Université Badji Mokhtar ANNABA
Faculté des sciences
Département d'informatique

Théorie des langages
Support de cours et TD

Réalisé par :
Dr. T. BENOUHIBA

(Dernière mise à jour : 15 avril 2011)

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Notions fondamentales en théorie des langages

1.1 Rappels sur la théorie des ensembles

1.1.1 Définitions

Définition 1 : Un ensemble est une collection d'objets sans répétition. Si un objet appartient à un ensemble A, on dit qu'il est élément de cet ensemble et l'on note $x \in A$. On distingue un ensemble particulier noté $\emptyset$ qui ne contient aucun élément.

Un ensemble est noté par {...} où les pointillés indiquent les éléments de l'ensemble ou une caractéristique de ses éléments. Il existe principalement trois moyens pour définir un ensemble :

- **Définition par extension** : cette définition consiste à donner tous les éléments d'un ensemble. Exemple : $\{0, 1, 2, 3, 4\}$. Cette définition n'est valable que si l'ensemble est fini.
- **Définition par compréhension** : cette définition consiste à définir les éléments d'un ensemble par les propriétés qui les définissent. En d'autres termes, on écrira $\{x|x$ vérifie une propriété $P(x)\}$. Exemple : $\{n \in \mathbb{N}|n \equiv 2 = 0\}$ définit les nombres entiers pairs. Bien évidemment, cette définition permet de représenter des ensembles finis ou infinis mais attention, quand même, au paradoxe de Russel (*l'ensemble qui contient tous les ensembles est-il un ensemble ou quoi exactement... ?*)
- **Définition par induction** : cette définition est très utile car elle permet de montrer facilement des propriétés vérifiées par les éléments. La définition par induction consiste à définir un ensemble par certains éléments triviaux (c'est-à-dire des éléments dont on sait, sans démonstration, qu'ils appartiennent à l'ensemble) et des règles d'induction permettant de retrouver d'autres éléments de l'ensemble en fonction de ceux connus jusqu'ici. La forme générale des règles d'induction est la suivante : $x \in A \Rightarrow f(x) \in A$ tel que f est un moyen permettant de construire d'autres éléments en fonction de l'argument. Exemple : l'ensemble des entiers peut être représenté comme suit : $\mathbb{N} = \{0; x \in \mathbb{N} \Rightarrow (x + 1) \in \mathbb{N}\}$.

  La définition par induction est utile car elle facilite la démonstration des théorèmes. Par exemple, tout le monde connaît le principe de démonstration par récurrence d'une propriété $P(n)$ $(n \in \mathbb{N})$:

– on montre d'abord que $P(0)$ est vérifiée
– on montre que si $P(n)$ est vérifiée alors $P(n+1)$ est vérifiée

Ce principe peut être facilement étendu à toute définition par induction. En effet, soit A un ensemble défini par induction par $\{triv(A); x \in A \Rightarrow f_1(x) \in A, ..., x \in A \Rightarrow f_n(x) \in A,\}$ où $triv(A)$ sont les éléments triviaux de A. Soit à démontrer qu'une propriété P est vérifiée par les éléments de A. On procède comme suit :
– pour chacun des éléments triviaux de A, s'assurer que P est vérifiée
– pour chaque règle d'induction $x \in A \Rightarrow f_i(x) \in A$, montrer que $P(x) \Rightarrow P(f_i(x))$ est vérifiée.

Ce type de démonstration est appelé démonstration par induction.

1.1.2 Opérations sur les ensembles

Nous pouvons définir plusieurs opérations sur les ensembles. Ces opérations permettent soit de comparer des ensembles soit de construire d'autres ensembles.

Comparaison des ensembles

Les ensembles peuvent être comparés de différentes manières selon les éléments qui les contiennent. Néanmoins, il existe une comparaison qui peut être appliquée à n'importe quels ensemble, il s'agit de l'inclusion. Formellement, on dit qu'un ensemble A est inclus ou égal à un autre ensemble B, et l'on note $A \subseteq B$, si $\forall x \in A$, $x \in B$. On dira alors que A est un sous-ensemble de B. Exemple : $\{a, b, c\} \subseteq \{a, b, c, d\}$. Nous pouvons définir une notion encore plus forte : l'égalité. Deux ensembles A et B sont égaux (c'est-à-dire qu'ils contiennent les mêmes éléments) si $A \subseteq B$ et $B \subseteq A$.

On peut alors annoncer certains résultats :
– $\emptyset \subseteq A$ quelque soit l'ensemble A
– $A \subseteq \emptyset \Rightarrow A = \emptyset$
– si A est défini par $\{x|P(x)$ est vérifiée$\}$ alors les éléments de tout sous-ensemble de A vérifient P.

Construction d'autres ensembles

Soient A et B deux sous-ensembles quelconques de Ω. Supposons que A (resp. B) soit défini par compréhension avec $\{x|P_A(x)\}$ (resp. $\{x|P_B(x)\}$). Supposons également que A (resp. B) soit défini par induction avec $\{triv(A); x \in A \Rightarrow f_A(x) \in A\}$ (resp. $\{triv(B); x \in B \Rightarrow f_B(x) \in B\}$). On définit alors les opérations suivantes :

– L'union, notée $A \cup B$, comporte tout élément appartenant à A *ou* B ($A \cup B = \{x|x \in A \vee x \in B\}$). Cette opération peut être définie par compréhension avec $\{x|P_A(x) \vee P_B(x)\}$. Elle peut même être définie par induction avec $\{triv(A) \cup triv(B); x \in A \Rightarrow f_A(x) \in A, x \in B \Rightarrow f_B(x) \in B\}$.

– L'intersection, notée $A \cap B$, comporte tout élément appartenant à A *et* B ($A \cap B = \{x|x \in A \wedge x \in B\}$). Elle peut être définie par compréhension avec $\{x|P_A(x) \wedge P_B(x)\}$. Malheureusement, on ne peut pas la définir par induction mais on note, quand même, que $triv(A) \cap triv(B)$ peuvent être considérés comme des éléments triviaux de $A \cap B$.

- La différence, notée $A - B$, comporte tout élément appartenant à A *et* qui n'appartient pas à B ($A - B = \{x | x \in A \wedge x \notin B\}$). Elle peut être définie par compréhension avec $\{x | P_A(x) \wedge \neg P_B(x)\}$. Malheureusement, on ne peut pas la définir par induction.

- Le complément, noté $\overline{A} = \Omega - A$. On peut le définir par compréhension avec $\{x | \neg P_A(x)\}$. Malheureusement, on ne peut pas le définir par induction.

- Le produit cartésien, noté $A \times B$, est l'ensemble des paires (a, b) telles que $a \in A$ et $b \in B$. On peut le définir par compréhension avec $\{(x, y) | P_A(x) \wedge P_B(y)\}$. Il peut également être défini par induction avec $\{triv(A) \times triv(B) | (x, y) \in A \times B \Rightarrow (f_A(x), f_B(y)) \in A \times B\}$.

- L'ensemble des parties de A, noté 2^A, est l'ensemble de tous les sous-ensembles de A.

Exemple 1 : $A = \{a, b\}$, $B = \{a, c\}$, $\Omega = \{a, b, c\}$:
- $A \cap B = \{a\}$;
- $A \cup B = \{a, b, c\}$;
- $A - B = \{b\}$;
- $\overline{A} = \{c\}$;
- $A \times B = \{(a, a), (a, b), (b, a), (b, c)\}$;
- $2^A = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

Mesures sur les ensembles

Ces opérations ont pour objectif de construire une valeur numérique caractérisant l'ensemble. La mesure la plus utilisée et celle de la cardinalité. Elle est notée $card(A)$ et représente le nombre d'éléments de A. Si l'ensemble est infini alors sa cardinalité est ∞.

Par ailleurs, on peut facilement montrer que $card(A \times B) = card(A).card(B)$ et que $card(2^A) = 2^{card(A)}$.

1.2 Théorie des langages

La théorie des langages utilise un certain nombre de concepts ainsi qu'une certaine terminologie. Nous allons d'abord définir certaines notions capitales qui sont les bases de cette théorie.

1.2.1 Notions sur les mots

Définition 2 : Un *symbole* est une entité abstraite (par abstraite, on veut dire que le symbole *abstrait* une idée). Les lettres et les chiffres sont des exemples de symboles fréquemment utilisés , mais des symboles graphiques peuvent également être considérés comme des symboles.

Exemple 2 :
- les chiffres 0,1,...,9 sont des symboles (ils servent à dénombrer) ;
- les lettres arabes, latines, cyrilliques, grecques, ... sont des symboles qui dénotent des voix ;

– les pictogrammes chinois sont des symboles qui dénotent des concepts, des idées, etc.

Définition 3 : Un *alphabet* est un ensemble de symboles. Il est également appelé le vocabulaire.

Définition 4 : Un *mot* (ou bien une chaîne) défini sur un alphabet A est une suite finie de symboles juxtaposés de A.

Exemple 3 :
– Le mot 1011 est défini sur l'alphabet $\{0, 1\}$
– Le mot 1.23 est défini sur l'alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$;

1.2.2 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 : $|\varepsilon| = 0$.

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

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

Propriété de la concaténation

Soient w, w_1 et w_2 trois mots définis sur l'alphabet A :
– $|w_1.w_2| = |w_1| + |w_2|$;
– $\forall a \in A : |w_1.w_2|_a = |w_1|_a + |w_2|_a$;
– $(w_1.w_2).w_3 = w_1.(w_2.w_3)$ (la concaténation est associative) ;
– $w.\varepsilon = \varepsilon.w = w$ (ε est un élément neutre pour la concaténation) ;

L'exposant

L'opération $w.w$ est notée par w^2. En généralisant, on note $w^n = \underbrace{w...w}_{n \text{ fois}}$. 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. On appelle mot miroir de w et on le note par w^R le mot obtenu en écrivant w l'envers, c'est-à-dire que $w^R = a_n...a_2a_1$. Il est donc facile de voir que $(w^R)^R = w$.

