Voici un exemple simple, mais assez complet.

$$\begin{aligned}
&G = \{T, N, R, S\} \text{ où :}\\
&\quad T = \{a, b, m, n\}\\
&\quad N = \{O, P\}\\
&\quad S = aO\\
&\quad R = \{O \to aO|bP \ \text{ et } \ P \to m|n\}
\end{aligned}$$

où le signe | signifie ou.

On peut donner deux exemples de génération d'éléments à partir de cette grammaire :

1. $aO \to aaO \to aaaO \to aaabP \to aaabn$
2. $aO \to aaO \to aabP \to aabm$

On comprend aisément le rôle des règles de productions et celui des lettres terminales qui vont arrêter la génération de la chaîne de caractère.

Un automate peut constituer un autre exemple intéressant. En effet, il est possible de traduire un automate par une grammaire appropriée.

$$\begin{aligned}G =& \{T, N, R, S\} \text{ où :}\\ & T = \{\text{états terminaux}\}\\ & N = \{\text{états non terminaux}\}\\ & S = \text{état initial}\\ & R = \{\text{fonction de transition}\}\end{aligned}$$

Pour plus de clarté, considérons l'automate de l'exercice 6, page 9, et tentons de le traduire en une grammaire.

$$\begin{aligned}G =& \{T, N, R, S\} \text{ où :}\\ & T = \{e_0\}\\ & N = \{e_0, e_1\}\\ & S = e_0\\ & R = \{\delta(e_0, a) = \perp\ ;\ \delta(e_0, b) = e_1\ ;\ \delta(e_1, a) = e_0\ ;\ \delta(e_1, b) = e_1\}\end{aligned}$$

Évidemment, l'existence d'une correspondance entre automate et expression régulière implique l'idée d'une formalisation des expressions régulières sous forme de grammaire. Nous n'aborderons pas ce cas ici.

Enfin, voici un autre exemple de grammaire où il n'y a pas de symbole terminaux.

$$\begin{aligned}G =& \{T, N, R, S\} \text{ où :}\\ & T = \emptyset\\ & N = \{A, B\}\\ & S = A\\ & R = \{A \to AB \;\text{ et }\; B \to A\}\end{aligned} \tag{1}$$

Celle-ci permet la génération de séquences infinies dont nous allons parler maintenant.

4.2 Génération de séquence

La production des éléments de la grammaire donnée dans le dernier exemple ci-dessus débute par l'axiome A. Ensuite, les deux règles de productions doivent être utilisées pour produire l'élément suivant : remplacer A par AB et B par A. Comme il n'y a qu'un A dans l'axiome, seule la première règle s'applique tout d'abord et on obtient AB. Puis, les deux règles doivent à nouveau être utilisées à partir de cet état pour produire le suivant.

On remplace donc les A par AB et B par A pour chaque lettre de AB. On obtient ainsi ABA. Et on continue indéfiniment. On obtient alors :

$$\begin{aligned}
&A\\
&AB\\
&ABA\\
&ABAAB\\
&ABAABABA\\
&ABAABABAABAAB
\end{aligned}\tag{2}$$

Ces chaînes n'ont pas encore de signification. Mais on peut déjà envisager différentes règles de productions permettant de générer de multiples chaînes de caractères. Beaucoup de questions peuvent alors se poser concernant les propriétés des chaînes ainsi constituées.

Voici enfin un autre exemple très connu de grammaire permettant de générer des chaînes auxquelles on peut attacher une signification par interprétation graphique :

$$G = \{T, N, R, S\} \text{ où :}\tag{3}$$
$$\begin{aligned}
&T = \{+, -\}\\
&N = \{F\}\\
&S = F\\
&R = \{F \rightarrow F + F - F - F + F\}
\end{aligned}$$

Notez la séparation entre les symboles terminaux, sans règles de production et le symbole non terminal auquel correspond une règle de production.

La génération des chaînes est alors la suivante :

$$\begin{aligned}
&F\\
&F + F - F - F + F\\
&F + F - F - F + F + F + F - F - F + F - F + F - F - \hookleftarrow\\
\hookrightarrow\ &F + F - F + F - F - F + F + F + F - F - F + F
\end{aligned}\tag{4}$$

Encore une fois, le sens de ces chaînes n'est pas encore évident. Par ailleurs, on voit aussi que la génération des caractères de ces chaînes devient très vite inaccessible au possibilités humaines en raison du nombre élevé de caractères produits. Une mécanisation de la production est donc rapidement nécessaire.

4.3 L-system

Les deux derniers exemples ci-dessus ((3) et (1)) sont des systèmes de Lindenmayer ou L-system. De quoi s'agit-il ? L'ouvrage de référence en la matière, *The Algorithmic Beauty of Plants* (Prusinkiewicz, 2004, p.2-3), en présente le contexte en ces termes :

> « In 1968 a biologist, Aristid Lindenmayer, introduced a new type of string-rewriting mechanism, subsequently termed L-systems. The essential difference between Chomsky grammars and L-systems lies in the method of applying productions. In Chomsky grammars productions are applied sequentially, whereas in L-systems they are applied in parallel and simultaneously replace all letters in a given word. This difference reflects the biological motivation of L-systems. Productions are intended to capture cell divisions in multicellular organisms, where many divisions may occur at the same time. Parallel production application has an essential impact on the formal properties of rewriting systems. »

L'objectif est donc de modéliser des mécanisme de division (cellulaire). Le moyen utilisé est un mécanisme de réécriture. Quoi de plus naturel donc que d'utiliser une grammaire précisément basée sur la réécriture. Les règles de production s'y prêtent bien et les deux exemples présentés ci-dessus en sont une belle démonstration. Il ne s'agit pas ici d'une division cellulaire, mais de la réplication d'un motif sur lui-même. Une grammaire en est ici l'origine et nous verrons par la suite qu'elle peut se traduire par une récursivité des algorithmes nécessaires pour la production de motifs conséquents.

