Chapitre 4

Les langages algébriques

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

1. Les automates à pile

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exemple 4.1 : un automate à pile

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

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

— $\delta(0, a, A) = (0, AA)$

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

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

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

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

Détaillons l'analyse de quelques mots :

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

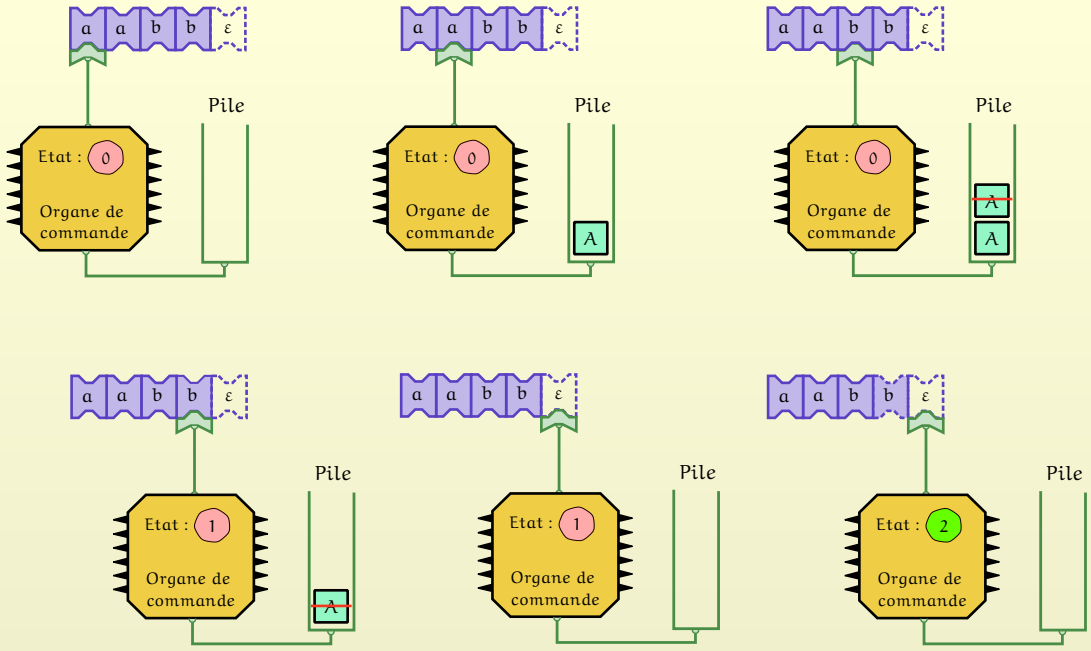

FIGURE 4.1 – Analyse du mot aabb

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

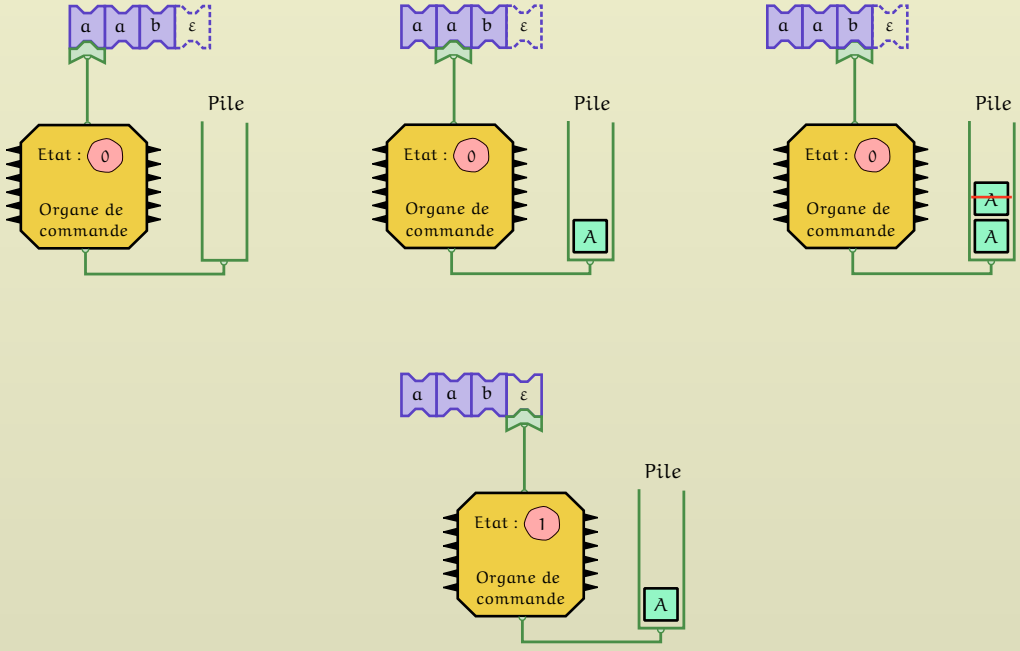

FIGURE 4.2 – Analyse du mot aab

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

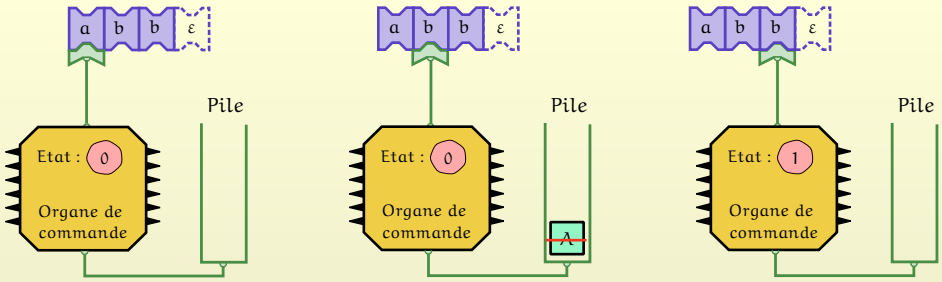

FIGURE 4.3 – Analyse du mot abb

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

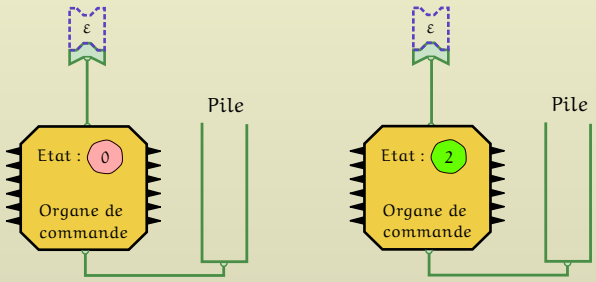

FIGURE 4.4 – Analyse du mot ε

1..1 Les automates à pile et le déterminisme

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

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

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

Exemple 4.2 : automate à pile déterministe

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

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

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

Exemple 4.3 : automate à pile non-déterministe

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

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

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

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

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

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

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

2. Les grammaires hors-contexte

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

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

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

Exemple 4.4 : grammaire hors-contexte

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

2..1 Arbre de dérivation

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

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

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

Exemple 4.5 : arbre de dérivation

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

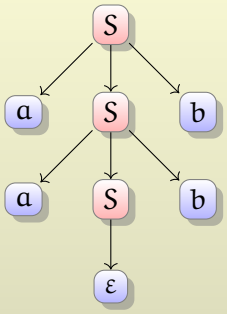

FIGURE 4.5 – Exemple d'un arbre de dérivation

2..2 Notion d'ambiguïté

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

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

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

Exemple 4.6 : lever l'ambiguïté

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