Certains mots, appelés palindromes, sont égaux à leur miroir. En d'autres termes, on lit la même chose dans les deux directions. Par ailleurs, on peut facilement vérifier que : $(u.v)^R = v^R.u^R$.

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.

Mots conjugués

Deux mots x et y sont dits conjugués s'il existe deux mots u et v tels que : $x = uv$ et $y = vu$.

1.2.4 Notions sur les langage

Définition 5 : Un langage est un ensemble (fini ou infini) de mots définis sur un alphabet donné. Évidemment, un langage peut être défini par extension, par compréhension ou par induction. Cette dernière manière porte une importance particulière pour les langages.

Exemple 4 :
- Langage des nombre binaires définies sur l'alphabet $\{0, 1\}$ (infini) ;
- Langage des mots de longueur 2 défini sur l'alphabet $\{a, b\}$=$\{aa, ab, ba, bb\}$;
- Langage Pascal (quel est son alphabet ?) ;
- Langue française (quel est son alphabet ?).

Opérations sur les langages

Les langages sont des ensembles. On peut alors leur appliquer n'importe quelle opération ensembliste. Cependant, vu la particularité des éléments constituant les langages (des mots), certaines de leurs opérations ne peuvent être appliquées qu'à eux.

Soient L, L_1 et L_2 trois langages définis sur l'alphabet X, on définit les opérations suivantes :
- L'union : notée par $+$ ou $|$ plutôt que $\cup$. $L_1 + L_2 = \{w | w \in L_1 \vee w \in L_2\}$;
- L'intersection : $L_1 \cap L_2 = \{w | w \in L_1 \wedge w \in L_2\}$;
- Le complément : $\overline{L} = \{$tous les mots w sur X$| w \notin L\}$;
- La concaténation (opération non ensembliste) : $L_1.L_2 = \{w | \exists u \in L_1, \exists v \in L_2 : w = uv\}$;
- Exposant (opération non ensembliste) : $L^n = \underbrace{L.L...L}_{n \text{ fois}} = \{w | \exists u_1, u_2, ...u_n \in L : w = u_1u_2...u_n\}$;

- Fermeture transitive de Kleene (opération non ensembliste) : notée $L^* = \bigcup_{i \geqslant 0} L^i$. 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^* ;
- Fermeture non transitive (opération non ensembliste) : $L^+ = \bigcup_{i>0} L^i$;
- Le langage miroir (opération non ensembliste) : $L^R = \{w | \exists u \in L : w = u^R\}$.

Propriétés des opérations sur les langages

Soit L, L_1, L_2, L_3 quatre langage définis sur l'alphabet A :
- $L^* = L^+ + \{\varepsilon\}$;
- $L_1.(L_2.L_3) = (L_1.L_2).L_3$;
- $L_1.(L_2 + L_3) = (L_1.L_2) + (L_1.L_3)$;
- $L.L \neq L$;
- $L_1.(L_2 \cap L_3) \neq (L_1 \cap L_2).(L_1 \cap L_3)$;
- $L_1.L_2 \neq L_2.L_1$.
- $(L^*)^* = L^*$;
- $L^*.L^* = L^*$;
- $L_1.(L_2.L_1)^* = (L_1.L_2)^*.L_1$;
- $(L_1 + L_2)^* = (L_1^* L_2^*)^*$;
- $L_1^* + L_2^* \neq (L_1 + L_2)^*$

1.3 Grammaire (système générateur de langage)

Nous avons dit plus haut que la définition des langages par induction revête une importance particulière. En effet, une telle définition permet de construire un langage infini avec un nombre restreint de concepts. De plus, elle facilite énormément les processus de démonstration des théorèmes grâce au principe de démonstration par induction (vu dans la section 1).

Cependant, l'avantage le plus important des définitions inductives est qu'elles permettent de comprendre les structures d'un langage ce qui facilite son appréhension. En effet, sans le contexte des langages, la règles d'induction possède un autre nom : la grammaire. Ainsi, une grammaire est un moyen permettant de décrire la construction des mots d'un langage. Elle a plusieurs avantages : elle permet de raisonner sur le langage, elle permet de construire des algorithmes efficaces pour le traitement des langages et enfin, et c'est ce qui est le plus important dans le vie courante, elle facilite l'apprentissage des langages.

Afin de mieux comprendre ce qu'est une grammaire et estimer réellement sa valeur, nous allons prendre un exemple.

1.3.1 Exemple introductif

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

- PHRASE $\rightarrow$ ARTICLE SUJET VERBE ARTICLE COMPLEMENT

- SUJET $\rightarrow$ "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 faisant 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

De même, à partir des phrases, on peut retrouver la catégorie de chaque mot en utilisant ces règles. 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 aux quelles on s'intéresse). Leur rôle consiste à construire les phrases du langage. Les symboles GARCON, 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, un exemple :
PHRASE → ARTICLE SUJET VERBE ARTICLE COMPLEMENT → "le" SUJET VERBE ARTICLE COMPLEMENT → "le" "garçon" SUJET VERBE ARTICLE COMPLEMENT → "le" "garçon" "voit" ARTICLE COMPLEMENT → "le" "garçon" "voit" "le" COMPLEMENT → "le" "garçon" "voit" "le" "livre"

1.3.2 Définition formelle des grammaires

Définition 6 : On appelle grammaire le quadruplet (V, N, X, R)
- 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 dits non-terminaux ou encore concepts ;
- S un non-terminal particulier appelé axiome (point de départ de la dérivation) ;
- R est un ensemble de règles de productions de la forme $g \to d$ tel que $g \in (V + N)^+$ et $d \in (V + N)^*$. La notation $g \to d$ est appelée une dérivation et signifie que g peut être remplacé par d.

Par convention, on utilisera les lettres majuscules pour les non-terminaux, et les lettres minuscules pour représenter les terminaux. Les règles de la forme $\varepsilon \to \alpha$ sont interdites. Pourquoi ?

Soit une suite de dérivations : $w_1 \to w_2 \to w_3 \to ... \to w_n$ alors on écrira : $w_1 \overset{*}{\to} w_n$. On dit alors qu'il y a une chaîne de dérivation qui mène de w_1 vers w_n.

Exemple 5 : Soit la grammaire $G = (\{a\}, \{S\}, S, \{S \to aS|\varepsilon\})$. On peut construire la chaîne de dérivation suivante : $S \to aS \to aaS \to aaaS$...

Définition 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 dérivation qui, partant de l'axiome

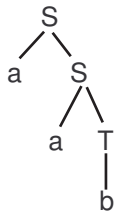
S, permet d'obtenir $u : S \xrightarrow{*} u$ (on appelle $S \xrightarrow{*} u$). Le langage de tous les mots générés par la grammaire G est noté $L(G)$. Notez qu'un expression, dérivée à partir de l'axiome, n'est considérée appartenant à $L(G)$ que si elle ne comporte aucun non-terminal.

Exemple 6 : Soit la grammaire $G = (\{a, b\}, \{S, T\}, S, \{S \to aS|aT, T \to bT|b\})$. Elle génère les mots abb et aab parce que $S \to aT \to abT \to abb$ et $S \to aS \to aaT \to aab$. On peut facilement voir alors que le langage généré par cette grammaire est : tous les mots sur $\{a, b\}$ de la forme $a^m b^n$ avec $m, n > 0$.

Définition 8 : Étant donnée une grammaire $G = (V, N, S, R)$, les arbres de syntaxe de G sont des arbres où les noeuds internes sont étiquetés par des symboles de N, et 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 syntaxique 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 7 : Soit la grammaire $G = (\{a, b\}, \{S, T\}, S, \{S \to aS|aT, T \to bT|b\})$. Elle génère le mot aab selon la chaîne de dérivation $S \to aS \to aaT \to aab$. Ce qui donne donc l'arbre syntaxique suivant :



1.3.3 Classification de Chomsky

La classification de Chomsky est un moyen permettant de maîtriser la complexité des langages ainsi que de celle des grammaires qui les génèrent. En effet, certains langage sont plutôt simples et peuvent être décrits par des grammaires facilement compréhensibles (comme l'exemple précédent). Cependant, il existe des langages d'une telle complexité que les grammaires qui les génèrent sont trop difficile à appréhender (par exemple, $\{a^n | n$ est premier$\}$).

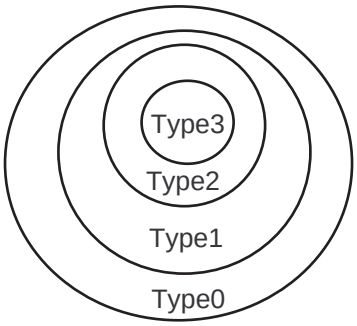
Il faut savoir que plus une grammaire est complexe à définir, plus les algorithmes qui la manipulent (si jamais ils existent) sont difficiles à apprendre et à mettre en œuvre. On arrive alors à une conclusion bien simple et pleine de bon sens : "utiliser des algorithmes simples pour des langages simples générés par des grammaires simples et réserver les algorithmes difficiles aux langages et grammaires complexes".