Mais, pour bien comprendre la signification des mécanismes de production des L-system dans la réalité, il faut maintenant en donner une interprétation graphique.

4.4 Interprétation graphique

Il s'agit ici d'utiliser les chaînes de caractères produites par les grammaires pour en faire une représentation graphique. L'articulation d'un mécanisme de production général et d'une analyse des briques composant les structures végétales est certainement à l'origine de cette idée. Elle a trouvé son expression complète grâce aux possibilités de dessin proposés par le langage de programmation Logo et la tortue (Turtle) qui lui est attachée.

L'interprétation consiste à utiliser les caractères de l'ensemble terminal N comme des instructions d'orientation de la tortue et les symboles non-terminaux comme des instruction de construction du chemin. Classiquement, on a :

F avancer d'un pas en avant,

$+$ tourner à gauche d'un angle donné,

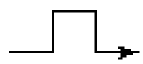
$-$ tourner à droite d'un angle donné.

Cette interprétation permet de suivre les chaînes de caractères produites par la grammaire pour en tracer une représentation graphique.

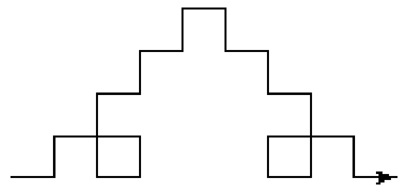
4.4.1 Koch

L'exemple donné par la grammaire (3), correspondant aux chaînes (4) peut ainsi prendre l'apparence des figures 1 pour un angle de 90°.

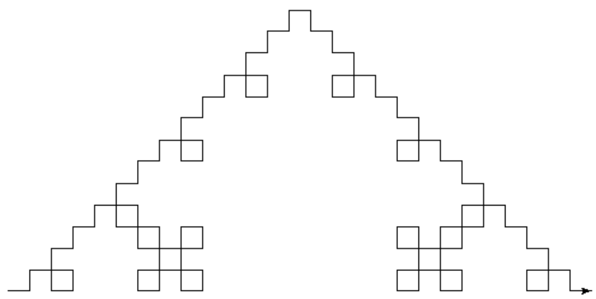
La figure 1(a) permet de suivre l'interprétation graphique à la main, caractère après caractère. Mais, on hésite déjà à le faire à la figure suivante pour des raisons évidentes.



(a) F+F-F-F+F



(b) F+F-F-F+F+F+F-F-F+F-F+F-F-F+F-F+F-F-F+F+F+F-F-F+F



(c) Plus encore ...

FIGURE 1 – *La courbe de Koch*

A part l'esthétique des figures 1, cette interprétation graphique semble ne pas apporter grand chose. Pourtant, les objets étudiés dans *The Algorithmic Beauty of Plants* (Prusinkiewicz, 2004, p.8-9) et reproduits[b] dans la figure 2 sont déjà plus intéressants et montrent des structures qui commencent à ressembler vaguement à des cristaux par exemple.

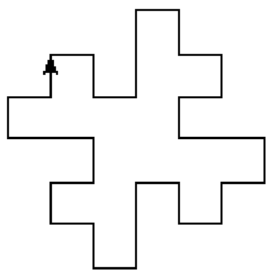
4.4.2 Modélisation des plantes

Un autre exemple d'utilisation d'une grammaire interprétée de manière particulière est donné par la modélisation de la croissance des plantes. Turtle est ici utilisé comme précédemment. Mais l'alphabet est étendu par deux nouveaux signes : les crochets [et]. Leur signification dans le cadre de Turtle est spéciale :

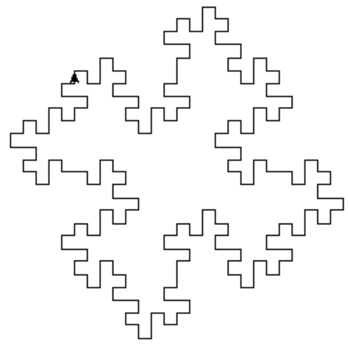
- **le crochet ouvrant** [signifie qu'il faut mémoriser la position et la direction courante de la tortue et
- **le crochet fermant**] signifie qu'il faut faire revenir la tortue à la position et direction précédemment mémorisée.

Cette opération de mémorisation est nécessaire pour permettre une structure d'arbre qui n'est pas possible sans cela puisque le parcours de la tortue est alors continu.

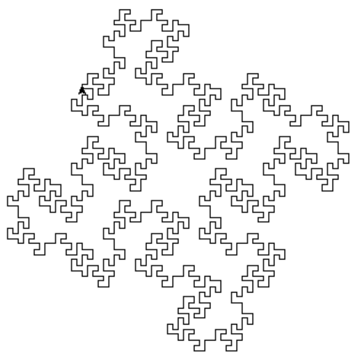
b. Le code permettant d'obtenir ces structures sera présenté plus loin.



(a) F-F+F+FF-F-F+F (axiome F-F-F-F)



(b) F-F+F+FF-F-F+F (axiome F-F-F-F)



(c) F+FF-FF-F-F+F+FF-F-F+F+FF+FF-F (axiome F-F-F-F)

FIGURE 2 – *Une autre courbe de Koch*

Voyons un exemple d'une telle grammaire.

$$\begin{aligned} G =& \{T, N, R, S\} \text{ où :} \\ & T = \{+, -, [,]\} \\ & N = \{F\} \\ & S = F \\ & R = \{F \rightarrow F[+F]F[-F]F\} \end{aligned} \tag{5}$$

Après deux fois l'application de la règle (deux générations), on obtient la chaîne suivante :

F[+F]F[-F]F[+F[+F]F[-F]F]F[+F]F[-F]F[-F[+F]F[-F]F]F[+F]F[-F]F

Encore une fois, l'interprétation n'est pas triviale, pas plus que la programmation de la mémorisation.

