DS7 (4 heures)

Lisez tout le texte avant de commencer. La plus grande importance sera attachée à la clarté, à la précision et à la concision de la rédaction. Toute réponse non justifiée ne sera pas prise en compte. Si vous repérez ce qui vous semble être une erreur d'énoncé, signalez la sur votre copie et poursuivez votre composition en expliquant les raisons de vos éventuelles initiatives. L'usage de tout dispositif électronique est interdit.

Le sujet comporte deux problèmes au choix. Le premier est de type MPI ; le second est de type MPI. Vous traiterez un seul problème en indiquant clairement votre choix au début de votre copie.*

Problème 1 (MPI)

Dans ce problème, on manipule des *grammaires algébriques*. Sauf mention contraire, l'alphabet des terminaux d'une grammaire est noté Σ et ses éléments sont en minuscules. L'alphabet des variables est noté V et ses éléments sont en majuscules. L'axiome d'une telle grammaire est noté S. On définit la taille d'une grammaire G par :

$$|G| = |\Sigma| + |V| + \sum_{X \to m \in \mathcal{R}} |m|$$

où $\mathcal{R}$ est l'ensemble de ses règles de production. On s'intéresse au problème du Mot défini comme suit.

$$\textsc{Mot} : \begin{cases} \textbf{Entrée :} & \text{une grammaire algébrique } G \text{ dont les terminaux sont les éléments de } \Sigma \text{ et } m \in \Sigma^*. \\ \textbf{Question :} & m \text{ appartient-il à } L(G)\,? \end{cases}$$

Grammaires propres et décidabilité du problème du mot

Une grammaire algébrique est dite *propre* si elle ne contient aucune règle de la forme $X \to \varepsilon$ ou $X \to Y$ avec X, Y des symboles non terminaux de la grammaire. On souhaite montrer que toute grammaire peut être rendue propre tout en engendrant les mêmes mots à l'exception du mot vide. Pour ce faire, on supprime les règles fautives dans l'ordre suivant.

- On supprime d'abord les règles de la forme $X \to \varepsilon$.
- On supprime ensuite les règles de la forme $X \to Y$.

Pour la première étape, on utilise l'algorithme suivant.

Algorithme 1 : Suppression des ε-productions

Entrée : une grammaire algébrique $G = (\Sigma, V, S, \mathcal{R})$
Sortie : une grammaire G' telle que $L(G') = L(G) \setminus \{\varepsilon\}$ et sans règle du type $X \to \varepsilon$

1 Calculer $E_0 = \{X \in V \mid X \to \varepsilon \in \mathcal{R}\}$
2 Calculer pour tout $n \geqslant 0$, $E_{n+1} = E_n \cup \{X \in V \mid X \to m \in \mathcal{R} \text{ avec } m \in E_n^*\}$
3 Noter $E = \bigcup_{n \in \mathbb{N}} E_n$
4 Introduire la fonction de substitution σ sur $\Sigma + V$ telle que :

$$\sigma : \begin{cases} a \mapsto a & \text{si } a \in \Sigma \\ X \mapsto X & \text{si } X \in V \text{ mais } X \notin E \\ X \mapsto X + \varepsilon & \text{si } X \in E \end{cases}$$

Étendre σ aux mots via $\sigma(m_1 \dots m_k) = \sigma(m_1) \dots \sigma(m_k)$
5 **pour** *chaque règle* $X \to m$ *de* G **faire**
6   **pour** *tout mot* $u \in \sigma(m)$ **faire**
7   Ajouter à un ensemble $\mathcal{R}'$ initialement vide la règle $X \to u$
8 Supprimer toutes les règles de la forme $X \to \varepsilon$ de $\mathcal{R}'$
9 **renvoyer** $(\Sigma, V, S, \mathcal{R}')$

Pour donner un exemple, si $X \to AbAB$ est une règle de $\mathcal{R}$ et que A appartient à l'ensemble E calculé ci-dessus mais pas B, $\sigma(AbAB) = (A+\varepsilon)b(A+\varepsilon)B = \{AbAB, AbB, bAB, bB\}$ et donc,lors du traitement de cette règle, on ajoutera à $\mathcal{R}'$ les quatre règles : $X \to AbAB \mid AbB \mid bAB \mid bB$.

