1 Transformations de grammaire

1. Soit l'alphabet $X = \{+, =, a\}$. Donner une grammaire algébrique pour le langage L dont chaque mot représente une addition correcte de deux suites de caractères a. Par exemple L contient le mot $aa + aaaa = aaaaaa$.

   Correction On constate que le langage demandé est de la forme $a^n + a^m = a^m a^n$. D'où la grammaire : $S \rightarrow aSa$
   $S \rightarrow a + Ma$
   $M \rightarrow aMa$
   $M \rightarrow a = a$

2. Soit le langage L_1 sur le vocabulaire $V = \{l'$, *homme*, *ours*, *qui*, *a*, *vu*$\}$ formé de l'ensemble des phrases finies de la forme *l'homme qui a vu l'ours*, *l'homme qui a vu l'homme qui a vu l'ours*, *l'homme qui a vu l'homme qui a vu ... qui a vu l'ours*.

   (a) Grammaire algébrique (*context-free*) engendrant L_1 :
   S → l'homme qui a vu S
   S → l'homme qui a vu l'ours

   (b) Grammaire régulière engendrant L_1.
   S → l' A
   A → homme B
   B → qui C
   C → a D
   D → vu E | vu S
   E → l' F
   F → ours

   Soit L_2 le langage engendré par la grammaire $\mathcal{G}_2$:
   S → NP Rel
   NP → l'homme | l'ours
   Rel → qui a vu NP Rel | qui a vu l'ours

   (c) Grammaire $\mathcal{G}_2$ sous forme normale de Chomsky :
   $S \rightarrow NP\ \ Rel$
   $NP \rightarrow L\ \ H \mid L\ \ O$
   $L \rightarrow l'$
   $H \rightarrow homme$
   $O \rightarrow ours$
   $Rel \rightarrow Q\ \ a\_vu\_NP\_Rel \mid Q\ \ a\_vu\_l'ours$
   $Q \rightarrow qui$
   $a\_vu\_NP\_Rel \rightarrow A\ \ vu\_NP\_Rel$
   $A \rightarrow a$
   $vu\_NP\_Rel \rightarrow V\ \ NP\_Rel$
   $V \rightarrow vu$
   $NP\_Rel \rightarrow NP\ \ Rel$
   $a\_vu\_l'ours \rightarrow A\ \ vu\_l'ours$
   $vu\_l'ours \rightarrow V\ \ l'ours$
   $l'ours \rightarrow L\ \ O$

(d) Grammaire $\mathcal{G}_3$ engendrant le langage L_3, sur-ensemble de L_2 mais dans lequel les symboles *ours* et *homme* sont **strictement** interchangeables :
Changer la règle : Rel → qui a vu l'ours
Par la règle : Rel → qui a vu NP

(e) Différences entre les langages L_1 et L_2 : L_2 doit finir par *l'ours*, mais *homme* et *ours* peuvent apparaître librement auparavant.

(f) Grammaire algébrique du langage $L_2 \setminus L_1$:
Il s'agit du langage où les chaînes finissent par *l'ours*, et où auparavant on a au moins une (autre) occurrence de *l'ours*. Avant d'avoir consommé cette occurrence 'intérieure' (non finale) de *l'ours*, on est comme en S. Après consommation, on peut soit terminer, soit consommer indifféremment *l'ours* ou *l'homme*.
S → l'homme qui a vu S
S → l'ours qui a vu A
A → l'ours
A → l'homme qui a vu A
A → l'ours qui a vu A

3. Mettre sous forme normale de Chomsky la grammaire définie par les règles de production suivantes $S \to AB \mid aS \mid a$
$A \to Ab \mid \varepsilon$
$B \to AS$

Correction Il faut commencer par nettoyer la grammaire. Suppression des ε-transitions : $S \to AB \mid B \mid aS \mid a$
$A \to Ab \mid b$
$B \to AS$
Puis suppression des cycles éventuels : pour cela on supprime les productions singulières : $S \to AB \mid AS \mid aS \mid a$
$A \to Ab \mid b$
$B \to AS$
Et enfin la forme quadratique : $S \to AB \mid AS \mid XS \mid a$
$X \to a$
$A \to AY \mid b$
$Y \to b$
$B \to AS$

4. (a) Montrer que la grammaire suivante est ambiguë :
$S \to TU; T \to ST \mid a; U \to US \mid b$
On prouve facilement que *abab* est possible via 2 dérivations.