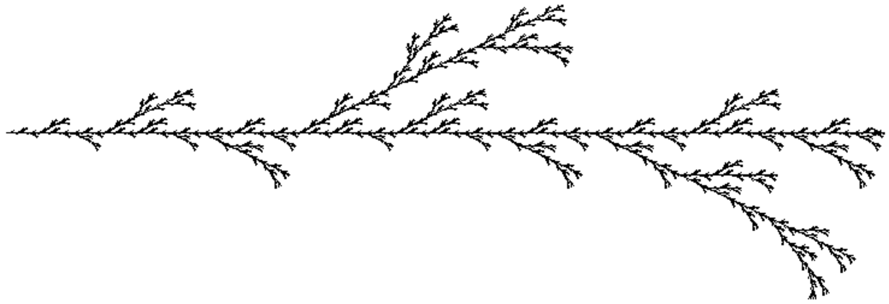


FIGURE 3 – *Modélisation d'une plante*

Pourtant, la figure 3 obtenue par l'interprétation graphique pour cinq génération et un angle de 25,7° est étonnante.

Évidemment, on s'imagine bien que le remplacement récursif d'un même motif à partir de points clés de celui-ci mène à la construction d'arbre de ce type. Mais la figure produite est tout de même remarquable dans la structure "botanique" qu'elle présente, même si une tige aussi rectiligne est peu probable dans la réalité.

Plusieurs autres règles peuvent être utilisées. Plusieurs angles aussi. Comme présenté dans *The Algorithmic Beauty of Plants* (Prusinkiewicz, 2004, p.25), on obtient alors des structures dont la ressemblance avec des plantes réelles est étonnante. La figure 4 présente les "plantes" obtenues avec les règles suivantes :

$$F \rightarrow F[+F]F[-F]F \tag{6}$$

$$F \rightarrow FF - [-F + F + F] + [+F - F - F] \tag{7}$$

On peut aussi illustrer l'utilisation de plusieurs règles simultanément à travers les exemples 8, 9 et 10 où l'axiome est X. Les deux règles sont appliquées successivement.

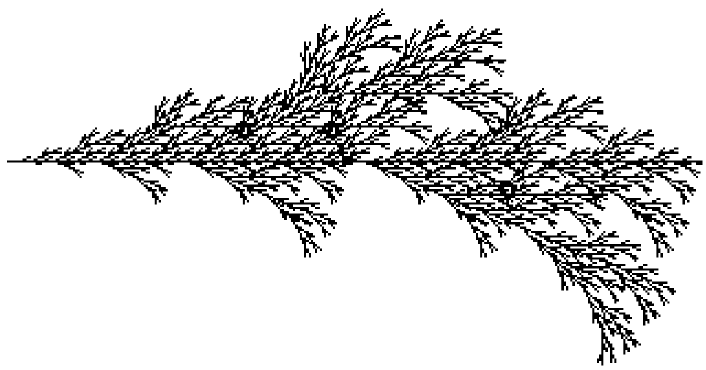
$$X \rightarrow F[+X]F[-X] + X \quad \text{et} \quad F \rightarrow FF \tag{8}$$

$$X \rightarrow F[+X][-X]FX \quad \text{et} \quad F \rightarrow FF \tag{9}$$

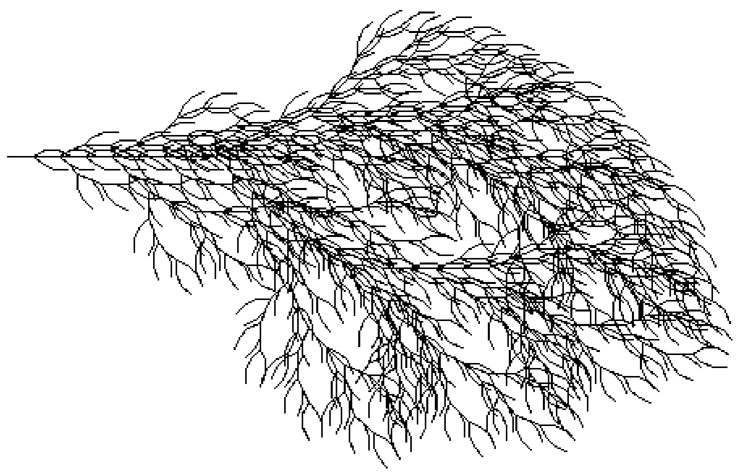
$$X \rightarrow F - [[X] + X] + F[+FX] - X \quad \text{et} \quad F \rightarrow FF \tag{10}$$

Le résultat de l'interprétation graphique est donné à la figure 5, page 18.

Bien évidemment, ces exemples ne sont pas construits au hasard. Une étude approfondie de la forme des chaînes nécessaires pour obtenir des structures réalistes de plantes doit être faite, ou tout au moins une réflection sur les motifs de base. On se reportera pour cela à l'adresse `http://algorithmicbotany.org/papers/` où sont présentés de nombreux exemples très crédibles. En ce qui nous concerne, l'objectif de ce petit cours n'étant pas d'étudier systématiquement ou de manière approfondie les plantes elles-mêmes, on laisse au lecteur le soin de tenter de trouver des chaînes crédibles et de les interpréter grâce à la programmation.



