



# LANGAGES ALGÈBRIQUES : AUTOMATES À PILE ET GRAMMAIRES-TYPE 2-

# 7

Dans ce chapitre nous allons élargir le champs d'étude selon la hiérarchie de Chomsky en s'intéressant aux langages de type 2. C'est une classe de langages appelés également langages algébriques et qui inclut les langages réguliers

- Générés par des grammaires hors contexte, les langages algébriques sont reconnus par des automates à pile, une extension des AEFs.
- Le niveau de complexité est inversement proportionnel au type du langage car le nombre d'algorithmes existants tend à diminuer en laissant la place à plus d'intuition.
- Les langages algébriques sont plus expressifs (par rapport aux langages réguliers) avec moins de restrictions (selon les grammaires à contexte libre).
- Ils sont plus intéressants et représentent (à un certain degré) la base de la plupart des langages de programmation.

## 7.1

## Automate à pile

### ✓ Théorème 7.1.1

Un langage  $L$  est dit algébrique si et seulement s'il existe un automate à pile qui reconnaît  $L$ .

### 7.1.1 Représentation formelle et fonctionnement

Un automate à pile (AP) est une machine abstraite semblable à un AEF, la seule différence c'est qu'elle possède une mémoire sous forme de pile pour analyser un mot.

#### 📌 Définition 7.1.1 Formellement

Un AP est défini par le sextuplet  $(X, \Pi, Q, s_0, F, \delta)$  tel que :

- $X$  : L'alphabet sur lequel les entrées (mots) sont définies.
- $\Pi$  : L'alphabet de la pile (l'ensemble des symboles à écrire dans la pile).  
▷  $\epsilon \in \Pi$  est un symbole signifiant que la pile est vide.
- $Q$  : L'ensemble des états.
- $s_0$  : L'état initial.
- $F$  : L'ensemble des états finaux ( $F \subseteq Q$ )

- $\delta$  : La fonction de transition d'une configuration à une autre selon : l'état actuel, symbole couramment lu et le contenu (sommet) de la pile tel que :  
 $\delta : Q \times (X \cup \{\varepsilon\}) \times \Pi \mapsto 2^Q \times \{(\Pi - \{\triangleright\})^* + \{\triangleright\}\}.$

### Définition 7.1.2 Configuration

Une configuration d'un AP est définie par un triplet  $(s_i, a, B)$  ce qui signifie que l'AP se trouve dans un état donné  $s_i \in Q$  en lisant un symbole donné  $a \in X$  et le sommet actuel de la pile est  $B$  (on peut mettre aussi le contenu de toute la pile).

### Définition 7.1.3 Transitions et actions possibles sur la pile

Une transition  $\delta(s_i, a, B) = (s_j, C)$  signifie que l'AP passe à l'état  $s_j$  en modifiant (empiler ou dépiler) ou pas le contenu de la pile dont le sommet sera  $C \in \Pi$ . Donc lors d'une transition dans un AP, il est possible non seulement de lire un symbole sur le ruban et changer d'état mais aussi effectuer une opération (dépiler/empiler) sur la pile :

- Empiler dans une pile vide :  $\delta(s_i, a, \triangleright) = (s_j, B\triangleright)$  signifie que si l'AP se trouvant dans l'état  $s_i$  lit  $a$  et sa pile est vide, il passe à l'état  $s_j$  et empile le symbole  $B$ .
- Empiler dans une pile non vide :  $\delta(s_i, a, B) = (s_j, AB)$  signifie que si l'AP se trouvant dans l'état  $s_i$  lit  $a$  et le sommet de sa pile est  $B$ , il passe à l'état  $s_j$  et empile un autre symbole  $A$ .
- Empiler sans rien lire :  $\delta(s_i, \varepsilon, B) = (s_j, AB)$  signifie que si l'AP se trouvant dans l'état  $s_i$  et le sommet de sa pile est  $B$ , il passe à l'état  $s_j$  et empile un autre symbole  $A$  sans avoir besoin de lire aucun symbole ( $\varepsilon$ )
- Dépiler :  $\delta(s_i, a, B) = (s_j, \varepsilon)$  signifie que si l'AP se trouvant dans l'état  $s_i$  lit  $a$  et le sommet de sa pile est  $B$ , il passe à l'état  $s_j$  et dépile un symbole (sommet) de sa pile.
- Transition sans modifier la pile :  $\delta(s_i, a, B) = (s_j, B)$  signifie que si l'AP se trouvant dans l'état  $s_i$  lit  $a$  et le sommet de sa pile est  $B$ , il passe à l'état  $s_j$  sans modifier le contenu de sa pile.

### Dédution 7.1.1 $w \in L(A)$ d'un point de vue AP

- Un mot  $w$  est accepté par un AP  $A$  si, après avoir lu tout le mot, l'AP se trouve dans un état final  $s_f \in F$ , peu importe le contenu de la pile.
- Ce qui veut dire que l'AP se trouve dans la configuration  $(s_f, \varepsilon, B)$ , tel que  $B = \triangleright$  ou un autre symbole de  $\Pi$ . Dans ce cas il est possible de vider le contenu restant de la pile après avoir lu le mot en entier.

➤ L'ensemble de tous les mots acceptés par  $A$  est représenté par le langage  $L(A)$

### Remarque 7.1.1 Le rejet d'un mot par un AP

Un mot  $w \notin L(A)$  est un mot rejeté par l'AP  $A$  dans l'un des trois cas suivants :

- Lorsque l'AP se trouve dans un  $s \in Q$  et le symbole en cours de lecture est  $a \in X$ , mais il n'y a pas de transition possible ( $A$  ne peut pas lire entièrement  $w$ )
- Si le mot  $w$  est entièrement lu par  $A$ , mais  $A$  n'est pas dans un état final.
- Si une opération incorrecte est menée sur la pile : dépiler une pile vide.

### Exemple Définir un AP

Prenons le langage  $L = \{a^n b^n | n \geq 0\}$  qui a été déjà montré comme non régulier dans le chapitre précédent et essayons de définir son AP :  $AP = (\{a, b\}, \{A, \triangleright\}, \{s_0, s_1, s_2\}, s_0, \{s_2\}, \delta)$ , tel que les transitions sont définies comme suit :

- $\delta(s_0, a, \triangleright) = (s_0, A\triangleright)$
- $\delta(s_0, a, A) = (s_0, AA)$
- $\delta(s_0, b, A) = (s_1, \varepsilon)$
- $\delta(s_1, b, A) = (s_1, \varepsilon)$
- $\delta(s_1, \varepsilon, \triangleright) = (s_2, \triangleright)$
- $\delta(s_0, \varepsilon, \triangleright) = (s_2, \triangleright)$

Dans cet exemple,  $w \in L$  veut dire que  $w$  contient autant de  $a$  que de  $b$  tel que toutes les occurrences de  $a$  précèdent celles de  $b$ . L'idée derrière la construction de cet AP est d'utiliser la pile comme un moyen pour compter le nombre d'occurrences de  $a$  (en empilant  $A$  après la lecture de chaque  $a$ ) et le comparer par la suite avec le nombre de  $b$  (en dépilant  $A$  après la lecture de chaque  $b$ )

