Chapitre 3

option informatique

Mots et langages

1. Introduction

L'objectif de ce chapitre est de fournir une introduction aux modèles utilisés en informatique pour décrire, représenter et effectuer des calculs sur des séquences finies de symboles. Avant d'introduire de manière formelle les concepts de base auxquels ces modèles font appel, nous allons présenter quelques-uns des grands domaines d'application de ces modèles, de façon à mieux saisir l'utilité d'une telle théorie.

1.1 Compilation

La compilation désigne la tâche consistant à transformer un ensemble de commandes écrites dans un langage de programmation de haut niveau en une série d'instructions exécutables par l'ordinateur (c'est-à-dire exprimées en *langage machine*). Parmi les tâches préliminaires qui incombent au compilateur se trouve l'identification des séquences de caractères qui forment des mots-clés du langage ou des noms de variables licites ou encore des nombres réels : cette étape s'appelle l'*analyse lexicale*.

Seconde tâche importante du compilateur : détecter les erreurs de syntaxe et pour cela identifier dans l'ensemble des séquences formant un programme celles qui sont correctement formées. Cela consiste par exemple à vérifier que les expressions arithmétiques sont bien formées, que les constructions du langage sont respectées, etc. Cette seconde étape s'appelle l'*analyse syntaxique*.
En fait, la tâche de l'analyseur syntaxique va même au delà de ces contrôles, puisqu'elle vise aussi à mettre en évidence la structure interne des séquences de symboles qu'on lui soumet, de manière à déterminer la séquence d'instructions à exécuter.

En d'autres termes, si on tente une analogie avec l'analyse d'une phrase exprimée dans un langage humain, l'analyse lexicale consiste à découper une phrase en mots appartenant à certaines catégories du langage (nom, verbe, adjectif, adverbe ...), l'analyse syntaxique à regrouper ces mots en phrases respectant les règles de la grammaire [1].

1.2 Bio-informatique

La génétique fournit un exemple naturel d'objets modélisables comme des séquences linéaires de symboles sur un alphabet fini. Chaque chromosome porteur du capital génétique est essentiellement formé de deux brins d'ADN ; chacun de ces brins peut être modélisé par une succession de nucléotides, chacun composé d'un phosphate, d'un sucre et d'une base azotée. Il existe quatre base différentes : la guanine G, l'adénine A, la cytosine C et la thymine T. L'information encodée dans ces bases déterminant une partie importante de l'information génétique, une modélisation utile d'un brin de chromosome (un gène) consiste en une simple séquence linéaire des bases qui le composent, soit en fait une (très longue) séquence définie sur un alphabet à quatre lettres {G,T,A,C}.

De ce modèle découlent plusieurs questions : comment déterminer la présence ou non d'un gène dans le patrimoine génétique d'un individu (autrement dit rechercher une séquence particulière de nucléotides dans un chromosome) ou encore détecter des ressemblances entre plusieurs fragments d'ADN. Ces ressemblances génétiques vont servir par exemple à localiser des gènes remplissant des mêmes fonctions ou encore à réaliser des tests de familiarité entre individus ou entre espèces.
Bref, rechercher des séquences ou mesurer des ressemblances constituent deux problèmes de base de la bio-informatique.

1. À ces deux étapes s'ajoute dans les langages humains une étape d'*analyse sémantique* visant à donner un sens à la phrase étudiée. Dans les langages informatiques cette étape se confond avec l'analyse syntaxique puisqu'une instruction n'est jamais ambiguë.

1.3 Recherche de motifs dans un texte

Tous les éditeurs de texte proposent une fonctionnalité permettant de déterminer les occurrences d'une chaîne de caractères dans un fichier texte. Certains proposent en outre une recherche élargie permettant de rechercher tout un ensemble de mots vérifiant un certain nombre de propriétés ; dans ce cas, on utilise la notion d'*expression régulière* pour exprimer précisément l'objet de la recherche. Par exemple, l'expression régulière `[Cc]h(at|ien)` décrit les quatre motifs `Chat`, `chat`, `Chien`, `chien` ; l'expression `(19|20)[0-9]{2}` décrit toute date comprise entre 1900 et 2099 ; l'expression `[a-z]+(\.[a-z]+)?@[a-z]+\.fr` reconnait les adresses email `henri@llg.fr` ou `henri.poincare@llg.fr` mais pas `henri@info.llg.fr` ni `henri.Pointcare@llg.fr`.
Vous trouverez en annexe les principales règles de construction des expressions régulières. Leur connaissance n'est en aucun cas un objectif de ce cours, mais une bonne partie de ce chapitre ainsi que le chapitre suivant étudient la théorie qui est à l'origine de leur invention.

Enfin, le motif de notre recherche peut ne pas être alphanumérique ; on peut souhaiter rechercher puis lire un code-barre dans une image. Il peut aussi être multidimensionnel : tout smartphone possède une application chargée de reconnaitre un QR code, de l'interpréter et d'ouvrir la page internet décrite par ce motif.

Figure 1 – Deux exemples de QR codes.

Cependant, la suite de ce cours se contentera d'aborder le problème de la recherche de motifs en dimension 1.

2. Mots et alphabets

Un *alphabet* est un ensemble fini Σ dont les éléments sont appelés les *lettres*. Une *mot* de l'alphabet Σ de longueur $n \in \mathbb{N}^*$ est une suite finie formée de n lettres. On note Σ^+ l'ensemble des mots, les lettres étant identifiées aux mots de longueur 1. On note $|s|$ la longueur du mot s. Enfin, si s est un mot et a une lettre, $|s|_a$ désigne le nombre d'occurrences de la lettre a dans le mot s. On a bien entendu :

$$|s| = \sum_{a \in \Sigma} |s|_a.$$

On adjoint à Σ^+ un mot, noté ε et appelé *mot vide*, de longueur nulle. On pose alors $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Concaténation
Étant donné deux mots $r = a_1 \cdots a_p$ et $s = b_1 \cdots b_q$, la *concaténation* de r et de s est le mot $rs = a_1 \cdots a_p b_1 \cdots b_q$. Il s'agit d'une loi associative possédant un élément neutre ε (autrement dit, cette loi muni Σ^* d'une structure de monoïde [2]). On peut donc définir par récurrence le mot r^n lorsque $r \in \Sigma^*$ et $n \in \mathbb{N}$ en posant :

$$r^0 = \varepsilon \qquad \text{et} \qquad \forall n \in \mathbb{N}^*, \quad r^n = r.r^{n-1} = r^{n-1}.r$$

2.1 Facteurs et sous-mots

Si u et v sont deux mots, on dit que u est un :

- *préfixe* (ou *facteur gauche*) de v s'il existe un mot $w \in \Sigma^*$ tel que $v = uw$;
- *suffixe* (ou *facteur droit*) de v s'il existe un mot $w \in \Sigma^*$ tel que $v = wu$;
- *facteur* de v lorsqu'il existe deux mots $x \in \Sigma^*$ et $y \in \Sigma^*$ tels que $v = xuy$.

Les préfixes et suffixes sont dits *propres* lorsque $w \neq \varepsilon$; les facteurs sont dits *propres* lorsque $x \neq \varepsilon$ ou $y \neq \varepsilon$.

Enfin, un *sous-mot* [3] d'un mot $u = a_1 \cdots a_n$ de longueur n (les a_i désignant ses lettres) est un mot $v = a_{\varphi(1)} \cdots a_{\varphi(p)}$ de longueur p, où $\varphi : [\![1, p]\!] \to [\![1, n]\!]$ est une application strictement croissante.

2. Un monoïde est un ensemble muni d'une opération interne associative et d'un élément neutre.
3. Attention, il y a ici désaccord avec la terminologie anglo-saxonne : *subword* signifie en fait facteur et c'est *scattered subword* qui est l'équivalent anglais de sous-mot.

Par exemple, “hippopoto” est un préfixe, “phobie” un suffixe, “monstro” un facteur, et “strophe” un sous-mot de “hippopotomonstrosesquippedaliophobie” [4].

• Quelques résultats combinatoires élémentaires

Lemme (Levi). — *Soient x, y, u et $v \in \Sigma^*$ tels que $uv = xy$. Alors il existe un unique mot $t \in \Sigma^*$ tel que l'une des deux conditions suivantes soit réalisée :*

- *$u = xt$ et $y = tv$;*
- *$x = ut$ et $v = ty$.*

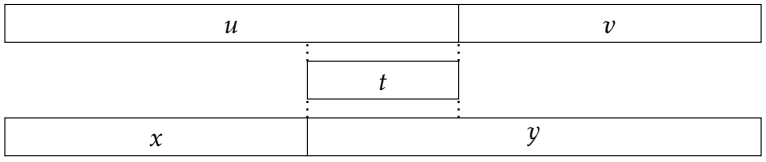


Figure 2 – Illustration du lemme de Levi lorsque $|u| \geqslant |x|$.

