جامعة باجي مختار عنابة
كلية علوم الهندسة
قسم الإعلام الآلي

Support de cours et de TD

Théorie des langages*

Préparé par Dr. T. BENOUHIBA

Parcours de licence en informatique

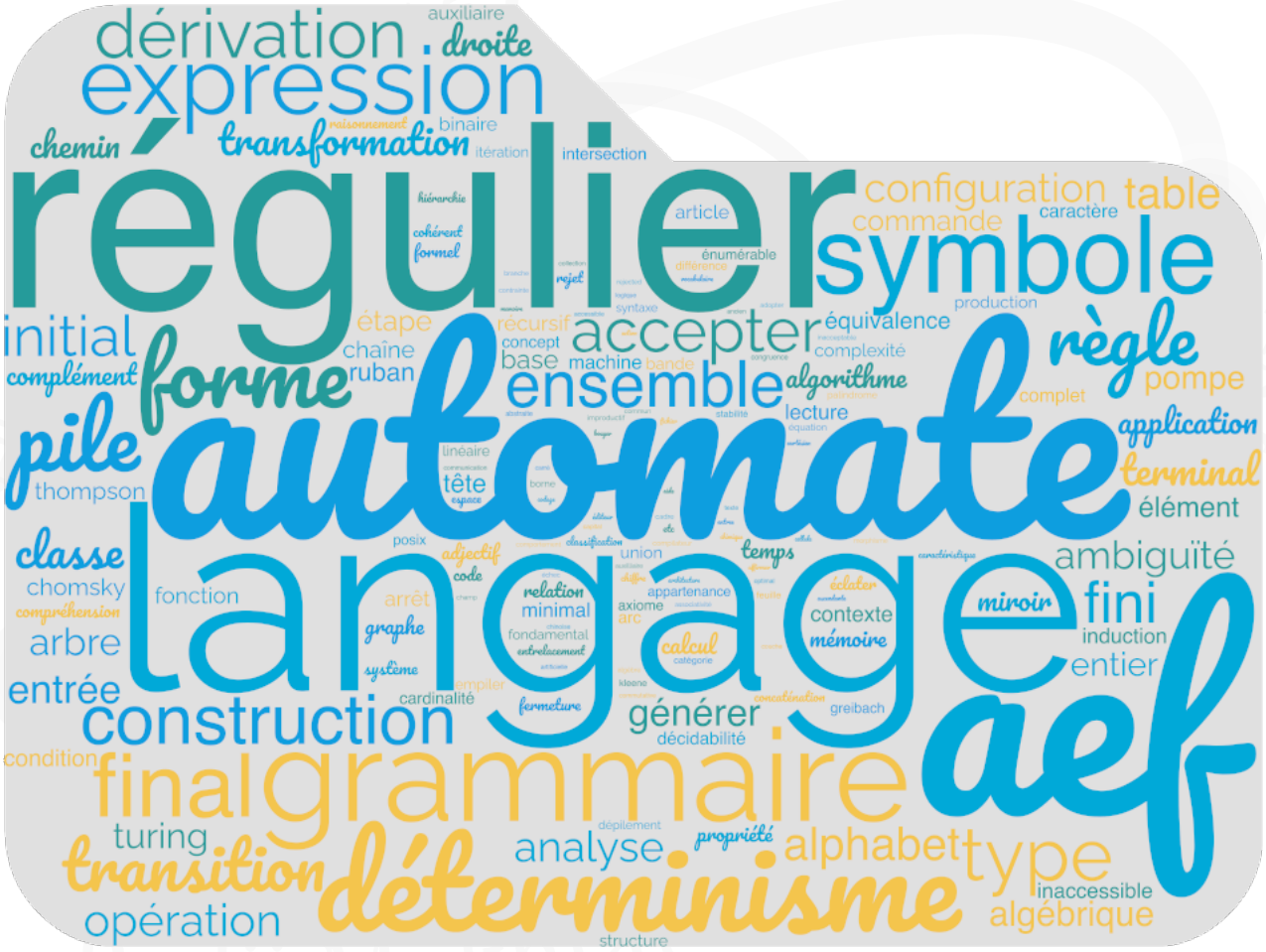

régulier
automate
langage
aef
grammaire
déterminisme
expression
dérivation
symbole
forme
pile
accepter
ensemble
règle
transformation
chemin
droite
auxiliaire
binaire
itération
intersection
configuration
table
commande
caractère
article
énumérable
différence
rejet
production
initial
étape
récursif
équivalence
chaîne
ruban
concept
base
machine
bande
complexité
algorithme
complément
lecture
complet
pompe
application
terminal
élément
thompson
tête
linéaire
espace
classe
chomsky
compréhension
fonction
arbre
entrée
construction
condition
posix
adjectif
code
union
temps
ambiguïté
arrêt
graphe
système
relation
minimal
fondamental
entrelacement
appartenance
axiome
arc
calcul
éclater
contexte
mémoire
miroir
fini
induction
entier
cardinalité
empiler
fermeture
générer
décidabilité
concaténation
greibach
final
turing
transition
opération
dépilement
analyse
propriété
alphabet
type
inaccessible
algébrique
structure

Table des matières

Première partie

Cours et TD

Chapitre 1

Notions fondamentales en théorie des langages

1. Rappels sur la théorie des ensembles

1..1 Définitions

Définition 1.1 : Un ensemble est une collection d'objets sans répétition, chaque objet est appelé un élément. L'ensemble ne contenant aucun élément est dit l'ensemble vide et est noté $\emptyset$.

Un ensemble peut être défini de plusieurs façons, chacune a ses avantages et ses inconvénients. Notons, au passage, le parallèle existant entre la théorie des ensembles et la logique des prédicats, ce qui fait que beaucoup de notions relatives aux ensembles sont définies en termes de la logique des prédicats.

Définition en extension

Définir un ensemble par extension revient simplement à donner la liste de ses éléments. Prenons l'exemple des chiffres entiers de 0 à 9 définis par l'écriture $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Cette définition est utile lorsque le nombre d'éléments d'un ensemble est fini et n'est pas très important. Notons que lorsqu'un élément x figure dans un ensemble A, cela est noté par $x \in A$. Par certains artefacts, on arrive à noter des ensembles infinis mais cela n'est pas pratique.

Définition par compréhension

Définir un ensemble A par compréhension revient à définir un prédicat P qui ne peut prendre la valeur "vraie" que si, et seulement si, il est évalué pour un élément de A, on écrira alors : $A = \{x|P(x)\}$. Comme exemple, nous pouvons définir l'ensemble des entiers naturels multiples de 3 par $\{x|x \bmod 3 = 0\}$ ($\bmod$ est l'opération modulo).
La définition par compréhension est plus pratique car elle permet de définir aisément des ensembles ayant un nombre infini d'éléments (comme c'est le cas pour l'exemple donné). Elle peut néanmoins mener à un paradoxe appelé paradoxe de Russel si on autorise P à être n'importe quel prédicat.

Définition par induction

Définir un ensemble A par induction revient à montrer comment les éléments de A sont construits. On parlera de preuve d'appartenance pour tout élément appartenant à A. Concrètement, la définition par induction de A nécessite de définir un ensemble fini d'éléments de A (appelé les triviaux de A et noté $triv(A)$) dont on suppose l'appartenance à A, une ou plusieurs fonctions $f_i : A \longrightarrow A$ et des règles de la forme :

$$x \rightarrow f_i(x)$$

signifiant que si x est un élément de A alors $f_i(x)$ est également un élément de A. On écrira $A = \{triv(A); x \in A \longrightarrow f_i(x) \in A\}$.
Comme exemple, on peut définir les entiers naturels par $\mathbb{N} = \{0; x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}\}$. La définition par induction permet, tout comme la définition par compréhension, de définir des ensembles infinis. Cependant, en permettant une meilleure structuration, elle peut simplifier la définition de certains ensembles complexes.

1..2 Démonstration de propriétés sur les ensembles

Souvent, on s'intéresse à une propriété Q censée être satisfaite par tout élément d'un ensemble A. Selon la définition adoptée de l'ensemble, on peut obtenir plusieurs types de démonstration :

Démonstration dans le cas d'une définition en extension

Dans le cas d'une définition en extension, Q est satisfaite par A si elle l'est pour tout élément A. En d'autres termes, il faut prendre chaque élément de A, le placer comme argument de Q puis vérifier si Q s'évalue à vrai. Évidemment, cette démonstration ne peut fonctionner que si l'ensemble A est fini. Elle n'est pas pratique si le nombre d'éléments est important.

Démonstration dans le cas d'une définition par compréhension

Supposons que $A = \{x|P(x)\}$. Dans ce cas, Q est satisfaite par A si on arrive à démontrer que $P(x) \rightarrow Q(x)$ en utilisant les axiomes de l'ensemble considéré. Bien sûr, on sera confronté au problème de complétude et démontrabilité des théorèmes (par exemple, on sait que quel que soit l'ensemble des axiomes définissant les entiers, il existera toujours des propriétés des nombres entiers que l'on ne pourra pas démontrer).

Démonstration dans le cas d'une définition par induction

Supposons que $A = \{triv(A); x \in A \rightarrow f(x) \in A\}$. Montrer qu'une propriété Q est satisfaite par A revient à :

1. Vérifier que Q est satisfaite pour tout élément de $triv(A)$ (faisable car cet ensemble est fini et généralement de petite taille) ;
2. Supposer que x satisfait Q, et démontrer que $f(x)$ satisfait également Q.

Cette méthode de démonstration s'appelle *démonstration par induction*. Elle est très utile pour démontrer des propriétés très complexes. Une de ses applications est la suivante : considérons l'ensemble des entiers naturels $\mathbb{N}$ défini précédemment (par induction), on cherche à vérifier la satisfaction d'une propriété Q sur les nombres entiers. On procède alors comme suit :

1. Vérifier que Q est satisfaite pour 0 ;
2. Supposer que Q est satisfaite pour un entier n et démontrer que Q est satisfaite pour $n + 1$.

On peut facilement s'apercevoir que le dernier raisonnement n'est autre que le raisonnement par récurrence. On vient, en fait, de montrer que ce raisonnement n'est qu'un cas particulier du raisonnement par induction.

1..3 Opérations sur les ensembles

On peut définir plusieurs opérations sur les ensembles. Certaines peuvent être définies pour toutes sortes de définitions d'ensembles, mais ce n'est malheureusement le cas dans tous les cas.

Inclusion et égalité

Soient A et B deux ensembles. On dit que A est inclus ou égal à B (et on note $A \subseteq B$) si $\forall x : x \in A \rightarrow x \in B$ (on dit que A est un sous-ensemble de B). On peut voir tout de suite que pour tout ensemble A, on a : $\emptyset \subseteq A$.

Si on utilise la définition par compréhension, c'est-à-dire $A = \{x|P(x)\}$ et $B = \{x|Q(x)\}$, alors $A \subseteq B$ tient si et seulement si $P \rightarrow Q$.

On parle d'égalité ($A = B$) lorsque $A \subseteq B$ et $B \subseteq A$. En compréhension, cela revient à montrer que : $P \leftrightarrow Q$.

Opérations binaires

Soit Ω un ensemble que l'on appellera l'ensemble référence. Soient A et B deux sous-ensembles quelconques de Ω définis par compréhension comme suit $A = \{x|P(x)\}$ et $B = \{x|Q(x)\}$, et définis par induction comme suit : $A = \{triv(A); x \rightarrow f(x)\}$ et $B = \{triv(B); x \rightarrow g(x)\}$. On définit alors les opérations suivantes :

- **Union :** notée $A \cup B$, elle comporte tout élément appartenant à A *ou* B, en d'autres termes, $A \cup B = \{x|x \in A \vee x \in B\}$. L'union est définie par compréhension comme suit : $A \cup B = \{x|P(x) \vee Q(x)\}$. Elle peut également être définie par induction comme suit : $A \cup B = \{triv(A) \cup triv(B); x \rightarrow f(x), x \rightarrow g(x)\}$.

- **Intersection :** notée $A \cap B$, elle comporte tout élément appartenant à A *et* B ; en d'autres termes, $A \cap B = \{x|x \in A \wedge x \in B\}$. L'intersection est définie par compréhension comme suit : $A \cup B = \{x|P(x) \wedge Q(x)\}$. Cependant, dans le cas général, on ne peut pas donner une définition inductive à l'intersection.

- **Différence :** notée $A - B$, elle comporte tout élément appartenant à A *et* qui n'appartient pas à B ; en d'autres termes, $A - B = \{x|x \in A \wedge x \notin B\}$. La différence est définie par compréhension comme suit : $A - B = \{x|P(x) \wedge \neg Q(x)\}$. Cependant, dans le cas général, on ne peut pas donner une définition inductive à la différence.

- **Complément :** notée $\overline{A} = \Omega - A$; en d'autres termes, $\overline{A} = \{x|x \in \Omega \wedge x \notin A\}$. Le complément est défini par compréhension comme suit : $\overline{A} = \{x|\neg P(x)\}$. Cependant, dans le cas général, on ne peut pas donner une définition inductive au complément.

- **Produit cartésien :** noté $A \times B$, c'est l'ensemble des paires (a, b) telles que $a \in A$ et $b \in B$; en d'autres termes, $A \times B = \{(x, y)|x \in A \wedge x \in B\}$. Le produit cartésien est défini par compréhension comme suit : $A \times B = \{(x, y)|P(x) \wedge Q(y)\}$. La définition par induction se fait comme suit : $A \times B = \{triv(A) \times triv(B); (x, y) \rightarrow (f(x), g(y))\}$.

Autres opérations

- **Cardinalité :** notée $card(A)$, c'est le nombre d'éléments de A. On a : $card(A \times B) = card(A)card(B)$.
- **L'ensemble des parties de** A **:** notée 2^A, c'est l'ensemble de tous les sous-ensembles de A. On a : $card(2^A) = 2^{card(A)}$.

Exemple 1.1 : calculs des opérations ensemblistes

$A = \{a, b\}$,$B = \{a, c\}$,$\Omega = \{a, b, c\}$:

- $A \cap B = \{a\}$;
- $A \cup B = \{a, b, c\}$;
- $A - B = \{b\}$;
- $\overline{A} = \{c\}$;
- $A \times B = \{(a, a), (a, b), (b, a), (b, c)\}$;
- $2^A = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

2. Théorie des langages

Le langage a pour vocation d'assurer la communication entre différentes entités. Bien qu'il soit d'usage de voir le langage comme une caractéristique humaine, ou un objet de prédilection de l'intelligence de l'homme, la notion de langage est bien plus universelle.

La notion de langage est étroitement liée à la notion de codage. Lorsque deux entités, ne partageant pas une mémoire commune, souhaitent échanger des informations, il y a besoin de coder ces informations pour les transmettre via un support (l'air, fil électrique, onde électromagnétique, substance chimique, etc). Le scénario de base est alors le suivant :

— L'entité A souhaite envoyer une information à l'entité B.

— L'entité A code son message selon des règles connues par A et B.

— Le message codé est envoyé via un médium.

— L'entité B reçoit le message et procède à son décodage en appliquant des règles connues par A et B.

— L'entité B récupère l'information envoyée par A.

Une condition nécessaire au fonctionnement de ce schéma est l'utilisation de règles de codage et décodage communes et connues par A et B. Imaginons, par exemple, que l'entité A ne parle que l'arabe envoie un message à la personne B ne comprenant que l'anglais : il est impossible d'envoyer l'information dans ce cas.

Étant un domaine de l'informatique théorique, la théorie des langages couvre plutôt les langages formels : les langages construits selon des modèles mathématiques stricts et qui servent généralement à communiquer avec une machine. Il est important ici de faire la différence entre un langage formel et un langage naturel (utilisé par les humains). Ce dernier, bien qu'ayant des bases mathématiques comme celui des langages formels, possède la caractéristique d'être ambigu et d'utiliser la même *construction syntaxique* pour signifier des choses différentes. Son étude relève généralement du domaine de l'intelligence artificielle et ne sera pas couverte par ce cours.

Afin de formaliser les langages étudiés, la théorie des langages introduit un certain nombre de concepts nécessaires à l'étude d'un langage. Ces concepts partent de la notion du symbole pour arriver au langage et grammaire tout en passant par la notion du mot (qui est la brique élémentaire qui construit un langage).

Une question fondamentale nous servira de points de départ pour donner diverses définitions aux langages : comment répondre à la question "est-ce qu'un mot appartient à un langage ?". On définira trois points de vue pour répondre à cette question :

— Définition ensembliste : c'est une définition basique pour un langage. Elle permet de définir des langages et de raisonner dessus (généralement par une définition par compréhension).

— Définition par grammaire : c'est expliquer comment construire les éléments d'un langage. Il s'agit de la notion clé pour développer des analyseurs de programmes notamment les compilateurs.

— Définition par automate : un automate est tout simplement la machine (ou l'algorithme) qui répond à la question posée par oui ou non. Son étude est cruciale car elle permet de dire, d'abord, si l'on peut répondre à la question et, le cas échéant, évaluer le temps nécessaire pour le faire.

Disposer de ces trois points de vue n'est pas un luxe intellectuel, car chacun permet de répondre à la question posée de manière complémentaire à ce que font les autres points. Par exemple, pour les langages de programmation usuels, il est d'usage que l'algorithme d'analyse (compilateur) soit

construit à partir de la grammaire du langage. L'algorithme construit sera par la suite modifié pour faire davantage d'analyses.

Dans ce qui suit, nous allons d'abord passer en revue plusieurs définitions nécessaires à la compréhension du concept de langage.

2..1 Notions sur les mots

Définition 1.2 : Un symbole est une entité abstraite. Les lettres et les chiffres sont des exemples de symboles utilisés fréquemment, mais des symboles graphiques, des sons ou tous types de signaux peuvent également être employés (idéogrammes, signal lumineux, fumée, signal électrique,...).

Définition 1.3 : Un alphabet est un ensemble de symboles. Il est également appelé le vocabulaire.

Définition 1.4 : Un mot (ou bien une chaîne) défini sur un alphabet A est une séquence finie de symboles juxtaposés de A.

Exemple 1.2 : des mots avec leurs alphabets

— Le mot 1011 est défini sur l'alphabet $\{0, 1\}$;

— Le mot 1.23 est défini sur l'alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$.

2..2 Longueur d'un mot

Si w est un mot, alors sa longueur est définie comme étant le nombre de symboles contenus dans w, elle est noté par $|w|$. Par exemple, $|abc| = 3, |aabba| = 5$.

En particulier, on note le mot dont la longueur est nulle par ε : $|\varepsilon| = 0$. Pour bien comprendre ε, on peut faire l'analogie entre les mots et chaînes de caractères en langage C. Un mot n'est autre qu'une chaîne de caractères, la chaîne "abc" représente tout simplement le mot abc. La chaîne vide "" (un tableau dont la première case contient $\backslash 0$) représente alors le mot ε.

On définit également la longueur d'un mot w par rapport à un symbole $a \in A$ ($|w|_a$) comme étant le nombre d'occurrences de a dans w. Par exemple, $|abc|_a = 1, |aabba|_b = 2$.

Remarque : Dans ce cours, on suppose que tous les mots considérés sont de taille finie. Il est possible aussi de considérer des mots de longueur infinie mais leur étude dépasse le cadre de ce cours.

2..3 Concaténation des mots

Soient w_1 et w_2 deux mots définis sur l'alphabet A. La concaténation de w_1 et w_2 est un mot w défini sur le même alphabet. w est obtenu en écrivant w_1 suivi de w_2, en d'autres termes, on colle le mot w_2 à la fin du mot w_1 :

$$w_1 = a_1...a_n,\ w_2 = b_1b_2...b_m$$
$$w = a_1...a_nb_1b_2...b_m$$

La concaténation est notée par le point, mais il peut être omis s'il n'y a pas d'ambiguïté. On écrira alors : $w = w_1.w_2 = w_1w_2$.

Propriétés de la concaténation

Soient w, w_1 et w_2 trois mots définis sur l'alphabet A :

— La concaténation n'est généralement pas commutative ;

— $(w_1.w_2).w_3 = w_1.(w_2.w_3)$ (la concaténation est associative) ;

— $w.\varepsilon = \varepsilon.w = w$ (ε est un élément neutre pour la concaténation) ;

— $|w_1.w_2| = |w_1| + |w_2|$;

— $\forall a \in A : |w_1.w_2|_a = |w_1|_a + |w_2|_a$;

Remarque : L'ensemble des mots muni de l'opération concaténation forme ce qu'on appelle *monoïde*. Cette structure algébrique possède des propriétés pouvant être utilisées dans l'analyse des langages (on parle, par exemple, de reconnaissance par morphisme sur un monoïde).

L'exposant

L'opération $w.w$ est notée par w^2. En généralisant, on note $w^n = \underbrace{w...w}_{n\ fois}$. En particulier, l'exposant 0 fait tomber sur ε : $w^0 = \varepsilon$ (le mot w est répété 0 fois).

Le mot miroir

Soit $w = a_1a_2...a_n$ un mot sur A ($a_i \in A$). On appelle mot miroir de w, et on le note par w^R, le mot obtenu en écrivant w à l'envers, c'est-à-dire que $w^R = a_n...a_2a_1$. Il est donc facile de voir que $(w^R)^R = w$ ainsi que $(u.v)^R = v^R.u^R$.

Certains mots, appelés palindromes, sont égaux à leurs miroirs. En d'autres termes, on lit la même chose dans les deux directions. Si l'on considère l'alphabet X, l'ensemble des palindromes sont les mots de la forme ww^R ou waw^R tel que a est un seul symbole.

Préfixe et suffixe

Soit w un mot défini sur un alphabet A. Un mot x (resp. y) formé sur A est un préfixe (resp. suffixe) de w s'il existe un mot u formé sur A (resp. v formé sur A) tel que $w = xu$ (resp. $w = vy$). Si $w = a_1a_2...a_n$ alors tous les mots de l'ensemble $\{\varepsilon, a_1, a_1a_2, a_1a_2a_3, ..., a_1a_2...a_n\}$ sont des préfixes de w. De même, tous les mots de l'ensemble $\{\varepsilon, a_n, a_{n-1}a_n, a_{n-2}a_{n-1}a_n, ...,\ a_1a_2...a_n\}$ sont des suffixes de w.

Entrelacement (mélange)

Soient u et v deux mots définis sur un alphabet A tels que $u = u_1u_2...u_n$ et $v = v_1v_2...v_m$. On appelle entrelacement de u et v (on le note par $u \sqcup v$) l'ensemble des mots w qui mélangent les symboles de u de v tout en gardant l'ordre des symboles de u et de v. Formellement, $w = u \sqcup v = w_1w_2...w_{n+m}$ tel que $w_k \in A$ et si $w_k = u_i$ alors $\nexists i', k' : k' > k \wedge i' < i \wedge w_{k'} = u_{i'}$ (la même contrainte s'applique aussi au mot v).
Par exemple, $ab \sqcup ac = \{abac, aabc, aacb, acab\}$ (notons que cette opération est commutative).

En particulier, lorsque v se réduit un seul symbole, l'opération d'entrelacement se résume à l'insertion dudit symbole quelque part dans le mot u. Si a est un symbole, alors $u \sqcup v = \{u_1.a.u_2 | u_1.u_2 = u\}$. Par exemple, $ab \sqcup c = \{cab, acb, abc\}$.

2..4 Notions sur les langages

Définition 1.5 : Un langage est un ensemble (fini ou infini) de mots définis sur un alphabet donné.

Exemple 1.3 : des langages différents

- Langage des nombre binaires définies sur l'alphabet $\{0, 1\}$ (infini) ;
- Langage des mots de longueur 2 défini sur l'alphabet $\{a, b\}$=$\{aa, ab, ba, bb\}$ (fini) ;
- Langage C (quel est le vocabulaire ?) ;

Opérations sur les langages

Soient L, L_1 et L_2 trois langages dont l'alphabet est X, on définit les opérations suivantes :

- **Union :** notée par $+$ ou $|$ plutôt que $\cup$. $L_1 + L_2 = \{w | w \in L_1 \vee w \in L_2\}$;
- **Intersection :** $L_1 \cap L_2 = \{w | w \in L_1 \wedge w \in L_2\}$;
- **Concaténation :** $L_1.L_2 = \{w | \exists u \in L_1, \exists v \in L_2 : w = uv\}$;
- **Exposant :** $L^n = \underbrace{L.L...L}_{n \text{ fois}} = \{w | \exists u_1, u_2, ...u_n \in L : w = u_1u_2...u_n\}$;
- **Fermeture transitive de Kleene :** notée $L^* = \sum_{i \geqslant 0} L^i$. En particulier, si $L = X$ on obtient X^* c'est-à-dire l'ensemble de tous les mots possibles sur l'alphabet X. On peut ainsi définir un langage comme étant un sous-ensemble quelconque de X^*. Une autre définition possible de L^* est la suivante : $L^* = \{w | \exists n \geqslant 0, \exists u_1 \in L, \exists u_2 \in L, ..., \exists u_n \in L, w = u_1u_2...u_n\}$;
- **Fermeture non transitive :** $L^+ = \sum_{i > 0} L^i$;
- **Le langage miroir :** $L^R = \{w | \exists u \in L : w = u^R\}$;
- **Le mélange ou l'entrelacement de langages :** $L \sqcup L' = \{u \sqcup v | u \in L, v \in L'\}$. En particulier, lorsque le langage L' se réduit à un seul mot composé d'un seul symbole a, on a : $L \sqcup a = \{u.a.v | (u.v) \in L\}$

Propriétés des opérations sur les langages

Soient L, L_1, L_2, L_3 quatre langages définis sur l'alphabet A :

— $L^* = L^+ + \{\varepsilon\}$;
— $L_1.(L_2.L_3) = (L_1.L_2).L_3$;
— $L_1.(L_2 + L_3) = (L_1.L_2) + (L_1.L_3)$;
— $L.L \neq L$;
— $L_1.(L_2 \cap L_3) \subseteq (L_1 \cap L_2).(L_1 \cap L_3)$;
— $L_1.L_2 \neq L_2.L_1$.
— $(L^*)^* = L^*$;
— $L^*.L^* = L^*$;
— $L_1.(L_2.L_1)^* = (L_1.L_2)^*.L_1$;
— $(L_1 + L_2)^* = (L_1^* L_2^*)^*$;
— $L_1^* + L_2^* \neq (L_1 + L_2)^*$

3. Les grammaires (systèmes générateurs de langages)

Afin de bien comprendre l'intérét des différents points de vue de définition des langages, on considère la situation suivante : une personne A veut apprendre à une personne B la langue française. Plusieurs approches sont possibles mais une des plus simples (et des plus naïves aussi) consiste à apprendre toutes les phrases en français. Si une telle prouesse est possible, on peut dire que la personne B connaît le français (même si le contre-exemple de la pièce chinoise nous indique que cela n'est pas forcément le cas). Malheureusement, dans bien des langages (y compris le français), le nombre de phrases (ou mots) et très important à retenir, voire infini. Cette méthode n'est pas alors efficace.