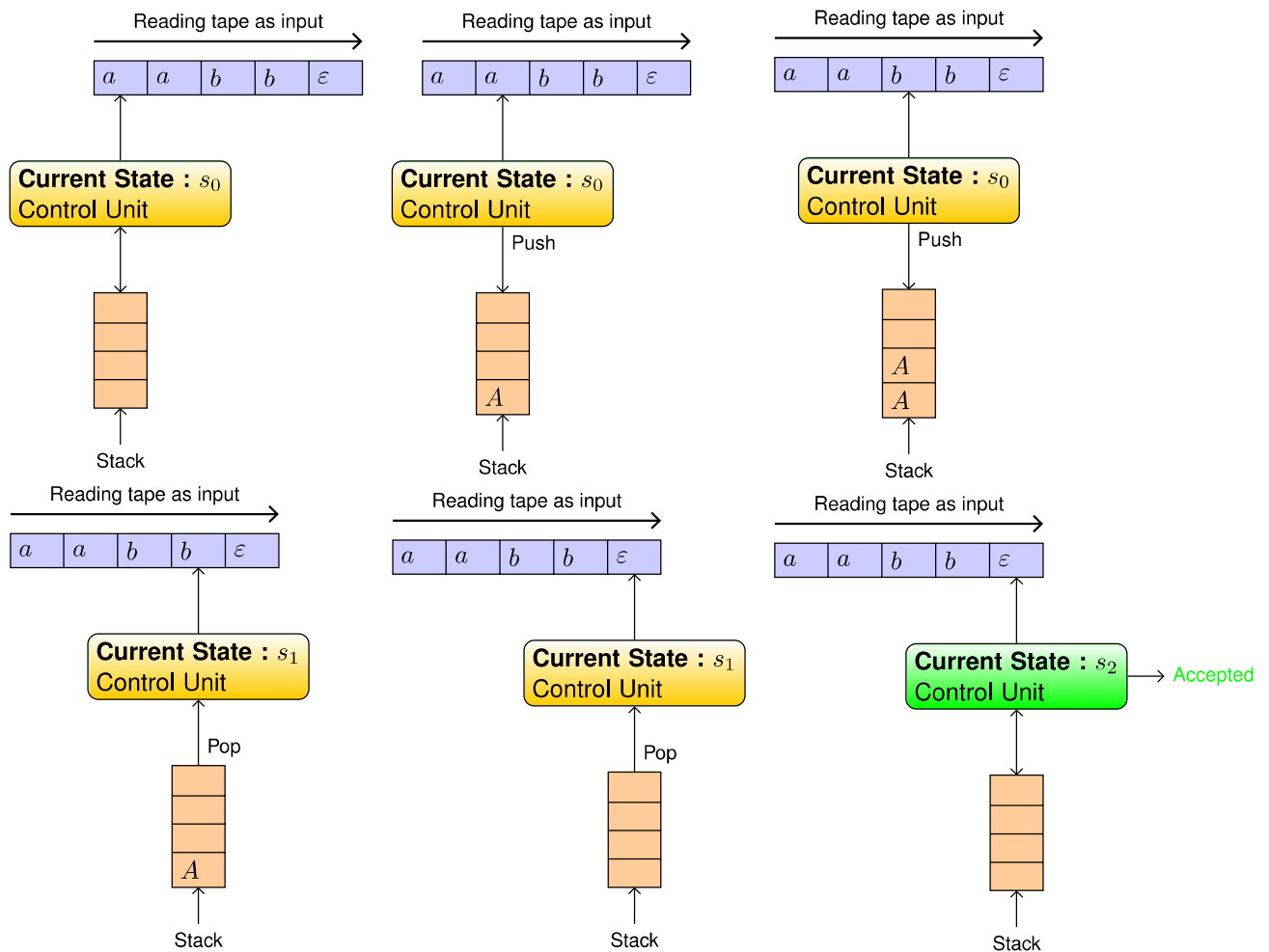


### Exemple Analyser des mots avec un AP

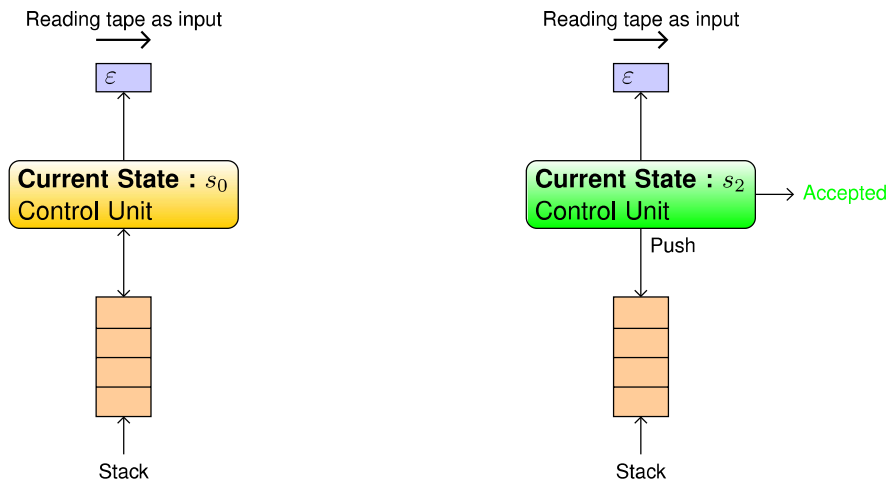
On va utiliser l'AP précédent pour analyser les mots suivants :

- 1  $w = aabb : \delta(s_0, a, \triangleright) \rightarrow \delta(s_0, a, A\triangleright) \rightarrow \delta(s_0, b, AA\triangleright) \rightarrow \delta(s_1, b, A\triangleright) \rightarrow \delta(s_1, \varepsilon, \triangleright) \rightarrow \delta(s_2, \varepsilon, \triangleright)$   
 Accepté