(b) La grammaire suivante est-elle ambiguë ? $S \to aSSb \mid ab$
Intuitivement, l'ambiguité si elle existe proviendrait de l'utilisation à un moment des deux règles différentes pour les deux symboles S. C'est-à-dire dérivation 1 : $aSSb \to a(aSSb)(ab)b$ versus dérivation 2 $aSSb \to a(ab)(aSSb)b$. Or on voit que les chaînes engendrées par les deux versions sont différentes : les chaînes commencent par au moins 3 a pour la dérivation 1, alors que les chaînes commencent par *aab* pour la dérivation 2.

5. **Grammaires propres :**

(a) Caractériser le langage reconnu par la grammaire suivante $S \longrightarrow cB \mid cS$
$B \longrightarrow aBb \mid \varepsilon$
Définir l'algorithme qui rend une grammaire ε-libre et l'appliquer à la grammaire ci-dessus.

Correction Le langage est $c * a^n b^n$ avec $n \geq 0$
Algorithme rendant une grammaire ε-libre : On commence par calculer l'ensemble des non terminaux dérivant epsilon directement ou indirectement :

```
calculer_epsilon_plus(grammaire)
{
    ( ## initialiser epsilon_plus : au départ l'ensemble vide)
    epsilon_plus := ∅
    epsilon_plus_grossit := vrai
    ( ## tant que epsilon_plus grossit)
    tant que epsilon_plus_grossit est vrai
    {
        epsilon_plus_grossit := faux
        pour chaque règle de grammaire
        {
            si la partie gauche n'est pas déjà dans epsilon_plus
            {
                si tous les symboles de la partie droite ∈ epsilon_plus
                (y compris partie droite vide)
                {
                    Ajouter la partie gauche à epsilon_plus
                    epsilon_plus_grossit := vrai
                }
            }
        }
    }
    return epsilon_plus
}
```

```
supprimer_epsilon_productions(grammaire)
{
    epsilon_plus := calculer_epsilon_plus(grammaire)
    new_rules := ∅
    pour chaque old_rule de grammaire
    {
        On remplace old_rule par un ensemble de nouvelle règles sigma_rules,
        construit de la manière suivante :
            Pour chaque symbole x de la partie droite de old_rule,
            tel que x ∈ epsilon_plus, on considère l'alternative :
                garder x dans la partie droite, ou bien l'enlever
        sigma_rules est constitué de toute la combinatoire de ces alternatives
        moins la chaîne vide.
        C'est à dire que si ε apparaît dans la combinatoire,
        on ne le retient pas dans sigma_rules
        D'où l'algo :
        sigma_rules := {ε}
        Pour chaque symbole x de la partie droite de old_rule
        {
            Pour chaque chaine de sigma_rules
            {
                si x ∈ epsilon_plus
                    remplacer dans sigma_rules,
                    chaine par le couple chaine et chaine+x
                sinon remplacer dans sigma_rules, chaine par chaine+x
            }
        }
        Ajouter à new_rules les règles sigma_rules, sauf celles de la forme X ⟶ ε
    }
    Si ε appartient au langage engendré par grammaire
    {
        Creer un nouvel axiome S' distinct de S (avec S = l'axiome de grammaire)
        Et Ajouter à new_rules les règles S' ⟶ ε|S
        return nouvelle grammaire (axiome = S', règles = new_rules)
    }
    Sinon
    {
        return nouvelle grammaire (axiome = S, règles = new_rules)
    }
}
```

Pour la grammaire citée, cela donne : $epsilon\_plus = B$ et la grammaire devient : $S \longrightarrow cB \mid c \mid cS$
$B \longrightarrow aBb \mid ab$

(b) Définir un algorithme qui détermine la liste des symboles ayant une contribu-

tion ; Utiliser l'algorithme pour simplifier (nettoyer) la grammaire
$S \longrightarrow A \mid B$
$A \longrightarrow aB \mid bS \mid b$
$B \longrightarrow AB \mid Ba$
$C \longrightarrow AS \mid b$