Une autre manière pour faire apprendre le français serait d'expliquer à la personne B comment construire des phrases en français, c'est ce qu'on appelle couramment les règles de grammaire (qui ne sont pas propres au français bien sûr). Au lieu d'apprendre toutes les phrases, il suffit d'expliquer à la personne B qu'une phrase simple en français est composée d'un sujet, d'un verbe et d'un complément d'objet direct (il faut bien entendu expliquer ces notions aussi). L'avantage de la grammaire en français est que le nombre de règles à retenir n'est pas important mais ces règles permettent de construire un grand nombre de phrases, d'où l'intérêt des grammaires pour définir les langages.

Les grammaires possèdent un autre intérêt capital : elles permettent de classer les langages en fonction de leur complexité. Grâce à la classification de Chomsky (voir plus bas), il est possible de classer les langages dans l'une des quatre classes allant de la classe la plus simple (classe 3) à la classe la plus complexe (classe 0).

3..1 Exemple introductif formalisé

Pour analyser une classe de phrases simples en français, on peut supposer qu'une phrase est composée de la manière suivante :

— PHRASE → ARTICLE SUJET VERBE COMPLEMENT

— SUJET → "garçon" ou "fille"

— VERBE → "voit" ou "mange" ou "porte"

— COMPLEMENT → ARTICLE NOM ADJECTIF

— ARTICLE → "un" ou "le"

— NOM → "livre" ou "plat" ou "wagon"

— ADJECTIF → "bleu" ou "rouge" ou "vert"

En effectuant des substitutions (on remplace les parties gauches par les parties droites) on arrive à générer les deux phrases suivantes :

Le garçon voit un livre rouge
Une fille mange le plat vert

On dit ici que PHRASE, ARTICLE, SUJET sont des concepts du langage ou encore des symboles non-terminaux (car ils ne figurent pas dans la phrase auxquelles on s'intéresse). Les symboles "garçon", "fille", "voit", "mange", etc sont des terminaux puisqu'ils figurent dans le langage final (les phrases).

Le processus de génération de la phrase à partir de ces règles est appelé *dérivation*. Voici comment est dérivée la première phrase :

- PHRASE → ARTICLE SUJET VERBE COMPLEMENT
→ le SUJET VERBE COMPLEMENT
→ le garçon VERBE COMPLEMENT
→ le garçon voit COMPLEMENT
→ le garçon voit ARTICLE NOM ADJECTIF
→ le garçon voit un NOM ADJECTIF
→ le garçon voit un livre ADJECTIF
→ le garçon voit un livre rouge

Pour la deuxième phrase, la dérivation est la suivante :

- PHRASE → ARTICLE SUJET VERBE COMPLEMENT
→ une SUJET VERBE COMPLEMENT
→ une fille VERBE COMPLEMENT
→ une fille mange COMPLEMENT
→ une fille mange ARTICLE NOM ADJECTIF
→ une fille mange le NOM ADJECTIF
→ une fille mange le plat ADJECTIF
→ une fille mange le plat vert

3..2 Formalisation

Définition 1.6 : On appelle grammaire formelle le quadruplet (V, N, X, R) tel que :

- V est un ensemble fini de symboles dits terminaux, on l'appelle également vocabulaire terminal ;
- N est un ensemble fini (disjoint de V) de symboles que l'on appelle non-terminaux ou encore concepts ;
- S est un non-terminal particulier appelé axiome (point de départ de la dérivation) ;
- R est un ensemble de règles de production de la forme $g \to d$ tel que $g \in (V + N)^+$ et $d \in (V + N)^*$.

Les règles de la forme $\varepsilon \to \alpha$ sont interdites. Pourquoi ?

Par convention, on utilisera les lettres majuscules pour les non-terminaux, et les lettres minuscules pour représenter les terminaux. Soit une suite de dérivations : $w_1 \to w_2 \to w_3 \to ... \to w_n$, alors on écrira : $w_1 \xrightarrow{*} w_n$. On dit alors qu'il y a une séquence de dérivation qui mène de w_1 vers w_n.

Définition 1.7 : Soit une grammaire $G = (V, N, S, R)$. On dit que le mot u appartenant à V^* est dérivé (ou bien généré) à partir de G s'il existe une suite de dérivations qui, partant de l'axiome S, permettent d'obtenir u : $S \xrightarrow{*} u$. Le langage de tous les mots générés par la grammaire G est noté $L(G)$. La suite des règles appliquées pour obtenir le mot s'appelle chaîne de dérivation.

Exemple 1.4 : grammaire générant le langage $\{a\}^*$

Soit la grammaire $G = (\{a\}, \{S\}, S, \{S \to aS|\varepsilon\})$ qui génère le langage $\{\varepsilon, a, aa, aaa\ , ...\} = \{a\}^*$ selon la figure 1.1. Les boîtes en bleu représentent des mots "non-terminaux" (ce ne sont pas des mots générés par la grammaire car ils comportent au moins un symbole non-terminal), les boîtes en gris représentent des mots générés par la grammaire (car ils ne comportent aucun symbole non-terminal).
La chaîne de dérivation du mot $aaaa$ selon cette grammaire est : $S \to aS \to aaS \to aaaS \to aaaaS \to aaaa$. On peut construire l'ensemble des mots possibles selon l'arbre de la figure 1.1.

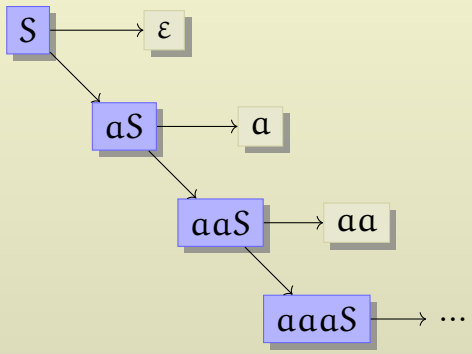


FIGURE 1.1 – Les dérivations possibles (attention, ceci n'est un pas arbre de dérivation)

Définition 1.8 : Étant donnée une grammaire $G = (V, N, S, R)$, les arbres de syntaxe de G sont des arbres où les nœuds internes sont étiquetés par des symboles de N, et les feuilles étiquetées par des symboles de V, tels que si le nœud p apparaît dans l'arbre et si la règle $p \to a_1...a_n$ (a_i terminal

ou non-terminal) est utilisée dans la dérivation, alors le nœud p possède n fils correspondant aux symboles a_i.

Si l'arbre syntaxique a comme racine l'axiome de la grammaire, alors il est dit arbre de dérivation du mot u tel que u est le mot obtenu en prenant les feuilles de l'arbre dans le sens gauche$\rightarrow$ droite et bas $\rightarrow$ haut.

Remarque : Attention : la figure 1.1 n'est un pas un arbre de dérivation !

3..3 Classification de Chomsky

Chomsky a établi une classification hiérarchique qui permet de classer les grammaires en quatre catégories. La classe d'une grammaire est déterminée grâce à la forme de ses règles de production. À chacune des catégories, on associe un type de machine minimale permettant d'analyser les langages générés par ces grammaires. Une grammaire de type i génère un langage de type j tel que $j \geqslant i$.

Soit $G = (V, N, S, R)$ une grammaire, les classes de grammaires de Chomsky sont :

— Type 3 ou grammaire régulière (à *droite*) : toutes les règles de production sont de la forme $g \rightarrow d$ où $g \in N$ et $d = aB$ tel que a appartient à V^* et B appartient à $N + \{\varepsilon\}$;

— Type 2 ou grammaire hors-contexte : toutes les règles de production sont de la forme $g \rightarrow d$ où $g \in N$ et $d \in (V + N)^*$;

— Type 1 ou grammaire contextuelle : toutes les règles sont de la forme $g \rightarrow d$ tel que $g \in (N + V)^+$, $d \in (V + N)^*$ et $|g| \leqslant |d|$. De plus, si ε apparaît à droite alors la partie gauche doit seulement contenir S (l'axiome). On peut aussi trouver la définition suivante des grammaires de type 1 : toutes les règles sont de la forme $\alpha B \beta \rightarrow \alpha \omega \beta$ tel que $\alpha, \beta \in (V + N)^*$, $B \in N$ et $\omega \in (V + N)^*$. **Dans ce cours, on utilise la première définition du type 1**.

— Type 0 : aucune restriction. Toutes les règles sont de la forme : $d \rightarrow g$, $g \in (V+N)^+$, $d \in (V+N)^*$

Le type retenu pour une grammaire est le plus petit qui satisfait les conditions. Il existe une relation d'inclusion entre les types de grammaires selon la figure 1.2.

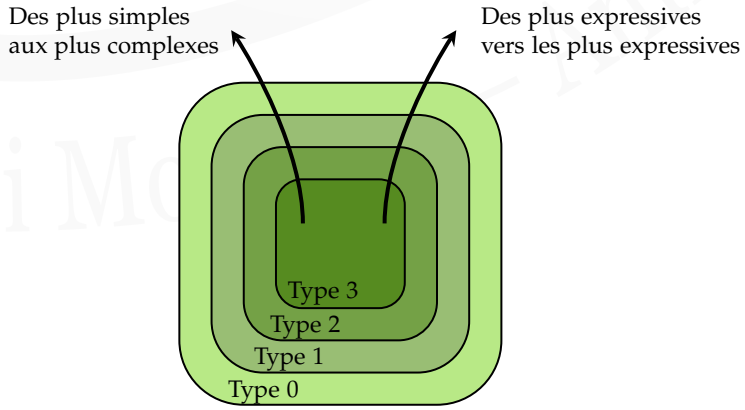


FIGURE 1.2 – La classification hiérarchique de Chomsky pour les grammaires

Exemple 1.5 : des grammaires de quelques langages

— La grammaire $(\{a,b,c\},\{S,T\},S,\{S \to abS|acT, T \to S|\varepsilon\})$ est une grammaire de type 3. On dit également qu'elle est régulière à droite. Le type du langage généré est forcément de type régulier (attention, on ne dit pas langage régulier à droite !). Un exemple d'un mot généré : $S \to abS \to abacT \to abac$ (le langage généré a la forme suivante : $\{((ab)^{p_i}ac)^q | p_i \geqslant 0, q > 0\}$.

— La grammaire $(\{a,b\},\{S,T\},S,\{S \to aSbc|\varepsilon\})$ est une grammaire de type 2 ou encore une grammaire hors-contexte. Pour trouver le type du langage généré, on doit montrer que l'on ne peut pas trouver une grammaire régulière qui génère le même le langage. C'est le cas ici (on l'accepte ici sans démonstration). Le type du langage généré est alors algébrique. Un exemple d'un mot généré est : $S \to aSbc \to aa \to bcbc \to aabcbc$ (le langage généré a la forme suivante : $\{a^n(bc)^n | n \geqslant 0\}$).

— La grammaire $(\{a,"\},\{S,B\},S,\{S \to "a", "a \to "B, Ba \to aaB, B" \to aa"\})$ est une grammaire de type 1 ou une grammaire contextuelle. Elle génère un langage contextuel car on peut montrer que le langage généré ne peut pas être généré par une grammaire hors-contexte. Un exemple d'un mot généré est : $S \to "a" \to "B" \to "aa" \to "Ba" \to "aaB" \to "aaaa"$ (le langage généré a la forme suivante : $\{"a^{2^p}" | p \geqslant 0\}$).

— La grammaire $(\{a\},\{S,T,U\},S,\{S \to UaT|a, T \to TT, aT \to Taa, UT \to U, U \to \varepsilon\})$ est une grammaire de type 0 (sans restriction). Elle génère un langage de type 0 ici. Un exemple de génération de mot est : $S \to UaT \to UaTT \to UaTTT \to UTaaTT \to UTaTaaT \to UTTaaaaT \to UTTaaaTaa \to UTTaaTaaaa \to UTTaTaaaaaa \to UTTTaaaaaaaa \to UTTaaaaaaaa \to UTaaaaaaaa \to Uaaaaaaaa \to aaaaaaaa$ ou encore a^8 (le langage généré a la forme suivante : $\{a^{2^p} | p \geqslant 0\}$).

Remarque : Pour la classe 3 des grammaires, il existe une autre forme possible appelée *grammaire régulière à gauche* (voir le TD). En réalité, tout langage régulier possède une grammaire régulière à droite et aussi une grammaire régulière à gauche (les deux formes sont alors équivalentes et l'on peut passer aisément d'une forme à l'autre).
Notez, cependant, qu'une grammaire régulière est soit à droite soit à gauche, le mélange est interdit. Une grammaire présentant à la fois les formes régulières à gauche et les formes régulières à droite est en réalité une grammaire hors-contexte.

4. Les automates

Les grammaires représentent un moyen qui permet de *décrire* un langage d'une manière que l'on peut qualifier d'inductive. Elles montrent comment les mots du langage sont dérivés.

Pour un langage donné L, on se propose de répondre à la question $w \in L$. On peut répondre à cette question de plusieurs façons. D'abord, on peut vérifier l'existence de w dans la liste des mots de L (impossible à réaliser si le langage est infini).

On peut également chercher une grammaire générant L puis vérifier si cette grammaire génère

w. On cherchera alors une chaîne de dérivation partant de l'axiome de la grammaire vers le mot. En fonction de la classe de grammaire du langage, la recherche de la chaîne peut être efficace ou non ou carrément impossible. Pour les langages de programmation usuels, c'est cette méthode qui est utilisée pour répondre à la question, les grammaires des langages sont alors minutieusement choisies pour que la recherche soit efficace.

Il existe en réalité un troisième moyen permettant de répondre è cette question : les automates. Un automate est une machine qui, après avoir exécuté un certain nombre d'opérations sur le mot, peut répondre à cette question par oui ou non.

Définition 1.9 : Un automate (ou une machine de Turing) est une machine abstraite qui permet de lire un mot et w de répondre à la question : "w appartient-il au langage ?" par oui ou non. Aucune garantie n'est cependant apportée concernant le temps d'analyse ou même la possibilité de le faire.

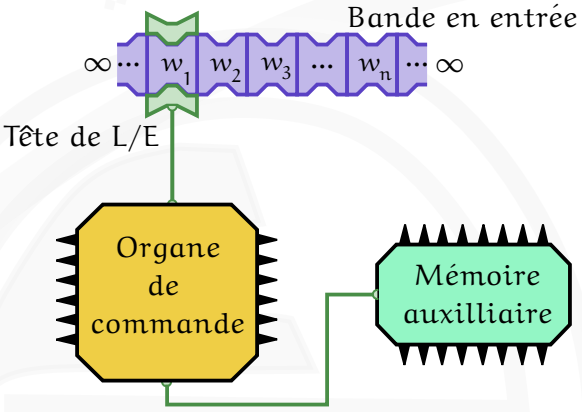


FIGURE 1.3 – Architecture d'une machine de Turing

Une machine de Turing (voir la figure 1.3) est composée de :

— Une bande ou un ruban en entrée finie ou infinie (des deux côtés) sur lequel sera inscrit le mot à lire. Le ruban est composé de cases, chacune stockant un seul symbole à la fois.

— Une tête de lecture/écriture qui peut lire ou écrire dans une seule case du ruban à la fois. Elle peut également se déplacer à droite ou à gauche sur le ruban.

— Un organe de commande qui permet de gérer un ensemble fini de pas d'exécution.

— Éventuellement, une mémoire auxiliaire de stockage.

Formellement, un automate contient au minimum :

— Un alphabet pour les mots en entrée noté X ;

— Un ensemble non vide d'états noté Q ;

— Un état initial noté $q_0 \in Q$;

— Un ensemble non vide d'états finaux $q_f \in Q$;

— Une fonction de transition (permettant de changer d'état) notée δ. Le changement d'état se fait en fonction de l'état en cours de l'automate, du symbole en cours de lecture par la tête et éventuellement du contenu de la mémoire auxiliaire.

4..1 Configuration d'un automate

L'analyse d'un mot par un automate se fait à travers une séquence de configurations.

Définition 1.10 : On appelle configuration d'un automate les valeurs de ses différents composants, à savoir la position de la tête L/E, l'état de l'automate et éventuellement le contenu de la mémoire auxiliaire (lorsqu'elle existe). On peut représenter sommairement la configuration par un triplet (q, a, mem) avec q : l'état de l'automate, a : la position de la tête L/E et mem : le contenu d'une éventuelle mémoire auxiliaire.

Il existe deux configurations spéciales appelées configuration initiale et configuration finale.

Définition 1.11 : La configuration initiale est celle qui correspond à l'état initial q_0 et où la tête de L/E est positionnée sur le premier symbole du mot à lire.

Définition 1.12 : Une configuration finale est celle qui correspond à un des états finaux q_f et où le mot a été entièrement lu.

On dit qu'un mot est **accepté** par un automate si, à partir d'une **configuration initiale**, on arrive à une **configuration finale** à travers une succession de configurations intermédiaires. En d'autres termes, un mot est accepté par une machine de Turing en suivant les étapes suivantes :

1. Inscrire le mot entier sur le ruban (le mot est de longueur finie) ;
2. Positionner la tête de L/E sur le premier symbole du mot ;
3. Mettre la machine à son état intial ;
4. Laisser tourner la fonction de transition jusqu'à arrêt de la machine ;
5. Le mot est accepté si (les deux conditions sont nécessaires) :
   (a) Le mot est entièrement lu (au moins une fois) ;
   (b) La machine se trouve dans un état final.

On dit aussi qu'un langage est accepté par un automate lorsque tous les mots de ce langage (et uniquement les mots de ce langage) sont acceptés par l'automate.

4..2 Classification des automates

Comme les grammaires, les automates peuvent être classés en 4 classes selon la hiérarchie de Chomsky :

— Type 3 ou automate à états finis (AEF) : il accepte les langages de type 3. Sa structure est la suivante :
   — bande en entrée finie ;
   — sens de lecture de gauche à droite ;
   — Pas d'écriture sur la bande et pas de mémoire auxiliaire.

   Un AEF est complètement dépourvue de mémoire.

— Type 2 ou automate à pile : il accepte les langages de type 2. Sa structure est similaire à l'AEF mais dispose en plus d'une mémoire organisée sous forme d'une **seule** pile infinie ;

— Type 1 ou automate à bornes linéaires (ABL) : il accepte les langages de type 1. Sa structure est la suivante :
  — Bande en entrée **finie** accessible en lecture/écriture ;
  — Lecture dans les deux sens ;
  — Pas de mémoire auxiliaire.

— Type 0 ou machine de Turing : il accepte les langages de type 0. Sa structure est la même que l'ABL mais la bande en entrée est infinie.

Le tableau 1.1 résume les différentes classes de grammaires, les langages générés et les types d'automates qui les acceptent :

Grammaire	**Langage**	**Automate**
Type 0	Récursivement énumérable	Machine de Turing
Type 1 ou contextuelle	Contextuel	Automate à borne linéaire
Type 2 ou hors-contexte	Algébrique	Automate à pile
Type 3 ou régulière	Régulier ou rationnel	Automate à états finis

TABLE 1.1 – Les types de grammaires, langages et automates (voir plus bas pour les langages récursivement énumérables)

5. De la décidabilité et de la complexité

5..1 Introduction à la décidabilité

Tous les langages ne se ressemblent pas. Par cela, on entend que le comportement des machines de Turing n'est pas le même pour tous les langages. Certains sont dits plus *complexes* que d'autres car la construction des machines de Turing correspondantes est plus laborieuse ou le temps d'analyse peut être très important. Certains langages ne possèdent même pas de machines de Turing qui les acceptent, on introduit alors la définition d'un langage "récursivement énumérable".

Définition 1.13 : Un langage L défini sur un alphabet X est dit *récursivement énumérable* si l'on peut trouver une machine de Turing qui peut décider de l'appartenance (c'est-à-dire lire tout le mot et s'arrêter avec un état final) de tout mot de L moyennant un nombre fini de configurations (la machine donne la réponse oui).

Les langages qui ne sont pas récursivement énumérables existent en réalité. Ce sont des langages dont l'analyse de certains mots ne peut jamais aboutir (la machine tourne à l'infini). Un exemple

simple de ces langages est l'ensemble des programmes en langages C qui se terminent. En effet, on peut démontrer qu'il est impossible de construire une machine de Turing qui peut décider, en un temps fini, si un programme quelconque écrit en langage C peut se terminer pour une entrée donnée (problème appelé *problème d'arrêt*).

Un langage récursivement énumérable ne garantit donc que la procédure d'analyse d'un mot qui ne lui appartient pas puisse se terminer. Ceci nous amène alors à une nouvelle définition.

Définition 1.14 : Un langage L défini sur un alphabet X est dit *récursif* si l'on peut trouver une machine de Turing qui peut décider de l'appartenance ou non d'un mot de X^* moyennant un nombre fini de configurations.

Pour un langage récursif, on dit que la procédure de son analyse est *décidable*. L'analyse d'un langage récursif se termine alors toujours quel que soit son résultat final. Étant donné que l'objectif final de la construction des automates est la construction d'un programme équivalent qui analyse des mots, il est tout à fait évident de s'intéresser uniquement à la catégorie des langages récursifs ici.

On doit noter ici que la définition utilise l'expression "un nombre fini de configurations". Cette définition est plus intéressante par rapport à une autre qui se base sur le temps d'exécution ou tout autre critère. Ceci permet en effet d'annoncer les résultats suivants :

— Le temps d'analyse des mots d'un langage récursif est fini.

— La taille de la mémoire utilisée pour analyser un mot d'un langage récursif est finie.

— La taille du ruban utilisé pour analyser un mot d'un langage récursif est finie.

5..2 Notions sur la complexité

Lorsqu'un langage est récursif, cela ne signifie pas forcément qu'il soit intéressant. En effet, si l'on s'intéresse au temps par exemple, l'analyse d'un mot, bien qu'elle dure un temps fini, peut prendre un temps considérable pouvant aller à des milliards de siècles sur le plus rapide des ordinateurs. De même, l'analyse d'un mot peut consommer un espace mémoire (ruban+mémoire auxiliaire) qui, bien que fini, peut aller à des bornes défiant toute technologie existante.

On voudrait dire par cette introduction que l'analyse d'un mot par une machine requiert généralement des ressources (temps, espace, énergie, bande passante d'un réseau, etc). Les langages récursifs utilisent un nombre fini de ressources, mais cela ne signifie pas qu'ils consomment peu de ressources.

Lors de l'analyse d'un mot d'une certaine taille, on appelle **complexité** le nombre de ressources mobilisées par l'analyse. Évidemment, vu la limitation des ressources (qui coûtent de l'argent en fin de compte), il est tout à fait justifié de penser à construire les machines qui en consomment le moins.

Traditionnellement, on définit la complexité comme étant la relation qui existe entre la taille du mot à analyser (n) et le nombre de ressources requises. Il est également d'usage de s'intéresser à deux types de ressources plus que d'autres :

— Temps : c'est-à-dire le temps nécessaire pour analyser le mot. On appelle cela la complexité en temps.

— Espace : c'est-à-dire la taille du ruban et la mémoire auxiliaire nécessaires pour analyser le mot. On appelle cela la complexité en espace.

On peut se baser sur un critère qui permet de représenter plusieurs types de complexité. En effet, si l'on s'intéresse au nombre de configurations nécessaires pour analyser un mot (en acceptation ou

en rejet), il est possible de quantifier :

— La complexité en temps : le passage d'une configuration à une autre se fait généralement en un temps fixe a. Si le nombre de configurations est $f(n)$ alors le temps d'analyse sera $a.(f(n)-1)$.

— La complexité en espace : en considérant la suite des configurations, il est possible, non sans difficulté, de quantifier le nombre de cases utilisées sur le ruban ainsi que la taille de la mémoire utilisée. Dans tous les cas, ce nombre est fonction de $f(n)$.

Dans ce qui suit, on s'intéressera alors au nombre de configurations possibles pour un mot donné comme étant le critère de complexité. Cette dernière peut aller de la plus simple des quantités (une seule configuration : $f(n) = 1$) à une fonction exponentielle ($f(n) = a^n$) ce qui rend la machine tout à fait inexploitable.