FIGURE 7.1 – Analyser le mot  $aabb$  par un AP

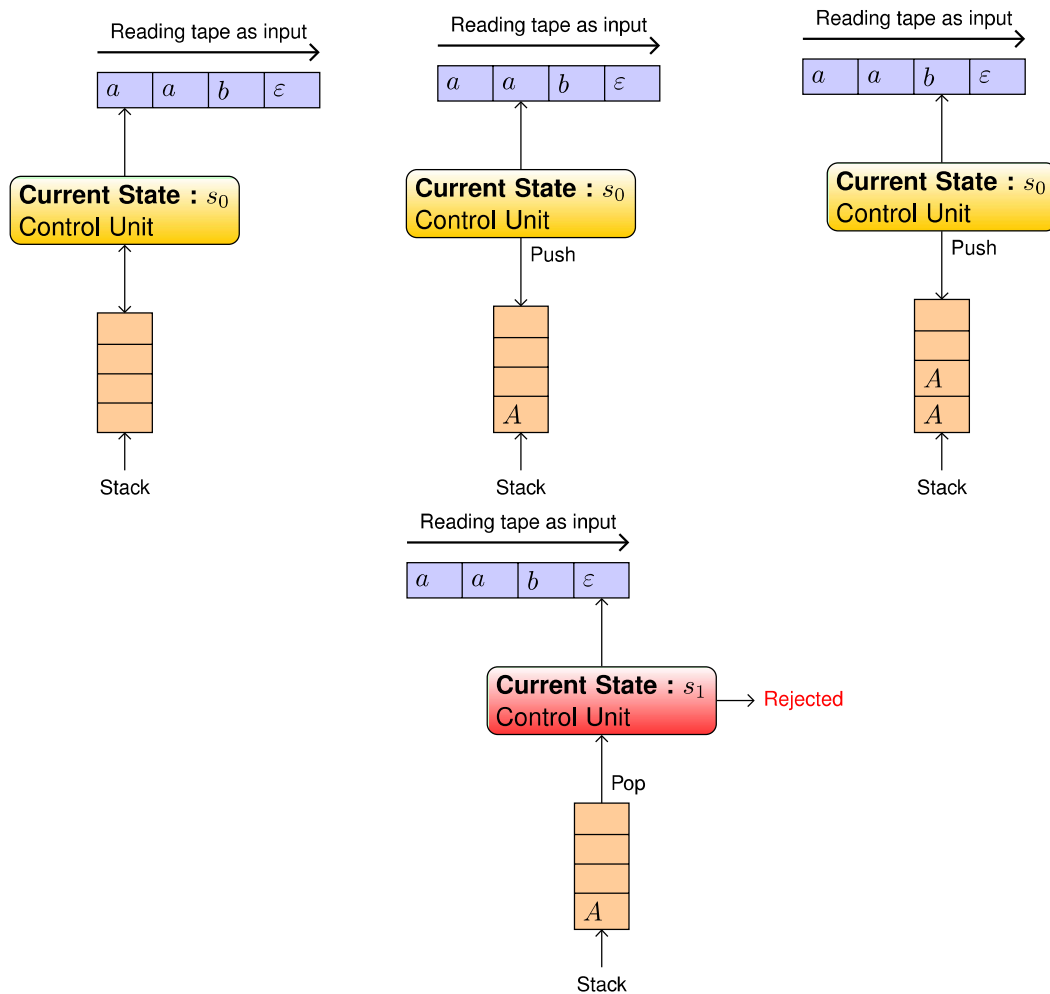


- 2  $w = \varepsilon : \delta(s_0, \varepsilon, \triangleright) \rightarrow \delta(s_2, \varepsilon, A\triangleright) \rightarrow$  Accepté

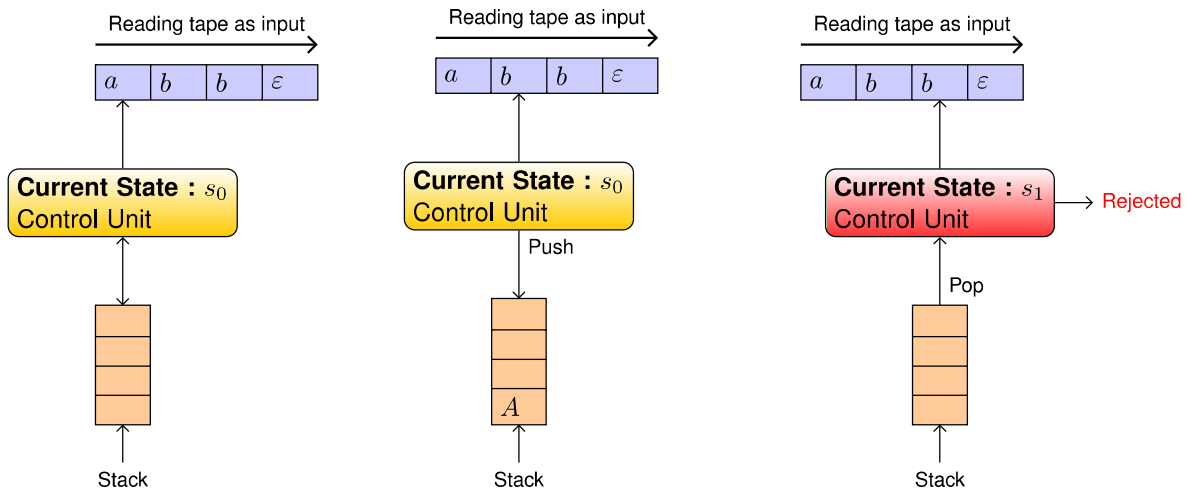
FIGURE 7.2 – Analyser le mot  $\varepsilon$  par un AP


- 3  $w = aab : \delta(s_0, a, \triangleright) \rightarrow \delta(s_0, a, A\triangleright) \rightarrow \delta(s_0, b, AA\triangleright) \rightarrow \delta(s_1, \varepsilon, A\triangleright)$  **Rejeté**

Car même si le mot est entièrement lu mais l'AP ne peut pas passer à l'état final car il n'y a pas de transition qui correspond à cette configuration.

 FIGURE 7.3 – Analyser le mot  $aab$  par un AP


- 4  $w = abb : \delta(s_0, a, \triangleright) \rightarrow \delta(s_0, b, A\triangleright) \rightarrow \delta(s_1, b, \triangleright)$  **Rejeté**  
Pourquoi ?

FIGURE 7.4 – Analyser le mot  $abb$  par un AP

### 7.1.2 Le déterminisme des automates à pile

La notion de déterminisme ne se limite pas seulement aux AEFs. Les APs peuvent être déterministes ou non-déterministes.

#### 📌 Définition 7.1.4 Un AP déterministe

Un AP  $A = (X, \Pi, Q, s_0, F, \delta)$  est dit déterministe si  $\forall (s_i, a, A) \in Q \times X - \{\varepsilon\} \times \Pi$ , il existe au plus une paire  $(s_j, B) \in X \times \Pi$  ( $A$  ou  $B$  désigne le sommet de la pile), tel que  $(s_i, a, A) = (s_j, B)$ .

#### 💬 Remarque 7.1.2 Les $\varepsilon$ -transitions dans un AP déterministe ?

Oui, il est possible d'avoir un AP déterministe avec des  $\varepsilon$ -transitions mais uniquement pour passer à un état final.

#### 📖 Exemple un AP déterministe

Nous voulons contruire un AP déterministe acceptant  $L = \{wcw^R \mid w \in (a|b)^*\}$  (des mots palindromes de longueur impaire).