Une question clé apparaît alors : comment mesurer la complexité d'une grammaire ou d'un langage ? On peut, d'ores et déjà, écarter la cardinalité d'un langage puisqu'elle ne permet pas de distinguer langage simple d'un langage complexe. Noam Chomsky a eu le génie de

remarquer que la complexité d'une grammaire (et celle du langage aussi) dépend de la forme des règles de production de celle-ci. Chomsky a ainsi proposé quatre classes (hiérarchiques) de grammaires (et de langages) de sorte qu'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*[1]) : toutes les règles de production sont de la forme $g \to d$ où $g \in N$ et $d = aB$ tel que a appartient à V^* et B appartient à $N \cup \{\varepsilon\}$;
- Type 2 ou **grammaire hors-contexte** : toutes les règles de production sont de la forme $g \to d$ où $g \in N$ et $d \in (V + N)^*$;
- Type 1 ou **grammaire contextuelle** : toutes les règles sont de la forme $g \to d$ 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 $\alpha B \beta \to \alpha \omega \beta$ tel que $\alpha, \beta \in (V + N)^*$, $B \in X$ et $\omega \in (V + N)^*$;
- Type 0 : aucune restriction. Toutes les règles sont de la forme : $d \to g$, $g \in (V + N)^+$, $d \in (V + N)^*$

Il existe une relation d'inclusion entre les types de grammaires selon la figure suivante :



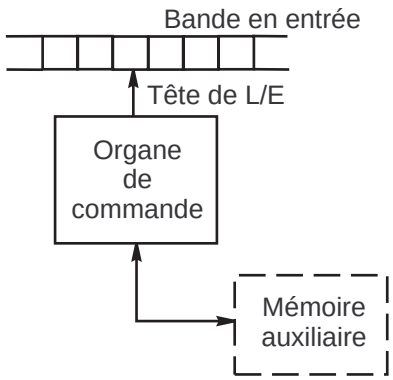
Le type retenu pour une grammaire est le plus petit qui satisfait les conditions. Pour trouver la classe d'un langage on procède cependant comme suit :
- Chercher une grammaire de type 3 qui le génère, si elle existe, le langage est de type 3 (ou **régulier**)
- Sinon, chercher une grammaire de type 2 qui le génère, si elle existe, le langage est de type 2 (ou **algébrique**)
- Sinon, chercher une grammaire de type 1 qui le génère, si elle existe, le langage est de type 1 (ou **contextuel**)
- Sinon, le langage est de type 0.

1.4 Les automates

Les grammaires représentent un moyen qui permet de *décrire* un langage d'une manière inductive. Elles montrent comment les mots du langage sont construits.

Considérons à présent un langage 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

1. Il existe également une forme régulière à gauche



dans la liste des mots de L (impossible à réaliser si le langage est infini). Si L est défini par compréhension, on peut alors vérifier si w respecte la propriété du langage. Si L est défini par une grammaire, on vérifie l'existence d'une chaîne de dérivation pour w, le cas échéant on conclut que $w \in L$.

Il existe en réalité un autre moyen permettant de répondre à 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 9 : Un automate est une machine abstraite qui permet de lire un mot et de répondre à la question : "un mot w appartient-il à un langage L ?" par oui ou non. Aucune garantie n'est cependant apportée concernant le temps de reconnaissance ou même la possibilité de le faire.

Un automate est composé de :
- Une bande en entrée finie ou infinie sur laquelle sera inscrit le mot à lire ;
- Un organe de commande qui permet de gérer un ensemble fini de pas d'exécution ;
- Eventuellement, une mémoire auxiliaire de stockage.

Formellement, un automate contient au minimum :
- Un alphabet pour les mots en 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.

1.4.1 Configuration d'un automate

Le fonctionnement d'un automate sur un mot se fait à travers un ensemble de configurations.

Définition 10 : On appelle configuration d'un automate en fonctionnement 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). Il existe deux configurations spéciales appelées configuration initiale et configuration finale.

Définition 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 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 reconnu par un automate si, à partir d'une configuration initiale, on arrive à une configuration finale à travers une succession de configurations intermédiaires . On dit aussi qu'un langage est reconnu par un automate lorsque tous les mots de ce langage sont reconnus par l'automate.

1.4.2 Classification des automates

Comme les grammaires, les automates peuvent être classés en 4 classes selon la hiérarchie de Chomsky. Il ne faut surtout pas oublier que la classification de Chomsky est une classification de complexité. Par conséquent, les automates reconnaissant les langages simples (par exemple ceux de la classe 3) sont plus simples que les automates reconnaissant les langages complexes. La classification de Chomsky pour les automates consiste à définir, pour chaque classe de langage, l'automate minimal permettant de répondre à la question "un mot *w* appartient-il à un langage ?"

Nous avons quatre classes d'automates :

- Type 3 ou **automate à états fini** (AEF) : il reconnaît 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.
- Type 2 ou **automate à pile** : il reconnaît les langages de type 2. Sa structure est similaire à l'AEF mais dispose en plus d'une mémoire organisée sous forme d'une pile infinie ;
- Type 1 ou **automate à bornes linéaires** (ABL) : il reconnaît 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 reconnaît les langages de type 0. Sa structure est la même que l'ABL mais la bande en entrée est infinie.

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

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

1.5 Exercices de TD

Exercice 1 : Déterminez l'alphabet pour chacun des langages suivants :
- Les nombres hexadécimaux ;
- Les nombres romains ;
- Les nombres réels en Pascal ;
- Les identificateurs en Pascal ;
- Le langage Pascal ;

Exercice 2 : Trouvez les langages correspondants aux définitions suivantes :
- Tous les mots sur $\{a, b, c\}$ de longueur 2 contenant un a ou un b mais pas les deux ;
- Tous les mots sur $\{a, b\}$ contenant au maximum deux a ou bien un b ;
- Tous les mots sur $\{a, b\}$ qui contiennent plus de a que de b ;
- Le langage L défini comme suit : $\{\varepsilon; u \in L \Rightarrow aaub \in L\}$

Exercice 3 : On note par $Conj(w)$ l'ensemble des conjugués de w. Trouvez $Conj(w)$ pour $w = \varepsilon$, $w = aacb$, $w = a^n b^n$.

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 : $L = \{ab, ac, \varepsilon\}$, $L = \{a^n b^m | n, m \geqslant 0\}$, $L = \{a^n b^m | n \geqslant m\}$.

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

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 : 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 8 : Donnez, sans démonstration, les langages générés par les grammaires suivantes. Dites, à chaque fois, de quel type s'agit-il ? :

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

Exercice 9 : Donnez les grammaires qui génèrent les langages suivants :

- Les nombres binaires ;
- Les mots sur $\{a, b\}$ qui contiennent le facteur aa

Exercice 10 : Soient G et G' deux grammaires qui génèrent respectivement les langages L et L'. Donnez une construction qui permet de trouver la grammaire de :

- $L.L'$;
- $L + L'$;
- L^* ;

Chapitre 2

Les automates à états finis (AEF)

Les AEF sont les plus simples des machines de reconnaissance de langages car ils ne comportent pas de mémoire. Par conséquent, les langages reconnus 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.

2.1 Généralités sur les AEF

Définition 13 : Un automate à états finis est une 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 du mot à reconnaître) ;
- Q est l'ensemble des états possibles ;
- q_0 est l'état initial ;
- F est l'ensemble des états finaux ($F \neq \emptyset, F \subseteq Q$). F représente l'ensemble des états d'acceptation ;
- δ est une fonction de transition qui permet de passer d'un état à un autre selon l'entrée en cours :

  $\delta : Q \times (X \cup \{\varepsilon\}) \mapsto 2^Q$

  $\delta(q_i, a) = \{q_{j_1}, ..., q_{j_k}\}$ ou $\emptyset$ ($\emptyset$ signifie que la configuration n'est pas prise en charge)

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, un mot est rejeté par un AEF 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 dit que l'automate est *bloqué*) :
- 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 reconnus par l'automate (notons le par $L(A)$) et le reste des mots (le complémentaire de L ou encore $X^* - L(A)$).

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

2.1.1 Représentation par table

La table possède autant de lignes qu'il y d'états dans l'automate de telle sorte que chaque ligne correspond à un état. Les colonnes correspondent aux différents symboles de l'alphabet. Si l'automate est dans l'état i et que le symbole a est le prochain à lire, alors l'entrée (i, a) de la table donne l'état auquel l'automate passera après avoir lu a. Notons la donnée de la table n'est suffisante pour définir tout l'automate puisqu'il faut préciser l'état initial et les états finaux.

Exemple 8 : L'automate qui reconnaît les mots de la forme $a^n b^m (n \geqslant 0, m > 0)$ est le suivant : $(\{a, b\}, \{0, 1\}, 0, \{1\}, \delta)$ tel que δ est donnée par la table :

État	a	b
0	0	1
1	-	1

Essayons à présent d'analyser certains mots par cet automate.
- le mot ab : $(0, a) \to (0, b) \to 1$, ab est accepté parce qu'il a été entièrement lu et l'état de sortie est état final ;
- le mot ba : $(0, b) \to (1, b)$, ba est rejeté car il n'a pas été entièrement lu.
- le mot a : $(0, a) \to 1$, a est rejeté car l'état de sortie n'est pas final bien que le mot soit entièrement lu.

2.1.2 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 un double rond. Le schéma suivant reprend l'automate précédent :

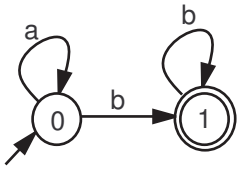


Figure 2.1 – L'automate de $a^n b^m (n \geqslant 0, m > 0)$

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 $a_1, ..., a_k$.

État	a	b
0	0,1	0
1	-	2
2	2	2

Table 2.1 – Table de transition de l'automate de la figure 2.2

2.2 Les automates et le déterminisme

2.2.1 Notion de déterminisme

En général, nous avons un certain nombre d'idées de ce qu'est un programme informatique. Parmi ces idées, on conçoit généralement que l'on connaît, à tout moment et avec précision, les étapes futures de l'exécution du programme. Cependant, cette vision est très naïve et ne prend pas en considération les environnements complexes dans lesquels peut s'exécuter le programme. Dans la plupart des cas, dû à la complexité des environnements, à leur caractère aléatoire et à l'absence d'une vision globale, la connaissance des étapes futures ne peut être qu'approximative. Le problème est encore plus accru si l'on utilise des algorithmes à caractère aléatoire. Dans ces situation, il arrive que l'exécution puisse avoir plusieurs évolution et que l'on ne puisse pas savoir *a priori* laquelle choisir. On dit alors que le programme est non déterministe (dans le cas contraire, on parle de programme déterministe).