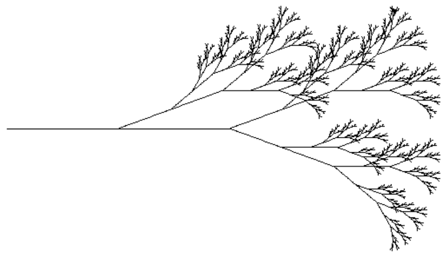
(a) nbiteration = 5, angle = 20° (axiome F ; règle 6)



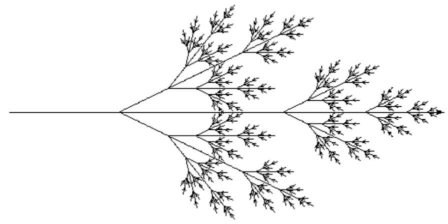
(b) nbiteration = 5, angle = 22,5° (axiome F ; règle 7)

FIGURE 4 – *Deux autres plantes*

Car, il est temps maintenant de considérer cet aspect moins formel, mais aussi important dans le cadre d'un cours d'informatique. Nous allons donc commencer par considérer le problème de la création de L-system de manière algorithmique. Cela nous permettra d'avoir une bonne vision des mécanismes principaux permettant la génération de ces systèmes.



(a) nbiteration = 7, angle = 20° (axiome X, règle 8)



(b) nbiteration = 7, angle = 25,7° (axiome X, règle 9)

(c) nbiteration = 5, angle = 22,5° (axiome X, règle 10)

Figure 5 – *Deux autres plantes*

4.5 Algorithmique

4.5.1 Création

Commençons par la construction de la chaîne de Lindenmayer. Nous avons vu que tout commence avec un axiome. Celui-ci peut être constitué de deux types de symboles : non-terminaux et terminaux. La production de la chaîne consiste quant à elle à remplacer un certain nombre de fois les symboles non-terminaux par une chaîne correspondante donné par une (ou plusieurs) règle. Pour simplifier la situation, on ne considérera ici qu'une seule règle. Il est donc nécessaire de disposer du nombre d'itération choisi, de l'axiome et de la règle qu'il faut initialiser. Ensuite viennent deux boucles. L'une pour répéter les remplacements des symboles non-terminaux par ceux de la règle et l'autre pour lire l'axiome. Finalement, au bout d'un nombre de génération donné, le résultat produit est un axiome "final" qui sera transmis pour interprétation graphique. L'algorithme 1, page 19, traduit ce raisonnement en pseudo-code.

Algorithme 1 – Création de la chaîne

Données : nbiteration, axiome, regle
Résultat : nouvel axiome
initialisation;
tant que *on est < nbiteration* **faire**
  tant que *on est < longueur de l'axiome* **faire**
    on lit un caractère de l'axiome;
    si *on trouve un caractère non terminal* **alors**
      on le remplace par la règle;
    sinon si *on trouve un caractère terminal* **alors**
      on le réécrit simplement;

4.5.2 Interprétation

Après la création de la chaîne vient son interprétation. Pour cela, il est nécessaire de disposer de l'axiome "final", de la longueur choisie des segments produit par la tortue et de l'angle correspondant à chaque rotation. L'interprétation graphique se fait ensuite par la lecture de chaque caractères de la chaîne et le lancement de l'opération correspondante par la tortue.

Si l'opération de mémorisation de la position de la tortue est simple, il faut analyser plus finement le retour à cette position. En effet, il est nécessaire de mémoriser plusieurs états différents avant d'y revenir successivement. Un tableau des états est donc nécessaire. De plus, une fois revenu sur un état mémorisé, il ne faut pas oublier de l'effacer. L'algorithme 2, page 20, résume tout cela en pseudo-code.

Dans le cas où plusieurs règles sont présentes, seule la procédure de création change avec deux boucles de lecture successives permettant d'effectuer le remplacement de deux caractères non-terminaux différents. Ces deux boucles ayant la même forme, on ne présentera pas ce cas. Mais le code donné en annexe 7.2 en donne une illustration compréhensible.

4.6 Programmation

4.6.1 Approche non récursive

Python est le langage utilisé. Mais,comme toujours dans le domaine de la programmation, il y a pour chaque problèmes beaucoup de solutions possibles. Même si l'objectif de ce cours n'est pas la programmation, deux solutions différentes vont être présentées, non récursive et récursive, répondant toutes les deux à la description algorithmique présentée au point 4.5. En effet, dans le cadre d'un cours sur les langages, il est une fois de plus intéressant de montrer la diversité des approches, sans pour autant se focaliser particulièrement sur les avantages des différents modes de programmation. Par contre, on abordera pas ces problèmes en programmation orientée objet ou à travers d'autres langages tels que java par exemple. Mais il est évidemment envisageable de le faire par la suite.

Dans la section 4.5, on a vu la décomposition du problème de génération des L-system en deux procédures d'une part de production des motifs à partir de la grammaire et d'autre part d'interprétation de ceux-ci par turtle. Le code complet est donné en annexe 7.1. On ne va détailler ici que les aspects non liés au langage de programmation utilisé. Cependant, il faut signaler qu'on a utilisé un découpage du problème en *fonction* pour des raisons de clarté uniquement et qu'on a aussi mélangé variables *globale* et *locales*, *procédures* et *fonctions* pour mettre en jeu ces notions dans un programme fonctionnel. Mais, on ne les définira pas ici.

La création de la chaîne consiste essentiellement en une boucle qui parcourt successivement tous les caractères.

```
while j < len(axiom):
            pointeur1 = axiom[j]
```

La lecture se fait en plaçant le caractère j de l'axiome dans une variable `pointeur1` qui permettra de décider du caractère terminal ou non d'un caractère. En effet, par remplacement de la règle dans les symboles non terminaux de l'axiome, la boucle va créer un nouvel axiome, nommé `axiome2`, qui sera utilisé comme axiome de départ à chaque itération. Ainsi, on a :

```
# Symboles non terminaux
            if pointeur1 == 'F':
                # remplacement de la regle dans l'axiome
```

Algorithme 2 – Interprétation graphique de la chaîne

Données : axiome, longueur, angle
variable : tableau des états mémorisés
Résultat : le tracé
initialisation;
tant que *on est < longueur de l'axiome* **faire**
- on lit un caractère de l'axiome;
- **si** *on trouve tel caractère* **alors**
  - on demande à la tortue de faire telle opération;
- **sinon si** *on trouve tel autre caractère* **alors**
  - on demande à la tortue de faire telle autre opération;
- **sinon si** *on trouve le caractère [* **alors**
  - on mémorise sa position et sa direction
- **sinon si** *on trouve le caractère]* **alors**
  - on récupère l'état, on y envoie la tortue et on efface cet état de la pile des état mémorisés

```
                    axiom2 = axiom2 + regle
```

On voit qu'on remplace le caractère `F` par la chaîne correspondante de la règle. Par contre, la lecture d'un symbole terminal ne donne lieu à aucun remplacement. On réécrit simplement ce symbole tel quel.