Preuve. Supposons par exemple $|u| \geqslant |x|$. x est un préfixe de uv donc de u, ce qui justifie l'existence d'un mot t tel que $u = xt$. Dans ce cas, l'égalité $uv = xy$ peut encore s'écrire $xtv = xy$ puis en simplifiant $tv = y$.
Le cas $|x| \geqslant |u|$ se traite de la même façon. □

Ce résultat est utilisé pour démontrer deux autres résultats élémentaires qui découlent de la non-commutativité de la concaténation.

Théorème. — *Soient x, y et $z \in \Sigma^*$ tels que $xy = yz$ et $x \neq \varepsilon$. Alors il existe deux mots u et v et un entier $k \in \mathbb{N}$ tels que :*

$$x = uv, \quad y = (uv)^k u = u(vu)^k, \quad z = vu.$$

Preuve. Si $|x| \geqslant |y|$, on peut appliquer le lemme de Levi : il existe un mot t tel que $x = yt$ et $z = ty$. Dans ce cas le résultat est bien acquis avec $u = y$, $v = t$ et $k = 0$.
Si $|x| \leqslant |y|$, raisonnons par induction sur la longueur de y.

- Si $|y| = 1$ on a aussi $|x| = 1$ car $x \neq \varepsilon$ et dans ce cas $x = y = z$. Le résultat est acquis avec $u = y$, $v = \varepsilon$ et $k = 0$.
- Si $|y| > 1$, supposons le résultat acquis pour tout mot de longueur inférieure, et appliquons le lemme de Levi : il existe un mot t tel que $y = xt$ et $y = tz$. On a donc $xt = tz$ et puisque $x \neq \varepsilon$, $|t| < |y|$. On peut donc appliquer l'hypothèse de récurrence et affirmer l'existence de deux mots u et v tels que $x = uv$, $t = (uv)^k u = u(vu)^k$ et $z = vu$. Il reste à calculer $y = xt = tz$ et obtenir $y = (uv)^{k+1} u = u(vu)^{k+1}$.

□

Théorème. — *Soient x et $y \in \Sigma^*$ tels que $xy = yx$, avec $x \neq \varepsilon$ et $y \neq \varepsilon$. Alors il existe un mot $u \in \Sigma^*$ et deux entiers i et j tels que $x = u^i$ et $y = u^j$.*

Preuve. Raisonnons par induction sur $|xy|$.

- Si $|xy| = 2$ alors $|x| = |y| = 1$ et $x = y$. Le résultat est acquis en posant $u = x = y$ et $i = j = 1$.
- Si $|xy| > 1$, supposons le résultat acquis pour tout mot de longueur inférieure et appliquons le théorème précédent : il existe deux mots u et v et un entier k tels que $x = uv = vu$ et $y = (uv)^k u = u(vu)^k$.
  Si $u = \varepsilon$ ou $v = \varepsilon$ alors $y = x^k$ ou $y = x^{k+1}$ et le résultat est acquis avec $u = x$, $i = 1$ et $j = k$ ou $j = k + 1$.
  Si $u \neq \varepsilon$ et $v \neq \varepsilon$, puisque $y \neq \varepsilon$ on a $|uv| = |x| < |xy|$ donc on peut appliquer l'hypothèse de récurrence : il existe un mot w et deux entiers i et j tels que $u = w^i$ et $v = w^j$. Dans ce cas, $x = w^{i+j}$ et $y = w^{(k+1)i+kj}$.

□

4. Ce qui signifie « la peur des grands mots ».

2.2 Distances entre mots

Distance préfixe
Il existe plus d'une façon de munir l'ensemble Σ^* d'une distance ; l'une d'entre-elles consiste à considérer le *plus long préfixe commun* à deux mots u et v, préfixe que nous allons désormais noter $\text{plpc}(u,v)$, et à définir :

$$d(u,v) = |uv| - 2|\text{plpc}(u,v)|.$$

On obtient ainsi une distance, appelée *distance préfixe*. On vérifie en effet que :

- $d(u,v) \geqslant 0$;
- $d(u,v) = 0 \iff u = v$;
- $d(u,w) \leqslant d(u,v) + d(v,w)$.

Preuve. De ces trois propriétés seule la troisième demande peut-être une justification.
Après simplification, celle-ci revient à prouver que :

$$|\text{plpc}(u,v)| + |\text{plpc}(v,w)| \leqslant |v| + |\text{plpc}(u,w)|$$

Le plus court des deux préfixes $\text{plpc}(u,v)$ et $\text{plpc}(v,w)$ est commun à u, v et w donc

$$\min(|\text{plpc}(u,v)|, |\text{plpc}(v,w)|) \leqslant |\text{plpc}(u,w)|.$$

Les deux préfixes $\text{plpc}(u,v)$ et $\text{plpc}(v,w)$ sont préfixes de $|v|$ donc

$$\max(|\text{plpc}(u,v)|, |\text{plpc}(v,w)|) \leqslant |v|.$$

D'où le résultat, en additionnant ces deux inégalités. □

Remarque. On obtient aussi des distances lorsque, au lieu de considérer la longueur des plus longs préfixes communs on considère celle des plus longs suffixes, des plus longs facteurs ou des plus longs sous-mots communs. Dans tous les cas, la seule propriété demandant une justification soigneuse est l'inégalité triangulaire. Dans le cas des suffixes, la preuve est identique à celle ci-dessus ; dans le cas des facteurs ou des sous-mots, c'est un peu plus délicat. Traitons par exemple le cas des sous-mots en désignant par plsmc le plus long sous-mot commun.
Notons I et J les indices des lettres de v qui correspondent à $\text{plsmc}(u,v)$ et à $\text{plsmc}(v,w)$. Alors

$$|\text{plsmc}(u,v)| + |\text{plsmc}(v,w)| = |\text{I}| + |\text{J}| = |\text{I} \cup \text{J}| + |\text{I} \cap \text{J}|.$$

Considérons le sous-mot de v constitué des lettres dont les indices appartiennent à $\text{I} \cap \text{J}$. Il s'agit d'un sous-mot commun à u, v et w donc $|\text{I} \cap \text{J}| \leqslant |\text{plsmc}(u,w)|$.
Puisque par ailleurs $|\text{I} \cup \text{J}| \leqslant |v|$ on peut conclure que $|\text{plsmc}(u,v)| + |\text{plsmc}(v,w)| \leqslant |v| + |\text{plsmc}(u,w)|$.

Distance d'édition
Une autre distance communément utilisée sur Σ^* est la *distance d'édition*, ou distance de Levenshtein : elle est définie comme étant le nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour transformer un mot u en un mot v. Par exemple, on passe du mot **`polynomial`** au mot **`polygonal`** en suivant les étapes :

- suppression de la lettre **`'i'`** : `polynomial` → `polynomal` ;
- remplacement du **`'n'`** par un **`'g'`** : `polynomal` → `polygomal` ;
- remplacement du **`'m'`** par un **`'n'`** : `polygomal` → `polygonal` ;

donc la distance d'édition entre ces deux mots est égal à 3.
Vous avez peut-être étudié l'année dernière un algorithme de calcul de la distance d'édition ; c'est un problème classique de la programmation dynamique.

De multiples variantes de cette notion de distance ont été proposées, utilisant des opérations différentes ou en considérant des poids variables pour les différentes opérations. Pour prendre un exemple réel, cet algorithme est utilisé en bio-informatique pour mesurer le degré de similarité de deux gènes appartenant à deux espèces cousines : en effet, ces deux espèces ont hérité des gènes d'un ancêtre commun mais au cours de l'évolution leurs gènes ont subi des mutations, se traduisant par l'apparition ou la suppression d'un ou plusieurs nucléotides, ou par la transformation de certains d'entre eux.
Cependant, pour des raisons liées à la structure moléculaire de ces nucléotides, un `A` va plus facilement muter en `G` et un `C` en `T` (et réciproquement). On attribuera alors un coût différent en fonction des substitutions observées.

A C C T C T [-] A A T C [T] A T T C G T [A] C T G C T A T T
A C C T C T [G] A A T C [C] A T T C G T [-] C T G C T A T T

FIGURE 3 – Deux gènes qui diffèrent de deux insertion/suppression et d'une substitution.

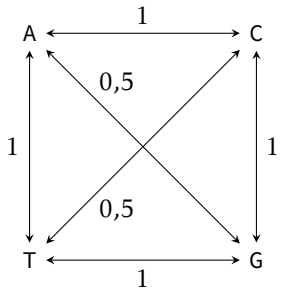


FIGURE 4 – Un exemple des coûts d'une substitution entre deux nucléotides.

2.3 Mots de Dyck

Nous allons maintenant nous intéresser aux mots sur l'alphabet constitué des deux lettres (et), c'est à dire les parenthèses ouvrante et fermante, et plus spécifiquement des mots correspondant aux suites de parenthèses intervenant dans une expression mathématique syntaxiquement correcte. Par exemple, l'expression : $2+(3\times(x-5)+(5-x)\times y)\times(y-x)$ fournit le mot bien parenthésé : (()())(), alors que les mots)(ou ()(sont mal parenthésés.
Par la suite et pour faciliter la lecture, on posera $\Sigma=\{a,b\}$, a désignant la parenthèse ouvrante et b la parenthèse fermante. Ainsi, le mot bien parenthésé que nous venons de donner en exemple s'écrira : *aababbab*.