Il est également à noter que vu la possibilité de construire plusieurs machines de Turing acceptant le même langage, on peut alors associer plusieurs complexités à un langage donné. Parmi toutes les machines existantes, on dira que la machine la moins complexe (celle avec le plus faible $f(n)$ en moyenne) représente la **complexité du langage**.

Une nouvelle fois, la classification de Chomsky nous aide à définir la complexité de certains types de langages. Par exemple, tous les langages réguliers ont une complexité dite linéaire, c'est-à-dire que $f(n)$ est donné par $a \times n$. Ce n'est pas le cas pour les autres classes. En effet, seule une sous-partie des langages algébriques possède une complexité linéaire. Par la suite, on appellera cette classe : les langages algébriques linéaires, c'est d'ailleurs la classe à laquelle appartiennent les langages de programmation usuels. Enfin, en général, la catégorisation des langages contextuels et des langages récursifs par rapport à la complexité est beaucoup plus difficile.

6. Exercices de TD

Exercice 1

Déterminez l'alphabet de chacun des langages suivants :

- Les nombres binaires ;
- Les nombres entiers éventuellement munis d'un signe ;
- Les nombres réels en C ;
- Les identifiants en C ;
- Le langage C ;

Exercice 2

Trouvez les langages correspondant aux définitions suivantes :

- Tous les mots sur $\{a, b, c\}$ de longueur 2 ne contenant pas un c ;
- Tous les mots sur $\{a, b\}$ contenant au maximum deux a ou bien un b ;
- Tous les mots sur $\{a, b\}$ contenant plus de a que de b ;
- Le langage L défini comme suit : $\varepsilon \in L$, si $u \in L$ alors $auab \in L$

Exercice 3

- Calculez $\varepsilon \sqcup\!\sqcup\, a$, $abca \sqcup\!\sqcup\, d$, $abca \sqcup\!\sqcup\, a$, $a^n \sqcup\!\sqcup\, b$.
- Calculez $\{a^n | n \geqslant 0\} \sqcup\!\sqcup\, a$, $\{a^n b^n | n \geqslant 0\} \sqcup\!\sqcup\, a$.

Exercice 4

On note par $Pref(L)$ l'ensemble suivant : $\{u | \exists w \in L : u \text{ est préfixe de } w\}$. Calculez $Pref(L)$ dans chacun des cas suivants : $L = \{ab, abc, \varepsilon\}$, $L = \{a^m b^n | m, n \geqslant 0\}$, $L = \{a^n b^n | n \geqslant 0\}$.
On note par $Suf(L)$ l'ensemble suivant : $\{u | \exists w \in L : u \text{ est suffixe de } w\}$. Calculez $Suf(L)$ pour les langages précédents.

Exercice 5

Définissez la fermeture de Kleene (L^*) pour chacun des langages suivants :

- $L = \{\varepsilon\}$;
- $L = \{a\}$;
- $L = \{a, ab\}$;
- $L = \{aa, ab, ba, bb\}$;

Exercice 6

Soit X un alphabet, trouvez les mots $w \in X^*$ qui vérifient :

— $w^2 = w^3$;

— $\exists v \in X^* : w^3 = v^2$;

Exercice 7

Donnez, sans démonstration, les langages générés par les grammaires suivantes. Dites, à chaque fois, de quel type s'agit-il ? (pour trouver la forme des mots, on commence d'abord par générer quelques mots à partir de la grammaire) :

— $G = (\{a\}, \{S\}, S, \{S \to aS|\varepsilon\})$;

— $G = (\{a\}, \{S\}, S, \{S \to aSa|\varepsilon\})$;

— $G = (\{a, b\}, \{S\}, S, \{S \to aSa|bSb|\varepsilon\})$;

Exercice 8

Précisez le type de chacune des grammaires suivantes ainsi que les types des langages qui en dérivent :

— $G = (\{a, b\}, \{S, T\}, S, \{S \to aabS|aT, T \to bS|\varepsilon\})$;

— $G = (\{a, b, c\}, \{S, T, U\}, S, \{S \to bSTa|aTb, T \to abS|cU, U \to S|\varepsilon\})$;

— $G = (\{x, +, *\}, \{S\}, S, \{S \to S + S|S * S|x\})$;

— $G = (\{0, 1, 2\}, \{S, T, C, Z, U\}, S, \{S \to TZ, T \to 0U1, T \to 01, U \to 0U1C|01C, C1 \to 1C, CZ \to Z2, 1Z \to 12\})$

— $G = (\{0, 1, 2\}, \{S, C, Z, T\}, S, \{S \to TZ, T \to 0T1C|\varepsilon, C1 \to 1C, CZ \to Z2, 1Z \to 1\})$;

— $G = (\{a, b, c\}, \{S, T\}, S, \{S \to Ta|Sa, T \to Tb|Sb|\varepsilon\})$

Exercice 9

Donnez les grammaires qui génèrent les langages suivants :

— Les nombres binaires ;

— Les mots sur $\{a, b\}$ qui contiennent le facteur a

Exercice 10

— Montrez par induction que pour tout $n \in \mathbb{N}$, les entiers s'écrivant de la forme $1.0^{2n}.1$ est multiple de 11 (attention il s'agit de la forme des entiers).

— Montrez par induction que tout entier palindrome de longueur paire est un multiple de 11.

Exercice 11

Soient G et G′ deux grammaires qui génèrent respectivement les langages L et L′. Donnez les construction qui permettent de trouver les grammaires de :

— L.L′ ;

— $L + L'$;

— L^* ;

Appliquez vos constructions proposées pour déduire la grammaire du langage $(\{a^m b^n | m, n \geqslant 0\} + \{c^m | m \geqslant 0\})^*$.

Chapitre 2

Les automates à états finis (AEF)

Les AEF sont les plus simples des machines d'analyse de langages car ils ne comportent pas de mémoire. Par conséquent, les langages acceptés par ce type d'automates sont les plus simples des quatre classes de Chomsky, à savoir les langages réguliers (type 3). Par ailleurs, les automates à états finis peuvent être utilisés pour modéliser plusieurs problèmes dont la solution n'est pas très évidente. La série de TD propose quelques exercices dans ce sens.

1. Généralités sur les AEF

Définition 2.1 : Un automate à états finis est machine abstraite définie par le quintuplet (X, Q, q_0, F, δ) tel que :

- X est l'ensemble des symboles formant les mots en entrée (l'alphabet des mots à analyser) ;
- Q est l'ensemble des états possibles ;
- q_0 est l'état initial ($q_0 \in Q$) ;
- F est l'ensemble des états finaux ($F \subseteq Q$) ou encore les états d'acceptation ;
- δ est une fonction de transition qui permet de passer d'un état à un autre selon l'entrée en cours :

  $\delta : Q \times (X \cup \{\varepsilon\}) \mapsto 2^Q$
  $\delta(q_i, a) = \{q_{j_1}, q_{j_2}, ..., q_{j_k}\}$ ou $\emptyset$ ($\emptyset$ signifie que la configuration n'est pas prise en charge ou encore que la transition n'existe pas)

Un mot est accepté par un AEF si, après avoir lu tout le mot, l'automate se trouve dans un état final ($q_f \in F$). En d'autres termes, lorsqu'un AEF n'a plus de transition à exécuter, un mot est rejeté dans deux cas :

- L'automate est dans l'état q_i, l'entrée courante étant a et la transition $\delta(q_i, a)$ n'existe pas (on n'arrive pas à lire tout le mot) ;
- L'automate arrive à lire tout le mot mais l'état de *sortie* n'est pas un état final.

Un AEF A est donc un *séparateur* (ou classifieur) des mots de X^* en deux parties : l'ensemble des mots acceptés par l'automate (notons-le par $L(A)$) et le reste des mots ($X^* - L(A)$).

Exemple 2.1 : un automate à états finis

Soit l'AEF défini par $(\{a, b\}, \{0, 1\}, 0, \{1\}, \delta)$ tel que :

$$\delta(0, a) = \{0\} \quad \delta(0, b) = \{1\}$$
$$\delta(1, a) = \emptyset \quad \delta(1, b) = \{1\}$$

Voici comment se fait l'analyse de différents mots :

— Le mot aab : on a la suite des configurations suivantes : $(0, a) \to (0, a) \to (0, b) \to (1, \varepsilon)$. Notons que ε dans la dernière configuration signifie que l'on est arrivé à la fin du mot. aab est accepté car l'état de sortie 1 est final et le mot a été entièrement lu. Schématiquement, nous pouvons imaginer le fonctionnement de l'automate selon la figure 2.2 :

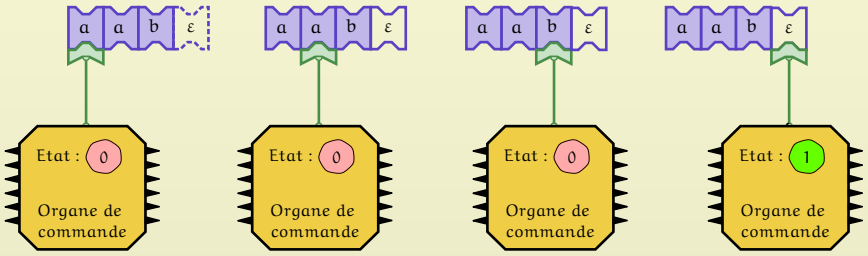


FIGURE 2.1 – Représentation imaginaire de l'analyse du mot aab

— Le mot ε : la seule configuration est $(0, \varepsilon)$. Le mot est rejeté car 0 n'est pas un état final même s'il a été entièrement lu. Schématiquement, nous pouvons imaginer le fonctionnement de l'automate selon la figure 2.2 :

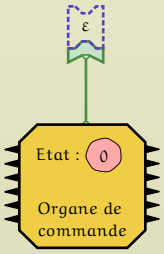


FIGURE 2.2 – Représentation imaginaire de l'analyse du mot ε

— Le mot ba : on a la suite des configurations suivantes : $(0, b) \to (1, a)$. Le mot n'est pas accepté car il n'a pas été entièrement lu (même si l'état de sortie est bien final). Schématiquement, nous pouvons imaginer le fonctionnement de l'automate selon la figure 2.3 :

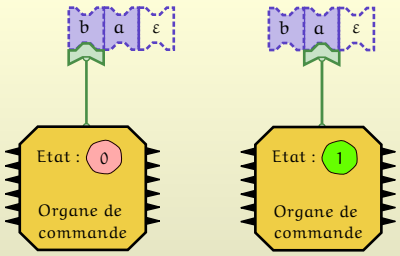


FIGURE 2.3 – Représentation imaginaire de l'analyse du mot ba

On peut facilement voir que le langage accepté par cet automate est $a^m b^n (m \geqslant 0, n > 0)$.

Un AEF peut être représenté de deux manières : soit par une table définissant la fonction de transition soit par un *graphe orienté*.

1..1 Représentation par table

La table possède autant de lignes qu'il y a d'états dans l'automate de telle sorte que chaque ligne corresponde à un état. Les colonnes correspondent aux différents symboles de l'alphabet. Si l'automate est dans l'état i et que le symbole j est le prochain à lire, alors l'entrée (i, j) de la table donne l'état auquel l'automate passera après avoir lu j. Notons que la définition par table n'est pas suffisante pour définir l'AEF entièrement étant donné que la table ne donne ni l'état initial ni les états finaux.

Exemple 2.2 : représentation tabulaire

L'automate précédent est représenté comme suit :

État	a	b
0	0	1
1	-	1

1..2 Représentation graphique

La représentation graphique consiste à représenter l'automate par un graphe orienté. Chaque état de l'automate est schématisé par un rond (ou sommet). Si la transition $\delta(q_i, a) = \{q_j\}$ est définie, alors on raccorde le sommet q_i au sommet q_j par un arc décoré par le symbole a. L'état initial est désigné par une flèche entrante au sommet correspondant tandis que les états finaux sont marqués par un double rond. Le schéma de la figure 2.4 reprend l'automate précédent.

Exemple 2.3 : représentation graphique

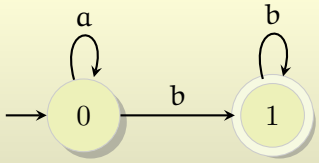


FIGURE 2.4 – L'automate acceptant $a^m b^n (m \geqslant 0, n > 0)$

Lorsqu'il y a plusieurs symboles $a_1, ...a_k$ tels que $(q_i, a_l) = \{q_j\}(l = 1..k)$ alors on se permet de décorer l'arc (q_i, q_j) par l'ensemble $a_1, ..., a_k$.

La définition d'un AEF en tant que graphe a un avantage, car elle permet de définir autrement l'acceptation d'un mot. Mais, on préférera la représentation tabulaire dans le cas où l'on fait des calculs sur les automates.

Définition 2.2 : Soit l'AEF $A = (X, Q, q_0, F, \delta)$. On appelle *chemin* toute séquence d'états $s_0, s_1, s_2, ..., s_n$ tels que $\forall i : s_i \in Q$ et $\forall i, \exists a_i \in X : s_{i+1} \in \delta(s_i, a_i)$. Son mot correspondant est alors : $a_0 a_1 a_2 ... a_{n-1}$.

Dans l'automate précédent, on peut, entre autres, définir les chemins suivants :

— Le chemin $0, 0, 0, 0$ correspondant au mot aaa ;
— Le chemin $0, 0, 0, 1$ correspondant au mot aab ;
— Le chemin $0, 1, 1, 1, 1, 1$ correspondant au mot $abbbb$;
— Le chemin $1, 1, 1, 1$ correspondant au mot bbb.

On peut maintenant donner une deuxième définition à l'acceptation d'un mot.

Définition 2.3 : Soit l'AEF $A = (X, Q, q_0, F, \delta)$. Un mot w est accepté par A s'il existe un chemin de A correspondant à w tel que le premier état est q_0 et le dernier état est final.

Ainsi, dans les mots précédents, le mot aaa est rejeté alors que aab et $abbbb$ sont acceptés. Le mot ba est, à son tour, rejeté car il ne correspond à aucun chemin de l'automate considéré. Le mot bbb est accepté aussi car on peut lui trouver un chemin partant de l'état initial et se terminant par un état final.

2. Les automates et le déterminisme

2..1 Notion de déterminisme

Un programme informatique est, en général, déterministe : c'est-à-dire que l'on connaît avec précision son comportement dans les différentes situations possibles. Ceci rend l'analyse des programmes plus simples. Malheureusement, tous les programmes ne peuvent pas être de la sorte car on peut parfois ignorer les conditions d'exécution (état d'une machine distante par exemple). Il est à noter que l'on préfère toujours une version déterministe d'un programme plutôt qu'une version

non-déterministe (le non-déterminisme peut induire des coûts importants surtout pour l'analyse et la maintenance).

Un AEF, étant une forme spéciale d'un programme informatique, agit similairement. En d'autres mots, étant donné un mot à analyser, on souhaite connaître avec certitude le prochain état de l'automate. Ceci revient à dire que si l'automate est dans l'état q_i et que l'entrée courante est a alors il existe au plus seul un état q_j tel que $\delta(q_i, a) = \{q_j\}$. Le cas échéant, on dit que l'automate est déterministe parce qu'il sait **déterminer** le prochain état à tout moment. Dans le cas inverse, l'automate doit choisir une action et la tester à terme, si l'acceptation n'est pas possible l'automate doit tester les autres éventualités.

Le non-déterminisme peut également provenir des transitions (arcs). En effet, rien n'interdit dans la définition des AEF d'avoir des ε-transitions, c'est-à-dire des transitions décorées avec ε. L'existence d'une ε-transition entre les états q_i et q_j signifie que l'on n'a pas besoin de lire un symbole [1] pour passer de q_i vers q_j (attention ! l'inverse n'est pas possible).

Voyons maintenant une définition formelle de la notion du déterminisme pour les AEF.

Définition 2.4 : Un AEF (X, Q, q_0, F, δ) est dit déterministe si les deux conditions sont vérifiées :

— $\forall q_i \in Q, \forall a \in X$, il existe au plus un état q_j tel que $\delta(q_i, a) = \{q_j\}$;

— L'automate ne comporte pas de ε-transitions.

Exemple 2.4 : automate non-déterministe sans ε-transition

Soit le langage des mots définis sur $\{a, b\}$ possédant le facteur ab. La construction d'un AEF non-déterministe est facile. La table suivante donne la fonction de transition (l'état initial est l'état 0 et l'état 2 est final).

État	a	b
0	0,1	0
1	-	2
2	2	2

La figure 2.5 reprend le même automate :

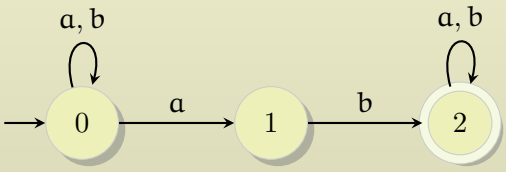


FIGURE 2.5 – L'automate des mots sur $\{a, b\}$ contenant le facteur ab

Exemple 2.5 : automate non-déterministe avec des ε-transitions

L'automate donné par la figure 2.6 accepte le même langage que le précédant mais en utilisant des ε-transitions.

1. Par abus de langage, on dit qu'on lit ε

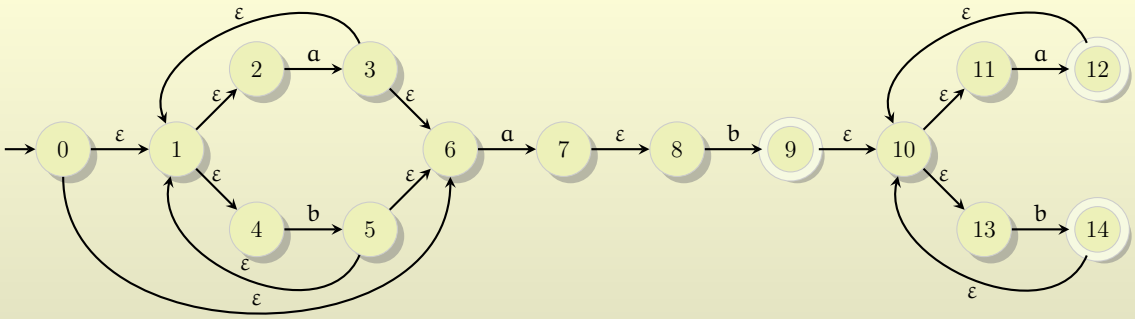


FIGURE 2.6 – L'automate acceptant les mots sur $\{a, b\}$ contenant le facteur ab (avec des ε-transitions)

2..2 Déterminisation d'un automate à états fini

Si on analyse le mot aba par l'AEF de la figure 2.5, on aura la première configuration $(0, a)$ suite à laquelle deux états sont possibles : soit 0 ou 1. À ce stade et vu que l'on ne connaît pas la suite du mot, on est incapable de déterminer quel état permet-il d'accepter le mot. Étant donné que dans un AEF, la tête L/E est censée bouger dans un seul sens, on se trouve dans la situation suivante : chaque fois que l'on n'arrive pas à déterminer quel est le prochain état, on en choisit un arbitrairement puis on continue l'analyse. Si cette dernière permet d'accepter le mot alors le choix était bon, sinon on revient au point de non-déterminisme pour choisir un autre état et continuer l'analyse. Pour le mot aba, cela se passe comme suit :

$$
\begin{array}{lllllll}
(0,a) & \to & (0,b) & \to & (0,a) & \to & (0,\varepsilon)\ \textbf{échec} \\
 & \downarrow & & & & \searrow & (1,\varepsilon)\ \textbf{échec} \\
 & \downarrow & & & & & \\
 & \downarrow & & & & & \\
 & \to & (1,b) & \to & (2,a) & \to & (2,\varepsilon)\ \textbf{succès}
\end{array}
$$

Cet exemple illustre bien le problème des automates non-déterministes. En effet, pour un mot de longueur n, la complexité de l'analyse peut, au pire cas, être $f(n) = a^n$ $(a > 1)$ avant de dire si le mot est accepté ou non. Pour $a = 2$ et un mot de 100 symboles (ce qui n'est pas important), le nombre de configurations peut atteindre l'ordre de 2^{100}, ce qui équivaut à 10^{30} configurations. L'analyse peut alors durer des milliers de siècles même sur l'ordinateur le plus rapide au monde. En revanche, si l'AEF est déterministe, le nombre de configurations est simplement $f(n) = n + 1$ seulement.

Nous pouvons alors déduire, dans un premier temps, que les automates non-déterministes ne sont guère intéressants vu qu'ils peuvent générer un coût très élevé lors de l'analyse, mais ce n'est pas le cas en réalité. Le plus souvent, il se trouve qu'il est beaucoup plus simple de concevoir un AEF non-déterministe que de construire un qui soit déterministe. De plus, comme on le verra dans le prochain chapitre, les langages réguliers sont souvent notés en utilisant *les expressions régulières*. Un algorithme (dit de Thompson) permet alors de construire l'AEF du langage à partir des expressions régulières mais l'AEF est presque toujours non-déterministe avec beaucoup d'ε-transitions.

Heureusement, lorsqu'il s'agit des langages réguliers, un théorème nous sera d'une grande utilité

car il établit l'équivalence entre les automates déterministes et ceux non-déterministes (la démonstration de ce théorème sort du cadre de ce cours).

Théorème 2.1

(dit de Rabin et Scott) Tout langage accepté par un AEF non-déterministe peut également également être accepté par un AEF déterministe.

Une conséquence très importante de ce théorème peut déjà être citée (en réalité, elle découle plutôt de la démonstration de ce théorème) :

Proposition 2.1

Tout AEF non-déterministe peut être transformé en un AEF déterministe.

Ce résultat établit que si l'on veut construire l'automate à états fini déterministe qui accepte les mots d'un certain langage, alors on peut commencer par trouver un AEF non-déterministe (ce qui est plus facile). Il suffit de le transformer, après, pour obtenir un automate à états finis déterministe.

2..3 Déterminisation d'un AEF sans ε-transition

En réalité, l'algorithme de déterminisation d'un AEF est général, c'est-à-dire qu'il fonctionne dans tous les cas (qu'il y ait des ε-transitions ou non). Cependant, il est plus facile de considérer cet algorithme sans les ε-transitions. Dans cette section, on suppose que l'on a un AEF A ne comportant aucune ε-transition.

Le principe de l'algorithme est de considérer des ensembles d'états plutôt que de simples états (dans l'algorithme suivant, chaque ensemble d'états représente un état futur de l'automate déterministe).

Algorithme de déterminisation d'un AEF sans les ε-transitions

1. Partir de l'état initial $E^{(0)} = \{q_0\}$ (c'est l'état initial du nouvel automate)
2. Construire $E^{(1)}$ (considéré comme étant un nouvel état) l'ensemble des états obtenus à partir de $E^{(0)}$ par un symbole a : $E^{(1)} = \bigcup_{q' \in E^{(0)}} \delta(q', a)$
3. Recommencer l'étape 2 pour tous les symboles possibles et pour chaque nouvel ensemble $E^{(i)}$: $E^{(i)} = \bigcup_{q' \in E^{(i-1)}} \delta(q', a)$
4. Tous les ensembles contenant au moins un état final du premier automate sont des états finaux
5. Renuméroter les états en tant qu'états simples

Pour illustrer cet algorithme, nous allons l'appliquer à l'automate donné par la figure 2.5. La table 2.1 donne les étapes d'application de l'algorithme (les états en gras sont des états finaux).

La figure 2.7 donne l'automate obtenu (remarquons qu'il n'est pas optimal). Cet automate n'est pas évident à trouver mais grâce à l'algorithme de déterminisation, on peut le construire automati-

État	a	b
0	0,1	0
0,1	0,1	0,2
0,2	0,1,2	0,2
0,1,2	0,1,2	0,2

$\Longrightarrow$

État	a	b
0′	1′	0′
1′	1′	2′
2′	3′	2′
3′	3′	2′

TABLE 2.1 – Déterminisation d'un AEF sans ε-transition

quement.

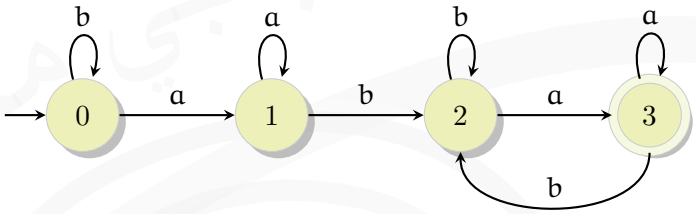


FIGURE 2.7 – L'automate déterministe qui accepte les mots ayant le facteur ab

2..4 Déterminisation avec les ε-transitions

La déterminisation d'un AEF contenant au moins une ε-transition est un peu plus compliquée puisqu'elle fait appel à la notion de l'ε-fermeture d'un ensemble d'états. Nous commençons donc par donner sa définition.

Définition 2.5 : Soit E un ensemble d'états. On appelle ε-fermeture de E l'ensemble des états incluant, en plus de ceux de E, tous les états accessibles depuis les états de E par un chemin correspondant au mot ε.