Soit l'AP  $A = (\{a, b\}, \{A, B\}, \{s_0, s_1, s_2\}, s_0, \{s_0\}, \delta)$  tel que  $\delta$  est définie par les transitions suivantes :

- |   |  |
|---|--|
| — $\delta(s_0, a, \triangleright) = (s_0, A\triangleright)$ | — $\delta(s_0, c, A) = (s_1, A)$                                     |
| — $\delta(s_0, b, \triangleright) = (s_0, B\triangleright)$ | — $\delta(s_0, c, B) = (s_1, B)$                                     |
| — $\delta(s_0, a, A) = (s_0, AA)$                           | — $\delta(s_0, c, \triangleright) = (s_1, \triangleright)$           |
| — $\delta(s_0, a, B) = (s_0, AB)$                           | — $\delta(s_1, a, A) = (s_1, \varepsilon)$                           |
| — $\delta(s_0, b, A) = (s_0, BA)$                           | — $\delta(s_1, b, B) = (s_1, \varepsilon)$                           |
| — $\delta(s_0, b, B) = (s_0, BB)$                           | — $\delta(s_1, \varepsilon, \triangleright) = (s_2, \triangleright)$ |

L'idée consiste à empiler des symboles  $A, B$  qui correspondent aux symboles lus  $a, b$ . Une fois qu'on arrive à lire un  $c$  on dépile en lisant les symboles de  $w^R$  (Une symétrie par rapport à  $c$ )

#### 📌 Définition 7.1.5 Un AP non-déterministe

Un AP est non-déterministe si pour la même configuration il existe au moins deux transitions (actions sur la pile) possibles. Dans ce cas, il est nécessaire d'examiner toutes les séquences de configurations possibles jusqu'à en trouver une qui réussisse

Comme on l'a vu avec un AEF non-déterministe, le temps nécessaire à un AP non-déterministe pour reconnaître un mot est généralement bien plus long par rapport à un AP déterministe

### Exemple un AP non-déterministe

Considérons maintenant le langage des mots palindromes de longueur paire  $L = \{ww^R \mid w \in (a|b)^*\}$ . Le problème qui se pose lors de la construction d'un AP acceptant  $L$  est l'absence d'un 'repère' (comme par exemple le symbole  $c$ ) indiquant la fin de  $w$  et le début de  $w^R$ . Sans une indication pareille l'AP ne saura pas à quel moment il faut arrêter d'empiler et procéder au dépilement. Par conséquent, l'AP sera non-déterministe.

- |   |  |
|---|--|
| — $\delta(s_0, a, \triangleright) = (s_0, A\triangleright)$ | — $\delta(s_0, b, B) = (s_0, BB)$                                    |
| — $\delta(s_0, b, \triangleright) = (s_0, B\triangleright)$ | — $\delta(s_0, b, B) = (s_1, \varepsilon)$                           |
| — $\delta(s_0, a, A) = (s_0, AA)$                           | — $\delta(s_1, a, A) = (s_1, \varepsilon)$                           |
| — $\delta(s_0, a, A) = (s_1, \varepsilon)$                  | — $\delta(s_1, b, B) = (s_1, \varepsilon)$                           |
| — $\delta(s_0, a, B) = (s_0, AB)$                           | — $\delta(s_1, \varepsilon, \triangleright) = (s_2, \triangleright)$ |
| — $\delta(s_0, b, A) = (s_0, BA)$                           | — $\delta(s_0, \varepsilon, \triangleright) = (s_2, \triangleright)$ |

Après la lecture du premier symbole, l'idée consiste d'une part à supposer que chaque symbole lu représente le dernier symbole de  $w$  (Un point de non-déterminisme). Dans ce cas, on commence à dépiler et continuer l'analyse. Mais s'il le mot est rejeté, il faut revenir au dernier point de non-déterminisme afin d'empiler et ainsi de suite.

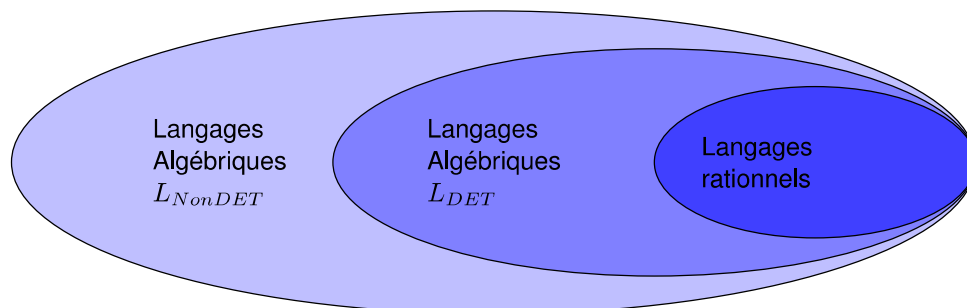
### Remarque 7.1.3 $L_{DET} \subset L_{NonDET}$

- Tout langage algébrique accepté par un AP déterministe (langage algébrique déterministe) est également reconnu par un AP non déterministe. Mais l'inverse n'est pas toujours vrai. Autrement dit, il existe des langages algébriques qui ne sont acceptés par aucun AP déterministe. Mais pourquoi ?
- Il n'est pas possible de transformer tout AP non-déterministe en AP déterministe. Cette transformation n'est pas systématique.

➤ Donc, l'ensemble des langages acceptés par les APs déterministes forment une sous-classe des langages algébriques. Ceci veut dire que l'ensemble des langages reconnus par des APs non-déterministes est beaucoup plus important que celui des langages reconnus par des APs déterministes.

$$L_{DET} \subset L_{NonDET}$$

FIGURE 7.5 – Langages algébriques déterministes et non-déterministes



### Remarque 7.1.4 Complexité

- Le temps d'analyse d'un mot par un AP déterministe, est linéaire en fonction de la taille de

l'entrée. En revanche, le temps d'analyse d'un AP non-déterministe peut être exponentiel (en fonction de la taille du mot à analyser). C'est la raison pour laquelle, les langages déterministes sont plus intéressants que ceux non-déterministes et qui ne pouvant pas être acceptés que par des APs non-déterministes

- Les APs non-déterministes sont plus expressifs que les APs déterministes

### Remarque 7.1.5 Stabilité

- Les langages déterministes sont stables par rapport au complément ( $\bar{L}$ ) mais ce n'est pas le cas pour les opérations :  $\cup, \cap, \cdot, *$ .
- Le reste des langages algébriques qui ne peuvent pas être déterministes sont plutôt stables par rapport à :  $\cup, \cdot, *$ , mais ce n'est pas le cas quand il s'agit de  $\cap$  et  $\bar{L}$ .
- Cependant, l'intersection d'un langage algébrique et avec un langage régulier est toujours algébrique. par exemple  $\{w | w = a^*b^*\} \cap \{a^n b^n | n \geq 0\}??$

## 7.2

## Grammaire hors-contexte


Les concepteurs de compilateurs et d'interpréteurs des langages de programmation dont la plupart sont des langages algébriques commencent souvent par obtenir une grammaire pour un langage étudié. La construction de ce genre d'analyseurs (parsers) qui sont couramment utilisés aussi en NLP (Natural Language Processing) et les domaines associés à l'IA/ML utilise des grammaires hors contexte. Il est ainsi intéressant d'étudier en particulier ce type de grammaire qui génèrent les langages algébriques.

Une autre raison, pour laquelle, il est plus intéressant de s'intéresser aux grammaires hors-contextes plus que les APs, c'est parce qu'il y a de nombreux algorithmes et outils traitant des formes particulières de grammaires (Chomsky, Greibach) qui ont pour objectif de faciliter la construction des arbres de dérivation.