On conviendra aisément que l'ensemble $\mathscr{D}$ des expressions bien parenthésées, appelées aussi *mots de* Dyck, peut être défini à l'aide des règles de construction suivantes :

$$\begin{cases} \varepsilon \in \mathscr{D} \\ (r,s) \in \mathscr{D}^2 \Longrightarrow arbs \in \mathscr{D} \end{cases}$$

Définissons maintenant un morphisme additif $\sigma : \Sigma^* \to \mathbb{Z}$ (la *valuation* d'un mot de Σ^*) en posant : $\sigma(a)=1$ et $\sigma(b)=-1$; nous disposons alors du résultat suivant :

THÉORÈME. — *Un mot m de Σ^* appartient à $\mathscr{D}$ si et seulement si :*

(i) $\sigma(m)=0$;

(ii) pour tout préfixe m' de m, on a $\sigma(m')\geqslant 0$.

Preuve. Notons $\mathscr{D}'$ l'ensemble des mots vérifiant les propriétés (i) et (ii) énoncées ci-dessus, et montrons que $\mathscr{D}=\mathscr{D}'$ en procédant par double inclusion.

- Soit $m\in\mathscr{D}$. Montrons par induction sur $|m|$ que m appartient à $\mathscr{D}'$.
  Si $|m|=0$, c'est clair puisque $m=\varepsilon$.
  Si $|m|>0$, supposons que tout mot de $\mathscr{D}$ de longueur strictement inférieure à $|m|$ appartienne à $\mathscr{D}'$. Le mot m s'écrit : $m=arbs$ avec $(r,s)\in\mathscr{D}^2$, donc par hypothèse d'induction, r et s appartiennent à $\mathscr{D}'$. Ainsi, $\sigma(m)=1+\sigma(r)-1+\sigma(s)=0$, et les préfixes de m sont :
  - a avec $\sigma(a)=1$;
  - ar' où r' est préfixe de r, et $\sigma(ar')=1+\sigma(r')\geqslant 1$;
  - arb avec $\sigma(arb)=1+0-1=0$;
  - $arbs'$ où s' est préfixe de s, et $\sigma(arbs')=1+0-1+\sigma(s')\geqslant 0$.

  On en déduit que m est élément de $\mathscr{D}'$, ce qui prouve par induction que $\mathscr{D}\subset\mathscr{D}'$.
- Soit maintenant $m\in\mathscr{D}'$, et montrons par induction sur $|m|$ que m appartient à $\mathscr{D}$.
  Si $|m|=0$ c'est clair puisqu'alors $m=\varepsilon$.
  Si $|m|>0$, supposons que tout mot de $\mathscr{D}'$ de longueur strictement inférieure à $|m|$ appartienne à $\mathscr{D}$, et notons $m=m_1\cdots m_p$, avec $m_i\in\{a,b\}$.

m_1 est un préfixe de m donc $\sigma(m_1) \geqslant 0$ et $m_1 = a$. Considérons alors l'ensemble $\mathrm{I} = \left\{k \in [\![2,p]\!] \;\middle|\; \sigma(m_1 \cdots m_k) = 0\right\}$; cet ensemble est non vide puisqu'il contient p, il possède donc un plus petit élément k. Puisque $k-1$ n'appartient pas à I, on a nécessairement $\sigma(m_1 \cdots m_{k-1}) \geqslant 1$, et donc $m_k = b$. Posons alors $r = m_2 \cdots m_{k-1}$ et $s = m_{k+1} \cdots m_p$; ainsi, $m = arbs$.

Puisque $\sigma(arb) = 0$, tout préfixe s' de s vérifie : $\sigma(s') \geqslant 0$, et $\sigma(s) = 0$. Ainsi, $s \in \mathscr{D}'$.

Puisque $\sigma(arb) = 0$, on a aussi $\sigma(r) = 0$. Enfin, si r' est un préfixe de r, $\sigma(ar') \geqslant 1$ (de par le caractère minimal de k) et donc $\sigma(r') \geqslant 0$. Ainsi, $r \in \mathscr{D}'$.

Par hypothèse d'induction, on peut affirmer que r et s appartiennent à $\mathscr{D}$, donc m aussi. Ainsi, $\mathscr{D}' \subset \mathscr{D}$.

□

Remarquons que la preuve que nous venons de faire montre aussi que tout mot non vide m de $\mathscr{D}$ se décompose *de manière unique* sous la forme $arbs$ (autrement dit, dans une expression bien parenthésée, toute parenthèse ouvrante ne peut correspondre qu'à une unique parenthèse fermante). En effet, nous venons de constater que arb est le plus petit préfixe propre de m de valuation nulle.

Nombres de Catalan

Il est évident qu'un mot de $\mathscr{D}$ est de longueur paire ; notons donc c_n le nombre de mots de $\mathscr{D}$ de longueur $2n$. La suite $(c_n)_{n\in\mathbb{N}}$ est définie par : $c_0 = 1$ et la relation : $\forall n \in \mathbb{N}$, $c_{n+1} = \sum_{p+q=n} c_p c_q$.

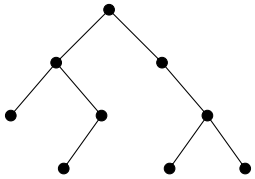
En effet, tout mot de longueur $2n+2$ s'écrit de manière unique sous la forme $arbs$, ou r et s sont des mots de L de longueurs respectives $2p$ et $2q$ vérifiant $p+q=n$.

Vous avez peut-être déjà rencontré cette suite (qui intervient dans de nombreux problèmes combinatoires) : c'est la suite des nombres de Catalan.

c_n est par exemple égal au nombre d'arbres binaires à n nœuds ; on peut en effet établir une bijection entre un mot bien parenthésé et un arbre binaire en convenant que :

- au mot vide est associé l'arbre nil ;
- à un mot bien parenthésé de la forme $arbs$ est associé l'arbre dont la racine a pour fils gauche l'arbre associé à r et pour fils droit l'arbre associé à s.

À l'inverse, lors d'un parcours en profondeur d'un arbre binaire, il suffit de noter par un a le passage à gauche d'un nœud et par un b le passage sous un nœud pour reconstituer le mot à partir de l'arbre. Par exemple, à l'arbre ci-dessous est associé le mot $aaabbaabbbabaabbab$:



On retrouve aussi la suite de Catalan lorsqu'on dénombre les chemins du plan situés dans le demi plan des ordonnées positives, débutant en $(0,0)$ et se terminant en $(2n,0)$, constitués d'une suite de segments de coordonnées vectorielles $(1,1)$ ou $(1,-1)$. Ici l'analogie avec les mots de Dick est plus évidente, en particulier

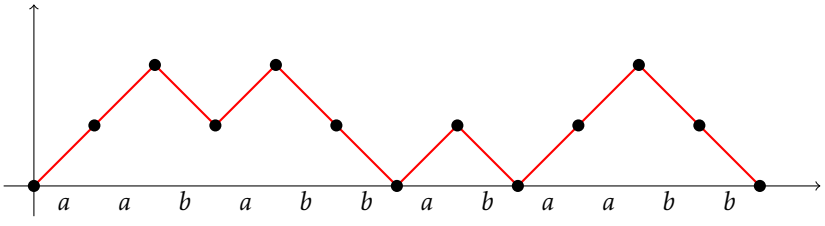


Figure 5 – Un chemin de Dick et le mot qui lui est associé.

avec la caractérisation énoncée dans le dernier théorème : si m' est préfixe de m, $\sigma(m')$ est l'ordonnée du point d'abscisse $|m'|$ du chemin de Dick correspondant.