Algorithme de calcul de ε-fermeture ϕ d'un ensemble E

```
1 ϕ ← E
2 répéter
3 |   pour tout état q de ϕ faire
4 |   |   ϕ ← ϕ ∪ δ(q, ε)
5 jusqu'à ce que l'ensemble ϕ ne change plus;
```

Exemple 2.6 : calcul des ε-fermetures

Considérons l'automate donné par la figure 2.6, calculons un ensemble d'ε-fermetures :

— ε-fermeture($\{0\}$) = $\{0, 1, 2, 4, 6\}$

— ε-fermeture($\{1, 2\}$) = $\{1, 2, 4\}$

— ε-fermeture($\{3\}$) = $\{1, 2, 3, 4, 6\}$

L'algorithme de déterminisation dans le cas général est le suivant (la fonction ε_f représente l'ε-fermeture) :

Algorithme de déterminisation d'un AEF avec des ε-transitions

1 Partir de l'état initial $E^{(0)} = \varepsilon_f(\{q_0\})$ (c'est l'état initial du nouvel automate)

2 Construire $E^{(1)}$ l'ensemble des états obtenus à partir de $E^{(0)}$ par un symbole a : $E^{(1)} = \varepsilon_f\left(\bigcup_{q' \in E^{(0)}} \delta(q', a)\right)$

3 Recommencer l'étape 2 pour tous les symboles possibles et pour chaque nouvel ensemble $E^{(i)}$: $E^{(i)} = \varepsilon_f\left(\bigcup_{q' \in E^{(i-1)}} \delta(q', a)\right)$

4 Tous les ensembles contenant au moins un état final du premier automate sont des états finaux

5 Renuméroter les états en tant qu'états simples

Exemple 2.7 : déterminisation avec des ε-transitions

Appliquons maintenant ce dernier algorithme à l'automate de la figure 2.6.

État	a	b
0,1,2,4,6	1,2,3,4,6,7,8	1,2,4,5,6
1,2,3,4,6,7,8	1,2,3,4,6,7,8	1,2,4,5,6,9,10,11,13
1,2,4,5,6	1,2,3,4,6,7,8	1,2,4,5,6
1,2,4,5,6,9,10,11,13	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,10,11,13,14
1,2,3,4,6,7,8,10,11,12,13	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,9,10,11,13,14
1,2,4,5,6,10,11,13,14	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,10,11,13,14
1,2,4,5,6,9,10,11,13,14	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,10,11,13,14

ce qui produit l'automate suivant (l'état initial est 0, les états finaux sont : 3, 4, 5 et 6) :

État	a	b
0	1	2
1	1	3
2	1	2
3	4	5
4	4	6
5	4	5
6	4	5

3. Minimisation d'un AEF déterministe

Si on prend deux automates déterministes acceptant le même langage, le nombre de configurations nécessaires pour analyser un mot ne dépend que de la taille de ce mot. A priori, on peut dire alors que le nombre d'états d'un automate importe peu pour déterminer le coût d'analyse d'un mot. Mais, ce n'est pas tout à fait correct.

D'abord signalons que l'opération de déterminisation a la fâcheuse tendance de produire beaucoup d'états. Pour un AEF non-déterministe comportant n états, la procédure de déterminisation peut produire jusqu'à $2^n - 1$ états possibles (ce qui est un nombre considérable). Or, stocker et explorer une petite structure mémoire n'est pas comme explorer une grande structure. Pour n assez grand, il se peut que la représentation mémoire d'un AEF nécessite plusieurs pages de mémoires, allant jusqu'à causer des défauts de pages (ce qui augmente le temple d'analyse). Sur un autre plan, dans les systèmes embarqués (où la dimension énergie est des plus capitales), nous avons tout l'intérêt à réduire le nombre de cellules mémoires occupées afin de minimiser la puissance électrique nécessaire pour maintenir les données.

En conclusion, il est tout à fait justifiable de vouloir s'intéresser à la réduction du nombre d'états nécessaires pour accepter un langage. D'ailleurs, on s'intéressera dans ce qui suit au calcul du nombre minimal d'états nécessaires à l'analyse d'un langage donné. Il est à noter, enfin, que si on peut trouver une multitude d'automates déterministes pour analyser le même langage, on ne peut trouver qu'un seul automate déterministe et minimal (avec un nombre minimal d'états) acceptant le même langage.

La minimisation s'effectue en éliminant les états dits inaccessibles et en *confondant* (ou fusionnant) les états acceptant le même langage.

3..1 Les états inaccessibles

Définition 2.6 : Un état est dit inaccessible s'il n'existe aucun chemin permettant de l'atteindre à partir de l'état initial.

D'après la définition, les états inaccessibles sont improductifs (il existe néanmoins des états improductifs qui ne sont pas inaccessibles), c'est-à-dire qu'ils ne participeront jamais à l'acceptation d'un mot. Ainsi, la première étape de minimisation d'un AEF consiste à éliminer ces états. L'étudiant peut, en guise d'exercice, écrire l'algorithme qui permet de trouver les états inaccessibles d'un AEF (indice : pensez à un algorithme de marquage d'un graphe).

3..2 Les états β-équivalents

Définition 2.7 : Deux états q_i et q_j sont dits β-équivalents s'ils permettent d'atteindre les états finaux à travers les mêmes mots. On écrit alors : $q_i \ \beta \ q_j$.

Par le même mot, on entend que l'on lit la même séquence de symboles pour atteindre un état final à partir de q_i et q_j. Par conséquent, ces états acceptent le même langage. La figure 2.8 montre un exemple d'états β-équivalents car l'état q_i atteint les états finaux via les mots a et b, de même pour l'état q_j. L'algorithme de minimisation consiste donc à fusionner simplement ces états pour n'en faire qu'un.

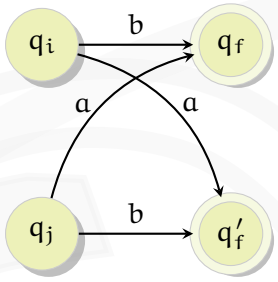


FIGURE 2.8 – Exemple d'états β-équivalents

Remarque : La relation β-équivalence est une relation d'équivalence. De plus, si q_i et q_j sont β-équivalents, alors $\forall x \in X$ (X étant l'alphabet), $\delta(q_i, x)$ et $\delta(q_j, x)$ sont également β-équivalents (puisque q_i et q_j acceptent le même langage). La relation β-équivalence est dite une relation de congruence.

Remarque : Le nombre de classes d'équivalence de la relation β-équivalence est égal au nombre des états de l'automate minimal car les état de chaque classe d'équivalence acceptent le même langage (ils seront fusionnés).

3..3 Minimiser un AEF

La méthode de réduction d'un AEF est la suivante :

1. *Nettoyer* l'automate en éliminant les états inaccessibles (cette étape n'est pas nécessaire si l'automate à minimiser a été obtenu par la procédure de déterminisation) ;
2. Regrouper les états congruents (appartenant à la même classe d'équivalence).

Dans l'algorithme suivant, chaque classe d'équivalence représentera un état dans l'automate minimal.

Algorithme de fusion des états β-équivalents d'un AEF $A = (X, Q, q_0, F, \delta)$

1 Créer deux classes d'états : $A = F$, $B = Q - F$

2 S'il existe un symbole a et deux états q_i et q_j d'une même classe tel que $\delta(q_i, a)$ et $\delta(q_j, a)$ n'appartiennent pas à la même classe, alors créer une nouvelle classe et séparer q_i et q_j. On laisse dans la même classe tous les états qui donnent des états appartenant à la même classe

3 Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus d'états à séparer

Les paramètres de l'automate minimal sont, alors, les suivants :

— Chaque classe d'équivalence est un état de l'automate minimal ;
— La classe qui contient l'ancien état initial représente l'état initial de l'automate minimal ;
— Toute classe contenant un état final devient un état final ;
— La fonction de transition est définie comme suit : soient A une classe d'équivalence obtenue, a un symbole de l'alphabet et q_i un état $q_i \in A$ tel que $\delta(q_i, a)$ est définie. La transition $\delta(A, a)$ est égale à la classe B qui contient l'état q_j tel que $\delta(q_i, a) = q_j$.

Exemple 2.8 : étapes pratiques de la minimisation

Soit à minimiser l'automate suivant (les états finaux sont les états 1 et 2 tandis que l'état 1 est initial) :

État	a	b
1	2	5
2	2	4
3	3	2
4	5	3
5	4	6
6	6	1
7	5	7

La première étape consiste à éliminer les états inaccessibles, il s'agit juste de l'état 7.

Pour le regroupement des états congruents, nous appliquons l'algorithme suivant :

1. Définir deux ensembles d'états : A : ensemble des états finaux, B : ensemble des états non finaux (si tous les états sont finaux, alors commencer avec un seul ensemble).
2. Procéder par itérations en vérifiant tous les ensembles d'états contenant plus qu'un état comme suit :
   (a) Pour un ensemble d'états E, pour chaque symbole a de l'alphabet, vérifier qu'il n'existe pas deux états $q, q' \in E$ tel que $\delta(q, a) \in J$, $\delta(q', a) \in J'$ et $J \neq J'$.
   (b) Si un tel symbole existe (on dit que l'ensemble n'est pas cohérent), procéder à la découpe de l'ensemble E comme dans l'algorithme précédent.
   (c) Sinon on passe au prochain symbole.
3. Si aucune modification n'a eu lieu au cours d'une itération, alors arrêter l'algorithme, sinon refaire une nouvelle itération.

Pour l'automate précédent, les étapes de détermination des classes d'équivalence sont les suivantes :

1. On crée d'abord les deux ensembles : $A = \{1, 2\}, B = \{3, 4, 5, 6\}$.
2. Itération 1 :
   (a) Pour l'ensemble A et le symbole a : $\delta(1, a) = 2 \in A$ et $\delta(2, a) = 2 \in A$, OK.
   (b) Pour l'ensemble A et le symbole b : $\delta(1, b) = 5 \in B$ et $\delta(2, b) = 4 \in B$, OK. L'ensemble A est cohérent pour le moment.
   (c) Pour l'ensemble B et le symbole a : $\delta(3, a) = 3 \in B$, $\delta(4, a) = 5 \in B$, $\delta(5, a) = 4 \in B$ et $\delta(6, a) = 6 \in B$. OK.
   (d) Pour l'ensemble B et le symbole b : $\delta(3, b) = 2 \in A$, $\delta(4, b) = 3 \in B$, $\delta(5, b) = 6 \in B$ et $\delta(6, b) = 1 \in A$. La classe doit être éclatée, alors on crée un nouvel ensemble $C = \{3, 6\}$ ce qui laisse l'ensemble $B = \{4, 5\}$.
3. Itération 2 (comme il y a eu des modifications au cours de la première itération).
   (a) On constate que l'ensemble A est cohérent (à vous de faire les vérifications).
   (b) On constate que l'ensemble B est cohérent (à vous de faire les vérifications).
   (c) On constate que l'ensemble C est cohérent (à vous de faire les vérifications).
   (d) On arrête alors l'algorithme car il n'y a plus de modifications possibles.

Le nouvel automate est donc le suivant (l'état initial est A) :

État	a	b
A	A	C
B	B	A
C	C	B

Remarque : L'automate obtenu est minimal et est unique, il ne peut plus être réduit. Si après la réduction d'un automate on obtient le même, alors cela signifie qu'il est déjà minimal. Par ailleurs, l'automate déterministe et minimal d'un AEF caractérise le langage accepté. En d'autres mots, si $L = L'$ (tous les deux étant réguliers), alors ils ont le même automate déterministe et minimal (on peut être amené à éliminer certains autres états non nécessaires néanmoins).

4. Opérations sur les automates

Notons d'abord que les opérations suivantes ne sont pas spécifiques aux AEF, elles peuvent plus ou moins être étendues à tout automate. Cependant, la complexité de ces opérations augmente inversement avec le type du langage.

4..1 Le complément

Soit A un automate <u>déterministe</u> défini par le quintuplet (X, Q, q_0, F, δ) acceptant le langage L. L'automate acceptant le langage inverse (c'est-à-dire $X^* - L$) est défini par le quintuplet $(X, Q, q_0, Q - F, \delta)$ (en d'autres termes, il suffit de changer les états finaux en états non finaux et vice-versa).

Cependant, cette propriété ne fonctionne que si l'automate est *complet* : la fonction δ est complètement définie pour toute paire $(q, a) \in Q \times X$ comme le montre les exemples suivants.

Exemple 2.9 : automate du langage complémentaire

Soit le langage des mots définis sur l'alphabet $\{a, b, c\}$ contenant le facteur ab. Il s'agit ici d'un automate complet, on peut, donc, lui appliquer la propriété précédente pour trouver l'automate des mots qui ne contiennent pas le facteur ab (notons que l'état 2 du deuxième automate correspond à une sorte d'état d'erreur auquel l'automate se branche lorsqu'il détecte le facteur ab. On dit que l'état 2 est *improductif*, étant donné qu'il ne peut pas générer de mots).

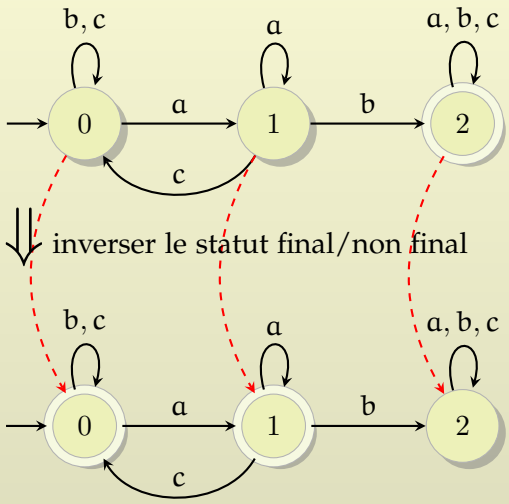


FIGURE 2.9 – Comment obtenir l'automate du langage complémentaire (d'un automate complet). Attention, les flèches rouges ne font pas partie de l'automate, elles illustrent seulement le changement de statut.

Soit maintenant le langage des mots définis sur $\{a, b, c\}$ contenant exactement deux a (figure 2.10). L'automate n'est pas complet, donc, on ne peut pas appliquer la propriété précédente à moins de le transformer en un automate complet. Pour le faire, il suffit de rajouter un état non final E (on le désignera comme un état d'erreur) tel que $\delta(2, a) = E$.

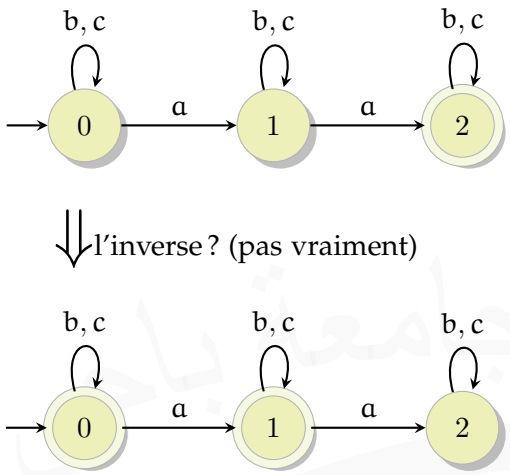


FIGURE 2.10 – Si l'automate n'est pas complet, on ne peut pas obtenir l'automate du langage inverse. L'automate obtenu accepte les mots contenant au plus 1 a.

Considérons à la fin l'automate de la figure 2.11. Les deux automates acceptant le mot a, ce qui signifie que les deux automates n'acceptent pas des langages complémentaires.

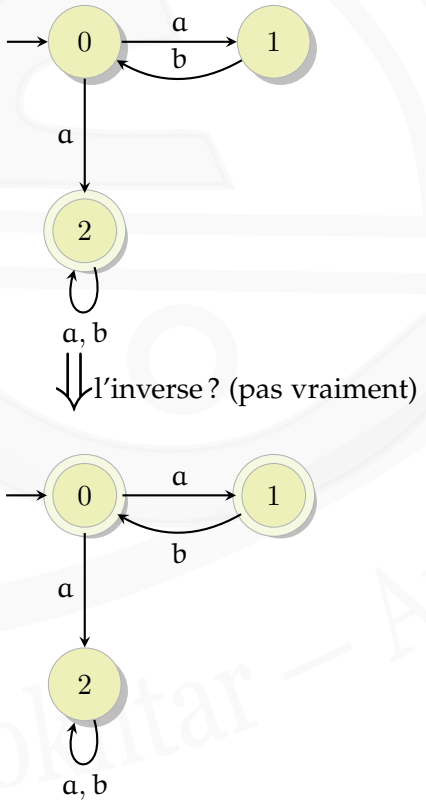


FIGURE 2.11 – Si l'automate n'est pas déterministe, on ne peut pas trouver l'automate du langage complémentaire.

Remarque : Le fait qu'un AEF ne soit pas déterministe ne représente pas un obstacle pour trouver l'automate du langage inverse. En effet, on pourra toujours le construire en procédant à une transformation. *Laquelle ?*

La construction d'une version complète d'un AEF (qui doit être déterministe) se fait comme suit :

1. Rajouter un nouvel état N ;
2. Les transitions manquantes mènent dorénavant vers N ;
3. Toute transition sortante de N revient au même état.

4..2 L'opération d'entrelacement

Nous verrons ici le cas simple de l'entrelacement d'un langage avec un symbole (voir la figure 2.12). Soit L un langage accepté par un AEF $A = (X, Q, q_0, F, \delta)$. On note par $A' = (X, Q', q'_0, F', \delta')$ une copie de l'automate A avec : Q' l'ensemble des états de Q auxquels on rajoute prime ($'$), par q'_0 l'état initial de A auquel on rajoute $'$, par F' l'ensemble des états de F auxquels on rajoute $'$ et par δ' la fonction de transition $\delta(q'_i, a) = q'_j$ si $\delta(q_i, a) = q_j$. L'automate acceptant le langage $L \sqcup \alpha$ (α est un seul symbole) est donné par $A'' = (X + \{\alpha\}, Q \cup Q', q_0, F', \delta'')$ tel que :

— $\delta''(q_i, x) = \delta(q_i, x)$, $q_i \in Q$;
— $\delta''(q'_i, x) = \delta'(q'_i, x)$, $q'_i \in Q'$;
— $\delta''(q_i, \alpha) = q'_i$, $q_i \in Q$.

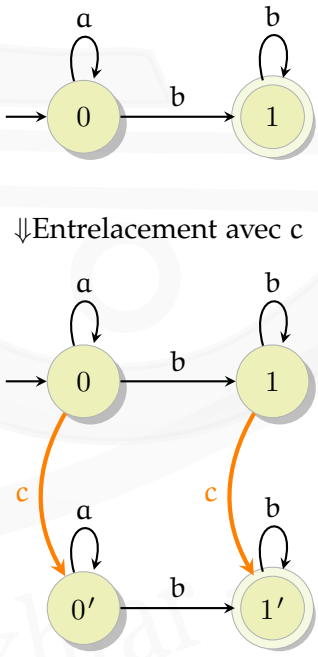


FIGURE 2.12 – Exemple d'entrelacement (ici c'est l'entrelacement avec le symbole c). Les arcs oranges sont les arcs (ayant été rajoutés) reliant les états du premier automate avec leurs correspondants du deuxième automates.

4..3 Produit d'automates

Définition 2.8 : Soient $A = (X, Q, q_0, F, \delta)$ et $A' = (X', Q', q'_0, F', \delta')$ deux automates à états finis. On appelle produit des deux automates A et A' l'automate $A'' = (X'', Q'', q''_0, F'', \delta'')$ défini comme suit :

- $X'' = X \cap X'$;
- $Q'' = Q \times Q'$;
- $q''_0 = (q_0, q'_0)$;
- $F'' = F \times F'$;
- $\delta''((q, q'), a) = \delta(q, a) \times \delta'(q', a)$

Cette définition permet de synchroniser l'analyse d'un mot par les deux automates. On pourra facilement démontrer que si w est un mot accepté par A'' alors il est également accepté par A et A'. Ainsi, si $L(A)$ est le langage accepté par A et $L(A')$ est le langage accepté par le langage A' alors l'automate A'' accepte l'intersection des deux langages : $L(A) \cap L(A')$.

Exemple 2.10 : produit d'automates

Considérons la figure 2.13.

- L'automate (1) accepte les mots sur $\{a, b, c\}$ contenant au moins deux a.
- L'automate (2) accepte les mots sur $\{a, b, c\}$ contenant au moins deux b.
- L'automate (3) représente le produit des deux automates.

Remarquons que dans l'automate (3), tout chemin qui part de l'état initial vers l'état final passe forcément par deux a et deux b (tout ordre est possible). Or, ceci est exactement le langage résultant de l'intersection des deux premiers langages.

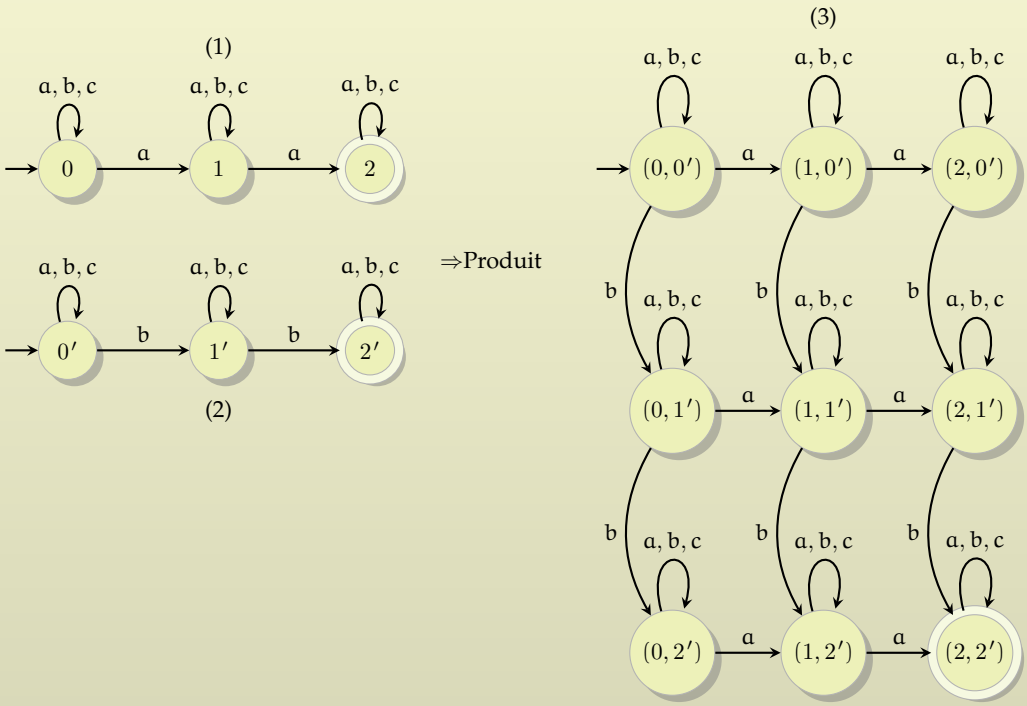


FIGURE 2.13 – Exemple de produit d'automates

4..4 Le langage miroir

Soit $A = (X, Q, q_0, F, \delta)$ un automate acceptant le langage $L(A)$. L'automate qui reconnaît le langage $(L(A))^R$ est accepté par l'automate $A^R = (X, Q, F, \{q_0\}, \delta^R)$ tel que : $\delta^R(q', a) = q$ si $\delta(q, a) = q'$.

En d'autres termes, il suffit juste d'inverser les sens des arcs de l'automate et le statut initial/final des états initiaux et finaux. Notons, par ailleurs, que l'automate A^R peut contenir plusieurs états initiaux ce qui signifie qu'il est le plus souvent non-déterministe. Pour éliminer le problème des multiples états initiaux, il suffit de rajouter un nouvel état initial et de le raccorder aux anciens états finaux par des ε-transitions.

Deuxième méthode de minimisation

La construction du miroir a une application surprenante, qui permet de minimiser un AEF quelconque. Soit A un AEF acceptant le langage $L(A)$. L'algorithme de minimisation est le suivant :

1. Construire l'AEF de $(L(A))^R$, appelons cet AEF A^R ;
2. Déterminiser A^R, appelons l'AEF résultat A^R_{det} ;
3. Construie l'AEF du miroir à partir de A^R_{det}, soit A' l'AEF résultant ;
4. Déterminiser A', le résultat représente l'AEF déterministe et minimal acceptant le langage $L(A)$.

Notons, enfin, que cet algorithme requiert que l'on accepte que l'AEF ait plusieurs états initiaux. Autrement, il ne fonctionne pas.

5. Simulation des AEF

Dans cette section, on s'intéresse à la construction d'un programme simulant le fonctionnement d'un AEF. On suppose alors que l'on dispose d'un AEF déterministe (pas forcément minimal) et que l'on veuille produire un programme en langage C qui le simule.

Tout d'abord, nous avons besoin de simuler certains éléments de la machine de Turing comme le ruban, la lecture, le déplacement de la tête, etc. Pour ce faire, on utilise les hypothèses suivantes :

— Le ruban est représenté par un tableau de caractères *word*. La tête de lecture est représentée par un entier `head` (initialisé à 0), la longueur du mot est stockée dans la variable entière `len`.

— Le déplacement de la tête se fait par la fonction `next()` qui renvoie le prochain caractère de la lecture. À la fin du mot, il renvoie 0.