Un AEF, étant une forme spéciale d'un programme informatique, peut être déterministe ou non. Par déterministe, on entend que l'on peut connnaître les états par lesquels passera l'automate avec une seule lecture du mot. 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 un état q_j tel que $\delta(q_i, a) = q_j$. Dans le cas inverse, l'automate doit choisir une action (un des états possibles) et la tester à terme, si la reconnaissance n'est pas possible l'automate doit tester les autres éventualités. On s'aperçoit alors que les automates non déterministes sont moins efficaces que les automates déterministes.

Cependant, en pratique, il est souvent plus facile de concevoir des automates non déterministes. Heureusement, cela ne nuit en rien au fonctionnement des automates. En effet, on verra plus tard un théorème énonçant que tout automate non déterministe peut être transformé en un automate déterministe.

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 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 14 : 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 9 : Soit le langage des mots définis sur $\{a, b\}$ possédant le facteur ab. La construction d'un AEF non déterministe est facile. La table 2.1 donne la fonction de transition (l'état initial est l'état 0 et l'état 2 est final). La figure 2.2 reprend le même automate :

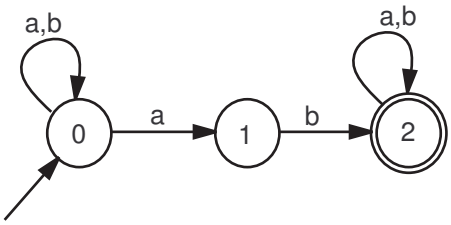


Figure 2.2 – L'automate des mots contenant le facteur ab

Exemple 10 : L'automate donné par la figure 2.3 reconnaît le même langage que le précédent mais en utilisant des ε-transitions.

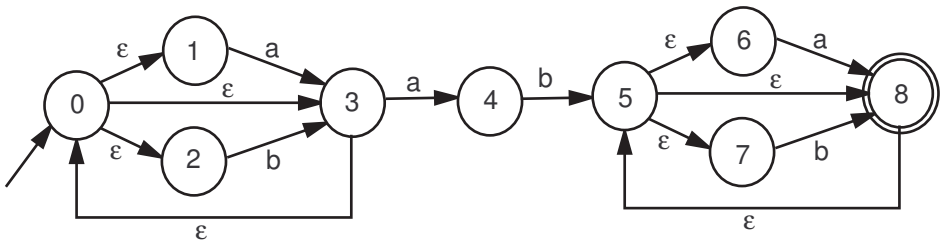


Figure 2.3 – L'automate reconnaissant les mots contenant le facteur ab (en ayant des ε-transitions)

2.2.2 Déterminisation d'un automate à états fini

Dans la section précédente, nous avons présenté la notion de déterminisme pour un automate[2]. Il est en général plus facile de concevoir des automates non déterministes surtout lorsque l'on utilise les *expressions régulières* pour dénoter les langages réguliers. Nous allons maintenant énoncer un théorème qui nous sera d'une grande utilité lorsqu'on travaille avec des AEF non déterministes (la démonstration de ce théorème sort du cadre de ce cours).

Théorème 1 : (appelé encore théorème de Rabin et Scott) Tout langage accepté par un AEF non déterministe est également 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 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 reconnaît 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. Notons que cette notion n'est pas limitées aux seuls AEF

2.2.3 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.

Algorithme : Déterminiser un AEF sans les ε-transitions

Principe : considérer des ensembles d'états plutôt que des états (dans l'algorithme suivant, chaque ensemble d'états représente un état du futur automate).

1- Partir de l'état initial $E^{(0)} = \{q_0\}$ (c'est l'état initial du nouvel automate) ;
2- Construire $E^{(1)}$ l'ensemble des états obtenus à partir de $E^{(0)}$ par la transition a :
$$E^{(1)} = \bigcup_{q' \in E^{(0)}} \delta(q', a)$$
3- Recommencer l'étape 2 pour toutes les transitions 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 deviennent 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 2.2. La table suivante illustre les étapes d'application de l'algorithme (les états en gras sont des états finaux) :

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

La figure 2.4 donne l'automate obtenu. Cet automate n'est pas évident à trouver mais grâce à l'algorithme de déterminisation, on peut le construire automatiquement.

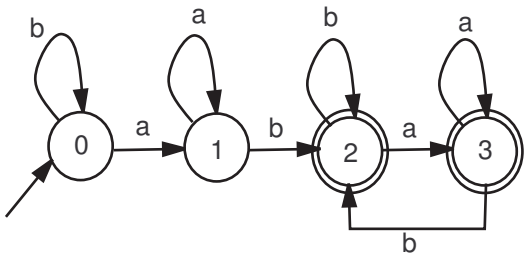


Figure 2.4 – L'automate déterministe qui reconnaît les mots ayant le facteur ab

2.2.4 Déterminisation avec les ε-transitions

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

Définition 15 : 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 étiqueté par le mot ε.

La construction des ε-transitions se fait donc d'une manière récursive. L'étudiant peut, en guise d'exercice, écrire l'algorithme permettant de construire un tel ensemble.

Exemple 11 : Considérons l'automate donné par la figure 2.3, calculons un ensemble d'ε-fermetures :
- ε-fermeture$(\{0\}) = \{0, 1, 2, 3\}$
- ε-fermeture$(\{1, 2\}) = \{1, 2\}$
- ε-fermeture$(\{3\}) = \{0, 1, 2, 3\}$

Algorithme : Déterminisation d'un AEF comportant des ε-transitions

Le principe de cet algorithme repose sur l'utilisation des ε-fermetures qui représenteront les états du nouvel automate.

1- Partir de l'ε-fermeture de l'état initial (elle représente le nouvel état initial) ;
2- Rajouter dans la table de transition toutes les ε-fermetures des nouveaux états produits avec leurs transitions ;
3- Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de nouvel état ;
4- Tous les ε-fermetures contenant au moins un état final du premier automate deviennent finaux ;
5- Renuméroter les états en tant qu'états simples.

Exemple 12 : Appliquons maintenant ce dernier algorithme à l'automate de la figure 2.3.

État	a	b
0,1,2,3	0,1,2,3,4	0,1,2,3
0,1,2,3,4	0,1,2,3,4	0,1,2,3,5,6,7,8
0,1,2,3,5,6,7,8	0,1,2,3,4,5,6,7,8	0,1,2,3,5,6,7,8
0,1,2,3,4,5,6,7,8	0,1,2,3,4,5,6,7,8	0,1,2,3,5,6,7,8

ce qui produit (surprise) l'automate suivant (c'est le même que celui donné par la figure 2.4) :

État	a	b
0	1	0
1	1	2
2	3	2
3	3	2

2.3 Minimisation d'un AEF déterministe

D'une manière général, moins un automate contient d'états, moins il prendra du temps à reconnaître un mot (par exemple, en déterminisant un AEF, on tombe le plus souvent sur un automate comportant plus d'états qu'il n'en faut). De plus, il prendra moins d'espace en mémoire s'il est question de le sauvegarder. Il est donc logique de vouloir minimiser ce temps en essayant de réduire le nombre d'états. Si on peut trouver une multitude d'automates pour reconnaître le même langage, on ne peut trouver néanmoins qu'un seul automate minimal reconnaissant le même langage. La minimisation s'effectue en éliminant les états dits inaccessibles et en *confondant* (ou fusionnant) les états reconnaissant le même langage.

2.3.1 Les états inaccessibles

Définition 16 : 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, c'est-à-dire qu'ils ne participeront jamais à la reconnaissance 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.

2.3.2 Les états β-équivalents

Définition 17 : Deux états q_i et q_j sont dits β-équivalents s'ils permettent d'atteindre les états finaux en utilisant 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 reconnaissent le même langage. La figure 2.5 montre un exemple d'états β-équivalents car l'état q_i atteint les états finaux via le mot a, de même pour l'état q_j. L'algorithme de minimisation consiste donc à fusionner simplement ces états pour n'en faire qu'un.

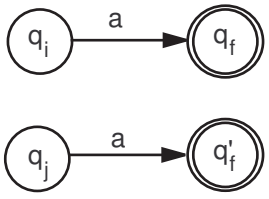


Figure 2.5 – Exemple d'états β-équivalents

Remarque 1 : La relation β-équivalence est une relation d'équivalence. De plus, $\forall x \in X$ (X étant l'alphabet), $\delta(q_i, x)$ et $\delta(q_j, x)$ sont également β-équivalents (puisque q_i et q_j reconnaissent le même langage). La relation β-équivalence est donc dite une relation de congruence.

Remarque 2 : 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 reconnaissent le même langage (ils seront fusionnés).

2.3.3 Minimiser un AEF

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

1. *Nettoyer* l'automate en éliminant les états inaccessibles ;
2. Regrouper les états congruents (appartenant à la même classe d'équivalence).

Algorithme : Regroupement des états congruents

Dans cet algorithme, chaque classe de congruence représentera un état dans l'automate minimal. **Il faut cependant noter que cet algorithme ne peut s'appliquer que si l'automate est déterministe.**

1- Faire deux classes : A contenant les états finaux et B contenant les états non finaux ;
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 un état d'arrivée dans la même classe ;
3- Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de classes à séparer ;

Les paramètres de l'automate minimal sont, alors, les suivants :
- Chaque classe de congruence est un état de l'automate minimal ;
- La classe qui contient l'ancien état initial devient 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 de congruence 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 13 : 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. Les étapes de détermination des classes de congruences sont les suivantes :

1. $A = \{1, 2\}, B = \{3, 4, 5, 6\}$;

2. $\delta(3, b) = 2 \in A$,$\delta(4, b) = 3 \in B$ ainsi il faut séparer 4 du reste de la classe B. Alors, on crée une classe C contenant l'état 4 ;
3. $\delta(3, b) = 2 \in A$, $\delta(5, b) = 6 \in A$ ainsi il faut séparer 5 du reste de la classe B. Mais inutile de créer une autre classe puisque $\delta(4, a) = 5 \in B$, $\delta(5, a) = 4 \in B$ et $\delta(4, b) = 3 \in B$ et $\delta(5, b) = 6 \in B$, il faut donc mettre 5 dans la classe C. $C = \{4, 5\}$ et $B = \{3, 6\}$;
4. Aucun autre changement n'est possible, alors on arrête l'algorithme.