Venons-en au calcul de c_n. Là encore, plusieurs preuves existent, la plus connue consistant à considérer la série entière :

$$f(x) = \sum_{n=0}^{+\infty} c_n x^n.$$

Supposons pour l'instant son rayon de convergence strictement positif. On effectue alors le produit de Cauchy :

$$f(x)^2 = \sum_{n=0}^{+\infty} \left(\sum_{p+q=n} c_p c_q \right) x^n = \sum_{n=0}^{+\infty} c_{n+1} x^n$$

qui conduit à la relation : $xf(x)^2 = f(x) - 1$.
Cette équation du second degré possède deux solutions lorsque $x > -1/4$, mais seule l'une d'entre elles est développable en série entière au voisinage de 0 : $f(x) = \dfrac{1 - \sqrt{1-4x}}{2x}$, avec un rayon de convergence égal à 1/4.
Il reste à développer cette fonction en série entière (passons sur les calculs) puis à invoquer l'unicité de ce dernier pour en déduire l'expression : $c_n = \dfrac{1}{n+1}\binom{2n}{n}$.
Il est aussi possible de tenir un raisonnement combinatoire en considérant un mot de longueur $2n$, comportant autant de a que de b, mais ne correspondant pas à une expression bien parenthésée : $m = m_1 \cdots m_k \cdots m_{2n}$.
Notons k le plus petit entier pour lequel $\sigma(m_1 \cdots m_k) < 0$. Nous avons nécessairement $m_k = b$ et $\sigma(m_1 \cdots m_{k-1}) = 0$, donc il y a autant de a que de b dans le préfixe $m_1 \cdots m_{k-1}$. Il y a donc un a de plus que de b dans le suffixe $m_{k+1} \cdots m_{2n}$. Associons alors au mot m le mot $m' = m_1 \cdots m_k \overline{m}_{k+1} \cdots \overline{m}_{2n}$ (en notant $\overline{a} = b$ et $\overline{b} = a$).
Posons $\mathrm{B}_1 = \left\{ m \in \Sigma^{2n} \;\middle|\; |m|_a = n \text{ et } |m|_b = n \right\}$ et $\mathrm{B}_2 = \left\{ m \in \Sigma^{2n} \;\middle|\; |m|_a = n-1 \text{ et } |m|_b = n+1 \right\}$. L'association entre m et m' permet d'établir une bijection entre $\mathrm{B}_1 \setminus \mathscr{D}$ et B_2.
De ceci résulte l'égalité : $\binom{2n}{n} - c_n = \binom{2n}{n+1}$, soit $c_n = \binom{2n}{n} - \binom{2n}{n+1} = \dfrac{1}{n+1}\binom{2n}{n}$.

3. Langages

On définit un langage L sur un alphabet Σ de la façon la plus simple qui soit : il s'agit de toute partie de Σ^*, autrement dit un ensemble de mots.
On peut décrire les langages en énumérant leurs éléments : $\mathrm{L} = \left\{ a^n b^n \;\middle|\; n \in \mathbb{N} \right\}$ est le langage des mots qui débutent par un certain nombre de a suivis du même nombre de b.
On peut définir un langage à l'aide d'une propriété, par exemple en considérant le langage des palindromes.
On peut enfin décrire un langage par des règles de construction, qu'on appelle dans ce cas une *grammaire formelle*. Par exemple, le langage des mots de Dick est défini par la grammaire : $\varepsilon \in \mathrm{L}$ et $(r,s) \in \mathrm{L}^2 \Longrightarrow arbs \in \mathrm{L}$.

Langages récursifs
Il existe un nombre dénombrable de mots dans Σ^*, mais le nombre de langages dans Σ^* est indénombrable. Parmi ceux-ci, tous ne sont pas à la portée des informaticiens : il existe des langages qui ne peuvent être énumérés par un algorithme. Plus précisément, on introduit les définitions suivantes :

Définition. — *Un langage* L *est dit :*
- récursivement énumérable *s'il existe un algorithme qui énumère tous les mots de* L ;
- récursif *s'il existe un algorithme qui, prenant un mot u de* Σ^*, *retourne* `true` *si u et dans* L *et* `false` *sinon.*

Tout langage récursif est récursivement énumérable : il suffit de parcourir les mots de Σ^* (par exemple en les classant par longueur croissante) et de soumettre chacun d'eux à l'algorithme qui détermine s'il appartient ou pas à L. La réciproque est fausse, mais de toute façon, seuls les langages récursifs ont un réel intérêt pratique puisqu'ils permettent de discriminer les mots de L et de $\Sigma^* \setminus \mathrm{L}$.

3.1 Opérations sur les langages

Les langages sont des ensembles et à ce titre, toutes les opérations ensemblistes leurs sont applicables : réunion (souvent notée avec l'opérateur + au lieu de $\cup$), intersection, complémentation.
Plus intéressant, la *concaténation* de deux langages L_1 et L_2 se définit par :

$$\mathrm{L}_1\mathrm{L}_2 = \left\{ u \in \Sigma^* \;\middle|\; \exists (x,y) \in \mathrm{L}_1 \times \mathrm{L}_2 \text{ tel que } u = xy \right\}$$

Cette opération de concaténation étant associative, on peut définir par récurrence le langage L^n en posant $L^0 = \{\varepsilon\}$ et $L^{n+1} = L^n L = L L^n$.
Attention cependant à ne pas confondre le langage $L^2 = \left\{uv \;\middle|\; u, v \in L\right\}$ avec le langage plus petit constitué des carrés de L : $\left\{u^2 \;\middle|\; u \in L\right\}$.
Finalement, l'opération de *fermeture de* Kleene (ou plus simplement l'*étoile*) d'un langage L se définit par :

$$L^* = \bigcup_{n\in\mathbb{N}} L^n.$$

De manière concrète, L^* est l'ensemble des mots que l'on peut construire en concaténant un nombre fini (éventuellement réduit à zéro) d'éléments de L.
Notons qu'on peut aussi définir $L^+ = \bigcup_{n\in\mathbb{N}^*} L^n$. À la différence de L^* qui contient toujours le mot vide ε, L^+ ne le contient que si L le contient. On a $L^+ = LL^*$.
D'autres opérations sur les langages existent ; on peut par exemple définir la *racine carrée* d'un langage L en posant :

$$\sqrt{L} = \left\{u \in \Sigma^* \;\middle|\; u^2 \in L\right\}.$$

Notons que pour tout $u \in L$ on a $u^2 \in L^2$ donc $u \in \sqrt{L^2}$. Ceci prouve que $L \subset \sqrt{L^2}$. En général, l'inclusion réciproque est fausse.
Enfin, on citera le *quotient* (gauche) de deux langages : si K et L sont deux langages, on pose :

$$K^{-1}L = \left\{v \in \Sigma^* \;\middle|\; \exists u \in K \text{ tel que } uv \in L\right\}.$$

3.2 Expressions et langages rationnels

L'un des objectifs de ce chapitre est d'introduire un formalisme efficace nous permettant de décrire certaines catégories de mots (autrement dit des langages) que l'on peut être amené à rechercher dans un texte (trouver par exemple tous les mots correspondant à une adresse mail valide dans une page internet). La caractéristique de ces langages est la possibilité de les décrire par des *motifs* (*patterns* en anglais), c'est-à-dire par des formules qu'on appelle des *expressions rationnelles*.

Définition. — *Soit Σ un alphabet. Les* expressions rationnelles [5] *sont définies inductivement par les propriétés suivantes :*

- *$\emptyset$ et ε sont des expressions rationnelles ;*
- *$\forall a \in \Sigma$, a est une expression rationnelle ;*
- *Si e_1 et e_2 sont des expressions rationnelles, alors $e_1 + e_2$, $e_1 e_2$ et e^* sont des expressions rationnelles.*

Remarque. On peut aussi noter $e_1|e_2$ au lieu de $e_1 + e_2$ et $e*$ au lieu de e^*.

L'interprétation d'une expression rationnelle est définie par les règles inductives suivantes :

- $\emptyset$ dénote le langage vide et ε le langage $\{\varepsilon\}$;
- $\forall a \in \Sigma$, a dénote le langage $\{a\}$;
- $e_1 + e_2$ dénote l'union des langages dénotés par e_1 et e_2 ;
- $e_1 e_2$ dénote la concaténation des langages dénotés par e_1 et e_2 ;
- e^* dénote la fermeture de Kleene du langage dénoté par e.

Un langage dénoté par une expression rationnelle sera qualifié de *langage rationnel*.

On notera Σ est une langage rationnel puisque dénoté par $\sum_{a\in\Sigma} a$. Il en est de même de Σ^* et $\Sigma^+ = \Sigma\Sigma^*$.

Exemples. Sur l'alphabet $\Sigma = \{a, b\}$:

- a^*b^* dénote le langage des mots débutant par un certain nombre de a suivi d'un certain nombre de b ;
- $(ab)^*$ dénote le langage des mots alternant la lettre a et la lettre b. Notons que l'expression $\varepsilon + a(ba)^*b$ dénote le même langage ;

5. On parle aussi d'*expressions régulières* par traduction de l'anglais *regular expression*.

- $(a+b)^*aaa(a+b)^*$ dénote le langage des mots dont aaa est un facteur ;
- $(a+ba)^*$ dénote le langage des mots dans lesquels chaque b est suivi d'un a ;
- $(a^*b)^*$ dénote le langage formé du mot vide et de tous les mots qui se terminent par un b. Il peut aussi être dénoté par $\varepsilon + (a+b)^*b$.

Théorème. — *L'ensemble* $\mathrm{Rat}(\Sigma)$ *des langages rationnels sur l'alphabet* Σ *est la plus petite partie de* $\mathscr{P}(\Sigma^*)$ *(au sens de l'inclusion) contenant* $\emptyset$, $\{\varepsilon\}$, $\{a\}$ *pour tout* $a \in \Sigma$ *et stable par réunion, concaténation et passage à l'étoile de* Kleene.

Arbre associé à une expression rationnelle
On peut naturellement associer à toute expression rationnelle un arbre binaire où les feuilles sont éléments de $\Sigma \cup \{\emptyset, \varepsilon\}$ et les nœuds les opérations $\{+, \cdot, *\}$. Ceci permet de prouver certaines propriétés relatives aux expressions rationnelles par induction sur la hauteur de l'arbre, qu'on appelle aussi *profondeur* de l'expression rationnelle.