Le code de la fonction peut être le suivant :

```
int next()
{
```

```
 if (head<len)
 return word[head++];
 else
 return 0;
}
```

La transcription d'un AEF $A = (X, Q, q_0, F, \delta)$ en un code C++ (car cela simplifie l'écriture seulement) se fait comme suit (le code peut bien sûr être amélioré par la suite) :

```
int state=q0;
char c=next();
int go=1;
while(go&&c)
{
 switch(state)
 {
  Pour chaque état q
  Mettre : case q :
  Mettre switch(c)
  //s'il y a des transitions sortantes, sinon ne rien mettre
  {
   Pour chaque symbole a tel que δ(q,a)=q'
   mettre :  case a:state=q';break;
   default:go=0;
  }
 }
 if (go) c=next();
}
if (c==0&&state|q ∈ F)
printf("%s","Accepted");|
else
printf("%s","Rejected");
```

A titre d'exemple, prenons l'automate de la figure 2.7, le code à écrire est alors le suivant :

```
int state=0;
char c=next();
int go=1;
while(go&&c)
{
    switch(state)
    {
        case 0 :
        switch(c)
        {
            case 'a' :
            state=1;
            break;
            case 'b' :
```

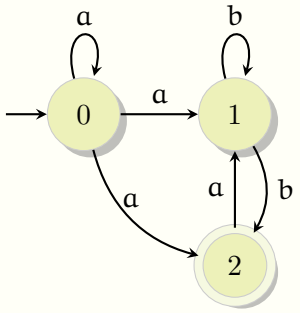
```
            state=0;
            break;
            default:
            go=0;
        }
        break;

        case 1 :
        switch(c)
        {
            case 'a' :
            state=1;
            break;
            case 'b' :
            state=2;
            break;
            default:
            go=0;
        }
        break;

        case 2 :
        switch(c)
        {
            case 'a' :
            state=3;
            break;
            case 'b' :
            state=2;
            break;
            default:
            go=0;
        }
        break;

        case 3 :
        switch(c)
        {
            case 'a' :
            state=3;
            break;
            case 'b' :
            state=2;
            break;
            default:
            go=0;
        }
        break;
    }
```

```
    if (go) c=next();
}
if (c==0&&(state==2||state==3))
printf("%s","Accepted");
else
printf("%s","Rejected");
```

6. Exercices de TD

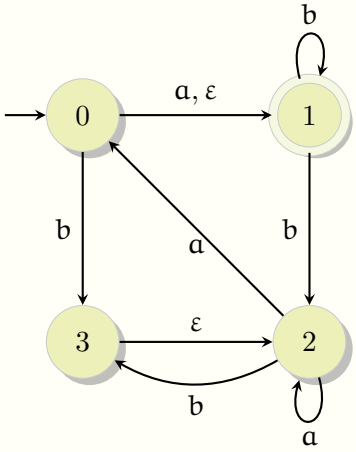
Exercice 1

Déterminisez et minimisez si nécessaire l'automate suivant :



Exercice 2

Déterminisez et minimisez si nécessaire l'automate suivant :



Exercice 3

Minimiser l'automate suivant (l'état initial est 1, les états finaux sont $\{3, 6, 8\}$) :

État	a	b	c
1	2	3	4
2	1	5	6
3	1	5	6
4	2	6	1
5	4	7	8
6	4	5	3
7	4	5	3
8	9	3	6
9	7	3	9

Exercice 4

Donnez les automates à états finis qui acceptent les langages suivants :

- Tous les mots sur $\{a, b, c\}$;
- Tous les mots sur $\{a, b, c\}$ qui se terminent avec un symbole différent de celui du début;
- Tous les mots sur $\{a, b, c\}$ qui contiennent le facteur ab;
- Tous les mots sur $\{a, b, c\}$ qui ne contiennent pas le facteur aba;
- Tous les mots sur $\{a, b, c\}$ qui ne contiennent pas le facteur aac;
- Tous les mots sur $\{a, b, c\}$ qui ne contiennent ni le facteur aba ni le facteur aac;
- Tous les mots sur $\{a, b, c\}$ tels que toutes les occurrences de c précèdent celle de b tout en ayant deux a.

Exercice 5

Donnez l'automate qui accepte tous les mots sur $\{a, b\}$ qui contiennent un nombre pair de a. Déduisez-en l'automate qui accepte le langage de tous les mots sur $\{a, b\}$ contenant un nombre pair de a et un nombre pair de b.

Exercice 6

Considérons l'ensemble des nombres binaires, donnez les automates qui acceptent :

- Les nombres multiples de 2;
- Les nombres multiples de 4;

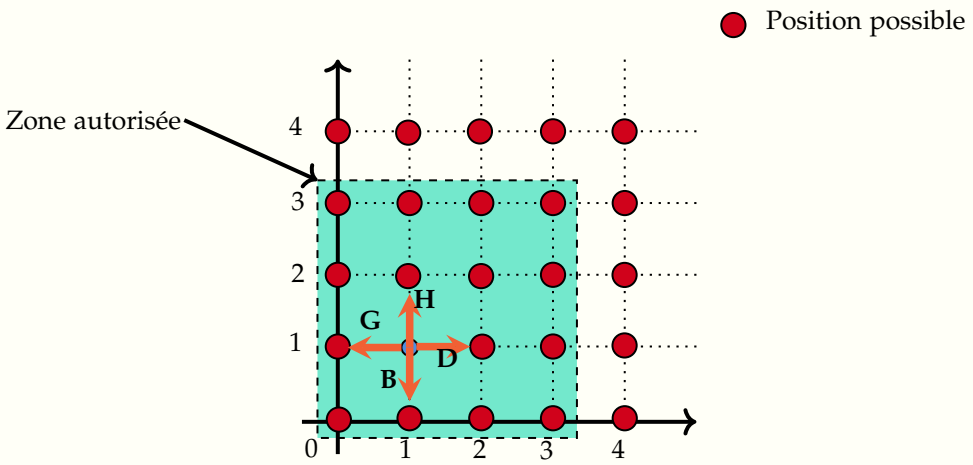
Déduisez-en l'automate qui accepte les multiples des nombres de la forme $2^n (n > 1)$. Si n est fixe, peut-on concevoir un automate qui accepte les multiples de tous les nombres binaires de la forme 2^i $(0 < i \leqslant n)$ et qui permet de simuler le calcul de i? Si oui, donnez l'automate, si non dites pourquoi?

Exercice 7

Donnez l'automate qui accepte les nombres binaires multiples de 3. Donnez ensuite deux façons pour construire l'automate qui accepte les multiples de 6.

Exercice 8

Soit un mobile pouvant bouger dans un environnement sous forme d'une matrice $\{0, 1, 2, 3\} \times \{0, 1, 2, 3\}$ selon la figure suivante.



Les mouvements possibles sont D (droite), G (gauche), H (haut) et B (bas). Le mobile prend ses ordres sous forme de mots composés sur l'alphabet $\{D, G, H, B\}$ (tout déplacement se fait d'une unité). Par exemple, si le mobile se trouve sur le point $(0, 0)$, alors le mot DHHG va situer le mobile sur le point $(0, 2)$. Ainsi, on peut parler de *langages* de déplacements permettant d'effectuer telle ou telle tâche. Donnez, si possible, les automates des déplacements (*langages*) suivants (on suppose que le mobile se trouve, au départ, au point $(0, 0)$) :

— Tout chemin assurant que le mobile reste dans la zone autorisée ;

— Les chemins qui n'entrent pas dans le carré $\{1, 2\} \times \{1, 2\}$;

— Les déplacements qui font revenir le mobile vers l'origine des coordonnées.

Exercice 9

Donnez un automate acceptant une date de la forme *jour/mois*. Faites attention aux dates invalides du type 30/02 (on considère que la date 29/02 est valide).

Exercice 10

Donnez un automate déterministe acceptant les nombres réels en langage C. Déduisez-en le programme qui accepte ces nombres.

Exercice 11

Donnez l'automate qui accepte les mots sur $\{a, b, c\}$ qui contiennent un a dans tout facteur de longueur $n \geqslant 1$.

Exercice 12

Dans un environnement virtuel, un robot se déplace en suivant les ordres donnés sous forme de mots (pour simplifier, on suppose que l'alphabet utilisé est égal à $\{a, b\}$). Par exemple, on peut lui donner l'ordre $abba$ pour lui ordonner d'aller à l'endroit ayant comme nom $abba$. Si le mot contient une séquence non répertoriée, alors l'emplacement du robot est indéterminé.

— Soient les emplacements ab et ba. Construire un automate déterministe permettant d'aller à deux états finaux différents selon la séquence lue (par exemple, si l'automate vient de lire ab alors il se trouve dans l'état 0, s'il lit ba alors il doit aller à l'état 1) ;

— Refaites la même chose avec les emplacements $\{aa, bb\}$ et $\{aba, abb\}$;

— Soit un automate qui accepte un seul emplacement u (vous pouvez considérer les automates précédents), quelle caractéristique peut-on donner au langage accepté par cet automate. Déduisez-en une méthode permettant de construire l'automate qui permet d'accepter deux emplacements distincts u et v.

Chapitre 3

Les langages réguliers

Les langages réguliers sont les langages générés par des grammaires de type 3 (ou encore les grammaires régulières). Ils sont acceptés par les automates à états finis. Le terme régulier vient du fait que les mots de tels langages possèdent une formes particulières pouvant être décrites par des expressions simples dites régulières. Ce chapitre introduit ces dernières et établit l'équivalence entre les trois représentations : expression régulière, grammaire régulière et AEF.

1. Les expressions régulières E.R

Définition 3.1 :
Soit X un alphabet quelconque ne contenant pas les symboles $\{*, +, |, ., (,)\}$. Une expression régulière est un mot défini sur l'alphabet $X + \{*, +, |, ., (,)\}$ permettant de représenter un langage régulier de la façon suivante :

— L'expression régulière ε dénote le langage vide ($L = \{\varepsilon\}$);
— L'expression régulière a ($a \in X$) dénote le langage $L = \{a\}$;
— Si r est une expression régulière qui dénote L alors $(r)^*$ (resp. $(r)^+$) est l'expression régulière qui dénote L^* (resp. L^+);
— Si r est une expression régulière dénotant L et s est une expression régulière dénotant L' alors $(r)|(s)$ est une expression régulière dénotant $L + L'$. L'expression régulière $(r).(s)$ (ou simplement $(r)(s)$) dénote le langage $L.L'$.

Les expressions régulières sont également appelées expressions *rationnelles*. L'utilisation des parenthèses n'est pas obligatoire si l'on est sûr qu'il n'y a pas d'ambiguïté quant à l'application des opérateurs $*, +, ., |$. Par exemple, on peut écrire $(a)^*$ ou a^* puisque l'on est sûr que $*$ s'applique juste à a. Par ailleurs, on se convient d'utiliser les priorités suivantes pour les différents opérateurs : 1)$*, +$, 2). et 3)|.

Exemple 3.1 : des expressions régulières

1. a^* : dénote le langage a^n ($n \geqslant 0$);
2. $(a|b)^*$: dénote les mots dans lesquels le symbole a ou b se répètent un nombre quelconque de fois. Elle dénote donc le langage de tous les mots sur $\{a, b\}$;

3. $(a|b)^*ab(a|b)^*$: dénote tous les mots sur $\{a, b\}$ contenant le facteur ab.

On peut également donner la signification suivante aux opérateurs des expressions régulières :

— r^* : signifie répéter r zéro ou plusieurs fois ;
— r^+ : signifie répéter r une ou plusieurs fois ;
— rs : signifie r suivie de s ;
— $r|s$: signifie soit r soit s (soit encore, r ou s).

1..1 Utilisation des expressions régulières

Les expressions régulières sont largement utilisées en informatique. On les retrouve plus particulièrement dans les *shells* des systèmes d'exploitation où ils servent à indiquer un ensemble de fichiers sur lesquels on voudrait appliquer un certain traitement. L'utilisation des expressions régulières en DOS, reprise et étendue par WINDOWS, est très limitée et ne concerne que le caractère "*" qui indique zéro ou plusieurs symboles quelconques ou le caractère "?" indiquant un symbole quelconque. Ainsi, l'expression régulière "f*" indique un mot commençant par f suivi par un nombre quelconque de symboles, "*f*" indique un mot contenant f et "*f*f*" indique un mot contenant deux f. L'expression "f?" correspond à n'importe quel mot de deux symboles dont le premier et f. Ce genre d'expressions régulières s'appelle expressions à *caractères jokers*.

L'utilisation la plus intéressante des expressions régulières est celle faite par UNIX et ses dérivés comme Linux (via la notation POSIX). Les possibilités offertes sont très vastes. Nous les résumons ici :

Expression	Signification
$[abc]$	les symboles a, b ou c
$[^abc]$	aucun des symboles a, b et c
$[a-e]$	les symboles de a jusqu'à e $\{a, b, c, d, e\}$
.	n'importe quel symbole sauf le symbole fin de ligne
$a*$	a se répétant 0 ou plusieurs fois
$a+$	a se répétant 1 ou plusieurs fois
$a?$	a se répétant 0 ou une fois
$a\|bc$	le symbole a ou b suivi de c
$a\{2,\}$	a se répétant au moins deux fois
$a\{,5\}$	a se répétant au plus cinq fois
$a\{2,5\}$	a se répétant entre deux et cinq fois
$\backslash x$	La valeur réelle de x (un caractère spécial)

Exemple 3.2 : des expressions régulières en notation POSIX

— $[^ab]*$: les mots qui ne comportent ni a ni b ;
— $[ab]*$: tous les mots sur $\{a, b\}$;
— $([^a] * a[^a] * a[^a]*)*$: les mots comportant un nombre pair de a ;
— $(ab\{, 4\})*$: en plus de ε, ce sont tous les mots commençant par a où chaque a est suivi

de quatre b au plus.

Utilisation des expressions régulières pour le calcul et la manipulation des données

Dans les éditeurs de texte supportant les expressions régulières (à titre d'exemple : notepad++, kate, gedit, brackets, atom, visual studio, ...), on peut faire une recherche basée sur les expressions régulières. Ces outils supportent aussi le remplacement basé sur les expressions régulières. Par exemple, si on veut remplacer tous les entiers par le mot "integer", on fournit les deux entrées suivantes : $[0-9]+$ et integer.

On peut aussi réaliser des remplacements contextuels via l'utilisation des parenthèses. Par exemple, l'expression régulière $([0-9])([0-9])$ représente une séquence de deux chiffres. Chaque paire de parenthèses correspond à un *groupe*. Ainsi, l'expression précédente renvoie deux groupes (notés respectivement par $\backslash 1$ et $\backslash 2$). Si on veut inverser l'ordre des séquences de deux entiers que l'on rencontre, on fournit alors les deux entrées suivantes : $([0-9])([0-9])$ et $\backslash 2\backslash 1$. Avec ces manipulations, on peut réaliser des calculs très intéressants sur une grande quantité de données.

À présent, nous allons quitter le merveilleux monde des E.R. POSIX, et revenir à ce cours. Nous allons, donc, reprendre la définition des expressions régulières (que l'on qualifiera de mathématiques) données par la section précédentes.

1..2 Expressions régulières ambiguës

Définition 3.2 :

Une expression régulière est dite *ambiguë* s'il existe au moins mot pouvant être mis en correspondance avec l'expression régulière de plusieurs façons.

Cette définition fait appel à la correspondance entre un mot et une expression régulière. Il s'agit, en fait, de l'opération qui permet de dire si le mot appartient au langage décrit par l'expression régulière. Par exemple, prenons l'expression régulière a^*b^*. Soit à décider si le mot aab est décrit ou non par cette expression. On peut écrire :

$$\underbrace{aa}_{a^*}\ \underbrace{b}_{b^*}$$

Ainsi, le mot est décrit par cette E.R. Il n'y a qu'une seule façon qui permet de le faire correspondre. Ceci est valable pour tous les mots de ce langage. L'E.R n'est donc pas ambiguë.

Considérons maintenant l'expression $(a|b)^*a(a|b)^*$ décrivant tous les mots sur $\{a, b\}$ contenant le facteur a. Soit à faire correspondre le mot $abaab$, on a :

$$\begin{aligned} abaab &= \underbrace{ab}_{(a|b)^*}.a.\underbrace{ab}_{(a|b)^*} \\ abaab &= \underbrace{aba}_{(a|b)^*}.a.\underbrace{b}_{(a|b)^*} \end{aligned}$$

Il existe donc au moins deux façons pour faire correspondre aab à l'expression précédente, elle est donc ambiguë.

L'ambiguïté pose un problème quant à l'interprétation d'un mot. Par exemple, supposons que, dans l'expression $(a|b)^*a(a|b)^*$, l'on veut comparer la partie à gauche du facteur a à la partie droite du mot. Selon la méthode de correspondance, le résultat est soit vrai ou faux ce qui est inacceptable dans un programme cohérent.

1..3 Comment lever l'ambiguïté d'une E.R ?

Il n'existe pas une méthode précise pour lever l'ambiguïté d'une E.R. Cependant, on peut dire que cette opération dépend de ce que l'on veut faire avec l'E.R ou plutôt d'une *hypothèse d'acceptation*. Par exemple, on peut décider que le facteur fixe soit le premier a du mot à analyser ce qui donne l'expression régulière : $b^*a(a|b)^*$. On peut également supposer que c'est le dernier a du mot à analyser ce qui donne l'expression régulière $(a|b)^*ab^*$.

2. Les langages réguliers, les grammaires et les AEF

Le théorème suivant établit l'équivalence entre les AEF, les grammaires régulières et les expressions régulières :

Théorème 3.1

(de Kleene) Soient Λ_{reg} l'ensemble des langages réguliers (générés par des grammaires régulières), Λ_{rat} l'ensemble des langages décrits par toutes les expressions régulières et Λ_{AEF} l'ensemble de tous les langages acceptés par un AEF. Nous avons, alors, l'égalité suivante :

$$\Lambda_{reg} = \Lambda_{rat} = \Lambda_{AEF}$$

Le théorème annonce que l'on peut passer d'une représentation à une autre du fait de l'équivalence entre les trois représentations. Par la suite, on verra comment passer d'une représentation à une autre.

2..1 Passage de l'automate vers l'expression régulière

Soit $A = (X, Q, q_0, F, \delta)$ un automate à états finis quelconque. On désigne par L_i le langage accepté par l'automate si son état initial était q_i. Par conséquent, trouver le langage accepté par l'automate revient à trouver L_0 étant donné que l'analyse commence à partir de l'état initial q_0. L'automate permet d'établir un système d'équations aux langages de la manière suivante :

— si $\delta(q_i, a) = q_j$ alors on écrit : $L_i = aL_j$;

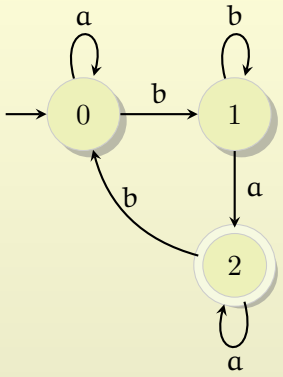
— si $q_i \in F$, alors on écrit : $L_i = \varepsilon$

— si $L_i = \alpha$ et $L_i = \beta$ alors on écrit : $L_i = \alpha|\beta$;

Il suffit ensuite de résoudre le système en précédant à des substitutions et en utilisant la règle suivante : la solution de l'équation $L = \alpha L|\beta$ ($\varepsilon \notin \alpha$) est le langage $L = \alpha^*\beta$ (attention ! si vous obtenez $L = \alpha L$ alors c'est la preuve d'une faute de raisonnement).

Exemple 3.3 : calcul de l'expression régulière

Soit l'automate donné par la figure suivante :



Trouvons le langage accepté par cet automate. Le système d'équations est le suivant :

1) $L_0 = aL_0|bL_1$

2) $L_1 = bL_1|aL_2$

3) $L_2 = aL_2|bL_0|\varepsilon$

Appliquons la deuxième règle sur les équations, on obtient alors après substitutions :

4) $L_0 = a^*bL_1$

5) $L_1 = b^*aL_2 = b^*a^+bL_0|b^*a^+$

6) $L_2 = a^*bL_0|a^*$

En remplaçant 5) dans 4) on obtient : $L_0 = a^*b^+a^+bL_0|a^*b^+a^+$. En appliquant le schéma de résolution donné plus haut, on trouve $L_0 = (a^*b^+a^+b)^*a^*b^+a^+$.

Remarque : Il est possible d'obtenir plusieurs expressions régulières associées à un AEF, selon l'ordre d'élimination des variables. Ceci est dû au fait que l'on peut trouver plusieurs expressions régulières différentes mais qui représentent, en fait, le même langage.

2..2 Passage de l'expression régulière vers l'automate

Il existe deux méthodes permettant de réaliser cette tâche. La première fait appel à la notion de *dérivée* tandis que la deuxième construit un automate comportant des ε-transitions en se basant sur les propriétés des langages réguliers.

Méthode des dérivées

Définition 3.3 : Soit w un mot défini sur un alphabet X. On appelle dérivée de w, par rapport à $a \in X$, le mot $u \in X^*$ tel que $w = au$. On note cette opération par : $u = w||a$.

On peut étendre cette notion aux langages. Ainsi la dérivée d'un langage par rapport à un mot $a \in X$ est le langage $L||a = \{u \in X^* | \exists w \in L : w = au\}$.

Exemple 3.4 : calcul des dérivées

— $L = \{1, 01, 11\}$, $L||1 = \{\varepsilon, 1\}$, $L||0 = \{1\}$.

Propriétés des dérivées

— $(\sum_{i=1}^{n} L_i)||a = \sum_{i=1}^{n}(L_i||a)$

— $(L.L')||a = (L||a).L' + f(L).(L'||a)$ tel que $f(L) = \{\varepsilon\}$ si $\varepsilon \in L$ et $f(L) = \emptyset$ sinon

— $(L^*)||a = (L||a)L^*$

Méthode de construction de l'automate par la méthode des dérivées

Soit r une expression régulière (utilisant l'alphabet X) pour laquelle on veut construire un AEF. L'algorithme suivant donne la construction de l'automate :

Algorithme de construction de l'AEF d'une E.R r par la méthode des dérivées

1 r est un nouveau langage

2 Dériver un nouveau langage disponible par rapport à chaque symbole de X

3 Recommencer l'étape 2 pour chaque nouveau langage obtenu jusqu'à ce qu'il n'y ait plus de nouveaux langages

4 Chaque langage obtenu correspond à un état de l'automate. L'état initial correspond à r. Si ε appartient à un langage obtenu alors l'état correspondant est final

5 **pour** *chaque relation de dérivation* $L_i||a = L_j$ **faire**

6 on crée une transition entre l'état associé à L_i et l'état associé à L_j et on la décore par a

Exemple 3.5 : application de la méthode des dérivées

Considérons le langage $L_0 = (a|b)^*a(a|b)^*$. On sait que :

— $(a|b)||a = \varepsilon$ donc $(a|b)^*||a = (a|b)^*$

Commençons l'application de l'algorithme :

— $L_0||a = [(a|b)^*a(a|b)^*]||a = ((a|b)^*a(a|b)^*)|(a|b)^* = (a|b)^* = L_1$

— $L_0||b = [(a|b)^*a(a|b)^*]||b = (a|b)^*a(a|b)^* = L_0$

— $L_1||a = [((a|b)^*a(a|b)^*)|(a|b)^*]||a = ((a|b)^*a(a|b)^*)|(a|b)^* = (a|b)^* = L_1$

— $L_1||b = [((a|b)^*a(a|b)^*)|(a|b)^*]||b = ((a|b)^*a(a|b)^*)|(a|b)^* = (a|b)^* = L_1$

Il n'y a plus de nouveaux langages, on s'arrête alors. L'automate comporte deux états q_0 (associé à $(a|b)^*a(a|b)^*$) et q_1 (associé à $((a|b)^*a(a|b)^*)|(a|b)^*$), il est donné par la table suivante (l'état initial est q_0 et l'état q_1 est final) :

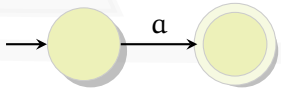
État	a	b
q_0	q_1	q_0
q_1	q_1	q_1

Remarque : Appliquée correctement, la méthode des dérives, bien qu'elle nécessite beaucoup de calcul, permet de construire l'AEF déterministe et minimal du langage. La difficulté de la méthode réside dans la difficulté de décider si deux expressions régulières sont équivalentes.