Enfin et selon la classification de Chomsky, il ne faut pas oublier que les langages algébriques sont aussi des langages contextuels mais on les traite comme algébriques afin de maîtriser la complexité d'analyse des mots.

### 7.2.1 Arbre syntaxique (Parse Tree)

Rappelons la forme des règles de dérivations  $g \rightarrow d$  qui caractérisent une grammaire hors-contexte  $G = (V, N, S, R) : g \in N$  et  $d \in (V + N)^*$ . Tant que  $g \in N$ , il est tout à fait possible de générer un mot par une grammaire à contexte libre, en construisant son arbre syntaxique de dérivation (vue précédemment).

 **Question.** Mais quel est l'avantage concret (l'application pratique) des arbres syntaxiques ?

Plus concrètement, pour exécuter un code écrit dans un langage de programmation (un langage algébrique généré par une grammaire hors-contexte), un compilateur traduit ce code (selon un processus appelé le **parsing**) sous une autre forme pour qu'il soit compilé et adapté à l'exécution. Une de ces formes de représentation de ce code est l'arbre syntaxique (parse tree). Ce sont des structures associées à des mots et à partir desquelles il est possible de calculer leur signification.

### Définition 7.2.1 Parsing

Le parsing (en anglais parsing) d'un mot avec une grammaire est le processus de construction de ou des arbres de dérivation pour ce mot, s'ils en existent.

### Exemple Arbre syntaxique d'un mot généré par une grammaire à contexte libre

Soit la grammaire  $G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSb | \varepsilon\})$ . (Que représente  $L(G)$  ?)

Pour prouver qu'un mot  $w = aabb \in L(G)$ , on va essayer de construire son arbre syntaxique :

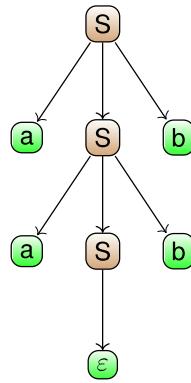


FIGURE 7.6 – Arbre syntaxique d'un mot généré par une grammaire hors-contexte

**? Question.** Mais cette structure associée à un mot est-elle nécessairement unique ?

## 7.2.2 L'ambiguïté d'une grammaire

L'ambiguïté dans une grammaire hors-contexte ressemble à celle que nous avons déjà abordée avec les ERs dans le chapitre précédent.

### 📌 Définition 7.2.2 Un mot ambigu

Un mot  $w \in L(G)$  ( $G$  est hors-contexte) est dit ambigu, s'il existe au moins deux arbres de dérivation possibles (en utilisant  $G$ ) qui lui sont associés

Deux arbres de dérivation (parse trees) différents ne veut pas dire deux dérivations différentes. La dérivation peut changer simplement selon l'ordre de remplacement des non-terminaux.

### 📌 Définition 7.2.3 Une grammaire ambiguë

Une grammaire  $G$  est dite ambiguë, si et seulement il existe au moins un mot qui peut être généré par deux ou plusieurs séquences de dérivation possibles. Cela veut dire qu'il existe un mot qui admet plusieurs arbres de dérivation. En d'autres termes, il existe au moins un mot ambigu  $w \in L(G)$

Cela se produit généralement, lorsqu'on a de la récursion (au niveau de l'axiome) à la fois à droite et à gauche. Mais l'inverse n'est pas toujours vrai. S'il n'y a pas cette récursion, ça ne veut pas dire que la grammaire n'est pas ambiguë (car la récursion peut se produire indirectement)



### Exemple d'une grammaire ambiguë

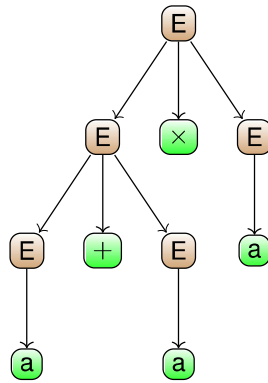
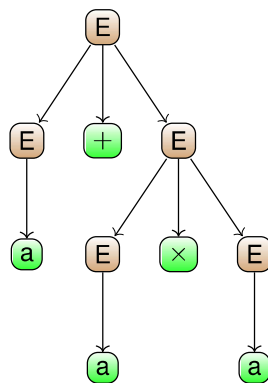
Soit la grammaire suivante qui génère un fragment d'un langage de programmation définissant les expressions arithmétiques :  $G = (\{a, \times, +, (, )\}, \{E\}, E, \{E \rightarrow E+E \mid E \times E \mid (E)a\})$ . Cette grammaire est ambiguë car elle génère au moins un mot ambigu.

Par exemple le mot (l'expression)  $w = a + a \times a$  est ambigu car il possède deux arbres de dérivation. Cette ambiguïté pose un problème dans l'évaluation arithmétique de ces expressions, car elle ne permet pas de respecter toujours l'ordre de priorité habituelle entre les opérateurs  $+$  et  $\times$ .

Arithmétiquement, si on remplace le dernier  $a$  par  $0$  on obtient  $a + a \times 0 = a + 0 = a$ . Mais selon le sens de la lecture sur l'arbre on aura deux évaluations différentes

- Le premier arbre : on regroupe les deux premières opérands de l'opérateur  $+$  comme étant la première opérande de l'opérateur  $\times$ . Cela donne  $a + a = 2a$  en premier lieu puis  $\times 0$ . On obtient donc  $0$  🚫
- Le deuxième arbre : La première opérande de l'opérateur  $+$  est un  $a$ . Alors que sa deuxième opérande est le résultat de  $a \times 0 = 0$ . Ce qui donne  $a + 0 = a$  (l'évaluation ou l'interprétation correcte)



FIGURE 7.7 – Premier arbre de dérivation de  $a + a \times a$ FIGURE 7.8 – Deuxième arbre de dérivation de  $a + a \times a$ **Déduction 7.2.1**

L'ambiguïté d'une grammaire ne signifie pas seulement deux analyses syntaxiques (ou plus) différentes du mot, mais également deux interprétations (résultats) différentes

Il faut noter que l'interprétation (la sémantique d'un mot) qui s'oppose à la syntaxe correspond à la signification d'un énoncé dans le cas d'une grammaire qui génère des phrases, à la valeur du calcul, ou encore à une séquence d'instructions à exécuter dans le cadre d'une grammaire qui génère un langage informatique.

**Remarque 7.2.1 Une grammaire ambiguë, mais à quoi ça sert ?**

Une grammaire ambiguë est rarement utile (et indésirable) pour un langage de programmation. Car pour un même programme (string) deux arbres d'analyse (ou plus) impliquent deux significations différentes (programmes exécutables). Alors qu'un programme doit avoir une interprétation unique.

**Comment lever l'ambiguïté d'une grammaire ?**

Certains langages peuvent être générés, à la fois, par des grammaires ambiguës et des grammaires non ambiguës. Ce qui veut dire que l'ambiguïté de certaines (pas toutes) grammaires peut être levée. Mais il n'y a pas de méthode précise qui fonctionne à tous les coups. L'idée de lever l'ambiguïté consiste généralement à introduire une hypothèse supplémentaire, qui peut changer la grammaire dans certains cas, mais qui doit préserver le langage généré.