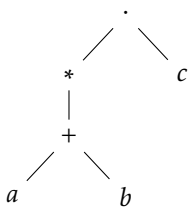


Figure 6 – Arbre associé à l'expression rationnelle $(a+b)^*c$.

Équivalences et réductions
La correspondance entre expression et langage n'est pas biunivoque : chaque expression dénote un unique langage, mais à un langage donné peuvent correspondre plusieurs expressions différentes. Par exemple, les expressions rationnelles a^*a^* et a^* dénotent le même langage des mots composés uniquement de la lettre a. On dit que deux expressions rationnelles sont *équivalentes* lorsqu'elles dénotent le même langage (quelques exemples sont donnés dans l'exercice 9).
Déterminer si deux expressions rationnelles sont équivalentes, ou encore chercher l'expression rationnelle la plus courte dénotant un langage donné sont, on s'en doute, des problèmes difficiles à résoudre de façon algorithmique. L'approche sans doute la plus efficace consiste à utiliser leur transformation en automates finis, qui sera abordée plus tard.

Remarque. On pourrait à juste titre s'étonner de la présence du symbole $\emptyset$ dans la définition d'une expression rationnelle. Et de fait, à l'exception de la description du langage vide ce symbole est inutile pour décrire un langage rationnel. Autrement dit :

Lemme. — *Si le langage dénoté par l'expression régulière e est non vide, il existe une expression équivalente* e' *ne contenant pas le symbole* $\emptyset$.

Preuve. Considérons une expression régulière e dénotant un langage non vide L et raisonnons par induction sur e.

- Si $e = \varepsilon$ ou $e \in \Sigma$, il suffit de poser $e' = e$.
- Si $e = e_1 + e_2$, alors $L = L_1 \cup L_2$, où L_1 et L_2 sont les langages dénotés respectivement par e_1 et e_2. Trois cas sont alors possibles :
  - si $L_1 = \emptyset$ alors $L = L_2 \neq \emptyset$ et par hypothèse d'induction il existe une expression régulière e_2' sans symbole $\emptyset$ tel que $e \equiv e_2 \equiv e_2'$;
  - si $L_2 = \emptyset$ alors $L = L_1 \neq \emptyset$ et par hypothèse d'induction il existe une expression régulière e_1' sans symbole $\emptyset$ tel que $e \equiv e_1 \equiv e_1'$;
  - Si $L_1 \neq \emptyset$ et $L_2 \neq \emptyset$ il existe des expressions e_1' et e_2' équivalentes à e_1 et e_2 et ne contenant pas le symbole $\emptyset$, et alors $e \equiv e_1' + e_2'$.
- Si $e = e_1 e_2$ aucune des deux expressions rationnelles e_1 et e_2 ne peut dénoter l'ensemble vide, donc par hypothèse d'induction il existe des expressions rationnelles e_1' et e_2' sans symbole $\emptyset$ équivalentes respectivement à e_1 et e_2 et alors $e \equiv e_1' e_2'$.

– si $e = e_1^*$ alors ou bien e_1 dénote le langage vide auquel cas $e \equiv \varepsilon$, ou bien e_1 ne dénote pas le langage vide auquel cas il est équivalent à une expression rationnelle e_1' n'utilisant pas le symbole $\emptyset$ et dans ce cas $e \equiv {e_1'}^*$.

□

Ce résultat nous permet de nous restreindre désormais aux expressions rationnelles ne contenant pas le symbole $\emptyset$. En particulier, on utilisera le type Caml suivant pour représenter en machine une expression rationnelle :

```
type regexp =
  | Epsilon
  | Const of string
  | Sum of regexp * regexp
  | Concat of regexp * regexp
  | Kleene of regexp ;;
```

Par exemple, l'expression rationnelle $(a+b)^*c$ (l'arbre associé à cette expression est représenté figure 6) est représentée en Caml par :

```
let e = Concat (Kleene (Sum (Const "a", Const "b")), Const "c") ;;
```

Lemme. — *Si le langage dénoté par l'expression régulière e est non vide, il existe une expression équivalente e' ne contenant ni le symbole $\emptyset$, ni le symbole ε, telle que e soit équivalente à ε, e' ou à $\varepsilon + e'$.*

Preuve. D'après le lemme précédent on peut déjà supposer que e ne contient pas le symbole $\emptyset$. Raisonnons alors de nouveau par induction sur e.

– Si $e = \varepsilon$ ou $e \in \Sigma$ le résultat est évident.

– Si $e = e_1 + e_2$, on applique l'hypothèse d'induction à e_1 et e_2 et suivant les cas on obtient l'une des formes équivalentes à e suivantes :

$+$	ε	e_2'	$\varepsilon + e_2'$
ε	ε	$\varepsilon + e_2'$	$\varepsilon + e_2'$
e_1'	$\varepsilon + e_1'$	$e_1' + e_2'$	$\varepsilon + (e_1' + e_2')$
$\varepsilon + e_1'$	$\varepsilon + e_1'$	$\varepsilon + (e_1' + e_2')$	$\varepsilon + (e_1' + e_2')$

– Si $e = e_1 e_2$, on applique l'hypothèse d'induction à e_1 et e_2 et suivant les cas on obtient l'une des formes équivalentes à e suivantes :

$\cdot$	ε	e_2'	$\varepsilon + e_2'$
ε	ε	e_2'	$\varepsilon + e_2'$
e_1'	e_1'	$e_1' e_2'$	$e_1' + e_1' e_2'$
$\varepsilon + e_1'$	$\varepsilon + e_1'$	$e_2' + e_1' e_2'$	$\varepsilon + (e_1' + e_2' + e_1' e_2')$

– Enfin, si $e = e_1^*$ on applique l'hypothèse d'induction à e_1 et on utilise l'une des formules suivantes :

$$\varepsilon^* = \varepsilon \qquad {e_1'}^* = {e_1'}^* \qquad (\varepsilon + e_1')^* = {e_1'}^*.$$

□

3.3 Langages locaux

Considérons un langage L sur un alphabet Σ et posons :

– $P(L) = \left\{ a \in \Sigma \;\middle|\; a\Sigma^* \cap L \neq \emptyset \right\}$ (l'ensemble des premières lettres des mots de L) ;

– $S(L) = \left\{ a \in \Sigma \;\middle|\; \Sigma^* a \cap L \neq \emptyset \right\}$ (l'ensemble des dernières lettres des mots de L) ;

– $F(L) = \left\{ u \in \Sigma^2 \;\middle|\; \Sigma^* u \Sigma^* \cap L \neq \emptyset \right\}$ (l'ensemble des facteurs de longueur 2 des mots de L).

– $N(L) = \Sigma^2 \setminus F(L)$.

De manière évidente nous avons :

$$L \setminus \{\varepsilon\} \subset \Big(P(L)\Sigma^* \cap \Sigma^* S(L)\Big) \setminus (\Sigma^* N(L) \Sigma^*)$$

ce qui signifie simplement que tout mot non vide de L a sa première lettre dans P(L), sa dernière lettre dans S(L), et qu'aucun de ses facteurs de longueur 2 n'est dans N(L).

On dit d'un langage L sur l'alphabet Σ qu'il est *local* lorsque cette inclusion est en fait une égalité. Plus précisément, on a :

Définition. — *Un langage* L *est dit* local *lorsqu'il existe deux parties* P *et* S *de* Σ *et une partie* N *de* Σ^2 *(l'ensemble des mots de longueur 2) tels que :*

$$L \setminus \{\varepsilon\} = (P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*).$$

Remarque. Si de telles parties existent, nécessairement P = P(L), S = S(L), N = N(L).

À première vue, l'intérêt de cette notion n'est pas évident, mais nous verrons plus tard que ces langages sont particulièrement bien adaptés à la reconnaissance efficace d'une expression rationnelle par un automate.

Exemples. On considère l'alphabet $\Sigma = \{a, b\}$. Alors :

– le langage dénoté par a^* est local (P = S = $\{a\}$, N = $\{ab, ba, bb\}$) ;

– le langage dénoté par $(ab)^*$ est local (P = $\{a\}$, S = $\{b\}$, N = $\{aa, bb\}$) ;

– en revanche les langages dénotés par $L_1 = a^* + (ab)^*$ et $L_2 = a^*(ab)^*$ ne sont pas locaux.

Pour montrer que les langages ci-dessus ne sont pas locaux, on calcule pour chacun d'eux $P(L_i) = \{a\}$, $S(L_i) = \{a, b\}$ et $F(L_i) = \{aa, ab, ba\}$, soit $N(L_i) = \{bb\}$. On observe alors que le mot *aba* est dans $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*)$ mais ni dans L_1, ni dans L_2, qui ne peuvent donc être locaux.