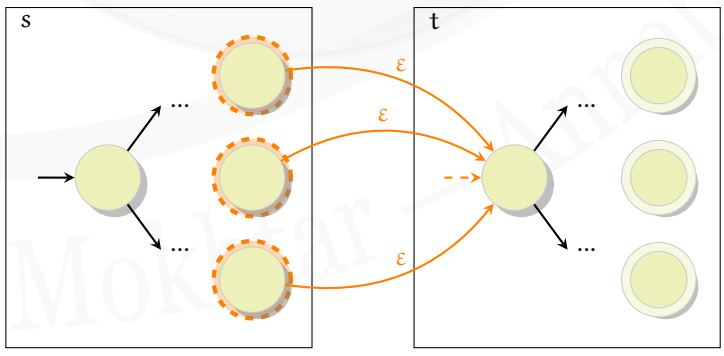
Méthode de Thompson

La méthode de Thompson permet de construire un automate en procédant à la décomposition de l'expression régulière selon les opérations utilisées. Soit r une E.R, alors l'algorithme à utiliser est le suivant :

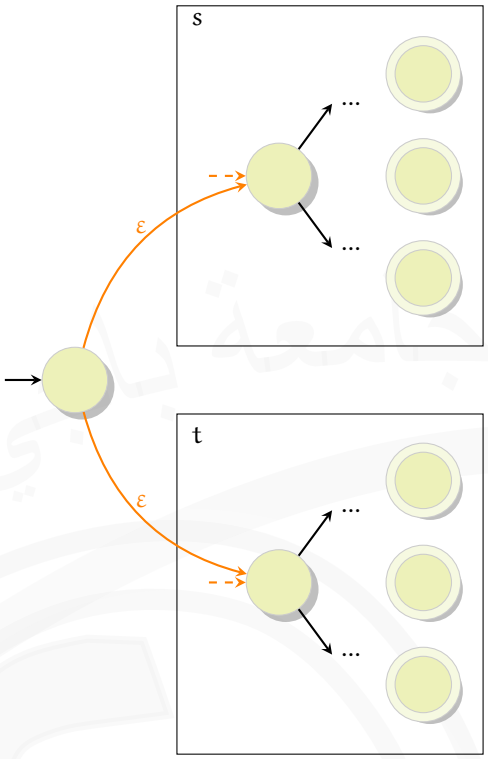
— Si $r = a$ (un seul symbole) alors l'automate est le suivant :



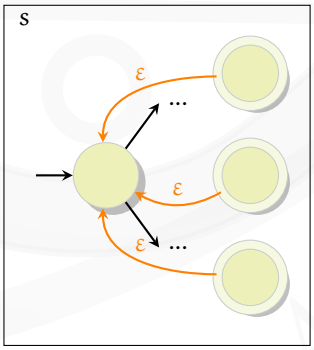
— Si $r = st$ alors il suffit de trouver l'automate A_s qui accepte s et l'automate A_t qui accepte t. Il faudra, ensuite relier chaque état final de A_s à l'état initial de A_t par une ε-transition. Les états finaux de A_s ne le sont plus et l'état initial de A_t ne l'est plus :



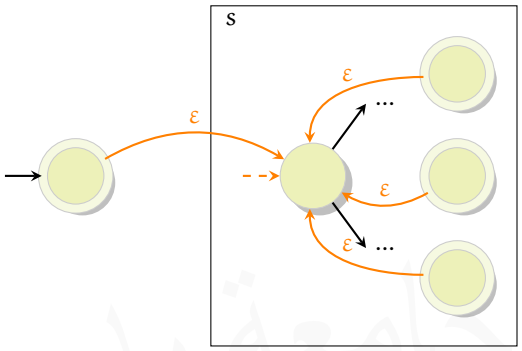
— Si $r = s|t$ alors il suffit de créer un nouvel état initial et le relier avec des ε-transitions aux états initiaux de A_s et A_t qui ne le sont plus :



— Si $r = s^+$ alors il suffit de relier les états finaux de A_s par des ε-transitions à son propre état initial :

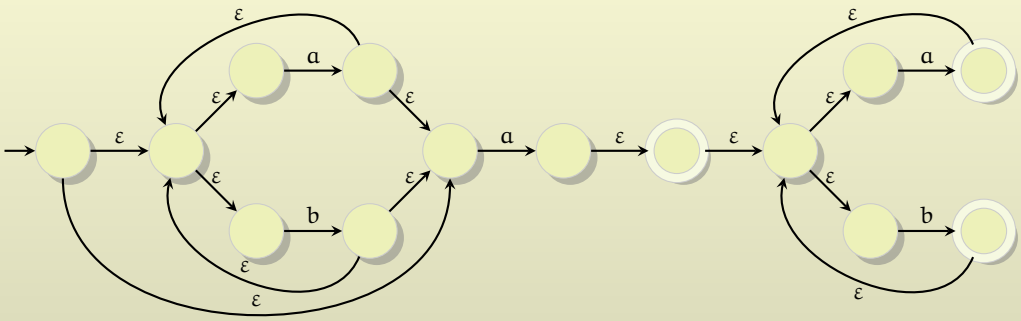


— Si $r = s^*$ alors il suffit de relier les états finaux de A_s par des ε-transitions à son état initial, créer un nouvel état initial et le relier à l'ancien état initial par une ε-transition. Enfin, le nouvel état initial est également final (dans certaines conditions, on peut trouver d'autres constructions possibles) :



Exemple 3.6 : application de la méthode de Thompson

Soit à construire l'automate du langage $(a|b)^*a(a|b)^*$, la méthode de Thompson donne l'automate suivant :



2..3 Passage de l'automate vers la grammaire

Du fait de l'équivalence des automates à états finis et les grammaires régulières, il est possible de passer d'une forme à une autre. Le plus facile étant le passage de l'automate vers la grammaire. Le principe de correspondance entre automates et grammaires régulières est très intuitif : il correspond à l'observation que chaque transition dans un automate produit exactement un seul symbole (au plus), de même que chaque dérivation dans une grammaire régulière *normalisée* (un concept qui sera présenté plus bas).

Soit $A = (X, Q, q_0, F, \delta)$ un AEF, la grammaire qui génère le langage accepté par A est $g = (V, N, S, R)$:

— $V = X$;

— On associe à chaque état de Q, un non-terminal. Ceci permet d'avoir autant de non-terminaux qu'il existe d'états dans A ;

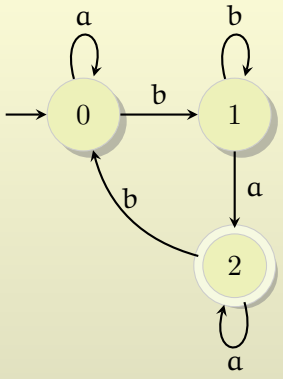
— L'axiome S est le non-terminal associé à l'état initial q_0 ;

— Soit A le non-terminal associé à q_i et B le non-terminal associé à q_j, si $\delta(q_i, a) = q_j$ alors la grammaire possède la règle de production : $A \to aB$;

— Si q_i est final et A est le non-terminal associé à q_i alors la grammaire possède la règle de production : $A \to \varepsilon$.

Exemple 3.7 : déduction de la grammaire régulière à droite

Soit l'AEF suivant :



La grammaire générant le langage accepté (en associant S à 0, T à 1 et U à 2) est : $(\{a, b\}, \{S, T, U\}, S, \{S \to aS|bT, T \to bT|aU, U \to aU|bS|\varepsilon\})$.

La forme engendrée par cet algorithme correspond à ce qu'on appelle grammaire régulière à droite (du fait que le non-terminal, dans la partie droite des règles, se trouve le plus à droite). En réalité, il existe une autre forme des grammaires de type 3, dite grammaire régulière à gauche.

Définition 3.4 : Soit $G = (V, N, S, R)$ une grammaire. G est dite régulière à gauche si toutes les règles de production sont de la forme $A \to B\alpha$ ou $A \to \alpha$ tel que $A, B \in N$ et $\alpha \in V^*$. Une grammaire régulière à gauche génère un langage régulier.

Notons ici qu'une grammaire est soit régulière à droite, soit à gauche, sinon elle n'est pas régulière. En d'autres termes, si on trouve des formes régulières à droite et à gauche, alors la grammaire n'est pas du type 3 (elle est de type 2).

Il n'est toutefois pas possible de déduire la grammaire régulière à gauche à partir de l'AEF Pour cela, il faut une petite gymnastique. Commençons d'abord par noter que si l'on prend une grammaire régulière droite à laquelle on applique le miroir sur toutes les parties droites des règles de production, on obtient une grammaire régulière à gauche générant le langage miroir. On peut alors proposer l'algorithme suivant. Soit A un AEF acceptant un langage L :

1. Construire l'AEF A^R acceptant le langage miroir (voir le chapitre 2 pour la méthode de construction).
2. Déduire la grammaire régulière à droite de L^R à partir de A^R.
3. Appliquer le miroir aux parties droites de toutes les règles de la grammaire obtenue. Le résultat représente la grammaire régulière à gauche générant L.

Notons, par le passage, que l'on s'intéresse généralement à une forme normalisée des grammaires régulières. Celle-ci est définie comme suit : soit $G = (V, N, S, R)$ une grammaire régulière à droite, elle est dite normalisée si toutes les règles de production sont de l'une des formes suivantes :

— $A \to aB$, $A, B \in N$, $a \in V$ $(|a| = 1)$;

— $A \to \varepsilon$, $A \in N$

2..4 Passage de la grammaire vers l'automate

D'après la section précédente, il existe une forme de grammaires régulières pour lesquelles il est facile de construire l'AEF correspondant. En effet, soit $G = (V, N, S, R)$ une grammaire régulière à droite, si toutes les règles de production sont de la forme : $A \to aB$ ou $A \to B$ $(A, B \in N, a \in V \cup \{\varepsilon\})$ alors il suffit d'appliquer l'algorithme suivant :

1. Associer un état à chaque non-terminal de N ;
2. L'état initial est associé à l'axiome ;
3. Pour chaque règle de production de la forme $A \to \varepsilon$, l'état q_A est final ;
4. Pour chaque règle $A \to aB$ alors créer une transition partant de q_A vers l'état q_B en utilisant l'entrée a ;
5. Pour chaque règle $A \to B$ alors créer une ε-transition partant de q_A vers l'état q_B.

Cependant, cet algorithme ne peut pas s'appliquer aux grammaires non normalisées. On peut néanmoins transformer toute grammaire régulière à droite à la forme normalisée. L'algorithme de transformation est le suivant. Soit $G = (V, N, S, R)$ une grammaire régulière :

1. Pour chaque règle de la forme $A \to wB$ $(A, B \in N$ et $w \in V^*)$ tel que $|w| > 1$, créer un nouveau non-terminal B' et éclater la règle en deux : $A \to aB'$ et $B' \to uB$ tel que $w = au$ et $a \in V$. Il faut recommencer la transformation jusqu'à obtenir des règles où le non-terminal à droite est précédé d'un seul non-terminal.
2. Pour chaque règle de la forme $A \to w$ $(A \in N$ et $w \in V*)$ tel que $|w| \geqslant 1$, créer un nouveau non-terminal B' et éclater la règle en deux : $A \to aB'$ et $B' \to u$ tel que $w = au$ et $a \in V$. Il faut recommencer la transformation jusqu'à obtenir une règle avec ε dans la partie droite.

Exemple 3.8 : transformation de grammaire vers la forme normalisée

Soit la grammaire régulière $G = (\{a, b\}, \{S, T\}, S, \{S \to aabS|bT, T \to aS|bb\})$. Le processus de transformation est le suivant :

— Éclater $S \to aabS$ en $S \to aS_1$ et $S_1 \to abS$;
— Éclater $S_1 \to abS$ en $S_1 \to aS_2$ et $S_2 \to bS$;
— Éclater $T \to bb$ en $T \to bT_1$ et $T_1 \to b$;
— Éclater $T_1 \to b$ en $T_1 \to bT_2$ et $T_2 \to \varepsilon$

Remarque : Il est possible de déduire une grammaire (qui ne sera pas forcément de type 3) à partir de l'expression régulière en utilisant un algorithme semblable à celui de Thompson. L'étudiant est invité à revoir le dernier exercice de la première série de TD pour une première idée de l'algorithme de construction.

3. Propriétés des langages réguliers

3..1 Stabilité par rapport aux opérations sur les langages

Les langages réguliers sont stables par rapport aux opérations de l'union, l'intersection, le complément, la concaténation et la fermeture de Kleene. La démonstration de ce résultat est très simple. Soient L et M deux langages réguliers désignés respectivement par les E.R r et s et respectivement acceptés par les automates A_r et A_s. Étant donné l'équivalence entre les langages réguliers et les AEF, nous avons :

— $L + M$ est régulier : l'AEF correspondant s'obtient en utilisant l'algorithme de construction de l'automate d'acceptation de l'E.R $r|s$;

— $L \cap M$ est régulier : l'AEF correspondant s'obtient en calculant le produit de A_r et A_s (voir le chapitre précédent) ;

— $\overline{L}$ est régulier : l'AEF correspondant s'obtient en rendant l'automate A_r déterministe et complet puis en inversant le statut final/non final des états (voir le chapitre précédent) ;

— L^R est régulier : l'AEF correspondant s'obtient en inversant le sens des arcs dans A_r et en inversant le statut initial/final des états (voir le chapitre précédent) ;

— L.M est régulier : l'AEF correspondant s'obtient en utilisant l'algorithme de construction de l'automate de $r.s$;

— L^* (resp. L^+) est régulier : l'AEF correspondant s'obtient en utilisant l'algorithme de construction de l'automate de r^* (resp. r^+) ;

Revenons un peu sur la stabilité des langages réguliers par rapport à l'union. Le résultat donné ci-haut montre que l'union finie de langages réguliers représente forcément un langage régulier. Ce qui signifie que tout langage fini est régulier. Cependant, l'union infinie de langages réguliers peut être de n'importe quel type (tout dépend des langages en question).

3..2 Les langages réguliers et la méthode des dérivées

Nous avons déjà vu deux méthodes de construction des AEF à partir de l'E.R : méthode des dérivées et méthode de Thompson. La première, nécessitant beaucoup de calculs, a le mérite de produire l'AEF déterministe et minimal du langage. La deuxième, très simple à appliquer et à programmer, a l'inconvénient de produire un AEF non-déterministe et comportant beaucoup d'états (notons qu'il existe des méthodes concurrentes à celle de Thompson, plus complexes mais produisant moins d'états de d'ε-transitions).

Si la méthode de Thompson requiert une expression régulière pour fonctionner, celle des dérivées ne le requiert pas car elle peut être appliquée à tout langage. On s'interrogera néanmoins sur le critère d'arrêt de cette méthode, le théorème suivant nous sera d'une grande aide (bien sûr, on suppose que la méthode des dérivées est convenablement appliquée) :

Théorème 3.2

La méthode des dérivées produit un nombre fini d'états pour tout langage régulier, et un nombre infini d'états pour tout langage non régulier.

Ainsi, la méthode des dérivées nous offre un premier moyen permettant de différencier les langages réguliers des non réguliers.

Exemple 3.9 : montrer que le langage $\{a^n b^n | n \geqslant 0\}$ n'est pas régulier

Soit le langage $L_0 = \{a^n b^n | n \geqslant 0\}$, appliquons-lui la méthode des dérivées. Nous avons alors :

- $L_0 || a = \{a^{n-1} b^n | n \geqslant 1\} = L_1$
- $L_1 || a = \{a^{n-2} b^n | n \geqslant 2\} = L_2$
- ...
- $L_i || a = \{a^{n-(i+1)} b^n | n \geqslant i+1\} = L_{i+1}$

On voit alors que les opérations de dérivations ne s'arrêtent jamais, produisant ainsi une infinité d'états. On en déduit que L_0 n'est pas régulier (en fait, il est algébrique).

3..3 Lemme de la pompe

Dans cette section, nous présentons d'autres critères permettant d'affirmer si un langage est régulier ou non. Rappelons d'abord que tout langage fini est un langage régulier. À présent, nous allons annoncer un critère dont la vérification permet de juger si un langage n'est pas régulier. Il s'agit en fait d'une condition nécessaire et non suffisante pour qu'un langage soit régulier. La démonstration de ce résultat sort du cadre du cours.

Proposition 3.1

Soit L un langage régulier infini défini sur l'alphabet X. Il existe alors un entier n tel que pour tout mot $w \in L$ et $|w| \geqslant n$, il existe $x, z \in X^*$ et $y \in X^+$ tels que :

- $w = xyz$;
- $|xy| \leqslant n$;
- $xy^i z \in L$ pour tout $i > 0$.

Il est à noter que cette condition est nécessaire et non suffisante. Il existe, en effet, des langages non réguliers vérifiant ce critère. Il existe néanmoins une condition nécessaire et suffisante que l'on va énoncer après avoir présenté un exemple d'utilisation du lemme de la pompe.

Exemple 3.10 : application du lemme de la pompe sur un langage régulier

Soit le langage $L = \{a^k b^l | k, l \geqslant 0\}$ (ou encore $a^* b^*$, il s'agit donc d'un langage régulier). Prenons $n = 1$ et vérifions le critère de la pompe. Soit un mot $w = a^k b^l$ $(k + l \geqslant 1)$. Si $k > 0$ alors il suffit de prendre $x = a^{k-1}, y = a$ et $z = b^l$, ainsi tout mot de la forme $xy^i z = a^{k+i-1} b^l$ appartient au langage. Si $k = 0$ alors il suffit de prendre $x = \varepsilon, y = b$ et $z = b^{l-1}$ pour vérifier le critère de la pompe.

Exemple 3.11 : application du lemme de la pompe sur un langage non régulier

Soit le langage $L = \{a^k b^k | k \geqslant 0\}$[a]. Supposons que le langage est régulier et appliquons le lemme de la pompe. Supposons que l'on a trouvé n qui vérifie le critère, ceci implique que pour tout mot $w \in L$, on peut le décomposer en trois sous-mots x, y et z tel que : $|xy| \leqslant n$, $y \neq \varepsilon$ et $xy^i z \in L$. Considérons le mot $a^{n+1} b^{n+1}$, toute décomposition de ce mot produit les mots $x = a^j$, $y = a^l$ et $z = a^{n+1-j-l} b^{n+1}$, $l > 0$ (si xy contient n symboles alors x et y ne contiennent que des a étant donné qu'il y a $(n + 1)$ a). Maintenant, le lemme de la pompe stipule que $xy^i z \in L$ pour tout $i > 0$ donc tout mot de la forme $a^j a^{il} a^{n+1-j-l} b^{n+1} = a^{(i-1)l+n+1} b^{n+1}$ appartient à L. Pour $i = 2$ la borne inférieure de $(i - 1)l$ est 1, ce qui signifie que le nombre de a est différent du nombre de b. Contradiction. Donc, le langage $a^k b^k$ n'est pas régulier.

a. Attention, l'expression $a^k b^k$ n'est pas régulière.

Exemple 3.12 : le lemme de la pompe peut être satisfait par un langage non régulier

Soit maintenant soit $L \subset b^*$ un langage non régulier arbitraire. Le langage :

$$a^+ L | b^*$$

satisfait le lemme de la pompe. Il suffit de prendre avec les notations du lemme, $n = 1$. Cet exemple illustre donc le fait que le lemme précédent ne constitue pas une condition nécessaire pour décider de la régularité d'un langage.

Il existe néanmoins une version nécessaire et suffisante du lemme de la pompe pour savoir, à coup sûr, si un langage est régulier.

Proposition 3.2

Soit L un langage infini défini sur l'alphabet X. L est régulier si et seulement s'il existe un entier $n > 0$ tel que pour tout mot $w \in X^*$ et $|w| \geqslant n$, il existe $x, z \in X^*$ et $y \in X^+$ tels que :

— $w = xyz$;

— $\forall u \in X^* : wu \in L \Leftrightarrow xy^i zu \in L$.

4. Exercices de TD

Exercice 1

Donnez une description de chacune des E.R suivantes, puis donnez leurs équivalentes en notation POSIX :

— $a(a|b)^*(b|\varepsilon)$;

— $(aaa)^*$;

— $(a|ab|abb)^*$

— $a(a|b)^*(aa|bb)^+$

— $(a|ab)^*$

Exercice 2

Donnez les expressions régulières suivantes en notation POSIX :

— Les identifiants en C ;

— Les nombres entiers multiples de 5 en base 10 ;

— Les mots sur $\{a, b, c\}$ contenant le facteur a^{1000}.

Exercice 3

Donnez une expression régulière pour chacun des langages suivants. En déduire à chaque fois l'automate correspondant en utilisant la méthode de Thompson ou celle des dérivées. Donnez leurs équivalentes en notation POSIX.

— Tous les mots sur $\{a, b, c\}$ commençant par a ;

— Tous les mots sur $\{a, b\}$ commençant par un symbole différent de leur dernier symbole ;

— Tous les mots sur $\{a, b, c\}$ contenant exactement deux a ;

— Tous les mots sur $\{a, b, c\}$ contenant au moins deux a ;

— Tous les mots sur $\{a, b, c\}$ contenant au plus deux a ;

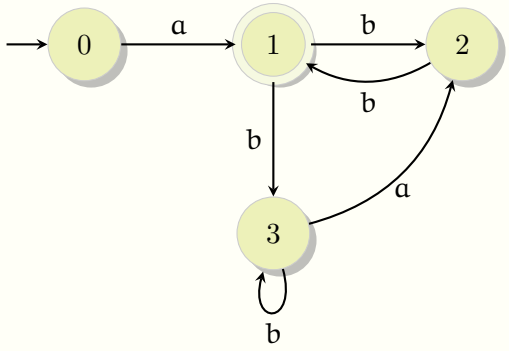
— Tous les mots sur $\{a, b, c\}$ ne contenant pas le facteur ab ;

— Tous les mots sur $\{a, b, c\}$ ne contenant pas le facteur aac ;

— Tous les entiers (en base dix) multiples de 5.

Exercice 4

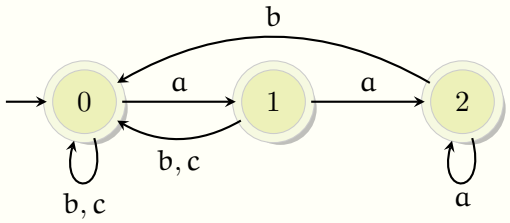
Soit l'automate suivant :



Trouvez le langage accepté par cet automate ainsi qu'une grammaire régulière qui le génère.

Exercice 5

Soit l'automate suivant :



1. Trouvez le langage régulier accepté par cet automate ;
2. Trouvez l'automate du langage complémentaire et déduisez son expression régulière.

Exercice 6

Soit le grammaire régulière : $G = (\{a, b\}, \{S, T\}, S, \{S \to abS|aS|bT|\varepsilon, T \to baT|bT|aS|\varepsilon\})$. Trouvez l'automate acceptant le langage généré par cette grammaire puis déduisez-en son expression régulière.

Exercice 7

En utilisant le lemme de la pompe, dites si les langages suivants sont réguliers. Si le critère de la pompe est vérifié, donnez un AEF ou une grammaire régulière générant le langage pour être sûr de la régularité :

— $a^n b^m$,$n \geqslant 0, m > 0$;
— a^{2n+1}, $n \geqslant 0$;
— $a^m b^n$, $m, n \geqslant 0$ et $n + m$ est pair ;
— $a^{2n} b^n$, $n \geqslant 0$
— a^{n^2}, $n \geqslant 0$;
— $b^m a^{n^2} + a^k$, $n, k \geqslant 0$, $m > 0$

Exercice 8

Donnez une expression régulière décrivant les langages suivants :

— tous les mots sur $\{a, b\}$ où chaque a est précédé d'un b ;
— tous les mots sur $\{a, b\}$ contenant à la fois les facteurs aa et bb ;
— tous les mots sur $\{a, b\}$ contenant soit aa soit bb mais pas les deux à la fois ;
— tous les mots sur $\{a, b\}$ ne contenant pas deux a consécutifs ;
— tous les mots sur $\{a, b, c\}$ où le nombre de a est multiple de 3.

Chapitre 4

Les langages algébriques

Malgré la panoplie d'algorithmes existant pour travailler sur les automates à états finis, ceux-ci restent limités aux seuls langages réguliers. Par exemple, dans le chapitre précédent, nous avons montré que le langage $\{a^n b^n | n \geqslant 0\}$ n'est pas régulier et, par conséquent, ne peut pas être reconnu par un AEF. Ce chapitre élargit un peu le champ d'études en s'intéressant aux langages algébriques qui représentent la couche qui suit immédiatement celle des langages réguliers dans la hiérarchie de Chomsky. Notons, cependant, que le niveau de complexité est inversement proportionnel au type du langage et, par conséquent, le nombre d'algorithmes existants tend à diminuer en laissant la place à plus d'*intuition*. Il reste à noter que les langages algébriques sont, tout de même, plus intéressants du fait de leur meilleure expressivité (par rapport aux langages réguliers) et qu'ils représentent (à un certain degré) la base de la plupart des langages de programmation.

1. Les automates à pile

Les automates à pile sont une extension des automates à états finis. Ils utilisent une (et une seule [1]) pile pour accepter les mots en entrée.

Définition 4.1 : Un automate à pile est défini par le sextuplet $A = (X, \Pi, Q, q_0, F, \delta)$ tel que :

- X est l'ensemble des symboles formant les mots en entrée (alphabet des mots à analyser) ;
- Π est l'ensemble des symboles utilisés pour écrire dans la pile (l'alphabet de la pile). Cet alphabet doit forcément inclure le symbole $\triangleright$ signifiant que la pile est vide ;
- Q est l'ensemble des états possibles ;
- q_0 est l'état initial ;
- F est l'ensemble des états finaux ($F \subseteq Q$) ;
- δ est une fonction de transition permettant de passer d'un état à un autre :

  $\delta : Q \times (X + \{\varepsilon\}) \times \Pi \mapsto 2^Q \times ((\Pi - \{\triangleright\})^* + \{\triangleright\})$
  $\delta(q_i, a, b) = (q_j, c)$ ou $\emptyset$ ($\emptyset$ signifie que la configuration n'est pas prise en charge)
  b est le sommet de la pile et c indique le nouveau contenu de la pile relativement à ce qu'il y avait avant le franchissement de la transition.

1. On peut montrer qu'un automate à deux piles est équivalent à une machine de Turing

Par conséquent, tout automate à états finis est en réalité un automate à pile à la seule différence que la pile du premier reste vide ou au mieux peut être utilisée dans une certaine limite que l'on ne dépasse jamais.

Une configuration d'un automate à pile consiste à définir le triplet (q, a, b) tel que q est l'état actuel, a est le symbole actuellement en lecture et b est le sommet actuel de la pile (on peut également mettre le contenu de toute la pile). Lorsque $b =$ " $\triangleright$ ", alors la pile est vide.

