On s'intéresse dans ce TP à la conversion d'une grammaire en forme normale de Chomsky, et à l'utilisation de cette dernière pour déterminer si un mot peut être généré par une grammaire hors-contexte.

On représente Σ et V comme des lettres minuscules et capitales de l'alphabet courant, de a à z et de A à Z respectivement. Une lettre de Σ sera donc implémentée par un objet de type `char`, dont le numéro ASCII sera compris entre 97 (a) et 122 (z). Une lettre de V sera implémentée par un objet de type `char`, dont le numéro ASCII sera compris entre 65 (A) et 90 (Z).

Une règle de production $X \to \alpha$ de G sera représentée par le type suivant :

```c
struct Regle {
    char X;
    char* alpha;
};

typedef struct Regle regle;
```

Ainsi, si `r`, de type `regle`, représente la règle $X \to \alpha$, alors `r.X` est égal à X et `r.alpha` est égal à α.

Une grammaire $G = (\Sigma, V, P, S)$ sera alors représentée par le type :

```c
struct Grammaire {
    int taille_V;
    int nb_prod;
    regle* Prod;
};

typedef struct Grammaire grammaire;
```

Si G est représenté par un objet `G` de type `grammaire`, alors `G.taille_V` correspond à la taille de V, avec la contrainte que V contient des lettres consécutives du début d'alphabet (par exemple `"ABCD"`), `G.nb_prod` est un entier correspondant au nombre de règles de production et `G.Prod` est un tableau de règles de production de taille `G.nb_prod`. Enfin, le symbole de départ est la lettre `'A'`.

On rappelle la définition suivante :

Définition

Soit $G = (\Sigma, V, P, S)$ une grammaire hors-contexte. On dit que G est en **forme normale de Chomsky** (FNC) si toutes les règles de production sont dans l'une des formes suivantes :

- $S \to \varepsilon$;
- $X \to YZ$ avec $Y, Z \in V \setminus \{S\}$;
- $X \to a$ avec $a \in \Sigma$.

Exercice 1

1. Écrire une fonction `bool terminal(char c)` qui renvoie `true` si la lettre c est un symbole terminal et `false` sinon. Écrire de même une fonction `bool variable(char c)` qui détermine si c est une variable ou non. On pourra utiliser `(int) c` pour convertir le caractère en numéro ASCII (même si c'est facultatif).
2. Écrire une fonction

   ```c
   grammaire creer_grammaire(int taille_V, char* tab_X, char** tab_alpha)
   ```

   qui prend en argument un entier correspondant à $|V|$, une chaîne de caractères `tab_X` correspondant aux variables (pouvant contenir des doublons), un tableau de chaînes `tab_alpha` supposé de même taille que `tab_X`, et renvoie la grammaire correspondante, c'est-à-dire contenant les règles `tab_X[i]` → `tab_alpha[i]`.

Indication : on rappelle que `int strlen(char* s)` *renvoie la taille de la chaîne de caractères* `s`, *en ayant inclus l'entête* `<string.h>`. *On ne fera pas d'allocation mémoire pour les chaînes de caractères.*

3. Écrire une fonction `void liberer_grammaire(grammaire G)` qui libère l'espace mémoire occupé par une grammaire.
4. Écrire une fonction `bool regle_chomsky(regle r)` qui prend en argument une règle r et détermine si la règle est valide pour une FNC.
5. En déduire une fonction `bool est_FNC(grammaire G)` qui détermine si une grammaire donnée est en FNC.
6. Tester la fonction précédente avec la grammaire G_0 définie par :
   - $A \to BbB$;
   - $B \to Ba \mid \varepsilon$.

   Tester avec la grammaire G_1 définie par :
   - $A \to BC \mid CB \mid DB \mid b$;
   - $B \to BE \mid a$;
   - $C \to b$;
   - $D \to BC$;
   - $E \to a$.

Exercice 2

L'algorithme de Cocke-Younger-Kasami est un algorithme de programmation dynamique qui permet de déterminer l'appartenance d'un mot au langage engendré par une grammaire en FNC. Soit $G = (\Sigma, V, P, S)$ une grammaire hors-contexte en FNC et $u = a_1 \ldots a_n \in \Sigma^*$. Pour $1 \leqslant i \leqslant j \leqslant n$, on pose $X_{ij} = \{X \in V \mid X \Rightarrow^* a_i \ldots a_j\}$. Par définition, on a l'équivalence $u \in L(G) \Leftrightarrow S \in X_{1n}$.