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

Etat	a	b
A	A	C
B	B	A
C	C	A

Remarque 3 : L'automate obtenu est minimal et est unique, il ne peut plus être réduit. Si après réduction on obtient le même automate, alors cela signifie qu'il est déjà minimal.

2.4 Opérations sur les automates

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

2.4.1 Le complément

Soit A un automate déterministe défini par le quintuplet (X, Q, q_0, F, δ) reconnaissant le langage L. L'automate reconnaissant le langage inverse (c'est-à-dire $\overline{L}$ ou $X^* - L$) est défini par le quintuplet $(X, Q, q_0, Q - F, \delta)$ (en d'autres termes, les états finaux deviennent 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 14 : 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 la 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 c'est un état *improductif* étant donné qu'il ne peut pas générer de mots).

Soit maintenant le langage des mots définis sur $\{a, b, c\}$ contenant exactement deux a (figure 2.7). 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 $\delta(2, a) = E$.

Considérons à la fin l'automate de la figure 2.8. Les deux automates reconnaissent le mot a ce qui signifie que les deux automates ne reconnaissent pas des langages complémentaires.

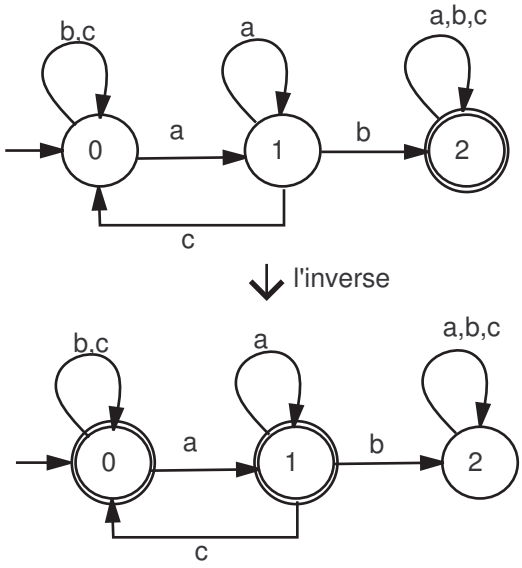


Figure 2.6 – Comment obtenir l'automate du langage complémentaire (d'un automate complet)

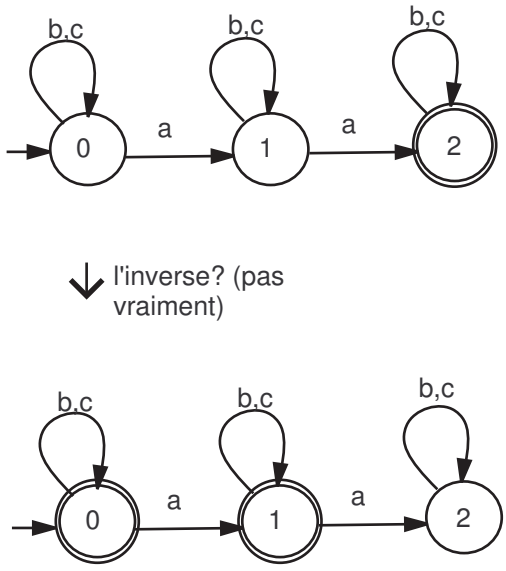


Figure 2.7 – Si l'automate n'est pas complet, on ne peut pas obtenir l'automate du langage inverse. L'automate obtenu reconnaît les mots contenant au plus 1 a.

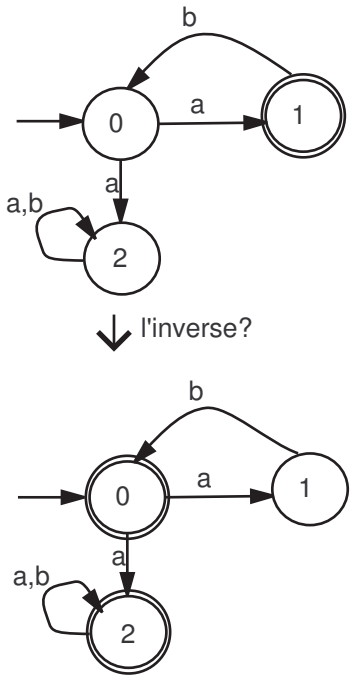


Figure 2.8 – Si l'automate n'est pas déterministe, on ne peut pas trouver l'automate du langage complémentaire.

Remarque 4 : 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 ?*

Remarque 5 : L'algorithme de déduction de l'automate du complémentaire peut en réalité être appliqué à un automate incomplet. Il suffit juste de le transformer en un automate complet et de lui appliquer, ensuite, l'algorithme. *Comment rendre un automate complet ?*

2.4.2 Produit d'automates

Définition 18 : Soit $A = (X, Q, q_0, F, \delta)$ 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 :

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

Cette définition permet de synchroniser la reconnaissance d'un mot par les deux automates. On pourra facilement démontrer que si w est un mot reconnu par A'' alors il l'est par

A, ceci est également le cas pour l'automate A'. Ainsi, si $L(A)$ est le langage reconnu par A et $L(A')$ est le langage reconnu par le langage A' alors l'automate A'' reconnaît l'intersection des deux langages : $L(A) \cap L(A')$.

Exemple 15 : Considérons la figure 2.9. L'automate (1) reconnaît les mots sur $\{a, b, c\}$ contenant deux a, l'automate (2) reconnaît les mots sur $\{a, b, c\}$ contenant deux b, l'automate (3) représente le produit des deux automates. Remarquons que dans l'automate (3), tout chemin qui mène 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.

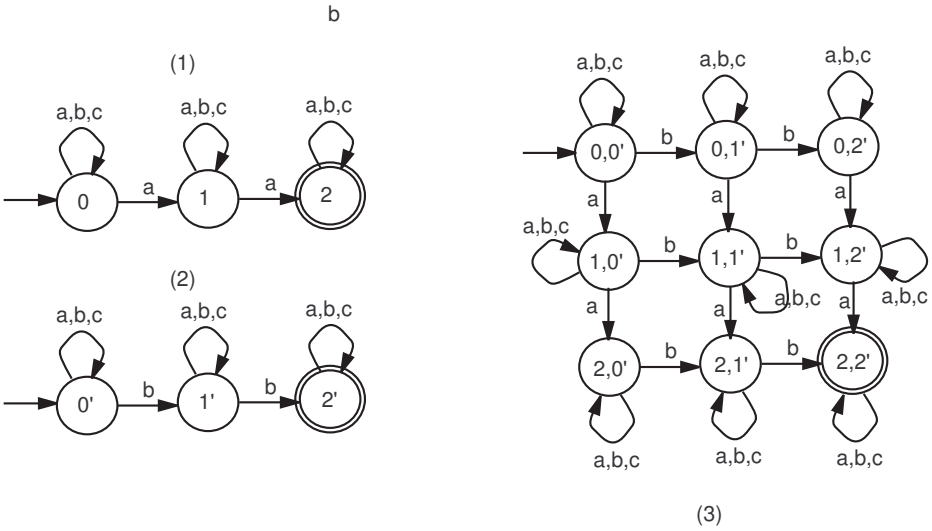


Figure 2.9 – Exemple d'intersection de produit d'automates

2.4.3 Le langage miroir

Soit $A = (X, Q, q_0, F, \delta)$ un automate reconnaissant le langage $L(A)$. L'automate qui reconnaît le langage $(L(A))^R$ est reconnu 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.

Minimisation par double déterminisation

L'algorithme permettent d'obtenir l'automate du langage miroir semble simple mais il permet, néanmoins, de réaliser des opérations très astucieuses. En particulier, il est bien utile pour minimiser un automate même si ce dernier n'est déterministe. Toutefois, pour que la construction fonctionne, il nous faudra modifier l'algorithme précédent comme suit : si l'automate de base possède plusieurs états finaux, alors l'automate du langage miroir possèdera

plusieurs états initiaux (on supprime l'action consistant à créer un nouvel état initial si l'automate possède plusieurs états finaux).

L'algorithme de minimisation d'un automate A est alors le suivant :

1. construire l'automate du langage miroir à partir de A, soit A^R l'automate obtenu ;
2. déterminiser A^R, soit A^R_d l'automate obtenu ;
3. construire l'automate du langage miroir à partir de A^R_d, soit A_{nd} l'automate obtenu ;
4. déterminiser A_{nd}, l'automate obtenu est l'automate minimal recherché.

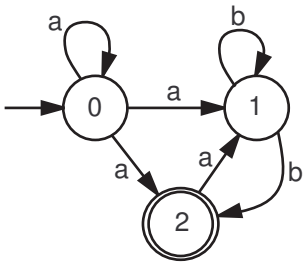
Par ailleurs, nous pouvons également énoncer les résultats suivants :

- Si L et M sont deux langages reconnus pas deux AEF, alors $L+M$ est également reconnu par un AEF ;
- Si L et M sont deux langages reconnus par deux AEF, alors $L.M$ est également reconnu par un AEF ;
- Si L est un langage reconnu par un AEF, alors L^* est également reconnu par un AEF ;

Le chapitre suivant traite des langages réguliers et des expressions régulières. On y abordera les opérations précédentes avec plus de détails étant donné qu'elles représentent les opérations de base lorsqu'on travaille avec les expressions régulières.

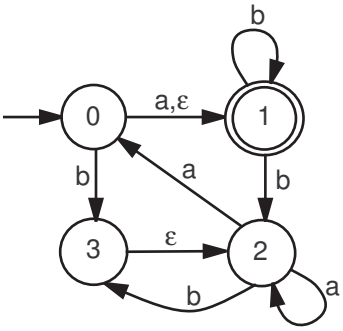
2.5 Exercices de TD

Exercice 1 : Soit l'automate suivant :



Déterminisez et minimisez-le si nécessaire.