Théorème. — *L'intersection de deux langages locaux est un langage local.*

Preuve. Considérons deux langages locaux L_1 et L_2 et posons :

$$P = P(L_1) \cap P(L_2), \quad S = S(L_1) \cap S(L_2), \quad N = N(L_1) \cup N(L_2).$$

Alors :

$$\begin{aligned}(L_1 \cap L_2) \setminus \{\varepsilon\} &= (L_1 \setminus \{\varepsilon\}) \cap (L_2 \setminus \{\varepsilon\}) = \Big(P(L_1)\Sigma^* \cap \Sigma^* S(L_1)\Big) \setminus (\Sigma^* N(L_1) \Sigma^*) \cap \Big(P(L_2)\Sigma^* \cap \Sigma^* S(L_2)\Big) \setminus (\Sigma^* N(L_2) \Sigma^*) \\ &= \Big(P(L_1)\Sigma^* \cap \Sigma^* S(L_1) \cap P(L_2)\Sigma^* \cap \Sigma^* S(L_2)\Big) \setminus \Big(\Sigma^* N(L_1)\Sigma^* \cup \Sigma^* N(L_2)\Sigma^*\Big) \\ &= (P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*)\end{aligned}$$

donc $L_1 \cap L_2$ est bien local. □

Remarque. Dans les exemples donnés plus haut nous avons constaté que ni la réunion ni la concaténation de deux langages locaux n'était pas nécessairement un langage local. On dispose cependant des résultats suivants :

Théorème. — *Si* L_1 *et* L_2 *sont deux langages locaux définis sur des alphabets* disjoints, *alors* $L_1 \cup L_2$ *est encore un langage local.*

Preuve. Notons Σ_1 et Σ_2 les alphabets sur lesquels sont respectivement définis L_1 et L_2, et $\Sigma = \Sigma_1 \cup \Sigma_2$.
On calcule $P(L_1 \cup L_2) = P(L_1) \cup P(L_2)$, $S(L_1 \cup L_2) = S(L_1) \cup S(L_2)$, $F(L_1 \cup L_2) = F(L_1) \cup F(L_2)$.

Considérons alors un mot u de $(P(L_1 \cup L_2)\Sigma^* \cap \Sigma^* S(L_1 \cup L_2)) \setminus (\Sigma^* N(L_1 \cup L_2)\Sigma^*)$ et décomposons-le en lettres : $u = a_1 a_2 \cdots a_p$.

– $a_1 \in P(L_1) \cup P(L_2)$; sans perte de généralité supposons par exemple $a_1 \in P(L_1)$. Alors $a_1 \in \Sigma_1$.

– $a_1 a_2 \in F(L_1) \cup F(L_2)$. Mais $F(L_1) \subset \Sigma_1^2$ et $F(L_2) \subset \Sigma_2^2$. Les deux alphabets étant disjoints et $a_1 \in \Sigma_1$ on a nécessairement $a_1 a_2 \in F(L_1)$ et $a_2 \in \Sigma_1$.

– De proche en proche on prouve que pour tout $i \in [\![1, p-1]\!]$, $a_i a_{i+1} \in F(L_1)$ et $a_{i+1} \in \Sigma_1$.

– Enfin, $a_p \in \Sigma_1$ et $a_p \in S(L_1) \cup S(L_2)$ donc $a_p \in S(L_1)$.

L_1 étant un langage local on en déduit que $u \in L_1$ et donc que $u \in L_1 \cup L_2$. □

Théorème. — *Si* L_1 *et* L_2 *sont deux langages locaux définis sur des alphabets* disjoints, *alors* L_1L_2 *est encore un langage local.*

Preuve. Avec les mêmes notations on a cette fois :

$$P(L_1L_2) = \begin{cases} P(L_1) & \text{si } \varepsilon \notin L_1 \\ P(L_1) \cup P(L_2) & \text{sinon} \end{cases} \qquad S(L_1L_2) = \begin{cases} S(L_2) & \text{si } \varepsilon \notin L_2 \\ S(L_1) \cup S(L_2) & \text{sinon} \end{cases}$$

$$F(L_1L_2) = F(L_1) \cup F(L_2) \cup S(L_1)P(L_2).$$

Considérons un mot u de $(P(L_1L_2)\Sigma^* \cup \Sigma^* S(L_1L_2)) \setminus (\Sigma^* N(L_1L_2)\Sigma^*)$ et décomposons-le en lettres : $u = a_1a_2\cdots a_p$.

– Si $a_1 \in \Sigma_2$ alors $\varepsilon \in L_1$ et $a_2 \in P(L_2)$. De proche en proche on prouve que $a_ia_{i+1} \in F(L_2)$ et $a_{i+1} \in \Sigma_2$. En particulier $a_p \in \Sigma_2$ donc $a_p \in S(L_2)$. Puisque L_2 est un langage local on en déduit que $u \in L_2$, et puisque $\varepsilon \in L_1$ on a aussi $u \in L_1L_2$.

– Si $a_1 \in \Sigma_1$, notons $a_1 \cdots a_k$ le plus long préfixe de u qui soit dans Σ_1^*. Alors $a_1 \in P(L_1)$ et $\forall i \in [\![1, k-1]\!]$, $a_ia_{i+1} \in F(L_1)$. Ensuite, de deux choses l'une :

  – si $k = p$ alors $a_p \in S(L_1)$ et $\varepsilon \in L_2$. Puisque L_1 est local on a $u \in L_1$ et puisque $\varepsilon \in L_2$ on a aussi $u \in L_1L_2$.

  – si $k < p$ alors $a_ka_{k+1} \in S(L_1)P(L_2)$ donc $a_{k+1} \in \Sigma_2$ et de proche en proche on prouve que $\forall j \in [\![k+1, p-1]\!]$, $a_ja_{j+1} \in F(L_2)$, $a_{j+1} \in \Sigma_2$ et $a_p \in S(L_2)$. Puisque L_1 et L_2 sont locaux on en déduit que $a_1 \cdots a_k \in L_1$ et $a_{k+1} \cdots a_p \in L_2$ et donc que $u \in L_1L_2$.

□

Enfin, on dispose du théorème suivant :

Théorème. — *La fermeture de* Kleene L^* *d'un langage local* L *est aussi un langage local.*

Preuve. On a $P(L^*) = P(L)$, $S(L^*) = S(L)$ et $F(L^*) = F(L) \cup S(L)P(L)$. En effet, considérons un mot $u \in L^* \setminus \{\varepsilon\}$. Alors $u = u_1u_2\cdots u_p$ avec $u_i \in L$.
La première lettre de u est aussi celle de u_1 donc est dans $P(L)$;
La dernière lettre de u est aussi celle de u_p donc est dans $S(L)$.
Un facteur de longueur 2 de u est :

– un facteur d'un des u_i auquel cas il est dans $F(L)$;

– ou de la forme xy où x est la dernière lettre de u_i et y la première lettre de u_{i+1}, auquel cas il est dans $S(L)P(L)$;

Posons donc $P = P(L)$, $S = S(L)$ et $N = \Sigma^* \setminus (F(L) + S(L)P(L))$. Pour montrer que L^* est local nous devons considérer un mot u de $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N\Sigma^*)$ et prouver que $u \in L^*$.
Parmi les facteurs de longueur 2 de u, considérons uniquement ceux qui appartiennent à $S(L)P(L)$; ils induisent une factorisation de u en mots : $u = u_1u_2\cdots u_p$ avec $u_i \in (P(L)\Sigma^* \cap \Sigma^* S(L)) \setminus (\Sigma^* N(L)\Sigma^*)$. Or par hypothèse L est local, donc chacun des u_i est un mot de L. On en déduit que $u \in L^*$. □

3.4 Expressions rationnelles linéaires

Nous l'avons vu, une expression rationnelle générale ne dénote pas nécessairement un langage local. Il y a cependant un cas particulier pour lequel c'est le cas : les expressions rationnelles *linéaires*.

Définition. — *Une expression rationnelle e est dite* linéaire *lorsque tout caractère de* Σ *apparaît au plus une fois dans e :*

$$\forall a \in \Sigma, \quad |e|_a \leqslant 1.$$

Par exemple, l'expression rationnelle $(ab)^*$ est linéaire, mais pas $(ab)^*a$.

L'intérêt de cette notion est justifié par le théorème suivant :

Théorème. — *Toute expression rationnelle linéaire dénote un langage local.*

Preuve. Raisonnons par induction structurelle sur l'expression rationnelle linéaire e.

- $\emptyset$ et ε dénotent respectivement les langages $\emptyset$ et $\{\varepsilon\}$ qui sont des langages locaux.
- si $a \in \Sigma$, a dénote le langage local $\{a\}$ avec $P = \{a\}$, $S = \{a\}$ et $N = \Sigma^2$.
- si $e = e_1 + e_2$ est une expression rationnelle linéaire, il en est de même de e_1 et e_2, et par hypothèse d'induction ces deux expressions dénotent des langages locaux. Mais e_1 et e_2 peuvent être définis sur des alphabets disjoints, donc e dénote aussi un langage local.
- si $e = e_1 e_2$ ou $e = e_1^*$ le raisonnement est identique.

□

Remarque. La réciproque de ce résultat est fausse : le langage aa^* est local mais ne peut être dénoté par une expression rationnelle linéaire.