**Exemple pour montrer comment lever l'ambiguïté d'une grammaire**

Reprenons l'exemple de la grammaire ambiguë précédente

$G = (\{a, \times, +, (, )\}, \{E\}, E, \{E \rightarrow E + E \mid E \times E \mid (E) \mid a\})$ .

$G$  sera transformée en  $G'$ , une version non ambiguë et qui consiste à donner :

- La priorité habituelle de l'opérateur  $\times$  (dont les opérands sont des facteurs notés par  $F$ ) par rapport à l'opérateur  $+$  (dont les opérands sont des termes notés par  $T$ )
- Une associativité à gauche de ces opérateurs

$G' = (\{a, \times, +, (, )\}, \{E, T, F\}, E, \{E \rightarrow E + T \mid T, T \rightarrow T \times F \mid F, F \rightarrow (E) \mid a\})$ .

Avec  $G'$  qui génère le même langage  $L(G)$ , on peut générer et construire le seul arbre syntaxique de  $w = a + a \times a$  (vu précédemment)

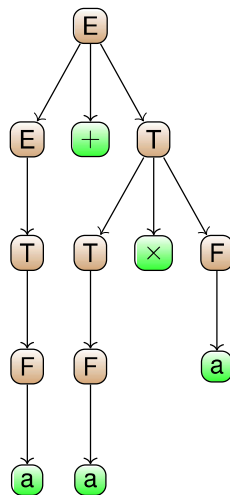


FIGURE 7.9 – L'arbre de dérivation de  $a + a \times a$  généré par  $G'$  (non-ambiguë)

### Remarque 7.2.2 Y-t-une méthode pour détecter l'ambiguïté d'une grammaire ?

Chercher l'ambiguïté ou non d'une grammaire est un problème indécidable car il n'existe aucun algorithme qui permet de décider correctement si une grammaire est non ambiguë (si elle existe) ou non.

### Remarque 7.2.3 L'ambiguïté d'une grammaire reflète-t-elle l'ambiguïté du langage généré ?

- L'ambiguïté est une propriété qui concerne les grammaires et pas les langages.
- Mais un langage hors-contexte est intrinsèquement ambigu ( $\{a^n b^n c^m\} \cup \{a^n b^m c^m\}$ ) si toutes les grammaires hors-contexte qui le génèrent sont ambiguës.  
 ➤ Ce langage serait absolument inadapté comme langage de programmation car il n'y aura pas de moyens pour se fixer sur une structure unique pour tous ses programmes.
- Les langages informatiques ne sont pas intrinsèquement ambigus, car lorsque l'on conçoit un nouveau langage informatique, il suffit de se prémunir contre les ambiguïtés.
- Les grammaires formelles utilisées pour les langages de programmation sont explicitement conçues pour limiter l'ambiguïté d'analyse.

## L'ambiguïté d'une grammaire et le non-déterminisme d'un AP ?

L'ambiguïté d'une grammaire qui génère  $L$  n'implique pas forcément le non-déterminisme d'un ou des APs acceptant  $L$

### Remarque 7.2.4 Ambiguïté et non-déterminisme

- Nous avons vu qu'il existe des langages algébriques non-déterministes (acceptés uniquement

par des APs non-déterministes) et il existe aussi des langages qui ne peuvent être générés que par des grammaires ambiguës.

► Mais si un langage  $L$  n'est accepté que par des automates à pile non-déterministes, cela ne veut pas dire que  $L$  est forcément généré par une grammaire ambiguë



### Exemple

C'est le cas de cette grammaire non-ambiguë  $G = (\{a, b\}, \{S\}, S), \{S \rightarrow aSa|bSb|\varepsilon\}$  qui génère les mots de  $L(G) = \{ww^R | w \in \{a, b\}^*\}$ . Chaque mot de  $L$  ne possède qu'un seul arbre de dérivation et pourtant il n'y a pas d'AP déterministe qui peut l'accepter (Nous avons plutôt vu son AP non déterministe précédemment)

## 7.2.3 Grammaire propre



### Définition 7.2.4 Grammaire propre

Une grammaire hors-contexte  $G = (V, N, S, R)$  est une grammaire propre si elle vérifie les critères suivants :

- Il n'y a que l'axiome qui peut générer  $\varepsilon$
- $\forall A \rightarrow u \in R$ , l'axiome ne doit pas figurer dans la partie droite  $u$  des règles.
- $G$  ne contient pas de productions unitaires. Autrement dit :  $\forall A \rightarrow u \in R : u \notin N$
- $G$  est réduite. Autrement dit :  $\forall A \in N$ ,  $A$  est utile, c'est à dire :
  - $A$  est atteignable (accessible) depuis l'axiome  $S : S \xrightarrow{*} \alpha A \beta$  tel que  $\alpha, \beta \in (N + V)^*$
  - $A$  est productif :  $A \xrightarrow{*} w$  tel que  $w \in V^*$

### Comment rendre une grammaire propre

N'importe quelle grammaire hors-contexte  $G$  peut être transformée en une grammaire propre en suivant les étapes suivantes :

- 1 Pour s'assurer que l'axiome ne figure pas dans la partie droite d'une règle, on peut rajouter un nouvel axiome  $S'$  à travers cette nouvelle règle  $S' \rightarrow S$ .
- 2 Éliminer les  $\varepsilon$ -productions (sauf celle de l'axiome qui produit  $\varepsilon$ )
  - (a) Déterminer l'ensemble  $E = \{A \in N - \{S'\} | A \xrightarrow{*} \varepsilon\}$  des non terminaux dérivables en  $\varepsilon$  (directement ou indirectement)
  - (b) Pour tout  $A \in E$ , modifier les productions  $B \rightarrow uAv$  contenant ce non-terminal, en remplaçant  $A$  par le mot vide  $\varepsilon$ . Ce qui permet d'ajouter la nouvelle règle  $B \rightarrow uv$
  - (c) Supprimer les règles  $A \rightarrow \varepsilon$



### Remarque 7.2.5 Des cas particuliers

- Si on a une règle  $B \rightarrow uAvAw$ , on ajoutera les règles  $B \rightarrow uvAw$ ,  $B \rightarrow uAvw$ ,  $B \rightarrow uvw$ .
- Si on a la règle  $B \rightarrow A$ , on ajouterait  $B \rightarrow \varepsilon$  sauf que ce type de règles est à supprimer, donc on réappliquera les étapes précédentes afin d'éliminer toutes les  $\varepsilon$ -productions

- 3 Éliminer les productions unitaires en appliquant les étapes suivantes sur  $R - \{S' \rightarrow \varepsilon\}$ 
  - (a) Chercher toutes les paires  $(A, B)$  tel que  $A \xrightarrow{*} B$
  - (b) Pour chaque paire  $(A, B)$ , rajouter aux règles de production de  $A$  toutes les productions de  $B \rightarrow u_1 | \dots | u_n$  ce qui donne  $A \rightarrow u_1 | \dots | u_n$

(c) Supprimer  $A \rightarrow B$

### Remarque 7.2.6

Cette suppression peut faire apparaître d'autres productions unitaires, c'est pour cela qu'il faut appliquer ces étapes d'une manière récursive

4 Réduire  $G$  en supprimant :

- Les non-terminaux improductifs
- Les non-terminaux inaccessibles.