Ainsi, au fur et à mesure de la lecture des caractères,on construit par concaténation une nouvelle chaîne `axiom2` qui sera à la base de la prochaine itération.

L'interprétation de la chaîne se fait à nouveau par lecture de celle-ci. Après création, elle est passée à la procédure d'interprétation via le paramètre `axiom`, contenant la chaîne de caractère qui sera lue. Ensuite, pour les symboles de déplacement et d'orientation, l'interprétation est simple :

```
# Interpreation
        if pointeur2 == 'F':
            forward(longueur)
        elif pointeur2 == '+':
            left(angle)
        elif pointeur2 == '-':
            right(angle)
```

Par contre, pour les symboles de mémorisation, la procédure est quelque peu plus complexe :

```
elif pointeur2 == '[':
            # memorisation de la position courante
            laposition.append(position())
            langle.append(heading())
        elif pointeur2 == ']':
            # retour a la derniere position et suppression de celle-ci
            up()
            goto(laposition[len(laposition)-1][0],\
                 laposition[len(laposition)-1][1])
            del laposition[len(laposition)-1]
            setheading(langle[len(langle)-1])
            del langle[len(langle)-1]
            down()
```

Tout d'abord, on utilise ici un tableau, nommé `laposition` pour mémoriser chaque couple abscisse et ordonnée de la tortue et un autre tableau, nommé `langle` pour mémoriser sa direction.

Puis, pour le retour à la position mémorisée, on réalise un `goto` aux coordonnées mémorisées et un `setheading` pour revenir à la direction enregistrée. Enfin, il ne faut pas oublier de supprimer des deux tableaux la position à laquelle on est revenu. Cela se fait par deux `del`. Notez que pendant le retour il ne faut pas que la tortue soit en position d'écriture. On lève donc le crayon avant le retour et on le baisse après.

En fin de compte, grâce à l'utilisation de fonctions (et procédures), on peut écrire le programme principal avec une grande clarté :

```
initialisation()
gram = grammaire()
## gram[0] retourne l'angle
## gram[1] retourne l'axiome
## gram[2] retourne la regle

axio = production(gram[1], gram[2])
interpretation(gram[0], axio)
```

L'initialisation se fait par le placement de la tortue et la mémorisation des paramètre, axiome et règle. Puis, vient naturellement la production et l'interprétation.

4.6.2 Approche récursive

Une autre approche de la génération de L-system fait usage d'une technique de programmation particulièrement intéressante : la récursivité. Il s'agit de l'appel d'une fonction à elle-même. Pour les L-system, il s'agit donc de ne plus générer la chaîne de caractère, mais de lire l'axiome en renvoyant simultanément à la fonction appelée, à chaque occurrence d'un caractère non terminal. Il n'est alors pas évident de suivre ce que fait le programme. C'est pourquoi on va maintenant reprendre une forme légèrement différente de celle donnée précédemment du flocon de Koch pour l'étudier dans le détail. Le code complet est donne en annexe 7.3. Seule la partie purement récursive est présentée ci-dessous :

```
def production(nbiteration, longueur, angle, axiome, regle):
    if nbiteration == 0 :
        forward(longueur)
    else:
        i = 0
        while i < len(regle):
            regle1 = regle[i]
            if regle1 == 'F':
                production(nbiteration-1, longueur/3,\
                                        angle, axiome, regle)
            if regle1 == '+':
                left(angle)
            if regle1 == '-':
                right(angle)
            i = i + 1
```

Et la figure 6 en donne l'interprétation graphique.

Le code est remarquable de concision, mais complexe à apréhender.

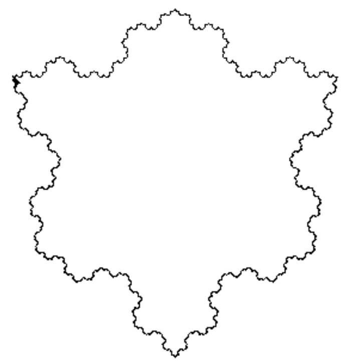


Figure 6 – *Flocon de Koch récursif.*

Le cœur de la récursivité se trouve dans le test de reconnaissance du caractère `F`. Il s'agit du caractère non terminal et il est nécessaire de le remplacer par l'ensemble des caractères de la règle. Mais, au lieu de réaliser simplement ce remplacement, on renvoie à la fonction de production. Cela n'a pas pour effet de remplacer le caractère non terminal par les symboles de la règle, mais de renvoyer sur une procédure, en l'occurrence celle dans laquelle on se trouve, qui va le faire. Or, cette fonction va encore s'appeler elle-même à la lecture du caractère `F` et ainsi de suite.

Le même processus va ainsi se répéter un nombre de fois déterminé par la variable `nbiteration`. Comme à chaque appel de la fonction de production on diminue la variable contenant le nombre d'itération, quand celle-ci devient nulle, on stoppe l'appel récursif et on trace un segment.

Ensuite, on retourne à la dernière procédure appelée pour réaliser l'instruction qui suit l'appel récursif selon la règle donnée, par exemple une instruction terminale de rotation. Et on continue pour réaliser une suite d'instructions terminales ou si non terminales, pour réaliser un appel récursif à la fonction principale.