Exercice 2 : Déterminisez et minimisez si nécessaire l'automate suivant :



Exercice 3 : Minimisez 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 reconnaissent les langages suivants :

1. Tous les mots sur $\{a, b, c\}$ commençant par c ;
2. Tous les mots sur $\{a, b, c\}$ dont le premier et le dernier symbole sont les mêmes ;
3. Tous les mots sur $\{a, b, c\}$ qui contiennent le facteur ab ou ac ;
4. Tous les mots sur $\{a, b, c\}$ qui ne contiennent pas le facteur aba ;

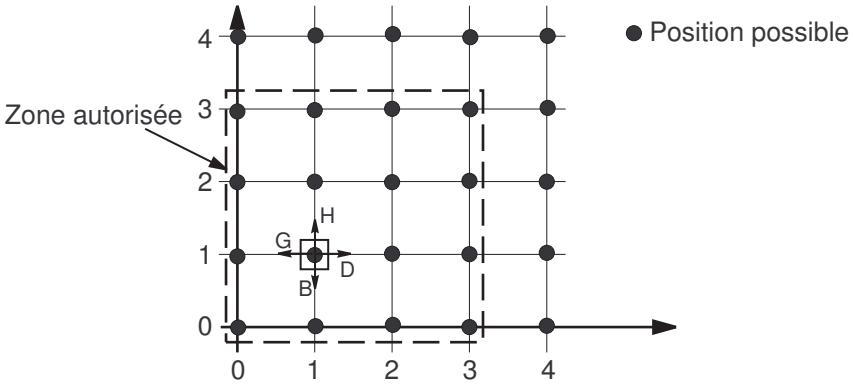
5. Tous les mots sur $\{a, b, c\}$ qui ne contiennent pas le facteur bca ;
6. Tous les mots sur $\{a, b, c\}$ qui ne contiennent ni le facteur aba ni le facteur bca ;
7. Tous les mots sur $\{a, b\}$ où tout facteur de longueur 3 contient au moins un a ;
8. Tous les mots sur $\{a, b\}$ de la forme $a^p b^q$ tel que $p + q$ est multiple de 2 ($(p, q > 0)$).

Exercice 5 : Donnez l'automate qui reconnaît tous les mots sur $\{a\}$ qui contiennent un nombre impair de a. Déduisez l'automate qui reconnaît le langage de tous les mots sur $\{a, b\}$ contenant un nombre impair de a et un nombre pair de b. Donnez alors l'automate reconnaissant tous les mots sur $\{a, b, c\}$ contenant un nombre impair de a et un nombre pair de b.

Exercice 6 : Considérons l'ensemble des nombres binaires, donnez les automates qui reconnaissent :
- Les nombres multiples de 2 ;
- Les nombres multiples de 4 ;
- Les nombres multiples de 2 mais pas 4.

Exercice 7 : Donnez l'automate qui reconnaît les nombres binaires multiples de 3. Donnez ensuite deux façons pour construire l'automate qui reconnaît les multiples de 6. Peut-on déduire l'automate des multiples de 9 à partir de l'automate des multiples de 3 ?

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 reconnaissant 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 reconnaissant les nombres réels en langage Pascal.

Chapitre 3

Les langages réguliers

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

3.1 Les expressions régulières E.R

Définition 19 : Soit X un alphabet quelconque ne contenant pas les symboles $\{*, +, |, ., (,)\}$. Une expression régulière est un mot défini sur l'alphabet $X \cup \{*, +, |, ., (,)\}$ 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 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 ait 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 convient à utiliser les priorités suivantes pour les différents opérateurs : 1)$*, +$, 2). et 3)|.

Exemple 16 :

1. a^* : dénote le langage régulier 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.

3.1.1 Utilisation des expressions régulières

Les expressions régulières sont largement utilisées en informatique. On les retrouve plus particulièrement dans les *shell* des systèmes d'exploitation où ils servent à indiquer un ensemble de fichiers sur lesquels on est appliqué 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 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.

L'utilisation la plus intéressante des expressions régulières est celle faite par UNIX. 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
\x	La valeur réelle de x (un caractère spécial)

Nous allons prendre des exemples intéressants :
- [^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})* les mots commençant par a où chaque a est suivi de quatre b au plus.

A présent, nous allons quitter le merveilleux monde de UNIX et revenir à ce cours. Nous allons, donc, reprendre la définition des expressions régulières données par la section 3.1.

3.1.2 Expressions régulières ambiguës

Définition 20 : 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 aab, 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.

3.1.3 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 de reconnaissance*. Par exemple, on peut décider que le facteur fixe soit le premier a du mot à reconnaître ce qui donne l'expression régulière : $b^*a(a|b)^*$. On peut également supposer que c'est le dernier a du mot à reconnaître ce qui donne l'expression régulière $(a|b)^*ab^*$. La série de TD propose quelques exercices dans ce sens.

3.2 Les langages réguliers, les grammaires et les automates à états finis

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

Théorème 2 : (Théorème 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 reconnus 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. Les sections suivantes expliquent comment passer d'une représentation à une autre.

3.2.1 Passage de l'automate vers l'expression régulière

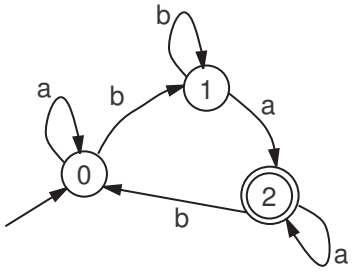
Soit $A = (X, Q, q_0, F, \delta)$ un automate à états fini quelconque. On note par L_i le langage reconnu par l'automate si son état initial était q_i. Par conséquent, trouver le langage reconnu par l'automate revient à trouver L_0 étant donné que la reconnaissance commence à partir

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 17 : Soit l'automate donné par la figure suivante :



Trouvons le langage reconnu 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 : $L_0 = a^*b^+a^+bL_0|a^*b^+a^+$ ce qui donne alors $L_0 = (a^*b^+a^+b)^*a^*b^+a^+$ (remarquez que l'on pouvait déduire la même chose à partir de l'automate...).

3.2.2 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.

Dérivée d'un langage

Définition 21 : Soit w un mot défini sur un alphabet X. On appelle dérivée de w par rapport à $u \in X^*$ le mot $v \in X^*$ tel que $w = uv$. On note cette opération par : $v = w||u$.

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

Exemple 18 :

- $L = \{1, 01, 11\}$, $L||1 = \{\varepsilon, 1\}$, $L||0 = \{1\}$, $L||00 = \emptyset$

Propriétés des dérivées

- $(\sum_{i=1}^{n} L_i)||u = \sum_{i=1}^{n}(L_i||u)$;
- $L.L'||u = (L||u).L' + f(L).(L'||u)$ tel que $f(L) = \{\varepsilon\}$ si $\varepsilon \in L$ et $f(L) = \emptyset$ sinon ;
- $(L^*)||u = (L||u)L^*$.

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 :

1. Dériver r par rapport à chaque symbole de X ;
2. Recommencer 1) pour chaque nouveau langage obtenu jusqu'à ce qu'il n'y ait plus de nouveaux langages ;
3. 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 ;
4. si $L_i||a = L_j$ alors 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 19 : Considérons le langage $(a|b)^*a(a|b)^*$. On sait que :
- $(a|b)||a = \varepsilon$ donc $(a|b)^*||a = (a|b)^*$

Commençons l'application de l'algorithme :
- $[(a|b)^*a(a|b)^*]||a = ((a|b)^*a(a|b)^*)|(a|b)^*$;
- $[(a|b)^*a(a|b)^*]||b = (a|b)^*a(a|b)^*$;
- $[((a|b)^*a(a|b)^*)|(a|b)^*]||a = ((a|b)^*a(a|b)^*)|(a|b)^*$;
- $[((a|b)^*a(a|b)^*)|(a|b)^*]||b = ((a|b)^*a(a|b)^*)|(a|b)^*$;

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) :

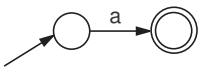
État	a	b
q_0	q_1	q_0
q_1	q_1	q_1

La méthode des dérivée ne permet pas seulement de construire l'AEF d'un langage, elle permet même de vérifier si un langage est régulier ou non. Pour cela, nous allons accepter le théorème suivant :

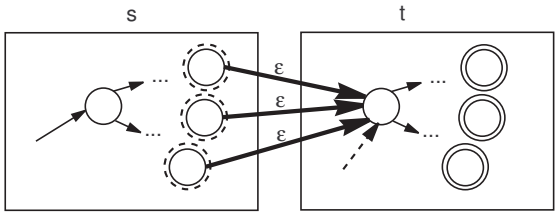
Théorème 3 :
Un langage est régulier si et seulement si le nombre de langages trouvés par la méthode des dérivées est fini. En d'autres termes, l'application de la méthode des dérivées à un langage non régulier produit un nombre infini des langages (ou encore d'états).