Question 1. Déterminer l'ensemble E dans le cadre de la grammaire G_0 d'axiome S caractérisée par les règles suivantes : $S \to AB \mid aA \mid Cb, A \to BB \mid a, B \to b \mid \varepsilon$ et $C \to c$. En déduire les règles de la grammaire qu'on obtient à partir de G_0 via l'algorithme 1.

Question 2. Justifier brièvement qu'en ligne 3 de l'algorithme 1, $E = \{X \in V \mid X \Rightarrow^* \varepsilon\}$.

Question 3. Montrer que pour une grammaire donnée il existe un entier n_0 tel que $\bigcup_{n \in \mathbb{N}} E_n = \displaystyle\bigcup_{n=0}^{n_0} E_n$. En déduire qu'à une petite modification près en ligne 2 qu'on explicitera, l'algorithme 1 termine.

Question 4. En s'inspirant éventuellement de l'algorithme 1, montrer que le problème suivant est décidable via un algorithme polynomial en $|G|$. On justifiera la complexité de l'algorithme proposé.

$$\textsc{Mot Vide} : \begin{cases} \textbf{Entrée :} & \text{une grammaire algébrique } G. \\ \textbf{Question :} & \varepsilon \in L(G)\,? \end{cases}$$

On poursuit la mise sous forme propre d'une grammaire en lui appliquant l'algorithme suivant.

Algorithme 2 : Suppression des règles variable-variable

Entrée : une grammaire algébrique $G = (\Sigma, V, S, \mathcal{R})$ sans règle de la forme $X \to \varepsilon$
Sortie : une grammaire G' telle que $L(G') = L(G)$ et sans règle du type $X \to Y$

1 **pour** *toute variable* $X \in V$ **faire**
2   Déterminer $E(X) = \{Y \in V \mid X \Rightarrow^* Y\}$
3 **pour** *tout* $X \in V$ **faire**
4   **pour** $Y \in E(X)$ **faire**
5   **pour** *toute règle* $Y \to m$ *avec* $m \notin V$ **faire**
6   Ajouter à $\mathcal{R}$ la règle $X \to m$
7 Supprimer toutes les règles de la forme $X \to Y$ de $\mathcal{R}$
8 **renvoyer** $(\Sigma, V, S, \mathcal{R})$

Question 5. Appliquer cet algorithme à la grammaire définie par les règles suivantes.

$$S \to A \mid AB \mid C \mid a \qquad A \to aA \mid B \mid C \qquad B \to a \mid b \qquad C \to A \mid cc$$

Question 6. Expliquer comment calculer l'ensemble de la ligne 2 en s'aidant d'un graphe orienté dont les sommets sont étiquetés par les variables et dont on précisera les arcs.

Question 7. Montrer que la complexité de l'algorithme 2 est polynomiale en la taille de G.

Question 8. Justifier qu'appliquer l'algorithme 1 à une grammaire G pour produire G puis appliquer l'algorithme 2 à G pour produire G'' permet d'obtenir une grammaire G'' propre.

Question 9. Si dans la question précédente on appliquait d'abord l'algorithme 2 puis l'algorithme 1, produirait-on une grammaire G' propre ? Justifier.

Question 10. On admet qu'appliquer l'algorithme 1 puis l'algorithme 2 à une grammaire G produit G' telle que $L(G') = L(G) \setminus \{\varepsilon\}$. La possibilité de rendre propre une grammaire sans trop changer le langage qu'elle engendre permet d'étudier la décidabilité du problème Mot.
Si $G = (\Sigma, V, S, \mathcal{R})$ est propre et qu'on dérive immédiatemment un mot $m \in (\Sigma + V)^+$ en un mot $m' \in (\Sigma + V)^*$ en utilisant une règle $X \to u$ avec $u \notin \Sigma$, que peut-on dire de la taille de m' par rapport à celle de m ?