• Algorithmes de calcul des ensembles P, S et F

Les différents résultats établis au sujet des langages locaux nous permettent d'écrire des algorithmes de calcul des ensembles P, S et F associés à une expression rationnelle linéaire, en procédant par induction structurelle. On observera que pour calculer l'ensemble des préfixes et suffixes du langage local dénoté par une expression rationnelle linéaire il est nécessaire de posséder une fonction qui détermine si ε appartient au langage. On commence donc par définir :

```
let rec motvide = function
  | Epsilon         -> true
  | Const _         -> false
  | Sum (e1, e2)    -> motvide e1 || motvide e2
  | Concat (e1, e2) -> motvide e1 && motvide e2
  | Kleene _        -> true ;;
```

On peut maintenant définir les fonctions qui calculent les ensembles P et S :

```
let rec prefixe = function
  | Epsilon                        -> []
  | Const a                        -> [a]
  | Sum (e1, e2)                   -> union (prefixe e1) (prefixe e2)
  | Concat (e1, e2) when motvide e1 -> union (prefixe e1) (prefixe e2)
  | Concat (e1, e2)                -> prefixe e1
  | Kleene e                       -> prefixe e ;;

let rec suffixe = function
  | Epsilon                        -> []
  | Const a                        -> [a]
  | Sum (e1, e2)                   -> union (suffixe e1) (suffixe e2)
  | Concat (e1, e2) when motvide e2 -> union (suffixe e1) (suffixe e2)
  | Concat (e1, e2)                -> suffixe e2
  | Kleene e                       -> suffixe e ;;
```

Pour calculer l'ensemble F il faut posséder une fonction qui calcule le produit cartésien de deux listes :

```
let rec produit l1 l2 = match (l1, l2) with
  | [], _     -> []
  | _, []     -> []
  | t1::q1, _ -> union (map (function x -> t1 ^ x) l2) (produit q1 l2) ;;
```

On en déduit la fonction calculant F :

```
let rec facteur = function
  | Epsilon         -> []
  | Const a         -> []
  | Sum (e1, e2)    -> union (facteur e1) (facteur e2)
  | Concat (e1, e2) -> let l = union (facteur e1) (facteur e2)
                       in union l (produit (suffixe e1) (prefixe e2))
  | Kleene e        -> union (facteur e) (produit (suffixe e) (prefixe e)) ;;
```

Illustrons l'usage de ces trois fonctions avec l'exemple de l'expression rationnelle linéaire $(a+b)^*c$:

```
# prefixe e ;;
- : string list = ["a"; "b"; "c"]
# suffixe e ;;
- : string list = ["c"]
# facteur e ;;
- : string list = ["aa"; "ab"; "ba"; "bb"; "ac"; "bc"]
```

4. Exercices

Exercice 1 Remplir la grille ci-dessous (à raison d'un caractère par case) de sorte que chaque mot, lu horizontalement ou verticalement, soit reconnu par l'expression régulière correspondante (en annexe figure la définition des expressions régulières étendues).

	`[RUTH]*(OE|EO)[RB]*`	`(BG|ON|KK)+[RIF]+`	`(MN|BO|FI)[EU]{2,}`	`(KT|AL|ET)+G`	`[OH](PR|AX|TR)+`
`[IT](O)*(BE|AD)*\1`					
`[NORMAL]+T{2}`					
`.*(XA|BE).*`					
`(EG|UL){2}[ALF]*`					
`[REQ]*(G|P)(.)+`					

Source : `https://regexcrossword.com/`

4.1 Mots et alphabets

Exercice 2

a) Soient u, v et w trois mots de Σ^* tels que u et v soient préfixes de w. Montrer que u est préfixe de v ou v est préfixe de u.

b) Soient a et b deux lettres de Σ et $u \in \Sigma^*$ tel que $au = ub$. Montrer que $a = b$ et $u \in \{a\}^*$.

c) Soient u et v deux mots de Σ^*. On suppose qu'il existe deux entiers non nuls p et q tels que $u^p = v^q$. Montrer qu'il existe un mot $w \in \Sigma^*$ et deux entiers m et n tels que $u = w^m$ et $v = w^n$.

Exercice 3 Soit Σ un alphabet. On définit sur Σ^* la relation $\mathscr{R}$ suivante :

$$u \mathscr{R} v \iff \exists x, y \in \Sigma^* \;\Big|\; u = xy \text{ et } v = yx.$$

a) Montrer que $\mathscr{R}$ est une relation d'équivalence.

b) Montrer que $u \mathscr{R} v$ si et seulement s'il existe un mot w tel que $uw = wv$.

c) Soit $n \in \mathbb{N}^*$. Montrer que $u \mathscr{R} v$ si et seulement si $u^n \mathscr{R} v^n$.

Exercice 4 Les mots de Fibonacci sur l'alphabet $\Sigma = \{a, b\}$ sont définis par les relations :

$$f_0 = \varepsilon, \quad f_1 = a, \quad f_2 = b \quad \text{et} \quad \forall n \geqslant 1,\ f_{n+2} = f_{n+1} f_n.$$

a) Montrer que pour tout $n \geqslant 3$ le suffixe de longueur 2 de f_n est ab si n est pair, ba si n est impair.

b) Notons, pour tout $n \geqslant 3$, g_n le préfixe de f_n obtenu en supprimant les deux dernières lettres de ce mot. Montrer que g_n est un palindrome.

Exercice 5 On considère l'alphabet $\Sigma = \{a, b\}$, qui sera représenté en Caml par le type :

```
type alphabet = a | b ;;
```

les mots sont représentés par le type *alphabet list*, et on note $\mathscr{D}$ l'ensemble des mots de Dick.

a) Définir une fonction de type *alphabet list –> bool* qui détermine si un mot appartient ou pas à $\mathscr{D}$.

Étant donné un mot bien parenthésé, on appelle *factorisation* de ce mot un découpage de ce dernier en produit d'expressions bien parenthésées, chacune d'entre elles étant appelé un *facteur* de ce mot.
Par exemple, $aababb = (()())$ est composé d'un seul facteur, $abaababb = ().(()())$ est composé de deux facteurs, et $aabbabaababb = (()).().(()())$ est composé de trois facteurs.

b) Réaliser une fonction calculant le nombre de facteurs que possède un mot de Dick, puis une fonction affichant cette factorisation.

Exercice 6 Dans cet exercice, on considère l'alphabet $\Sigma = \{a, b\}$. On définit sur Σ^* l'application $r \longmapsto \overline{r}$ par les égalités :

$$\overline{\varepsilon} = \varepsilon \qquad \text{et} \qquad \forall s \in \Sigma^*, \quad \overline{as} = \overline{s}b \quad \overline{bs} = \overline{s}a.$$

a) Calculer $\overline{aababa}$, puis montrer que pour tout $(r, s) \in (\Sigma^*)^2$, $\overline{rs} = \overline{s}\,\overline{r}$.

On utilise comme dans l'exercice précédent les types *alphabet* et *alphabet list* pour représenter respectivement les lettres de Σ et les mots de Σ^*.

b) Définir une fonction caml effectuant la transformation $r \longmapsto \overline{r}$.

Prenons maintenant une feuille de papier, et plions-la n fois dans le sens vertical, en repliant à chaque fois la moitié droite sur la gauche. Les plis de la feuille, une fois redépliée, forment une suite de creux et de bosses.

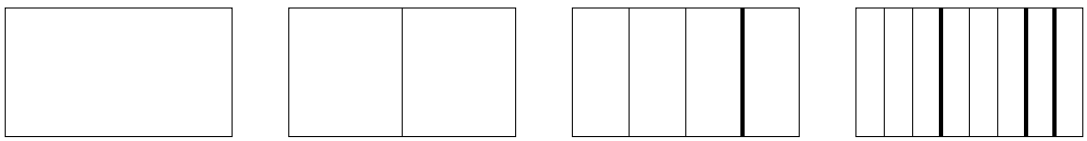


Figure 7 – Un trait fin représente un creux, un trait épais une bosse.

Un pliage de la feuille peut donc être codé par un mot de Σ^* en représentant par a un creux et par b une bosse. Ainsi, les premiers mots qu'on obtient sont : ε, a, aab, $aabaabb$. On note $u_0 = \varepsilon$, $u_1 = a$, $u_2 = aab$, $u_3 = aabaabb$, ... la suite de mots engendrée par les pliages successifs.

c) Exprimer u_{n+1} en fonction de u_n, et en déduire une fonction Caml construisant les mots de cette suite.

Les mots de ce langage étant préfixes les uns des autres, on peut considérer le « mot infini » w dont ils sont tous préfixes. On note w_n la n-ème lettre de ce mot.

d) Que vaut w_n lorsque n est impair ? Montrer ensuite que pour tout $n \in \mathbb{N}$, $w_n = w_{2n}$. En déduire un algorithme calculant w_n. Quelle est la 2000-ème lettre de w ?