Correction On construit N_0 l'ens. des symboles produisant directement un $\alpha \in X^*$. Puis tant que le point fixe n'est pas atteint, on applique la récurrence : constuire N_i l'ensemble des symboles produisant directement un $\alpha \in (X \cup N_{i-1})^*$.
$i = 0$
$N_i = \emptyset$
repeat
$i = i + 1$
$N_i = N_{i-1} \cup \{A \in V \,/\, A \rightarrow \alpha \text{ et } \alpha \in (N_{i-1} \cup X)^*\}$
jusqu'à $N_i = N_{i-1}$
$N_e = N_i$
Si $S \in N_e$ alors $L(S) \neq \emptyset$.
Cet algorithme fournit la liste de symboles ayant une contribution, on peut donc débarrasser la grammaire de tous les symboles sans contribution (et les règles les concernant). Pour la grammaire citée cela donne : $N_1 = \{A, C\}$ $N_2 = \{A, C, S\}$ $N_3 = \{A, C, S\}$ $L_G(B) = \emptyset$, donc B ss cont.

Version simplifiée :
$S \longrightarrow A$
$A \longrightarrow bS \mid b$
$C \longrightarrow AS \mid b$

(c) Définir un algorithme qui recherche les symboles inaccessibles ; Utiliser l'algorithme pour simplifier (encore) la grammaire précédente.

Correction On initialise l'ensemble des accessibles $Access := \emptyset$, et on utilise une fonction récursive "augmenter_accessibles", qui propage l'accessibilité à partir d'un non terminal A :
augmenter_accessibles(A) :
si $A \in Access$, STOP
sinon
ajouter A à $Access$
Pour toute règle de la forme $A \longrightarrow \alpha$
pour chaque non terminal B apparaissant dans α
augmenter_accessibles(B)

(d) Proposer un algorithme qui supprime d'une grammaire toutes les productions singulières

Correction

Algorithme A partir d'une grammaire G :

i. Pour tout $A \in V$,
construire $N_A = \{B \in V \,/\, A \stackrel{*}{\longrightarrow} B\}$

ii. Construire G' : Ajouter toutes les règles non singulières de G.
Pour tout $A \in V$, pour toute règle non singulière $B \longrightarrow \alpha$ avec $B \in N_A$, ajouter $A \longrightarrow \alpha$ à G'.

6. Eliminer la récursivité gauche de la grammaire ETF : $G = E \rightarrow E + T \mid T, T \rightarrow T \times F \mid F, F \rightarrow (E) \mid a$

Correction Rappel du principe d'élimination de la récursivité : On commence par empêcher de se retrouver dans la situation : $A_i \longrightarrow A_j\alpha$ et $A_j \longrightarrow A_i\beta$. Pour cela, pour toute paire A_i, A_j on laisse (arbitrairement) $A_j \longrightarrow A_i\beta$ mais on remplace intelligemment $A_i \longrightarrow A_j\alpha$, en y remplaçant A_j par toutes ses parties droites possibles.
Algorithme :

Pour tout i de 1 à n faire {
  Pour tout j de 1 à $i-1$ faire {
    Si $A_i \longrightarrow A_j\alpha$ existe, la remplacer par : $A_i \longrightarrow \delta_1\alpha \mid \delta_2\alpha \mid \ldots \mid \delta_h\alpha$
      où $A_j \longrightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_h$
  }
  Éliminer la récursivité immédiate des A_i-productions
}

Ici, choisir un ordre pour les non terminaux : par exemple $A1 = T, A2 = E, A3 = F$.
Pour i=1 ($Ai = T$) Elimination récursivité directe de T :
  remplacer $T \longrightarrow T \times F|F$ par les règles
  $T \longrightarrow FT'$ et $T' \longrightarrow \times FT'|\varepsilon$
Pour i=2 ($Ai = E$), faire
  -Pour j=1 ($Aj = T$) faire : si $E \longrightarrow T\alpha$ existe ? réponse oui avec $\alpha = \varepsilon$.
    On remplace $E \longrightarrow T$ par $E \longrightarrow FT'$
  -Eliminer la récursivité directe de E : On remplace $E \longrightarrow E + T|T$ par les règles $E \longrightarrow TE'$ et $E' \longrightarrow +TE'|\varepsilon$
Pour i=3 ($Ai = F$), faire
  -Pour j=1 ($Aj = T$), faire : si $F \longrightarrow T\alpha$ existe ? réponse non
  -Pour j=2 ($Aj = E$), faire : si $F \longrightarrow E\alpha$ existe ? réponse non
  -Eliminer la récursivité directe de E : rien à faire