Les exemples suivants illustrent comment peut-on interpréter une transition :

— $\delta(q_0, a, \triangleright) = (q_1, B\triangleright)$ signifie que si l'on est dans l'état q_0, si le symbole actuellement lu est a et si la pile est vide alors passer à l'état q_1 et empiler le symbole B ;

— $\delta(q_0, a, A) = (q_1, BA)$ signifie que si l'on est dans l'état q_0, si le symbole actuellement lu est a et si le sommet de la pile est A alors passer à l'état q_1 et empiler le symbole B ;

— $\delta(q_0, \varepsilon, A) = (q_1, BA)$ signifie que si l'on est dans l'état q_0, s'il ne reste plus rien à lire et que le sommet de la pile est A alors passer à l'état q_1 et empiler le symbole B ;

— $\delta(q_0, a, A) = (q_1, A)$ signifie que si l'on est dans l'état q_0, si le symbole actuellement lu est a et si le sommet de la pile est A alors passer à l'état q_1 et ne rien faire à la pile ;

— $\delta(q_0, a, A) = (q_1, \varepsilon)$ signifie que si l'on est dans l'état q_0, si le symbole actuellement lu est a et si le sommet de la pile est A alors passer à l'état q_1 et dépiler un symbole de la pile.

Un mot w est accepté par un automate à pile si après avoir lu tout le mot w, l'automate se trouve dans un état final. Le contenu de la pile importe peu du fait que l'on peut toujours la vider (comment ?). Par conséquent, un mot est rejeté par un automate à pile :

— Si lorsque aucune transition n'est possible, l'automate n'a pas pu lire tout le mot ou bien il se trouve dans un état non final.

— Si une opération incorrecte est menée sur la pile : dépiler alors que la pile est vide.

Exemple 4.1 : un automate à pile

L'automate acceptant les mots $a^n b^n$ (avec $n \geqslant 0$) est le suivant : $A = (\{a, b\}, \{A, \triangleright\}, \{q_0, q_1, q_2\}, q_0, \{q_2\}, \delta)$ tel que δ est définie par :

— $\delta(0, a, \triangleright) = (0, A\triangleright)$

— $\delta(0, a, A) = (0, AA)$

— $\delta(0, b, A) = (1, \varepsilon)$

— $\delta(1, b, A) = (1, \varepsilon)$

— $\delta(1, \varepsilon, \triangleright) = (2, \triangleright)$

— $\delta(0, \varepsilon, \triangleright) = (2, \triangleright)$

Détaillons l'analyse de quelques mots :

1. Le mot $aabb$: $(0, a, \triangleright) \rightarrow (0, a, A\triangleright) \rightarrow (0, b, AA\triangleright) \rightarrow (1, b, A\triangleright) \rightarrow (1, \varepsilon, \triangleright) \rightarrow (2, \varepsilon, \triangleright)$. Le mot est accepté. Schématiquement, l'analyse se fait selon la figure 4.1.

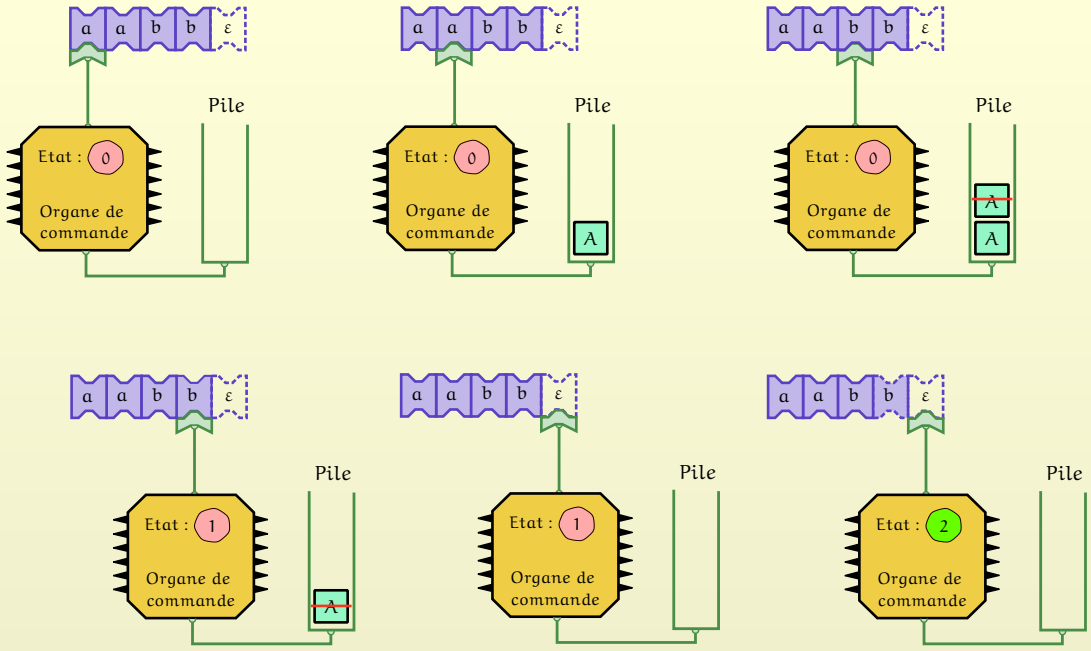


FIGURE 4.1 – Analyse du mot aabb

2. Le mot aab : $(0, a, \triangleright) \rightarrow (0, a, A\triangleright) \rightarrow (0, b, AA\triangleright) \rightarrow (1, \varepsilon, A\triangleright)$. Le mot n'est pas accepté car il n'y a plus de transitions possibles alors que l'état de l'automate n'est pas final. Schématiquement, l'analyse se fait selon la figure 4.2.

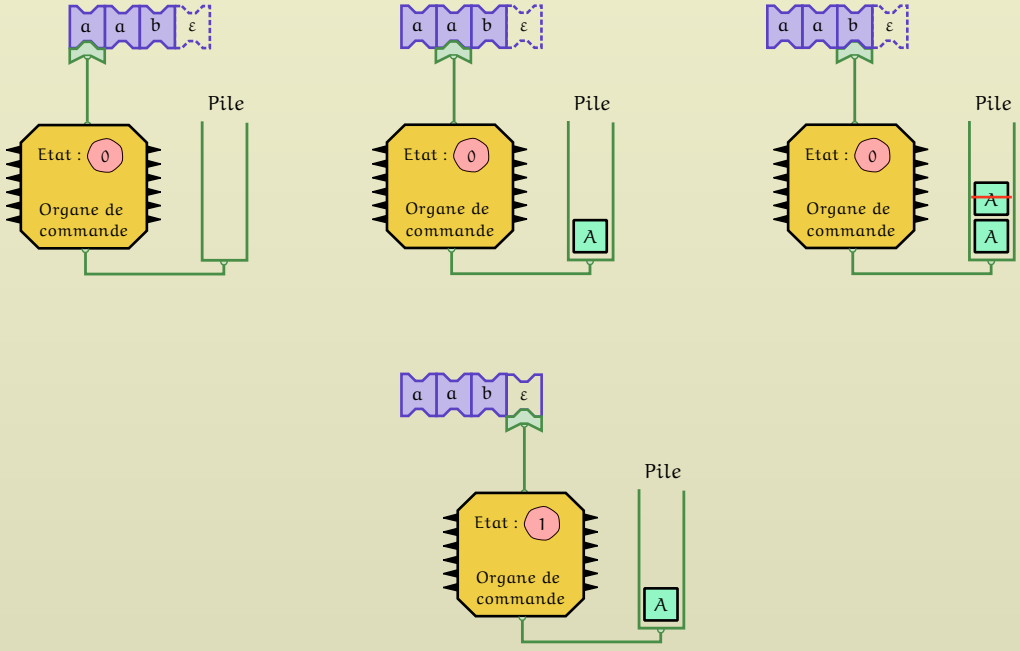


FIGURE 4.2 – Analyse du mot aab

3. Le mot abb : $(0, a, \triangleright) \rightarrow (0, b, A\triangleright) \rightarrow (1, b, \triangleright)$. Le mot n'est pas accepté car on n'arrive pas à lire tout le mot. Schématiquement, l'analyse se fait selon la figure 4.3.

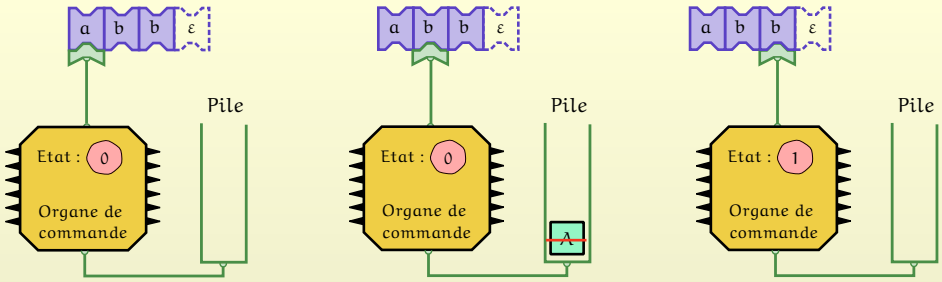


FIGURE 4.3 – Analyse du mot abb

4. Le mot $\varepsilon : (0, \varepsilon, \triangleright) \rightarrow (2, \varepsilon, \triangleright)$. Le mot est accepté. Schématiquement, l'analyse se fait selon la figure 4.4.

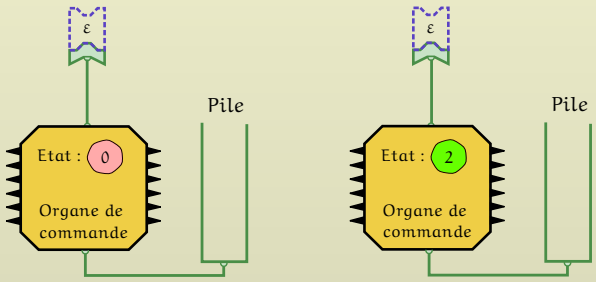


FIGURE 4.4 – Analyse du mot ε

1..1 Les automates à pile et le déterminisme

Comme nous l'avons signalé dans le chapitre des automates à états finis, la notion du déterminisme n'est pas propre à ceux-là. Elle est également présente dans le paradigme des automates à pile. On peut donc définir un automate à pile déterministe par :

Définition 4.2 : Soit l'automate à pile défini par $A = (X, \Pi, Q, q_0, F, \delta)$. A est dit déterministe si $\forall q_i \in Q, \forall a \in (X \cup \{\varepsilon\}), \forall A \in \Pi$, il existe au plus une paire $(q_j, B) \in (Q \times \Pi^*)$ tel que $\delta(q_i, a, A) = (q_j, B)$. Pour les ε-transitions, on les autorise dans un automate déterministe uniquement pour passer à un état final.

En d'autres termes, un automate à pile non-déterministe possède plusieurs actions à entreprendre lorsqu'il se trouve dans une situation déterminée. L'analyse se fait donc en testant toutes les possibilités jusqu'à trouver celle permettant d'accepter le mot.

Exemple 4.2 : automate à pile déterministe

Considérons le langage suivant : wcw^R tel que $w \in (a|b)^*$. La construction d'un automate à pile est facile, son idée consiste à empiler tous les symboles avant le c et de les dépiler dans l'ordre après (l'état initial est 0 et l'état 2 est final) :

— $\delta(0, a, \triangleright) = (0, A\triangleright)$

— $\delta(0, b, \triangleright) = (0, B\triangleright)$
— $\delta(0, a, A) = (0, AA)$
— $\delta(0, a, B) = (0, AB)$
— $\delta(0, b, A) = (0, BA)$
— $\delta(0, b, B) = (0, BB)$
— $\delta(0, c, A) = (1, A)$
— $\delta(0, c, B) = (1, B)$
— $\delta(0, c, \triangleright) = (1, \triangleright)$
— $\delta(1, a, A) = (1, \varepsilon)$
— $\delta(1, b, B) = (1, \varepsilon)$
— $\delta(1, \varepsilon, \triangleright) = (2, \triangleright)$

Exemple 4.3 : automate à pile non-déterministe

Considérons maintenant le langage ww^R tel que $w \in (a|b)^*$, les mots de ce langage sont des palindromes mais on ne sait pas quand est-ce qu'il faut arrêter d'empiler et procéder au dépilement. Il faut donc supposer que chaque symbole lu représente le dernier symbole de w, l'utiliser pour commencer à dépiler et continuer l'analyse, si cela ne marche pas il faut donc revenir empiler, et ainsi de suite (l'automate est alors non-déterministe) :

— $\delta(0, a, \triangleright) = (0, A\triangleright)$
— $\delta(0, b, \triangleright) = (0, B\triangleright)$
— $\delta(0, a, A) = (0, AA)$
— $\delta(0, a, B) = (0, AB)$
— $\delta(0, b, A) = (0, BA)$
— $\delta(0, b, B) = (0, BB)$
— $\delta(0, a, A) = (1, \varepsilon)$
— $\delta(0, b, B) = (1, \varepsilon)$
— $\delta(1, a, A) = (1, \varepsilon)$
— $\delta(1, b, B) = (1, \varepsilon)$
— $\delta(1, \varepsilon, \triangleright) = (2, \triangleright)$

Malheureusement, nous ne pouvons pas transformer tout automate à pile non-déterministe en un automate déterministe. En effet, la classe des langages acceptés par des automates à pile non-déterministes est beaucoup plus importante que celle des langages acceptés par des automates déterministes. Si L_{DET} est l'ensemble des langages acceptés par des automates à pile déterministes et L_{NDET} est l'ensemble des langages acceptés par des automates à pile non-déterministes (en fait c'est l'ensemble de tous les langages algébriques), alors :

$$L_{DET} \subset L_{NDET}$$

Il est important à noter ici que tout langage algébrique déterministe peut également être accepté par un automate à pile non-déterministe.

En pratique, on ne s'intéresse pas vraiment aux langages non-déterministes (c'est-à-dire, ne pouvant être accepté qu'avec un automate à pile non-déterministe) du fait que le temps d'analyse peut être exponentiel en fonction de la taille du mot à analyser (pour un automate déterministe, le temps d'analyse est linéaire en fonction de la taille du mot à analyser).

Nous allons à présent nous intéresser aux grammaires qui génèrent les langages algébriques puisque c'est la forme de ces grammaires qui nous permettra de construire des automates à pile (notamment grâce à l'analyse descendante ou ascendante en théorie de compilation).

2. Les grammaires hors-contexte

Nous avons déjà évoqué ce type de grammaires dans le premier chapitre lorsque nous avons présenté la hiérarchie de Chomsky. Rappelons-le quand même :

Définition 4.3 : (Rappel) Soit $G = (V, N, S, R)$ une grammaire quelconque. G est dite hors-contexte ou de type de 2 si tous les règles de production sont de la forme : $\alpha \to \beta$ tel que $\alpha \in N$ et $\beta \in (V+N)^*$.

Un langage généré par une grammaire hors-contexte est dit langage algébrique. Notons que nous nous intéressons, en particulier, à ce type de langages (et par conséquence à ce type de grammaires) du fait que la plupart des langages de programmation sont considérés comme algébriques (en réalité, ces langages sont contextuels mais on préfère les considérer comme algébriques afin de maîtriser la complexité d'analyse des mots).

Exemple 4.4 : grammaire hors-contexte

(Rappel) Soit la grammaire $G = (\{a, b\}, \{S\}, S, R)$ générant le langage $\{a^n b^n \mid n \geqslant 0\}$. R comporte les règles : $S \to aSb|\varepsilon$. D'après la définition, cette grammaire est hors-contexte et le langage $\{a^n b^n | n \geqslant 0\}$ est algébrique.

2..1 Arbre de dérivation

Vu la forme particulière des grammaires hors-contextes (présence d'un seul symbole non-terminal à gauche), il est possible de construire un arbre de dérivation pour un mot généré.

Définition 4.4 : (Rappel) Étant donné une grammaire $G = (V, N, S, R)$, les arbres de syntaxe de G sont des arbres dont les nœuds internes sont étiquetés par des symboles de N, les feuilles étiquetés par des symboles de V, tels que : si le nœud p apparaît dans l'arbre et si la règle $p \to a_1 ... a_n$ (a_i terminal ou non-terminal) est utilisée dans la dérivation, alors le nœud p possède n fils correspondant aux symboles a_i.

Si l'arbre de syntaxe a comme racine S, alors il est dit arbre de dérivation du mot u tel que u est le mot obtenu en prenant les feuilles de l'arbre dans le sens gauche→droite et bas→haut.

Exemple 4.5 : arbre de dérivation

Reprenons l'exemple précédent, le mot aabb est généré par cette grammaire par la chaîne : $S \rightarrow aSb \rightarrow aaSbb \rightarrow aa\varepsilon bb = aabb$. L'arbre de dérivation est donnée par la figure 4.5.

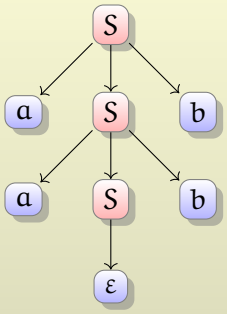


FIGURE 4.5 – Exemple d'un arbre de dérivation

2..2 Notion d'ambiguïté

Nous avons déjà évoqué la notion de l'ambiguïté lorsque nous avons présenté les expressions régulières. Nous avons, alors, défini une expression régulière ambiguë comme étant une expression régulière pouvant *coller* à un mot de plusieurs manières.

Par analogie, nous définissons la notion de l'ambiguïté des grammaires. Une grammaire est dite ambiguë si elle peut générer au moins un mot de plusieurs manières. En d'autres termes, si on peut trouver un mot généré par la grammaire et possédant au moins deux arbres de dérivation, alors on dit que la grammaire est ambiguë. Notons que la notion de l'ambiguïté n'a rien à avoir avec celle du non-déterminisme. Par exemple, la grammaire $G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSa|bSb|\varepsilon\})$ génère les mots ww^R tel que $w \in (a|b)^*$ (elle n'est pas ambiguë). Bien qu'il n'existe aucun automate à pile déterministe acceptant les mots de ce langage, tout mot du langage ne possède qu'un seul arbre de dérivation. Il est à noter également que certains langages algébriques ne peuvent être générés que par des grammaires ambiguës.

L'ambiguïté de certaines grammaires peut être levée comme le montre l'exemple suivant :

Exemple 4.6 : lever l'ambiguïté

Soit la grammaire $G = (\{0, 1, +, *\}, \{E\}, E, \{E \rightarrow E + E|E * E|(E)|0|1\})$. Cette grammaire est ambiguë car le mot 1+1*0 possède deux arbres de dérivation (figures 4.6 et 4.7). Or ceci pose un problème lors de l'évaluation de l'expression (précisons que l'évaluation se fait toujours de gauche à droite et bas en haut)[a]. Le premier arbre évalue l'expression comme étant 1+(1*0) ce qui donne 1. Selon le deuxième arbre, l'expression est évaluée comme étant (1+1)*0 ce qui donne 0 ! Or, aucune information dans la grammaire ne permet de préférer l'une ou l'autre forme.

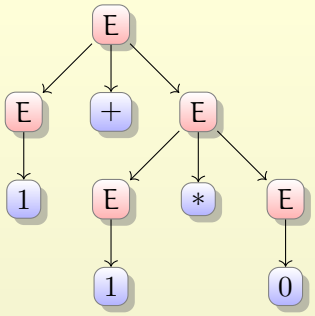


FIGURE 4.6 – Un premier arbre de dérivation

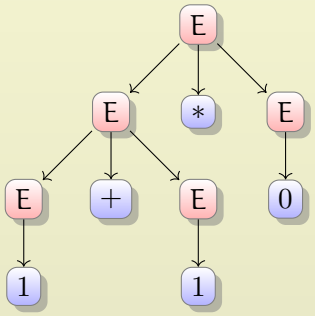


FIGURE 4.7 – Deuxième arbre de dérivation

D'une manière générale, pour lever l'ambiguïté d'une grammaire, il n'y a pas de méthodes qui fonctionne à tous les coups. De plus, il faut savoir que le problème de décider si une grammaire est ambiguë est non décidable : il n'existe aucun algorithme prenant une grammaire hors-contexte en entrée et décide correctement si elle est ambiguë ou non.
L'idée de lever l'ambiguïté consiste généralement à introduire une hypothèse supplémentaire (ce qui va changer la grammaire dans certains cas) en espérant que le langage généré reste le même. Par exemple, la grammaire $G' = (\{+, *, 0, 1\}, \{E, T, F\}, E, \{E \to E + T|T, T \to T * F|F, F \to (E)|0|1\})$ génère le même langage que G mais a l'avantage de ne pas être ambiguë. La transformation introduite consiste à donner une priorité à l'opérateur * par rapport à l'opérateur + et une associativité à gauche des opérateurs + et *.

a. Nous considérons le contexte de l'algèbre de Boole

2..3 Équivalence des grammaires hors-contextes et les automates à pile

Le théorème suivant établit l'équivalence entre les grammaires hors-contextes et les automates à pile.

Théorème 4.1

Pour tout langage généré par une grammaire hors-contexte, il existe un automate à pile (déterministe ou non) qui l'accepte. Réciproquement, pour tout langage accepté par un automate à pile, il existe une grammaire hors-contexte qui le génère.

Dans le paradigme des langages algébriques, il est plus intéressant de s'intéresser aux grammaires (rappelons qu'il n'existe pas d'expression régulière ici !). Ceci est dû à l'existence de nombreux algorithmes et outils traitant plutôt des formes particulières de grammaires (Chomsky, Greibach) cherchant ainsi à faciliter la construction des arbres de dérivation.

3. Simplification des grammaires hors-contextes

3..1 Les grammaires propres

Une grammaire hors-contexte (V, N, S, R) est dite *propre* si elle vérifie :

— $\forall A \to u \in R : u \neq \varepsilon$ ou $A = S$;

— $\forall A \to u \in R :$ S ne figure pas dans u ;

— $\forall A \to u \in R : u \notin N$;

— Tous les non-terminaux sont utiles, c'est-à-dire qu'ils vérifient :

  — $\forall A \in N :$ A est atteignable depuis S : $\exists \alpha, \beta \in (N + V)^* : S \xrightarrow{*} \alpha A \beta$;

  — $\forall A \in N :$ A est productif : $\exists w \in V^* : A \xrightarrow{*} w$.

Il est toujours possible de trouver une grammaire propre pour toute grammaire hors-contexte. En effet, on procède comme suit :

1. Rajouter une nouvelle règle $S' \to S$ tel que S' est le nouvel axiome (pour éviter que l'axiome ne figure dans la partie droite d'une règle) ;
2. Éliminer les règles $A \to \varepsilon$:

   — Calculer l'ensemble $E = \{A \in N + \{S'\} | A \xrightarrow{*} \varepsilon\}$ (les non-terminaux qui peuvent produire ε) ;

   — Pour tout $A \in E$, la règle $B \to \alpha A \beta$ de R est remplacée par la nouvelle règle $B \to \alpha\beta$

   — Enlever les règles $A \to \varepsilon$

3. Eliminer les règles $A \xrightarrow{*} B$, on applique la procédure suivante sur R privée de $S' \to \varepsilon$:

   — Calculer toutes les paires (A, B) tel que $A \xrightarrow{*} B$

   — Pour chaque paire (A, B) trouvée

     — Pour chaque règle $B \to u_1|...|u_n$ rajouter la règle $A \to u_1|...|u_n$

   — Enlever toutes les règles $A \to B$

4. Supprimer tous les non-terminaux non-productifs
5. Supprimer tous les non-terminaux non-atteignables.

4. Les formes normales

4..1 La forme normale de Chomsky

Soit $G = (V, N, S, R)$ une grammaire hors-contexte. On dit que G est sous forme normale de Chomsky si les règles de G sont toutes de l'une des formes suivantes :

— $A \to BC$, $A \in N$, $B, C \in N - \{S\}$

— $A \to a$, $A \in N$, $a \in V$

— $S \to \varepsilon$

L'intérêt de la forme normale de Chomsky est que les arbres de dérivations sont des arbres binaires ce qui facilite l'application de pas mal d'algorithmes.

Il est toujours possible de transformer n'importe quelle grammaire hors-contexte pour qu'elle soit sous la forme normale de Chomsky. Notons d'abord que si la grammaire est propre, alors cela facilitera énormément la procédure de transformation. Par conséquent, on suppose ici que la grammaire a été rendue propre. Donc toutes les règles de $G = (V, N, S, R)$ sont sous l'une des formes suivantes :

— $S \to \varepsilon$

— $A \to w$, $w \in V^+$

— $A \to w$, $w \in ((N - \{S\}) + V)^*$

La deuxième forme peut être facilement transformée en $A \to BC$. En effet, si

$$w = au, u \in V^+$$

alors il suffit de remplacer la règle par les trois règles $A \to A_1A_2$, $A_1 \to a$ et $A_2 \to u$. Ensuite, il faudra transformer la dernière règle de manière récursive tant que $|u| > 1$.

Il reste alors à transformer la troisième forme. Supposons que :

$$w = w_1A_1w_2A_2...w_nA_nw_{n+1} \text{ avec } w_i \in V^* \text{ et } A_i \in (N - \{S\})$$

La procédure de transformation est très simple. Si $w_1 \neq \varepsilon$ alors il suffit de transformer cette règle en :

$$\begin{aligned} A &\to B_1B_2 \\ B_1 &\to w_1 \\ B_2 &\to A_1w_2A_2...w_nA_nw_{n+1} \end{aligned}$$

sinon, elle sera transformée en :

$$\begin{aligned} A &\to A_1B \\ B &\to w_2A_2...w_nA_nw_{n+1} \end{aligned}$$

Cette transformation est appliquée de manière récursive jusqu'à ce que toutes les règles soient des règles normalisées.