Exercice 7 Dans cet exercice, nous allons nous intéresser aux mots ne possédant pas deux facteurs consécutifs égaux, autrement dit les mots sans facteurs carrés : ils ne peuvent s'écrire : rs^2t avec $|s| \geqslant 1$.

a) Montrer que si Σ ne contient que deux lettres, tout mot de quatre lettres ou plus possède un facteur carré.

Considérons maintenant l'alphabet $\Sigma = \{a, b\}$, et le morphisme σ défini par : $\sigma(a) = ab$ et $\sigma(b) = ba$.

b) Montrer que pour tout $n \in \mathbb{N}^*$, $\sigma^{n-1}(a)$ est préfixe de $\sigma^n(a)$; on peut donc considérer le mot m de longueur infinie tel que $\sigma^n(a)$ soit préfixe de m pour tout entier $n \in \mathbb{N}$. Ce mot infini est connu comme étant la suite de Thue-Morse.

c) On pose $\Sigma_1 = \{ab, ba\}$. Montrer que si $s \in \Sigma_1^*$, alors asa et bsb n'appartiennent pas à Σ_1^*, puis que pour tout $n \in \mathbb{N}^*$, $\sigma^n(a) \in \Sigma_1^*$.

d) En déduire que m ne possède pas de facteur de la forme r^2x, où r est un mot et x la première lettre du mot r.

On considère enfin le mot μ (infini) formé du nombre de b compris entre deux a consécutifs du mot m.

e) Montrer que l'alphabet $\{0, 1, 2\}$ est suffisant pour l'écrire, et qu'il ne possède pas de facteurs carrés.

f) Écrire une fonction Caml permettant de le générer.

4.2 Expressions et langages rationnels

Exercice 8 Soit $\Sigma = \{a, b\}$ un alphabet, et a et b deux de ses lettres. Donner une description en français des langages dénotés par les expressions rationnelles suivantes :

$$(\varepsilon + \Sigma)(\varepsilon + \Sigma), \quad (\Sigma^2)^*, \quad (b + ab)^*(a + \varepsilon), \quad (ab^*a + b)^*.$$

Exercice 9 Donner des expressions rationnelles qui dénotent les langages suivants :

- les mots sur l'alphabet $\Sigma = \{a, b\}$ qui contiennent au moins un a ;
- les mots sur l'alphabet $\Sigma = \{a, b\}$ qui contiennent au plus un a ;
- les mots sur l'alphabet $\Sigma = \{a, b\}$ tels que toute série de a soit de longueur paire ;
- les mots sur l'alphabet Σ dont la longueur n'est pas divisible par 3 ;
- les mots sur l'alphabet $\Sigma = \{a, b\}$ tels que deux lettres consécutives soient toujours distinctes ;
- les mots sur l'alphabet $\Sigma = \{a, b, c\}$ tels que deux lettres consécutives soient toujours distinctes.

Exercice 10 e et f étant des expressions rationnelles quelconques, montrer que les expressions rationnelles suivantes dénotent les mêmes langages :

$$(e + f)^*, \qquad e^*(e + f)^*, \qquad (e^* + f)^*, \qquad (e^*f^*)^*, \qquad (e^*f)^*e^*.$$

Exercice 11 Montrer que l'intersection des deux langages dénotés respectivement par $(b^*a^2b^*)^*$ et $(a^*b^2a^*)^*$ est rationnel.
Montrer que le complémentaire du langage dénoté par $(a + b)^*b$ est rationnel.

Exercice 12 **Lemme d'Arden.**
Montrer qu'il existe un unique langage L sur l'alphabet $\Sigma = \{a, b\}$ vérifiant $L = aL + b$, et qu'il s'agit du langage dénoté par a^*b.
Plus généralement, si A et B sont deux langages sur un même alphabet tel que $\varepsilon \notin A$, montrer que l'équation $L = AL + B$ admet pour unique solution $L = A^*B$.

Application. Sur $\Sigma = \{a, b\}$ on note L_1 le langage des mots ayant un nombre pair de b et L_2 le langage des mots ayant un nombre impair de b. Écrire deux relations linéaires liant L_1 et L_2 puis utiliser le lemme d'Arden pour les résoudre.

4.3 Langages locaux

Exercice 13 Montrer que si L est un langage local, le langage L′ des facteurs des mots de L est aussi local.

Exercice 14 Montrer qu'un langage L sur l'alphabet Σ est local si et seulement si :

$$\forall u, v, u', v' \in \Sigma^*, \quad \forall a \in \Sigma, \qquad (uav \in L \text{ et } u'av' \in L) \Longrightarrow uav' \in L.$$

4.4 Exercices divers

Exercice 15 On appelle *code* sur un alphabet Σ tout langage L qui vérifie la propriété suivante :

$$(u_1u_2\cdots u_p = v_1v_2\cdots v_q \quad \text{avec} \quad u_i, v_j \in \mathrm{L}) \Longrightarrow (p = q \quad \text{et} \quad \forall i \in [\![1,p]\!],\ u_i = v_i)$$

(autrement dit, toute factorisation d'un mot de L^* est unique).

a) Construire un code de quatre mots sur l'alphabet $\Sigma = \{a, b\}$.

b) Justifier que $\mathrm{L}_1 = \{a, ab, ba\}$ n'est pas un code puis que $\mathrm{L}_2 = \{a, ab\}$ en est un.

c) Démontrer que si tous les mots de L ont même longueur alors L est un code.

d) Soit $u, v \in \Sigma^*$ deux mots distincts. Montrer que $\{u, v\}$ est un code si et seulement si $uv \neq vu$.

e) Soit L un langage tel qu'aucun mot de L ne soit préfixe propre d'un autre mot de L. Montrer que L est un code. De tels codes sont appelés des *codes préfixes*.

f) Montrer que le produit de deux codes préfixes est encore un code préfixe.

L'algorithme de Sardinas et Patterson permet de décider si un langage L de cardinal fini est un code. Il se décrit de la manière suivante :

(i) **initialisation** : on pose $\mathrm{S}_0 = \mathrm{L}^{-1}\mathrm{L} \setminus \{\varepsilon\}$;

(ii) **itération** : $\mathrm{S}_{i+1} = \mathrm{S}_i^{-1}\mathrm{L} \cup \mathrm{L}^{-1}\mathrm{S}_i$;

(iii) **arrêt** : s'il existe i tel que $\varepsilon \in \mathrm{S}_i$, L n'est pas un code ; s'il existe $i > j$ tel que $\mathrm{S}_i = \mathrm{S}_j$, L est un code.

Rappel : si K et L sont deux langages, $\mathrm{K}^{-1}\mathrm{L} = \left\{v \in \Sigma^* \;\middle|\; \exists u \in \mathrm{K},\ uv \in \mathrm{L}\right\}$.

g) Appliquer l'algorithme de Sardinas et Patterson pour déterminer lesquels parmi les langages suivants sont des codes :

$\mathrm{L}_1 = \{ab, baa, abba, aabaa\}$ $\quad \mathrm{L}_2 = \{aaa, aba, abb, abaab\}$ $\quad \mathrm{L}_3 = \{a, abba\}$

$\mathrm{L}_4 = \{b, aa, ab, ba\}$ $\quad \mathrm{L}_5 = \{aa, ab, ba, bb, baaababa\}$ $\quad \mathrm{L}_6 = \{a, ab, bba, bbab, bbbb\}$

h) Justifier la terminaison de cet algorithme (on admettra sa validité).

Annexe : expressions régulières étendues

Les expressions régulières sont utilisées par des langages de programmation (Perl, PHP, module `re` de Python), des éditeurs de texte (emacs, vim), des commandes unix (`grep`). Malheureusement, malgré un effort de normalisation chacun de ces langages ou programmes utilise en partie ses propres notations. Celles présentées ici répondent au standard de normalisation POSIX ; certaines d'entre elles demandent à être adaptées suivant le langage ou le programme utilisé.

quantificateurs	
`n*`	0 ou plus *n*
`n+`	1 ou plus *n*
`n?`	0 ou 1 *n*
`n{2}`	exactement 2 *n*
`n{2,}`	au moins 2 *n*
`n{2,4}`	2, 3 ou 4 *n*

caractères spéciaux	
`\`	caractère d'échappement
`\n`	nouvelle ligne
`\s`	espace
`\w`	un caractère a-z, A-z ou 0-9 ou _
`\d`	un chiffre 0-9

intervalles	
`.`	tout caractère
`(a|b)`	*a* ou *b*
`[abc]`	*a*, *b* ou *c*
`[^abc]`	tout caractère sauf *a*, *b*, *c*
`[a-z]`	caractère alphabétique en minuscule
`[A-Z]`	caractère alphabétique en majuscule
`[0-9]`	chiffre
`(...)`	délimite un groupe
`\n`	le n^{e} groupe

méta-caractères
`^ [ . $ { * ( ) + \ | ? < >`

Les méta-caractères doivent être précédés du caractère d'échappement.
Par exemple, l'expression régulière ci-dessous peut être utilisée pour reconnaître une adresse mail valide :

```
\w+([-+.']\w+)*@\w+([-.]\w+)*\.[a-zA-Z]{2,6}
```