Méthode de Thompson

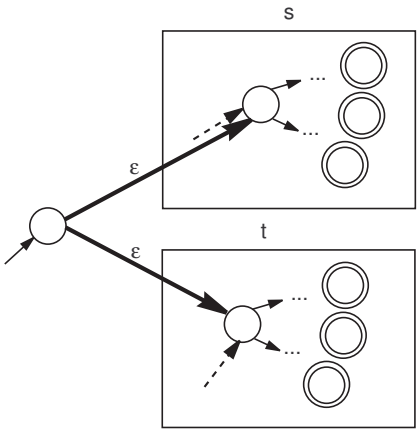
La méthode de Thompson permet de construire un automate en procédant à la décomposition de l'expression régulière selon les opérations utilisées. Soit r une E.R, alors l'algorithme à utiliser est le suivant :
- Si $r = a$ (un seul symbole) alors l'automate est le suivant :



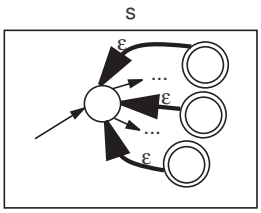
– Si $r = st$ alors il suffit de trouver l'automate A_s qui reconnaît s et l'automate A_t qui reconnaît t. Il faudra, ensuite relier chaque état final de A_s à l'état initial de A_t par une ε-transition. Les états finaux de A_s ne le sont plus et l'état initial de A_t ne l'est plus :



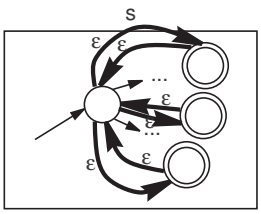
– Si $r = s|t$ alors il suffit de créer un nouvel état initial et le relier avec des ε-transitions aux états initiaux de A_r et A_s qui ne le sont plus :



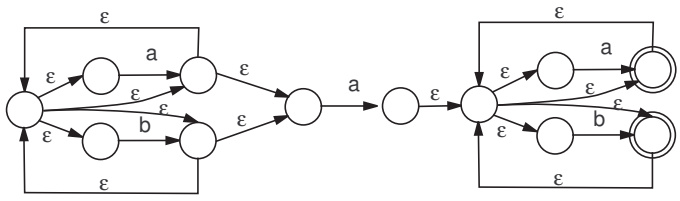
– Si $r = s^+$ alors il suffit de relier les états finaux de A_s par des ε-transitions à son état initial :



– Si $r = s^*$ alors il suffit de relier les états finaux de A_s par des ε-transitions à son état initial puis relier ce dernier à chacun des états finaux par des ε-transitions :



Exemple 20 : Soit à construire l'automate du langage $(a|b)^*a(a|b)^*$, la méthode de Thompson donne l'automate suivant :



3.2.3 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 symbole, de même que chaque dérivation dans une grammaire régulière *normalisée*. Soit $A = (X, Q, q_0, F, \delta)$ un AEF, la grammaire qui génère le langage reconnu 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 : $A \to \varepsilon$.

Il s'agit ici d'une grammaire régulière à droite. L'étudiant peut, en guise d'exercice, trouver une méthode permettant de construire la grammaire régulière à gauche. 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 :

- $A \to a$, $A \in N$, $a \in V$;
- $A \to aB$, $A, B \in N$, $a \in V$;
- $A \to \varepsilon$, $A \in N$

Voir la paragraphe suivant pour transformer une grammaire régulière en sa forme normale.

3.2.4 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 de production de la forme $A \to a$ ($a \in V$), alors créer un nouvel état final q_f et une transition partant de l'état q_A vers l'état q_f avec l'entrée a ;
5. Pour chaque règle $A \to aB$ alors créer une transition partant de q_A vers l'état q_B en utilisant l'entrée a ;
6. 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 à toutes les grammaires régulières. On peut néanmoins transformer toute grammaire régulière à la forme donnée ci-dessus. 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 $a \in V$;
2. Pour chaque règle de la forme $A \to w$ ($A \in N$ et $w \in V*$) tel que $|w| > 1$, créer un nouveau non-terminal B' et éclater la règle en deux : $A \to aB'$ et $B' \to u$ tel que $w = au$ et $a \in V$;
3. Recommencer 1) et 2) jusqu'à ce qu'il n'y ait plus de nouveaux non terminaux ;

Enfin, l'élimination des règles $A \to B$ ($A, B \in N$), on applique l'algorithme suivant (il faut bien entendu enlever toute règle de la forme $A \to A$) :

1. Pour chaque règle $A \to B$ tel que $B \to \beta_1|\beta_2|...|\beta_n$
   - Rajouter une règle $A \to \beta_1|\beta_2|...|\beta_n$
   - Supprimer la règle $A \to B$
2. Refaire l'étape 1) jusqu'à ce qu'il n'y ait plus de règles de la forme $A \to B$

Exemple 21 : 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 $T_1 \to bT_1$ et $T_1 \to b$;

3.3 Propriétés des langages réguliers

3.3.1 Stabilité par rapport aux opérations sur les langages

Les langages réguliers sont stables par rapport aux opérations de l'union, l'intersection, le complémentaire, 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 reconnus par les automates A_r et A_s. Étant donnée 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 de reconnaissance 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 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^+) ;
- L'union finie de langages réguliers représente un langage régulier. Ainsi, on peut dire que tout langage fini représente un langage régulier.
- L'union infinie de langages réguliers peut ou non être un langage régulier.

3.3.2 Lemme de la pompe

Dans cette section, nous présentons les critères permettant d'affirmer si un langage est régulier ou non. Avant toute discussion, on peut déjà annoncer que tout langage fini est un langage régulier. En effet, la construction d'un AEF reconnaissant ce type de langages est facile. L'étudiant peut en guise d'exercice trouver une méthode permettant de trouver l'AEF d'un langage fini quelconque, sa grammaire ainsi que l'E.R qui le représente.

A 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 suffisante et non nécessaire pour qu'un langage soit régulier. La démonstration de ce résultat sort du cadre du cours.

Proposition 2 : 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 :
- $w = xyz$;
- $|xy| \leqslant n$;
- $xy^iz \in L$ pour tout $i > 0$.

Il est à noter que cette condition est suffisante et non nécessaire. 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 22 : Soit le langage $L = a^kb^l$ (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 $w = a^kb^l$ $(k + l \geqslant 1)$. Si $k > 0$ alors il suffit de prendre $x = a^{k-1}, y = a$ et $z = b^l$, ainsi tout mot de la forme $xy^iz = a^{k+i-1}b^l$ 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 pomme.

Exemple 23 : Soit le langage $L = a^kb^k$ (avec $k \geqslant 0$)[3]. 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 $xy^iz \in L$. Considérons le mot $a^{n+1}b^{n+1}$, toute décomposition de ce mot produire 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 $xy^iz \in L$ pour tout $i > 0$ donc tout mot de la forme $a^ja^{il}a^{n+1-j-l}b^{n+1} = a^{(i-1)l+n+1}b^{n+1}$ appartient à L. Pour $i = 2$ la borne inférieure de

3. Attention, l'expression a^kb^k n'est pas régulière.

$(i-1)l$ 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.

Exemple 24 : 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.

Proposition 3 :

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 :
- $w = xyz$;
- $\forall u \in X^* : wv \in X^* \Leftrightarrow xy^i zu \in L$.

3.4 Exercices de TD

Exercice 1 : Donnez une description de chacune des E.R suivantes, puis donnez leurs équivalents en UNIX :

- $a(a|b)^*(b|\varepsilon)$;
- $(aaa)^*$;
- $(a|ab|abb)^*$
- $a(a|b)^*(aa|bb)^+$
- $(b|ab)^*$

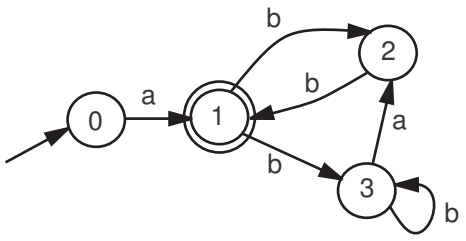
Exercice 2 : Donnez les expressions régulières UNIX pour les langages suivants :

- les identificateurs en Pascal ;
- les noms de fichiers en DOS ;
- les nombres entiers multiples de 5 ;
- les mots sur $\{a, b, c\}$ contenant le facteur a^{1000}.

Exercice 3 : Donnez une expression régulière pour chacun des langages suivants ainsi qu'une expression régulière en UNIX. Déduisez à chaque fois l'automate correspondant en utilisant la méthode de Thompson ou celle des dérivées :

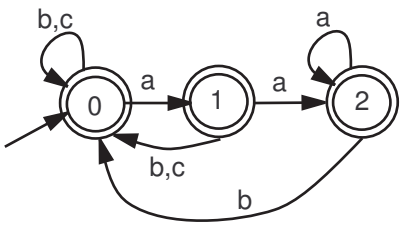
1. Tous les mots sur $\{a, b, c\}$ ne commençant pas par a ;
2. Tous les mots sur $\{a, b, c\}$ dont le premier et le dernier symbole sont les mêmes ;
3. Tous les mots sur $\{a, b, c\}$ contenant exactement trois a ;
4. Tous les mots sur $\{a, b, c\}$ contenant au moins trois a ;
5. Tous les mots sur $\{a, b, c\}$ contenant au plus trois a ;
6. Tous les mots sur $\{a, b, c\}$ ne contenant pas la facteur bc ;
7. Tous les mots sur $\{a, b, c\}$ ne contenant pas le facteur acc
8. Tous les entiers (en base dix) multiples de 5.

Exercice 4 : Soit l'automate suivant :



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

Exercice 5 : Soit l'automate suivant :



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

Exercice 6 : Soit le grammaire régulière : $G = (\{a, b\}, \{S, T\}, S, \{S \rightarrow aaS|bbS|bT|aT|\varepsilon, T \rightarrow bT|aT|aaS|bbS\})$. Trouvez l'automate reconnaissant le langage généré par cette grammaire puis en déduire 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}, $n \geqslant 0$;
- $a^n b^m$, $n, m \geqslant 0$ 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 ainsi que leurs équivalents en UNIX :

- 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 2 ;

Exercice 9 : Dites si les expressions rationnelles suivantes sont ambiguës. Si oui, levez l'ambiguïté :

- $(ab)^*(ba)^+$;
- $((aa)^+b^*)^*$;
- $(a|b)^*ab(a|b)^*$

Chapitre 4

Les langages algébriques

Malgré la panoplie d'algorithmes existants pour travailler sur les automates à états finis, ceux-là sont 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'étude en s'intéressant aux langages algébriques qui représentent la couche qui suit immédiatement celle des langages réguliers dans la hiérarchie de Chomsky. Remarquons, 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*.

4.1 Les automates à piles

Les automates à piles sont une extension des automates à états finis. Ils utilisent une (et une seule) pile pour reconnaître les mots en entrée.