Il est très difficile de suivre chaque étape de ce processus. Cependant, la figure 7 tente d'en donner une représentation graphique qui montre l'appel et le retour des processus dans l'ordre des flèches numérotées.

On voit que toute une série de procédures sont progressivement lancées les unes dans les autres. Puis, arrivées à leur terme, le tracé des segment se fait et elles se terminent par un retour à la procédure qui les a lancées. Finalement, tout aussi progressivement, on termine chacune des procédures imbriquées.

On a déjà remarqué que le code est plus concis que celui issu d'une simple lecture-interprétation. Par contre, il est moins clair, demanderait donc pour la modification une relecture attentive qui peut prendre du temps et il faudrait se demander s'il est plus performant.

Voilà. Comme le but de ce cours n'est pas la programmation récursive, nous finirons là cette analyse en précisant cependant les limites de celle-ci. En effet, avec la récursivité on utilise toujours la même règle. L'axiome n'est donc pas utilisé ou considéré égal à `F`.

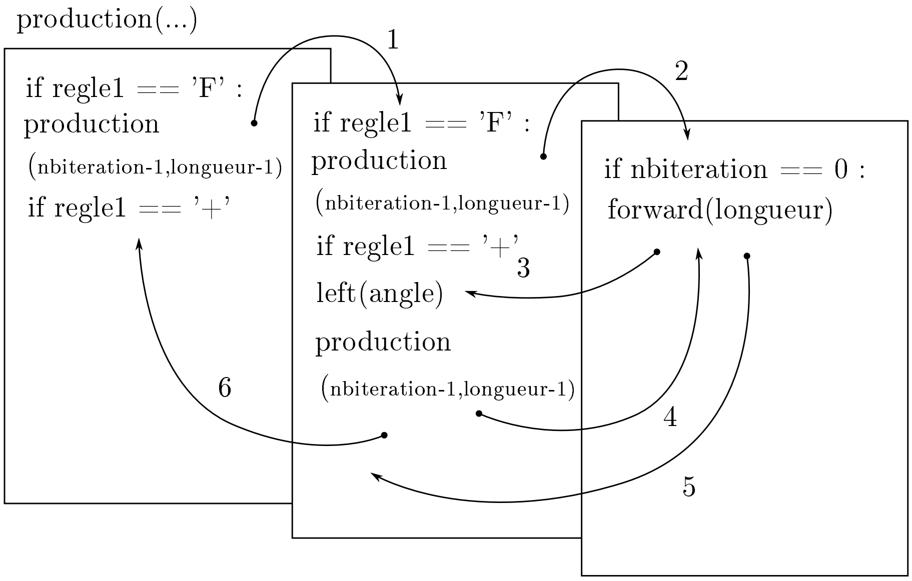


Figure 7 – *Flocon de Koch récursif.*

Certes, on peut envisager des moyens de contourner ce problème en réalisant l'axiome hors de la procédure récursive, en utilisant une récursivité indirecte ou en remplaçant à un moment bien précis la règle par l'axiome. Mais cela complique le code et l'avantage en concision de la récursivité s'amenuise.

5 Conclusion

Nous avons parcouru avec cette introduction aux L-system beaucoup de notions d'informatiques importantes. Les langages, leur reconnaissance, les grammaires, l'algorithmique et finalement différents modes de programmation nous ont permis de comprendre le rôle essentiel joué par les structures formelles que l'informatique théorique met à notre disposition. Les limites de ce cours ne nous ont cependant pas permis d'aborder d'autres notions tout aussi importantes comme les réseaux de Pétri ou les machines de Turing. Par contre, l'étude des L-system marque clairement la puissance de la modélisation informatique des structures biologiques. Plus, elle fait entrevoir une correspondance fascinante entre le développement des organisme vivants et des constructions répétitives très structurées, finalement très bien décrites par … le langage mathématique.

6 Solutions des exercices

1 L'alphabet des chiffres binaires est : $\Sigma = \{0, 1\}$ et le langage des nombres binaires à trois chiffres est :

$$L = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

2 En considérant les quatre nucléotides composant l'ADN, la thymine (T), la cytosine (C), l'adénine (A) et la guanine (G), on peut définir l'alphabet nécessaire ainsi :

$$\Sigma = \{A, C, G, T\}$$

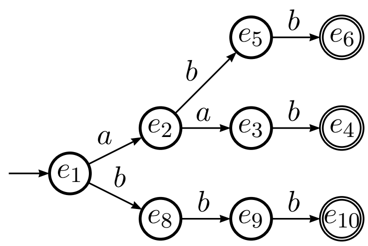
Les codons sont quant à eux des ensemble de trois nucléotides, pris parmi les quatre disponibles dans l'alphabet. Le langage ainsi formé est :

$$L = \{AAC, ACT, ACT, CTG, GTC, \ldots\}$$

et la combinatoire donne pour un arrangement avec répétition de trois éléments pris parmi quatre :

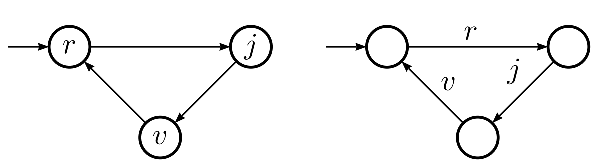
$$\overline{A_3^4} = 4^3 = 64 \text{ possibilités}$$

3 Il s'agit d'un automate simple sans grande difficultés. Il est déterministe puisque de chaque état ne part qu'une seule transition de même type.



4 Le problème peut sembler complexe car l'énoncé du mot allumé fait immédiatement penser à éteint. Naturellement, on pense à un état allumé et à un autre éteint. Comme il y a trois couleurs, on s'imagine alors six états et leur représentation avec des transitions du type "s'allume" ou "s'éteint" ne va pas de soit.