1. Pour $i \in [\![1, n]\!]$, que vaut X_{ii} ?
2. Montrer que pour $j - i > 0$, $X \in X_{ij}$ si et seulement s'il existe $i \leqslant k < j$ et $Y \in X_{ik}$, $Z \in X_{k+1,j}$ tels que $X \to YZ \in P$.

L'algorithme CYK pour un mot non vide utilise les questions précédentes de la manière suivante :

Entrée : grammaire $G = (\Sigma, V, P, S)$ en FNC et $u = a_1 \ldots a_n \in \Sigma^+$.
Début algorithme
  Poser tous les $X_{ij} = \emptyset$.
  Pour $i = 1$ à n **Faire**
    Initialiser X_{ii}.
  Pour $\delta = 1$ à $n - 1$ **Faire**
    Pour $i = 1$ à $n - \delta$ **Faire**
      $j \leftarrow i + \delta$.
      Pour $k = i$ à $j - 1$ **Faire**
        Pour $X \to YZ \in P$ **Faire**
          Si $Y \in X_{ik}$ et $Z \in X_{k+1,j}$ **Alors**
            $X_{ij} \leftarrow X_{ij} \cup \{X\}$.
  Renvoyer $S \in X_{1n}$.

Pour une grammaire `G`, on représente un ensemble X_{ij} par un tableau de booléens `bool* Xij` de taille $|V|$ tel que `Xij[ind]` vaut `true` si et seulement si la variable d'indice `ind` (en commençant à 0 pour la variable A) est X_{ij}.

3. Écrire une fonction `bool CYK(grammaire G, char* u)` qui détermine si $u \in L(G)$, en supposant

$u \neq \varepsilon$.

Indication : si `X` *est une variable, alors* `(int) X - (int) 'A'` *renvoie son indice compris entre* 0 *et* $|V| - 1$.

4. Tester la fonction précédente avec la grammaire G_1 et les mots *aaaba* (vrai), *aabab* (faux) et *aaaaa* (faux). Quel est le langage engendré par G_1 ? Est-ce cohérent ?
5. Tester la fonction précédente avec la grammaire G_2 définie par :
   - $A \to FC \mid FD \mid FE \mid FG \mid \varepsilon$;
   - $B \to FC \mid FD \mid FE \mid FG$;
   - $C \to BD$;
   - $D \to GB$;
   - $E \to BG$;
   - $F \to a$;
   - $G \to b$,

   et les mots *abaabb* (vrai), *bbaaba* (faux) et *ababab* (vrai). Quel est le langage engendré par G_2 ? Est-ce cohérent ?
6. Modifier la fonction précédente pour prendre en compte le cas du mot vide et tester avec les grammaires G_1 et G_2.
7. Déterminer la complexité de l'algorithme de Cocke-Younger-Kasami en fonction de n et $|P|$.

Exercice 3

On souhaite convertir une grammaire G quelconque en une grammaire en FNC faiblement équivalente à G. On va décomposer le travail en plusieurs étapes :

- éliminer les ε-productions, c'est-à-dire les règles de la forme $X \to \varepsilon$;
- éliminer les productions unitaires, c'est-à-dire de la forme $X \to Y$, avec $Y \in V$;
- éliminer les symboles terminaux en dehors des règles de la forme $X \to a$;
- raccourcir les membres droits des règles contenant des variables ;
- rajouter un nouveau symbole de départ qui n'apparaît pas du côté droit d'une règle, en lui ajoutant éventuellement une ε-production.

1. En écrivant des fonctions pour effectuer chacune de ces transformations, écrire une fonction `grammaire FNC(grammaire G)` qui effectue la conversion d'une grammaire quelconque en grammaire en FNC.

   On pourra travailler avec une liste chaînée de règles plutôt qu'un tableau.
2. Tester cette fonction sur la grammaire G_0 et sur la grammaire G_3 définie par :
   - $A \to BC$;
   - $B \to DE$;
   - $C \to aBC \mid \varepsilon$;
   - $D \to b \mid cAc$;
   - $E \to dDE \mid \varepsilon$.