Définition 22 : Un automate à pile est défini par le sextuple $A = (X, \Pi, Q, q_0, F, \delta)$ tel que :
- X est l'ensemble des symboles formant les mots en entrée (alphabet des mots à reconnaître) ;
- 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 ($F \neq \emptyset$, $F \subseteq Q$) ;
- δ est une fonction de transition permettant de passer d'un état à un autre :

  $\delta : Q \times (X \cup \{\varepsilon\}) \times \Pi \mapsto Q \times (\Pi - \{\triangleright\})^*$
  $\delta(q_i, a, b) = (q_j, c)$ 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

Par conséquent, tout automate à états fini 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.

Une configuration d'un automate à pile consiste donc à définir le triplet (q, a, b) tel que q est l'état actuel, a et le symbole actuellement en lecture et b est le sommet actuel de 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 m'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 avec 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 25 : L'automate reconnaissant 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 :

- $\delta(q_0, a, \triangleright) = (q_0, A\triangleright)$
- $\delta(q_0, a, A) = (q_0, AA)$
- $\delta(q_0, b, A) = (q_1, \varepsilon)$
- $\delta(q_1, b, A) = (q_1, \varepsilon)$
- $\delta(q_1, \varepsilon, \triangleright) = (q_2, \triangleright)$
- $\delta(q_0, \varepsilon, \triangleright) = (q_2, \triangleright)$

Détaillons la reconnaissance de quelques mots :

1. Le mot $aabb$: $(q_0, a, \triangleright) \rightarrow (q_0, a, A\triangleright) \rightarrow (q_0, b, AA\triangleright) \rightarrow (q_1, b, A\triangleright) \rightarrow (q_1, \varepsilon, \triangleright) \rightarrow (q_2, \varepsilon, \triangleright)$. Le mot est reconnu ;
2. Le mot aab : $(q_0, a, \triangleright) \rightarrow (q_0, a, A\triangleright) \rightarrow (q_0, b, AA\triangleright) \rightarrow (q_1, \varepsilon, A\triangleright)$. Le mot n'est pas reconnu car il n'y a plus de transitions possibles alors que l'état de l'automate n'est pas final
3. Le mot abb : $(q_0, a, \triangleright) \rightarrow (q_0, b, A\triangleright) \rightarrow (q_1, b, \triangleright)$. Le mot n'est pas reconnu car on n'arrive pas à lire tout le mot ;
4. Le mot ε : $(q_0, \varepsilon, \triangleright) \rightarrow (q_2, \varepsilon, \triangleright)$. Le mot est reconnu.

4.1.1 Les automates à piles 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 piles. On peut donc définir un automate à pile déterministe par :

Définition 23 : 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

$\delta(q_i, a, A) = (q_j, B)$.

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

Exemple 26 : 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 q_2 est final) :

- $\delta(q_0, a, \triangleright) = (q_0, A\triangleright)$
- $\delta(q_0, b, \triangleright) = (q_0, B\triangleright)$
-
- $\delta(q_0, a, A) = (q_0, AA)$
- $\delta(q_0, a, B) = (q_0, AB)$
- $\delta(q_0, b, A) = (q_0, BA)$
- $\delta(q_0, b, B) = (q_0, BB)$
-
- $\delta(q_0, c, A) = (q_1, A)$
- $\delta(q_0, c, B) = (q_1, B)$
- $\delta(q_0, c, \triangleright) = (q_1, \triangleright)$
-
- $\delta(q_1, a, A) = (q_1, \varepsilon)$
- $\delta(q_1, b, B) = (q_1, \varepsilon)$
- $\delta(q_1, \varepsilon, \triangleright) = (q_2, \triangleright\varepsilon)$

Considérons maintenant le langage ww^R tel que $w \in (a|b)^*$, les mots de ce langage sont palindromes mais on ne sait 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, le dépiler, continuer la reconnaissance si cela ne marche pas, il faut donc revenir empiler, et ainsi de suite :

- $\delta(q_0, a, \triangleright) = (q_0, A\triangleright)$
- $\delta(q_0, b, \triangleright) = (q_0, B\triangleright)$
-
- $\delta(q_0, a, A) = (q_0, AA)$
- $\delta(q_0, a, B) = (q_0, AB)$
- $\delta(q_0, b, A) = (q_0, BA)$
- $\delta(q_0, b, B) = (q_0, BB)$
-
- $\delta(q_0, a, A) = (q_1, \varepsilon)$
- $\delta(q_0, b, B) = (q_1, \varepsilon)$
- $\delta(q_0, a, B) = (q_1, \varepsilon)$
- $\delta(q_0, b, A) = (q_1, \varepsilon)$
-
- $\delta(q_1, a, A) = (q_1, \varepsilon)$
- $\delta(q_1, b, B) = (q_1, \varepsilon)$
- $\delta(q_1, \varepsilon, \triangleright) = (q_2, \triangleright\varepsilon)$

Malheureusement, nous ne pouvons pas transformer tout automate à piles non déterministe en un automate déterministe. En effet, la classe des langages reconnus par des automates

à piles non déterministes est beaucoup plus importantes que celle des langages reconnus par des automates déterministes. Si L_{DET} est l'ensemble des langages reconnus par des automates à piles déterministes et L_{NDET} est l'ensemble des langages reconnus par des automates à piles non déterministes, alors :

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

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.

4.2 Les grammaires hors-contextes

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 24 : 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 : $\alpha \rightarrow \beta$ tel que $\alpha \in N$ et $\beta \in (V + N)^*$.

Un langage généré par une grammaire hors-contexte est dit langage hors-contexte. 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 hors-contextes. Ceci est dû au fait que le temps d'analyse des mots générés par certaines grammaires hors-contextes est linéaire.

Exemple 27 : Soit la grammaire générant le langage $a^n b^n$ ($n \geqslant 0$) : $G = (\{a, b\}, \{S\}, S, R)$ tel que R comporte les règles : $S \rightarrow aSb|\varepsilon$. D'après la définition, cette grammaire est hors-contexte et le langage $a^n b^n$ est hors-contexte.

4.2.1 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 25 : Etant donnée une grammaire $G = (V, N, S, R)$, les arbres de syntaxe de G sont des arbres dont les noeuds 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 $p \rightarrow a_1 ... a_n$ (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 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 28 : 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 4.1.

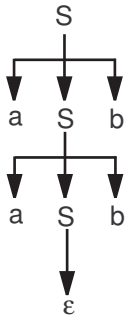


Figure 4.1 – Exemple d'un arbre de dérivation

4.2.2 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 manière.

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 plus d'une manière. 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 $G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSb|bSa|\varepsilon\})$ génère les mots ww^R tel que $w \in (a|b)^*$. Bien qu'il n'existe aucun automate à pile déterministe reconnaissant les mots de ce langage, tout mot ne possède qu'un seul arbre de dérivation.

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

Exemple 29 : 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 4.2 et 4.3). La grammaire est donc ambiguë. Or ceci pose un problème lors de l'évaluation de l'expression (rappelons que l'évaluation se fait toujours de gauche à droite et bas en haut)[4]. 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.

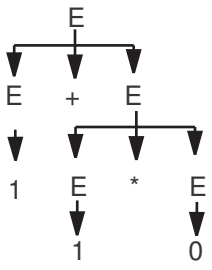


Figure 4.2 – Un premier arbre de dérivation

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

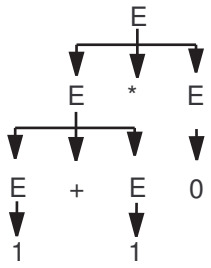


Figure 4.3 – Deuxième arbre de dérivation

D'une manière générale, pour lever l'ambiguïté d'une grammaire, il n'y a pas de méthodes qui fonctionne à tous les coups. Cependant, l'idée consiste généralement à introduire une hypothèse supplémentaire (ce qui va changer la grammaire) en espérant que le langage généré soit le même. Par exemple, la grammaire $G' = (\{+, *, 0, 1\}, \{E, T, F\}, E, \{E \rightarrow E + T|T, T \rightarrow T * F|F, F \rightarrow (E)|0|1\})$ 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 +.

4.2.3 Équivalence des grammaires hors-contextes et les automates à piles

Le théorème suivant établit l'équivalence entre les grammaires hors-contextes et les automates à piles. Néanmoins, ni le théorème ni sa preuve ne fournissent des moyens pour le passage d'une forme à une autre.

Théorème 4 : Tout langage hors-contexte est un langage algébrique et vice-versa. En d'autres termes, pour tout langage généré par une grammaire hors-contexte, il existe un automate à pile (déterministe ou non) qui le reconnaît. Réciproquement, pour tout langage reconnu 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.

4.3 Simplification des grammaires hors-contextes

4.3.1 Les grammaires propres

Une grammaire hors-contexte (V, N, S, R) est dite propre si elle vérifie :
- $\forall A \rightarrow u \in R : u \neq \varepsilon$ ou $A = S$;
- $\forall A \rightarrow u \in R : S$ ne figure pas dans u ;
- $\forall A \rightarrow u \in R : u \notin N$;
- 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 ;
2. Éliminer les règles $A \to \varepsilon$:
   - Calculer l'ensemble $E = \{A \in N \cup \{S'\} | A \xrightarrow{*} \varepsilon\}$;
   - Pour tout $A \in E$, pour toute règle $B \to \alpha A \beta$ de R
     - Rajouter la 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.

4.4 Les formes normales

4.4.1 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 S 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 :

$$A \to B_1 B_2$$
$$B_1 \to w_1$$
$$B_2 \to A_1 w_2 A_2 ... w_n A_n w_{n+1}$$

sinon, elle sera transformée en :

$$A \to A_1 B$$
$$B \to w_2 A_2 ... w_n A_n w_{n+1}$$

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

Exemple 30 : 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 : $S' \to S, S \to aSb|abS|bSa|baS$. En éliminant les formes $A \to B$, on obtient alors la grammaire propre $S' \to aSb|abS|bSa|baS, S \to aSb|abS|bSa|baS$. 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)

4.4.2 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.