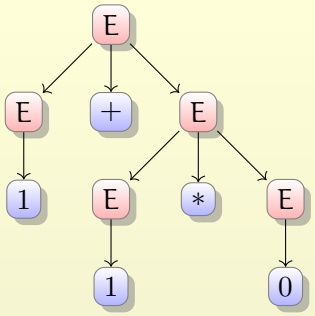

FIGURE 4.6 – Un premier arbre de dérivation

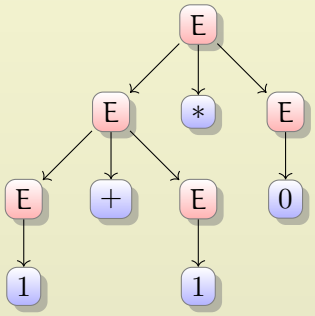

FIGURE 4.7 – Deuxième arbre de dérivation

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

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

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

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

Théorème 4.1

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

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

3. Simplification des grammaires hors-contextes

3..1 Les grammaires propres

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

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

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

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

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

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

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

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

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

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

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

   — Enlever les règles $A \to \varepsilon$

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

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

   — Pour chaque paire (A, B) trouvée

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

   — Enlever toutes les règles $A \to B$

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

4. Les formes normales

4..1 La forme normale de Chomsky

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

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

— $A \to a$, $A \in N$, $a \in V$

— $S \to \varepsilon$

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

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

— $S \to \varepsilon$

— $A \to w$, $w \in V^+$

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

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

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

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

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

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

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

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

sinon, elle sera transformée en :

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

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

Exemple 4.7 : construction de la forme normale de Chomsky

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

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

4..2 La forme normale de Greibach

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

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

— $A \to a$, $a \in V$

— $S \to \varepsilon$

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