Pour s'en sortir simplement, il faut au contraire penser "quel est la couleur du feu". Cela suppose qu'on ne considère que l'état allumé du feu. On peut alors dire rouge, puis jaune, puis vert, puis rouge, ... On pourrait aussi dire, c'est le rouge qui s'allume, puis c'est le jaune qui s'allume, puis ... Cela nous permet alors de représenter la succession des états allumé du feu par :



En particulier, l'automate ci-dessus permet la reconnaissance de la chaîne de caractère rjvrjvrjv... qui est une représentation de la succession des couleurs (allumées) du feu de circulation.

5 On a vu qu'il y a 64 possibilités pour former un acide aminé constitué d'un triplet choisi parmi quatre nucléotides. Il peut sembler alors que l'automate correspondant doive être très complexe. Mais au contraire, en raison du fait qu'on peut mettre n'importe quel nucléotide à chaque place parmi les trois disponibles nous permet de constituer un automate particulièrement simple :

où la notation C|G|T|U signifie soit C, soit G, soit T, soit U.

On constate ainsi la concision de la description de l'automate par rapport à l'exposé de tout les cas possibles.

6 On doit tout d'abord faire un b, puis un nombre quelconque (voir nul) de b. Comme l'état de droite n'est pas final, il faut ensuite revenir à celui de gauche par un a. Enfin, on peut faire tout cela un nombre quelconque de fois, y compris aucune. L'expression rationnelle correspondante est :

$$(bb^*a)^*$$

7 De manière très naturelle, on a :

$$[A..Z]\{2\}[0..9]\{5\}$$

8 Il faut raisonner à partir de la dernière transition avant l'état final. Il s'agit toujours d'une transition b. Donc, l'expression régulière finira par un b. Puis, on a soit deux b, soit un a suivi d'un a ou b. Cela se traduit par :

$$(bb|a(a|b))b$$

7 Annexes

7.1 Code python plante

```
 # -*- coding: utf-8 -*-
from turtle import *

# Generation de plante par grammaire L-system

def initialisation():
    # Definition des variables independantes de la grammaire
    global nbiteration, longueur
    nbiteration = 5
    longueur = 3

def grammaire():
    # Definition de la grammaire
    angle = 25.7
    axiom = 'F'

    regle = 'FF-[-F+F+F]+[+F-F-F]'
    #regle = 'F[+F]F[-F][F]'
    #regle = 'F[+F]F[-F]F'
    #regle = 'F+FF-FF-F-F+F+FF-F-F+F+FF+FF-F'

    return angle, axiom, regle

def production(axiom, regle):
    # Generation de la chaine de caracteres
    i = 0
    while i < nbiteration:
        j = 0
        axiom2 = ''
        while j < len(axiom):
            pointeur1 = axiom[j]

            # Symboles non terminaux
            if pointeur1 == 'F':
                # remplacement de la regle dans l'axiome
                axiom2 = axiom2 + regle

            # Symboles terminaux
            if pointeur1 == '+':
```

```
                axiom2 = axiom2 + '+'
            if pointeur1 == '-':
                axiom2 = axiom2 + '-'
            if pointeur1 == '[':
                axiom2 = axiom2 + '['
            if pointeur1 == ']':
                axiom2 = axiom2 + ']'

            j = j + 1
        axiom = axiom2
        i = i + 1
    print axiom
    return axiom

def interpretation(angle, axiom):
    # Interpretation de la chaine par turtle
    k = 0
    laposition = []
    langle = []
    while k < len(axiom):
        pointeur2 = axiom[k]

        # Interpreation
        if pointeur2 == 'F':
            forward(longueur)
        elif pointeur2 == '+':
            left(angle)
        elif pointeur2 == '-':
            right(angle)
        elif pointeur2 == '[':
            # memorisation de la position courante
            laposition.append(position())
            langle.append(heading())
        elif pointeur2 == ']':
            # retour a la derniere position et suppression de celle-ci
            up()
            goto(laposition[len(laposition)-1][0], laposition[len(laposition)-1][1])
            del laposition[len(laposition)-1]
            setheading(langle[len(langle)-1])
            del langle[len(langle)-1]
            down()

        k = k + 1
```

```
# Placement initial de la tortue
up()
backward(495)
down()

# Programme principal
## gram[0] retourne l'angle
## gram[1] retourne l'axiome
## gram[2] retourne la regle

initialisation()
gram = grammaire()

axio = production(gram[1], gram[2])
interpretation(gram[0], axio)

#quit()
```