Exemple 4.7 : construction de la forme normale de Chomsky

Soit la grammaire dont les règles de production sont : $S \to aSbS|bSaS|\varepsilon$, on veut obtenir sa forme normalisée de Chomsky. On commence par la rendre propre, donc on crée un nouvel axiome et on rajoute la règle $S' \to S$ puis on applique les transformations citées plus haut. On obtient alors la grammaire suivante : $S' \to S, S \to aSb|abS|bSa|baS|ab|ba$. En éliminant les formes $A \to B$, on obtient alors la grammaire propre $S' \to aSb|abS|bSa|baS|ab|ba, S \to aSb|abS|bSa|baS|ab|ba$. La grammaire continue bien sûr à comporter la règle $S' \to \varepsilon$.
Nous allons, à présent, transformer juste les productions de S étant donné qu'elles sont les mêmes que celles de S'.

- Transformation de $S \to aSb$
  - $S \to AU$ (finale)
  - $A \to a$ (finale)
  - $U \to Sb$ qui sera transformée en :
    - $U \to SB$ (finale)
    - $B \to b$ (finale)
- Transformation de $S \to abS$
  - $S \to AX$ (finale)
  - $X \to bS$ qui sera transformée en :
    - $X \to BS$ (finale)
- Transformation de $S \to bSa$
  - $S \to BY$ (finale)
  - $Y \to Sa$ qui sera transformée en :
    - $Y \to SA$ (finale)
- Transformation de $S \to baS$
  - $S \to BZ$ (finale)
  - $Z \to aS$ qui sera transformée en :
    - $Z \to AS$ (finale)
- Transformation de $S \to ab$ (idem pour ba)
  - $S \to ab$ devient $S \to CD$ avec $C \to a$ et $D \to b$
  - $S \to ba$ devient $S \to DC$

4..2 La forme normale de Greibach

Soit $G = (V, N, S, R)$ une grammaire hors-contexte. On dit que G est sous la forme normale de Greibach si toutes ses règles sont de l'une des formes suivantes :

— $A \to aA_1A_2...A_n$, $a \in V$, $A_i \in N - \{S\}$

— $A \to a$, $a \in V$

— $S \to \varepsilon$

L'intérêt pratique de la mise sous forme normale de Greibach est qu'à chaque dérivation, on détermine un préfixe de plus en plus long formé uniquement de symboles terminaux. Cela permet de construire plus aisément des analyseurs permettant de retrouver l'arbre d'analyse associé à un mot généré. Cependant, la transformation d'une grammaire hors-contexte en une grammaire sous la forme normale de Greibach nécessite plus de travail et de raffinement de la grammaire. Nous choisissons de ne pas l'aborder dans ce cours.

5. Exercices de TD

Exercice 1

Construisez un automate à pile pour les langages suivants. Donnez à chaque fois un mot accepté et un autre qui ne l'est pas en montrant les étapes d'analyse :

1. $\{a^{2n}b^n | n > 0\}$
2. $\{a^p b^q | p, q \geqslant 0, p \neq q\}$
3. $\{a^p b^q | p + q = 2k, k \geqslant 0\}$
4. $\{a^p b^q a^q b^p | p, q \geqslant 0\}$
5. $\{wc^n | w \in (a|b)^*, |w| = n, n \geqslant 0\}$
6. Les mots sur $\{a, b\}$ tels que tout préfixe contient plus de a que de b.

Exercice 2

Construisez un automate à pile des langages suivants. Dites à chaque fois s'il s'agit d'un langage déterministe ou non, puis donnez une grammaire hors-contexte qui génère le langage :

1. $\{a^p b^q a^p | p, q \geqslant 0\}$
2. $\{a^n b^n | n > 0\} + \{a^{2n} b^n | n > 0\}$
3. Tous les mots sur $\{a, b\}$ ayant autant de a que de b
4. $\{a^n b^n | n > 0\} + \{a^m b^n | m, n > 0\}$

Exercice 3

1. Soit la grammaire $G = (\{\neg, f, t, \wedge, (,)\}, \{S\}, S, \{S \to S \wedge S | \neg S | (S) | t | f\})$.
   (a) Montrez que les mots $f \wedge t$, $\neg t \wedge t$ et $t \wedge t \wedge \neg f$ sont générés par G. Montrez que G est ambiguë.
   (b) Quel est le type du langage généré par G ? pourquoi ?
   (c) Si on attache G au contexte des expressions booléennes, dites en quoi l'ambiguïté de G dérange.
   (d) Donnez une forme propre de G, ainsi que sa forme normale de Chomsky.
2. Soit la grammaire $G' = (\{\neg, f, t, \wedge, (,)\}, \{S, T\}, S, \{S \to T \wedge S | \neg S | T, T \to (S) | t | f\})$.
   (a) Montrez que les mots $f \wedge t$, $\neg t \wedge t$ et $t \wedge t \wedge \neg f$ sont générés par G'. G' est-elle ambiguë ?
   (b) Qu'a-t-on fait à G pour obtenir G' ? Est-ce que cette transformation est unique ?

Deuxième partie

Support de TP

Série 1

Prise en main

1. Éléments nécessaires pour la réalisation des exercices

Afin de réaliser les différents activités de cette série de TP, il est indispensable de d'avoir :

— Connaissance de certains éléments du langage Python :
  — Syntaxe de base du langage : indentation, affectations, structures de contrôle (`if` , `while` , `for` , ...), fonctions, etc.
— Des notions générales en algorithmique.
— A noter que la série présente uniquement les fonctions de bases. Elle ne peut, en aucun cas, se substituer de la documentation officielle du langage.

Eléments à apprendre : (exercice 1) les listes en Python

→ Une liste est définie par la syntaxe : `[...]`

◇ Exemple : `my_list=[1,2,5,6]` , `z=[]`

◇ Une liste peut contenir n'importe quel type de données (même hétérogènes). Elle peut même contenir des listes : `[1,2,[3,4],4.5]`

→ L'indexation se fait par `list[index]` :

◇ Exemple : `my_list[2]` renvoie 5

◇ Mise à jour : `my_list[1]=9`

→ On peut sélectionner un *slice* (une partie) d'une liste :

◇ Exemple : `my_list[0:2]` renvoie [1,2]

◇ Exemple : `my_list[1:]` renvoie [2,5,6]

◇ Exemple : `my_list[:3]` renvoie [1,2,5]

→ Opérations sur les listes :

◇ Longueur : `len(m_list)` renvoie 4

◇ Le nombre d'occurrence d'un élément : `my_list.count(e)`

◇ Concaténation : `[1,2]+[4,6]` donne [1,2,4,6]

◇ Répéter une liste : `[1,2]*3` donne [1,2,1,2,1,2]

◇ Vérifier si un élément figure dans une liste : `x in my_list` (résultat booléen)

◇ La fonction range : `list(range(0,5))` construit la liste `[0,1,2,3,4]` .

→ Programmer avec les listes :

◇ Parcourir les éléments d'une liste : `for e in my_list:print(e)`

◇ Parcourir les éléments par leurs indices :
```
for i in range(len(my_list)):print(my_list[i])
```

◇ Parcourir les indices et les éléments d'une liste :
```
for i,e in enumerate(my_list):print(i,"=>",e)
```

→ Compréhension : générer une liste d'éléments vérifiant un critère donné :

◇ Générer les nombres paires de my_list : `[ e for e my_list if e % 2==0 ]`

◇ Renvoyer tous les nombres multiples de 5 inférieurs à 100 : `[ e for e in range(101) if e % 5==0 ]`

Exercice 1

Écrire les fonctions suivantes (il faut écrire un programme testant les fonctions construites) :

Activité 1

Soustraire deux listes (garder les éléments de la première liste qui ne figurent pas dans la deuxième.

En-tête

```
def substract(L1,L2)
```

Test

```
substract([5,8,12,1,13,17],[6,7,12,5,17])
```

Activité 2

Répéter les éléments d'une liste en fonction d'une autre liste (exemple : L1=[6,1] et L2=[2,4], le résultat est : [[6,6],[1,1,1,1]]

En-tête

```
def repeat_by_list(L1,L2)
```

Test

```
repeat_by_list([6,1],[2,4])
```

Activité 3

Réarranger les éléments d'une liste : premier élément, dernier élément, deuxième élément, avant-dernier élément, troisième élément, ...

En-tête

```
def symmetric_browse(L)
```

Test

```
symmetric_browse([1,3,5,7,9,8,6,4,2])
```

Eléments à apprendre : (exercice 2) les chaînes de caractères en Python

→ La plupart des fonctions des listes sont applicables aux chaînes.

→ Une chaîne de caractères est une séquence de caractères (placées entre deux " ou deux '). On peut également avoir une chaînes sur plusieurs lignes avec les délimiteurs """.

◇ `my_str="abcdef"`

→ Une chaîne est une liste immuable

◇ `L'affectation s[2]="v"` provoque une erreur

→ Opérations sur les chaînes :

◇ La fonction `chr(...)` donne le caractère dont le code ASCII ou unicode est passé comme paramètre (exemple `chr(1585)`

◇ La fonction `ord(...)` donne le code ASCII ou unicode du caractère passé comme paramètre (exemple : `ord("a")`)

◇ Longueur : `len(my_str)` renvoie 5

◇ Le nombre d'occurrence d'un caractère : `my_str.count(e)`

◇ Concaténation : `"ab"+"cd"` donne "abcd"

◇ Répéter une chaîne : `"abc"*3` donne "abcabcabc"

◇ Vérifier si un élément figure dans une liste avec le test `x in my_list` (le résultat est booléen). Ceci fonctionne pour un caractère ou pour une chaîne (test de sous-chaînes).

◇ Recherche le premier (resp. le dernier) indice d'une sous-chaîne `my_str.find(sub_str)` (resp. `my_str.rfind(...)`)

◇ Joindre les éléments d'une liste pour en faire une chaîne `my_str.join(my_list)` (il faut que la éléments de la liste soient des chaînes de caractères).

◇ Formater une chaînes selon un format (option 1) : `str_format(a1,a2,...)` . Exemple : `"{}+{}={}".format(3,4,3+4)` .

◇ Formater une chaînes selon un format (option 2, disponible à partir de Python 3.6) : `f"..."` . Exemple : `f" {3}+{4}={3+4} "`.

◇ Diviser une chaînes de caractères par rapport à un séparateur : `my_str.split(sub)`

◇ Remplacer toutes les occurrences d'une chaînes par une autre chaîne : `my_str.replace(sub1,sub2)`

Exercice 2

Écrire les fonctions suivantes (il faut écrire un programme testant les fonctions construites) :

Activité 1

La fonction crange permettant de générer une chaîne contenant tous les caractères entre deux caractères (par exemple, `crange(["a","c"],["0","2"])` donne `"abc012"`).

En-tête

```
def crange(*cinterv)
```

Test

```
crange(["a","z"],["A","Z"],["0","9"])
```

Activité 2

Améliorer la fonction replace pour qu'elle prenne deux listes de chaînes de caractères [s1,s2,...], [t1,t2,...]. La fonction change s1 par t1, ensuite s2 par t2, etc.

En-tête

```
def replace_many(s,L1,L2)
```

Test

```
replace_many("aabc",["a","b","c"],["b","a","d"])
```

Activité 3

Implémenter une fonction replace conditionnelle. La chaîne sub1 est remplacée par sub3 uniquement si sub1 est suivie par sub2, sinon elle est remplacée par sub4.

En-tête

```
def conditional_replace(s,sub1,sub2,sub3,sub4)
```

Test

```
conditional_replace("abc=ded","=","f","=e","=f")
```

Série 2

Deuxième série : opérations sur les mots et les langages

1. Éléments nécessaires pour la réalisation des exercices

Afin de réaliser les différents activités de cette série de TP, il est indispensable de d'avoir :

— Connaissance du type `string` en Python.
  — Ne pas hésiter à consulter la documentation officielle
— Notions sur la récursivité
— Suivi le cours/TD du premier chapitre

Exercice 1

Écrire des les fonctions Python permettant de calculer :

Activité 1

La fonction `flatten_word(w_tuple)` transforme une liste de tuples représentant les symboles et leurs nombres en une chaîne de caractères. Par exemple, `flatten_word(("a",2),("b",4),("c",3))` renvoie la chaîne "aabbbbccc".

En-tête

```
def flatten_word(w_tuple)
```

Test

```
flatten_word(("a",2),("b",4),("a",2),("b",6),("c",1))
```

Activité 2

La fonction `unflatten_word(word)` effectue la transformation inverse de la fonction précédente. Par exemple, pour le mot "aabbbbccc", la fonction renvoie `("a",2),("b",4),("c",3)` .

En-tête

```
unflatten_word(word)
```

Test

```
unflatten_word("aabbbbccc")
```

Exercice 2

Écrire des les fonctions Python permettant de calculer :

Activité 1

La fonction `prefixes(word)` retourne la liste de tous les préfixes du mot *word*.

En-tête

```
def prefixes(word)
```

Test

```
prefixes("aaabbbccc")
```

Activité 2

La fonction `factors(word)` retourne la liste de tous les facteur du mot *word*.

En-tête

```
def factors(word)
```

Test

```
factors("aaabbbccc")
```

Activité 3

Une fonction permettant de calculer l'entrelacement d'un mot avec un symbole.

En-tête

```
def shuffle_symbol(word,c)
```

Test

```
shuffle_word("abba","c")
```

Activité 4

Une fonction permettant de calculer l'entrelacement de deux mots.

En-tête

```
def shuffle(w1,w2)
```

Test

```
shuffle_word("aaa","bbb")
```

Exercice 3

Pour chacun des langages suivants, écrire une fonction permettant d'accepter le langage. En d'autres termes, étant donné un mot du langage, la fonction doit décider si le mot appartient au langage ou non.

Activité 1

L_1 : les mots sur $\{a, b, c\}$ tel que le nombre de a est pair, le nombre de b est multiple de 3 et le nombre de c est multiple de 6. La lecture du mot se fait de gauche à droite uniquement (chaque symbole est lu une seule fois).

En-tête

```
def accept_l1(word)
```

Test

`accept_l1("ccabbaccbcc")` et `accept_l1("ccabbc")`

Activité 2

$L_2 = \{a^{2p}b^{3q}c^{2p+3q} | p, q \geqslant 0\}$. La lecture du mot se fait de gauche à droite uniquement (chaque symbole est lu une seule fois).

En-tête

```
def accept_l2(word)
```

Test

`accept_l2("aabbbccccccc")` et `accept_l2("aabbbcc")`

Activité 3

$L_3 = \{ww^R | w$ est un mot quelconque sur $\{"a", ..., "z"\}\}$. La lecture du mot se fait de gauche à droite uniquement (chaque symbole est lu une seule fois).

En-tête

```
def accept_l3(word)
```

Test

`accept_l3("abba")` et `accept_l3("aaabbcaa")`

Activité 4

$L_4 = \{wuw | w$ et u sont des mots quelconques sur $\{"a", ..., "z"\}\}$. w a une longueur non nulle.

En-tête

```
def accept_l4(word)
```

Test

`accept_l4("abcdabcd")` et `accept_l4("aababb")`

Activité 5

$L_5 = \{a^{2^n} | n \geqslant 0$. Il n'est pas autorisé de compter le nombre de a puis de le tester.

En-tête

```
def accept_l5(word)
```

Test

`accept_l5("aaaaaaaa")` et `accept_l5("aaaaaa")`

Activité 6

$L_6 = \{a^{n^2} | n \geqslant 0$. Il n'est pas autorisé de compter le nombre de a puis de le tester.

En-tête

```
def accept_l6(word)
```

Test

`accept_l6("aaaaaaaaa")` et `accept_l6("aaaaaa")`

Exercice 4

Soit la grammaire dont les règles de production sont données par : $S \rightarrow aSbS|\varepsilon$.

Activité 1

Écrire une fonction Python permettant de générer tous les mots dont la longueur est inférieure à une certaine limite.

En-tête

```
def generate(lim)
```

Test

```
generate(5)
```

Activité 2

Ecrire une deuxième fonction permettant de vérifier si un mot est généré par cette grammaire.

En-tête

```
def is_generated(w)
```

Test

```
is_generated("aabb") et is_generated("ababba")
```

Série 3

Troisième série : les AEF

1. Éléments nécessaires pour la réalisation des exercices

Afin de réaliser les différents activités de cette série de TP, il est indispensable de :

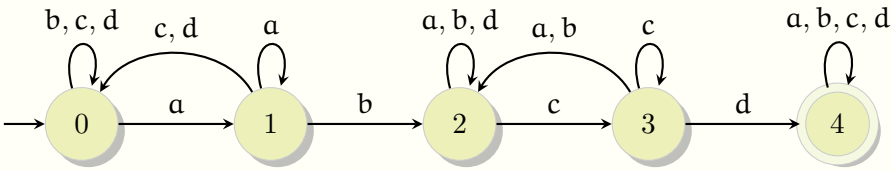
— Connaître le fonctionnement des AEF.

— Avoir des notions sur la récursivité

— Suivre le cours/TD du deuxième chapitre

Exercice 1

On considère le langage de tous les mots sur $\{a, b, c, d\}$ ayant le facteur ab et cd tel que l'occurrence de ab précède celle de cd. On considère alors l'AEF suivant :



Activité 1

Ecrire une fonction simulant cet automate. Pour chaque mot analysé, compter le nombre d'étapes nécessaires pour accepter ou rejeter les mots.

En-tête

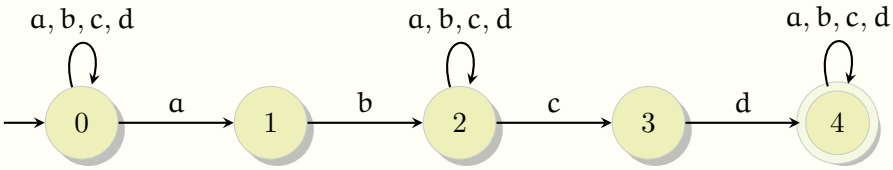
```
def simulate_fsa(word)
```

Test

`simulate_fsa("cdbaabbcdb")` et `simulate_fsa("acdbab")`

Exercice 2

On reprend le langage de l'exercice précédent. On utilise cette fois, l'AEF non-déterministe suivant :



Activité 2

Ecrire une fonction simulant cet automate. Pour chaque mot analysé, compter le nombre d'étapes nécessaires pour accepter ou rejeter les mots.

En-tête

```
def simulate_fsa(word)
```

Test

`simulate_ndet_fsa("cdbaabbcdb")` et `simulate_ndet_fsa("acdbab")`

Exercice 3

On s'intéresse aux chaînes de caractères contenant 0 et/ou 1 censées représenter des nombres binaires.

Activité 3

Ecrire une fonction permettant de calculer le reste de division sur 3 d'un nombre binaire représenté par une chaînes de caractères. Il n'est pas autorisé de transformer la chaîne en un entier ni d'utiliser les opérations d'addition, soustraction, multiplication, division ou modulo.

En-tête

```
def remainder_by_3(word)
```

Test

```
remainder_by_3("001001111")
remainder_by_3("111011")
remainder_by_3("101010")
```

Série 4

Quatrième série : les expressions régulières

1. Éléments nécessaires pour la réalisation des exercices

Afin de réaliser les différents activités de cette série de TP, il est indispensable de d'avoir :

— Syntaxe des expressions régulières (notation POSIX) expliquée dans le chapitre 3 du cours.

Exercice 1

Utilisez un éditeur de texte supportant les expressions régulières (Notepad++, GEdit, Kate, Visual Code Studio, etc). A noter que dans le cas de Visual Studio Code, le nommage de groupes dans la substitution se fait par $1, $2, etc. au lieu \1, \2, etc.

Activité 1

Chargez le fichier `data.txt` dans l'éditeur de votre choix. Trouvez comment rechercher les motifs suivants :

Tâches à réaliser

1. les mots commençant par une majuscule
2. les mots se trouvant à la fin d'une phrase
3. les mots placés entre parenthèses
4. les phrases ne contenant aucun chiffre
5. les mots se répétant deux fois de suite tel que les deux occurrences sont séparées par deux points (:)

Activité 2

Utilisez l'éditeur pour apporter les modifications suivantes.

Tâches à réaliser

1. les mots entre parenthèses seront placés entre crochets
2. chaque phrase se tient sur une ligne séparée
3. lorsqu'un mot (composé de lettres) est suivi par un nombre suivi lui-même par le premier mot, alors remplacer ce motif par le mot suivi par le nombre uniquement (supprimer la deuxième occurrence)

Eléments à apprendre : Module re en Python

Le module `re` implémente les expressions régulières et offre un certains nombre de fonctions utiles :

→ La fonction `match(reg_ex,string)` permet de chercher un motif correspondant `reg_ex` dans `string` . S'il y a une occurrence, alors un objet `re.Match` est retourné, sinon `None` .

◇ `m=re.match("a+","aabc")` retourne un objet `m` . En invoquant `m.group(0)` , on obtient `"aa"` . Dans le cas `m=re.match("ba+","aabc")` , on obtient `None`.

→ La fonction `findall(reg_ex,string)` permet de rendre une liste des chaînes de `string` correspondant à `reg_ex` .

◇ `re.findall("a+","aabaaaacbcaaba")` retourne `["aa","aaaa","aa","a"]` .

→ La fonction `fullmatch(reg_ex,string)` est similaire à `match` sauf que la correspondance se fait avec toute la chaîne et non pas une sous-chaîne seulement.

◇ L'appel de `re.fullmatch("a+","aabc")` retourne `None` . Cependant, l'appel `re.fullmatch("a+bc","aabc")` retourne un objet Match correspondant à la chaîne toute entière.

Exercice 2

On considère un fichier contenant des lignes dont la forme doit être : $entier_1$: $entier_2$,..., $entier_n$. Tous les entiers sont positifs (sans possibilité d'utiliser des signes).

Activité 1

Ecrivez une fonction gardant uniquement les lignes correctement écrites.

En-tête

```
def filter_file(file)
```

Test

```
def filter_file("data2.txt")
```
(le fichier sera fourni)

Activité 2

Ecrivez une fonction qui garde uniquement les lignes correctes dans le cas suivant : pour une ligne $entier_1$: $entier_2$,..., $entier_n$, il faut que $entier_1 = \sum_2^n entier_i$.

En-tête

```
def context_filter_file(file)
```

Test

```
def context_filter_file("data2.txt")
```

Eléments à apprendre : les dictionnaires en Python

→ Un dictionnaire est défini par : `{...}` (attention à la différence avec les ensembles).

◇ Exemple : `my_dict={"a' :5,"b" :2,"e" :" e" :5,"f" :1," z":[6,7]}`

→ Accès et mises à jour :

◇ Accès : `my_dict[key]` . Exemple : `my_dict["a"]` donne 5

◇ L'accès à une clé inexistante génère une erreur (`KeyError`). On peut tester l'existence d'une clé dans un dictionnaire par la condition : `key in my_dict`

◇ Mise à jour : `my_dict[key]=value`

→ Programmer avec les dictionnaires :

◇ Parcourir les clés d'un dictionnaire : `for k in my_dict:print(k)`

◇ Parcourir les valeurs d'un dictionnaire (option 1) :
```
for k in my_dict:print(my_list[k])
```

◇ Parcourir les valeurs d'un dictionnaire (option 2) :
```
for v in my_dict.values():print(v)
```

◇ Parcourir les clés et valeurs d'un dictionnaire :
```
for k,v in my_dict.item():print(k," :",v)
```

Exercice 3

On veut concevoir une fonction permettant de saisir les informations d'un enregistrement.

Activité 2

La saisie est guidée par un dictionnaire dont les clés représentent les champs de l'enregistrement. Les valeurs du dictionnaire contiennent deux clés : la première est `description` et représente une chaîne de caractères à afficher avant la saisie du champ, la deuxième est `forme` et représente une expression régulière modélisant les valeurs admissibles pour le champs. La fonction doit retourner le dictionnaire saisi.

En-tête

```
def input_record(record)
```

Test

Tester la fonction avec les champs suivants :

- **Code** : la description à afficher est `"Code de l'étudiant"` . Ce champ commence par `"UN"` suivi de 6 chiffres.
- **Nom** : la description à afficher est `"Nom de l'étudiant"` . Ce champ commence par une lettre suivie de lettres et/ou espaces.
- **Prénom** : la description à afficher est `"Prénom de l'étudiant"` . Ce champ commence par une lettre suivie de lettres et/ou espaces.
- **Date de naissance** : la description à afficher est `"Date de naissance"` . Ce champ spécifie le jour par un entier à un ou deux chiffres, le mois sur un ou deux chiffres et l'année sur quatre chiffres.
- **Email** : la description à afficher est `"Email de l'étudiant"` . Ce champ commence par une suite non-vide de caractères, suivie de `"@"` , suivi d'une suite non-vide de caractères. L'adresse doit contenir `"@"` une seule fois.
- **Note** : la description à afficher est `"Note de l'étudiant"` . Ce champ permet de saisir la note d'un étudiant sous la forme xx.xx, où x est un chiffre et seul le premier chiffre est obligatoire.

Références

— J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata theory, Languages and Computation, Addison-Wesley, 2001.

— A. Aho, J. Ullman, Concepts fondamentaux de l'informatique, Dunod, Paris, 1993.