### Remarque 7.2.7

Les règles de production qui contiennent des non-terminaux improductifs ou inaccessibles sont inutiles et peuvent être supprimées sans aucune influence sur le langage généré par la grammaire

### Exemple Rendre une grammaire propre

Soit la grammaire  $G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSbS \mid bSaS \mid \varepsilon\})$

1. On crée un nouvel axiome  $S'$  en ajoutant la nouvelle règle  $S' \rightarrow S$
2. On élimine les  $\varepsilon$ -productions :  $\{S \rightarrow \varepsilon\}$  en remplaçant  $S$  par  $\varepsilon$  dans la partie droite des règles en questions :
  - $S \rightarrow aSbS$  est remplacée par :  $S \rightarrow abS, S \rightarrow aSb, S \rightarrow ab$
  - $S \rightarrow bSaS$  est remplacée par :  $S \rightarrow baS, S \rightarrow bSa, S \rightarrow ba$
  - $S \rightarrow S'$  est remplacée par  $S' \rightarrow \varepsilon$
3. On élimine les productions unitaires :  $\{S \rightarrow S'\}$  en ajoutant les règles suivantes :
  - $S' \rightarrow abS \mid aSb \mid baS \mid bSa \mid ab \mid ba$
4. Il n'y a pas de non-terminaux improductifs ni inaccessibles

Alors nous obtenons la grammaire propre  $G'(\{a, b\}, \{S, S'\}, S', \{S' \rightarrow abS \mid aSb \mid baS \mid bSa \mid ab \mid ba \mid \varepsilon, S \rightarrow abS \mid aSb \mid baS \mid bSa \mid ab \mid ba\})$

## 7.2.4 Forme normale de Chomsky (FNC)

La mise sous forme normale de Chomsky d'une grammaire hors-contexte permet simplifier la construction des arbres de dérivations qui deviennent des arbres binaires (cela facilite l'application de nombreux algorithmes).

### Définition 7.2.5 Une grammaire sous la forme normale de Chomsky (FNC)

Une grammaire hors-contexte  $G = (V, N, S, R)$  est sous forme normale de Chomsky (FNC) si et seulement si toutes ses règles de production  $R$  sont écrites selon l'une des formes suivantes :

- $A \rightarrow BC$ .
- $A \rightarrow a$ .

Tel que :  $A \in N, B, C \in N - \{S\}$  et  $a \in V$ . Sachant que la règle  $S \rightarrow \varepsilon$  est autorisée.

### Théorème 7.2.1

Pour toute grammaire hors-contexte il existe une grammaire hors-contexte sous forme normale de Chomsky qui génère le même langage.

Ce théorème affirme qu'il est toujours possible de transformer n'importe quelle grammaire hors-contexte sous la forme normale de Chomsky.

## Convertir une grammaire en FNC

Soit  $G = (V, N, S, R)$  une grammaire hors-contexte. Il est préférable que  $G$  soit propre sinon, on commence par rendre  $G$  propre car cela facilitera la transformation :

- 1 Transformer  $G$  en une grammaire propre ce qui nous donne des règles  $R$  sous l'une des formes suivantes :
  - $A \rightarrow w, w \in V^+$  (Seulement l'axiome  $S$  qui produit  $\varepsilon$ ,  $S \rightarrow \varepsilon$ )
  - $A \rightarrow w_1 A_1 \dots w_n A_n w_{n+1}, w_i \in V^*, A_i \in N - \{S\}$  (sans production unitaire)
- 2 Convertir les règles de type  $A \rightarrow w, w \in V^+$  d'une manière récursive tel que
  - Si  $|w| = 1$  ( $w = a, a \in V$ ) alors, garder la règle  $A \rightarrow a$  (règle normalisée)
  - Sinon, si  $|w| \geq 2$ , ( $w = au, a \in V, u \in V^+$ ) alors, remplacer  $A \rightarrow w$  par  $A \rightarrow A_1 A_2, A_1 \rightarrow a$  et  $A_2 \rightarrow u$ .
  - Convertir  $A_2 \rightarrow u$  en appliquant les deux étapes précédentes (récursivité).

Le but ici est d'associer à chaque terminal, un non-terminal.

- 3 Convertir les règles de type  $A \rightarrow w_1 A_1 w_2 A_2 \dots w_n A_n w_{n+1}, w_i \in V^*, A_i \in N - \{S\}$ 
  - Si cette règle commence par au moins un terminal ( $w_1 \neq \varepsilon$ ), remplacez-la par :
    - $A \rightarrow B_1 B_2$  (règle normalisée)
    - $B_1 \rightarrow w_1$  (On revient à la conversion de l'étape (2))
    - $B_2 \rightarrow A_1 w_2 A_2 \dots w_n A_n w_{n+1}$  (Cette forme est transformée selon le cas suivant)
  - Sinon, elle est remplacée par :
    - $A \rightarrow A_1 B$  (règle normalisée)
    - $B \rightarrow w_2 A_2 \dots w_n A_n w_{n+1}$

### Remarque 7.2.8

De la même manière, cette transformation est appliquée récursivement jusqu'à ce que toutes les règles soient des règles normalisées

### Exemple Normaliser une grammaire (sous FNC)

Prenons l'exemple de la grammaire propre obtenue dans l'exemple précédent :  $G'(\{a, b\}, \{S, S'\}, S', \{S' \rightarrow abS|aSb|baS|bSa|ab|ba|\varepsilon, S \rightarrow abS|aSb|baS|bSa|ab|ba\})$

Les transformations suivantes seront appliquées sur les productions de  $S$ , les mêmes que celles de  $S'$

1.  $G'$  est déjà propre.
2. Convertir les règles :  $S \rightarrow ab|ba$  en
  - $S \rightarrow ab$  devient  $S \rightarrow AB$ , normalisée
  - $S \rightarrow ba$  devient  $S \rightarrow BA$ , normalisée
  - $A \rightarrow a, B \rightarrow b$  normalisées
3. Convertir les règles  $S \rightarrow abS|aSb|baS|bSa$  :
  - $S \rightarrow abS$  devient :
    - $S \rightarrow AU_1$  normalisée
    - $U_1 \rightarrow bS$  qui devient :  $U_1 \rightarrow BS$  normalisée
  - $S \rightarrow aSb$  devient :
    - $S \rightarrow AU_2$  normalisée
    - $U_2 \rightarrow Sb$  qui devient :  $U_2 \rightarrow SB$  normalisée
  - $S \rightarrow baS$  devient :
    - $S \rightarrow bU_3$  qui devient  $S \rightarrow BU_3$  normalisée
    - $U_3 \rightarrow aS$  qui devient  $U_3 \rightarrow AS$  normalisée

—  $S \rightarrow bSa$  devient :

- $S \rightarrow bU_4$  qui devient  $S \rightarrow BU_4$  *normalisée*
- $U_4 \rightarrow Sa$  qui devient  $U_4 \rightarrow SA$  *normalisée*

## 7.2.5 Forme normale de Greibach (FNG)

Il existe une autre forme de normalisation d'une grammaire hors-contexte qui permet de construire plus aisément des analyseurs de langages. C'est la forme normale de Greibach (FNG) dont on va aborder seulement la définition car le travail de transformation en FNG d'une grammaire est plus élaboré et plus raffiné.