7.2 Code python plante à deux règles

```
# -*- coding: utf-8 -*-
from turtle import *

# Generation de plante par grammaire L-system

def initialisation():
    # Definition des variables independantes de la grammaire
    global nbiteration, longueur
    # Pour le flocon de Koch
    #nbiteration = 3
    #longueur = 2

    # Pour le flocon de Koch Island
    #nbiteration = 3
    #longueur = 2

    # Pour la plante
    #nbiteration = 5
    #longueur = 5

    # Pour la plante 2
```

```
    #nbiteration = 5
    #longueur = 5

    # Pour la plante 3
    #nbiteration = 4
    #longueur = 8

    # Pour la plante 4
    #nbiteration = 7
    #longueur = 1

    # Pour la plante 5
    #nbiteration = 7
    #longueur = 1

    # Pour la plante 6
    nbiteration = 5
    longueur = 4

def grammaire():
    # Definition de la grammaire
    # Pour le flocon de  Koch
    #angle = 90
    #axiom = 'F-F-F-F'
    #regle = 'F+F-F-F+F'

    # Pour le flocon de Koch Island
    #angle = 90
    #axiom = 'F-F-F-F'
    #regle = 'F-F+F+FF-F-F+F'

    # Pour la plante
    #angle = 25.7
    #axiom = 'F'
    #regle = 'F[+F]F[-F]F'

    # Pour la plante 2
    #angle = 20
    #axiom = 'F'
    #regle = 'F[+F]F[-F][F]'

    # Pour la plante 3
    #angle = 22.5
```

```
    #axiom = 'F'
    #regle = 'FF-[-F+F+F]+[+F-F-F]'

    # Pour la plante 4
    #angle = 20
    #axiom = 'X'
    #regle1 = 'F[+X]F[-X]+X'
    #regle2 = 'FF'

    # Pour la plante 5
    angle = 25.7
    axiom = 'X'
    regle1 = 'F[+X][-X]FX'
    regle2 = 'FF'

    # Pour la plante 6
    angle = 22.5
    axiom = 'X'
    regle1 = 'F-[[X]+X]+F[+FX]-X'
    regle2 = 'FF'

    return angle, axiom, regle1, regle2

def production(axiom, regle1, regle2):
    # Generation de la chaine de caracteres
    i = 0
    while i < nbiteration:
        j = 0
        axiom2 = ''
        while j < len(axiom):
            pointeur1 = axiom[j]

            # Symboles non terminaux
            if pointeur1 == 'X':
                # remplacement de la regle dans l'axiome
                axiom2 = axiom2 + regle1

            # Symboles terminaux
            if pointeur1 == '+':
                axiom2 = axiom2 + '+'
            if pointeur1 == '-':
                axiom2 = axiom2 + '-'
            if pointeur1 == '[':
```

```
                axiom2 = axiom2 + '['
            if pointeur1 == ']':
                axiom2 = axiom2 + ']'
            if pointeur1 == 'F':
                axiom2 = axiom2 + 'F'

            j = j + 1
        axiom = axiom2
        print axiom

        m = 0
        axiom3 = ''
        while m < len(axiom):
            pointeur2 = axiom[m]

            # Symboles non terminaux
            if pointeur2 == 'F':
                # remplacement de la regle dans l'axiome
                axiom3 = axiom3 + regle2

            # Symboles terminaux
            if pointeur2 == '+':
                axiom3 = axiom3 + '+'
            if pointeur2 == '-':
                axiom3 = axiom3 + '-'
            if pointeur2 == '[':
                axiom3 = axiom3 + '['
            if pointeur2 == ']':
                axiom3 = axiom3 + ']'
            if pointeur2 == 'X':
                axiom3 = axiom3 + 'X'

            m = m + 1
        axiom = axiom3
        i = i + 1

    print axiom
    return axiom

def interpretation(angle, axiom):
    # Interpretation de la chaine par turtle
    k = 0
    laposition = []
```

```
    langle = []
    while k < len(axiom):
        pointeur2 = axiom[k]

        # Interpreation
        if pointeur2 == 'F':
            forward(longueur)
        elif pointeur2 == '+':
            left(angle)
        elif pointeur2 == '-':
            right(angle)
        elif pointeur2 == '[':
            # memorisation de la position courante
            laposition.append(position())
            langle.append(heading())
        elif pointeur2 == ']':
            # retour a la derniere position et suppression de celle-ci
            up()
            goto(laposition[len(laposition)-1][0], laposition[len(laposition)-1][1])
            del laposition[len(laposition)-1]
            setheading(langle[len(langle)-1])
            del langle[len(langle)-1]
            down()

        k = k + 1

# Placement initial de la tortue
up()
backward(400)
down()

# Programme principal
## gram[0] retourne l'angle
## gram[1] retourne l'axiome
## gram[2] retourne la regle

initialisation()
gram = grammaire()
axio = production(gram[1], gram[2], gram[3])
interpretation(gram[0], axio)

#quit()
```

7.3 Code python flocon de Koch récursif

```
# -*- coding: utf-8 -*-
from turtle import *

# Generation de plante par grammaire L-system

def initialisation():
    # Definition des variables independantes de la grammaire
    # Pour le flocon de Koch
    nbiteration = 3
    longueur = 200
    angle = 45
    axiome = 'F'
    regle = 'F+F--F+F'
    return nbiteration , longueur , angle , axiome , regle

def production(nbiteration , longueur , angle , axiome , regle):
    if nbiteration == 0 :
        forward(longueur)
    else:
        i = 0
        while i < len(regle):
            regle1 = regle[i]
            if regle1 == 'F':
                production(nbiteration -1, longueur/3,\
                                angle , axiome , regle)
            if regle1 == '+':
                left(angle)
            if regle1 == '-':
                right(angle)
            i = i + 1

# Placement initial de la tortue
up()
backward(200)
down()

# Programme principal
## init[1] : nbiteration
## init[2] : longueur
## init[3] : angle
```

```
## init[4] : axiome
## init[5] : regle

init = initialisation()
production(init[0], init[1], init[2], init[3], init[4])
right(120)
production(init[0], init[1], init[2], init[3], init[4])
right(120)
production(init[0], init[1], init[2], init[3], init[4])
```

Références

Buchs, D. (2011). Langages formels.

Desgraupes, B. (2008). *Expressions régulières.* Pearson. Simple, clair et efficace. Mais peu théorique.

Gastin, P. (2009). Langages formels.

Hammel, P. P. M. (1994). *Language-Restricted Iterated Function Systems, Koch Constructions, and L-systems.* University of Calgary SIGGRAPH '94 Course Notes. ACM Press.

Prusinkiewicz, A. L. P. (2004). *The Algorithmic Beauty of Plants.* Springer-Verlag. Ouvrage référence publié sur `http://algorithmicbotany.org/papers/`.

Renault, J. C. F. (2006). *Interprétation graphique d'une grammaire L-System.* Ecole Polytechnique de l'Universite de Tours.

Togni, O. (2007). Langages et automates.