### 🎯 Définition 7.2.6 Une grammaire sous forme normale de Greibach (FNG)

Une grammaire hors-contexte  $G = (V, N, S, R)$  est sous forme normale de Greibach (FNG) si et seulement si toutes ses règles de production  $R$  sont sous la forme suivante :

$A \rightarrow aX, a \in V, X \in N^*$ . Sachant que la règle  $S \rightarrow \varepsilon$  est autorisée.

### ✅ Théorème 7.2.2

Pour toute grammaire hors-contexte, il existe une grammaire hors-contexte sous forme normale de Greibach qui génère le même langage.

La forme des productions d'une grammaire mise sous FNG, montre déjà l'intérêt de cette normalisation. On voit bien qu'à chaque dérivation d'un mot, on détermine un préfixe (que des terminaux à gauche) de plus en plus long. Cela permet de construire plus aisément des APs à partir des grammaires, et par conséquent des analyseurs syntaxiques seront plus facilement implémentables.

## 7.2.6 Grammaires hors-contextes et AP

Il a été montré que les grammaires hors-contextes ainsi les automates à piles ont un pouvoir de représentativité équivalent. Les deux représentations décrivent la classe des langages algébriques.

### ✅ Théorème 7.2.3 équivalence entre une grammaire hors-contexte et un AP

Pour tout langage  $L$  généré par une grammaire hors-contexte  $G = (V, N, S, R)$ , il existe un automate à pile  $A$  (déterministe ou non-déterministe) tel que  $L(G) = L(A)$

Par conséquent, il est possible de construire un AP à partir d'une grammaire  $G$ . Pour faciliter la construction,  $G$  est souvent mise sous FNG. (Les étapes de constructions ne sont pas abordées dans ce cours).

## 7.3

## Série d'exercices de TD N°5

**Exercice 1 : Construire un AP et Analyser des mots avec**

Construisez l'AP de chacun des langages suivants. avant d'analyser la liste associée des mots .

1.  $L_1 = \{a^{2n}b^n | n > 0\}$ , mots à analyser :  $\{aabb, aaaabb\}$
2.  $L_2 = \{w \in \{a, b\}^* | |w|_a = |w|_b\}$ , mots à analyser :  $\{aabbabba, aba\}$
3.  $L_3 = \{wc^n | w \in \{a, b\}^*, |w| = n \geq 0\}$ , mots à analyser :  $\{abc, abbccc, cc\}$
4.  $L_4 = \{a^p b^q | p, q \geq 0, p \neq q\}$ , mots à analyser :  $\{aabb, aab, abb\}$
5.  $L_5 = \{a^i b^j | i + j = 2k, k \geq 0\}$ , mots à analyser :  $\{aaab, aaba, bb, abb\}$

**Exercice 2 : AP non-déterminite**

Construisez un AP non déterministe pour chacun des langages suivants :

1.  $L_1 = \{a^i b^j a^i | i, j \geq 0\}$ . Analysez les mots suivants :  $\{aba, abab, abaa, aabab, aaa\}$
2.  $L_2 = \{0^i 1^j 2^{i+j} | i, j \geq 0\}$ . Analysez les mots suivants :  $\{0122, 122\}$
3.  $L_3 = \{a^i b^j c^k | i = j \vee i = k, i, j, k > 0\}$
4.  $L_4 = \{0^i 1^j 2^k | i, j, k \geq 0, i + k = j\}$ .

**Exercice 3 : Ambiguïté d'une grammaire hors-contexte, Arbres de dérivation**

Soient les grammaires et les mots suivants :

1.  $G_1 = (\{ / \}, \{ S \}, S, \{ S \rightarrow S / S | \varepsilon \})$   
 $w_1 = //$
2.  $G_2 = (\{ a, b \}, \{ S \}, S, \{ S \rightarrow S \wedge S | S \vee S | S \Rightarrow S | S \Leftrightarrow S | \neg S | a | b \})$   
 $w_2 = a \wedge b \Leftrightarrow b$
3.  $G_3 = (\{ f, t, \neg, (, ) \}, \{ S \}, S, \{ S \rightarrow S \wedge S | \neg S | t | f \})$   
 $w_3 = f \wedge t, w'_3 = \neg t \wedge t, w''_3 = t \wedge t \wedge \neg f,$ 
  - Que représente  $L(G_i)$
  - Montrez à chaque fois que le mot  $w_i \in L(G_i)$  (en utilisant les arbres syntaxiques)
  - Dites à chaque fois si on peut déduire que  $G_i$  est ambiguë ou non ? Pourquoi ?

**Exercice 4 : Forme Normale de Chomsky**

Mettez les grammaires suivantes sous forme normale de Chomsky (FNC) :

1.  $G = (\{ f, t, \neg, (, ) \}, \{ S \}, S, \{ S \rightarrow S \wedge S | \neg S | t | f \})$
2.  $G = (\{ 0, 1 \}, \{ S, Z, E \}, S, \{ S \rightarrow ZSZ | 0E, Z \rightarrow E | S, E \rightarrow 1 | \varepsilon \})$

---

# RÉFÉRENCES

Nous nous sommes principalement inspirés des livres et des supports suivants pendant l'élaboration de ce cours :

1. Michael Sipser, "Introduction to the theory of computation" 2nd Edition, Course Technology (February 15, 2005).
2. Hollos Stefan, Hollos J. Richard "Finite Automata and Regular Expressions : Problems and Solutions", Abrazol Publishing (August 12, 2013).
3. John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Rotwani, "Introduction to Automata Theory, Languages, and Computation", 2nd Edition Addison-Wesley (2001-2014)
4. Alexander Meduna, "Formal Languages and Computation Models and Their Applications". 1st edition CRC Press (Taylor & Francis group), Auerbach publication (February 11, 2014).
5. Yuval Noah Harari, "Sapiens : Une brève histoire de l'humanité", Albin Michel (2 septembre 2015).
6. BERLIOUX P., LEVY M., "Théorie des langages", Notes de cours, École nationale supérieure d'informatique et de mathématiques appliquées, Grenoble, 2018 ;
7. AISSANI S., "Theorie des langages", Notes de cours, Université A. Mira de Béjaïa, Faculté des Sciences Exactes, Département d'Informatique 2019.
8. Benouhiba T., "Theorie des langages", Notes de cours, Université Badji Mokhtar de Annaba, Faculté des technologies , Département d'Informatique 2020.