Question 11. En déduire que si un mot $m \neq \varepsilon$ est engendré par une grammaire propre G alors il est dérivable en moins de $k \in \mathbb{N}$ dérivations avec un entier k qu'on précisera.

Question 12. En déduire un algorithme permettant de décider le problème du mot dont on évaluera la complexité. Quelle est, *a priori*, la classe de complexité du problème Mot ?

Grammaires sous forme normale de Chomsky

Une grammaire est dite sous *forme normale de Chomsky* si toutes ses règles sont de l'une des formes suivantes.

- $X \to a$ avec $a \in \Sigma$ une lettre de l'alphabet des terminaux.
- $X \to YZ$ avec $Y, Z \in V$ des lettres de l'alphabet des non terminaux.

On souhaite montrer que toute grammaire G peut être mise sous forme normale de Chomsky sans modifier le langage engendré, à ε près. Pour ce faire, on propose de suivre l'algorithme suivant.

Algorithme 3 : Mise sous forme normale de Chomsky

Entrée : une grammaire algébrique $G = (\Sigma, V, S, \mathcal{R})$
Sortie : une grammaire G' sous forme normale de Chomsky telle que $L(G') = L(G) \setminus \{\varepsilon\}$

Rendre G propre à l'aide des algorithmes de la partie précédente
pour *toute règle de la forme* $X \to m_1 \dots m_k$ *avec* $k \geqslant 3$ **faire**
    Supprimer cette règle de $\mathcal{R}$
    Ajouter à $\mathcal{R}$ les règles $X \to m_1 X_1$, $X_1 \to m_2 X_2$, $X_2 \to m_3 X_3$, ..., $X_{n-3} \to m_{n-2} X_{n-2}$, $X_{n-2} \to m_{n-1} m_n$ où les X_i sont de nouvelles variables qu'on ajoute à V
pour *toute lettre* $a \in \Sigma$ **faire**
    Ajouter une variable X_a à V
    pour *toute règle* $X \to m \in \mathcal{R}$ *avec* $|m| = 2$ **faire**
        Remplacer toutes les occurrences de a dans m par X_a
    Ajouter une règle $X_a \to a$ à $\mathcal{R}$
renvoyer *la grammaire obtenue par les modifications précédentes de* V *et* $\mathcal{R}$

Les modifications effectuées par l'algorithme 3 entre les lignes 2 et 4 portent dans la suite le nom d'*étape quadratique*. Les modifications effectuées par l'algorithme 3 entre les lignes 5 et 9 portent dans la suite le nom d'*étape variable*.

Question 13. Appliquer cet algorithme à la grammaire définie par les règles suivantes.

$$S \to ASA \mid aB \qquad A \to B \mid a \qquad B \to b \mid \varepsilon$$

Question 14. Expliquer brièvement pourquoi la grammaire obtenue via l'algorithme 3 engendre le même langage que la grammaire en entrée, à ε près.

Question 15. Montrer qu'en fin de l'algorithme 3, la grammaire obtenue est bien sous forme normale de Chomsky.

On se propose à présent d'implémenter partiellement la mise sous forme normale de Chomsky en OCaml. On pourra utiliser les fonctions des modules `List` et `Array`. Plus précisément, on se donne une grammaire propre. L'objectif est de la mettre sous forme normale de Chomsky en lui appliquant l'étape quadratique puis l'étape variable. On se dote du type `symbole`.

```
type symbole =  Term of char | Var of char | Dupl of char * int | X of char
type regle = { gauche : symbole ; droite : symbole list }
type grammaire = regle list
```

Il permet de représenter les lettres impliquées dans les règles d'une grammaire.

- Si $a \in \Sigma$ est un symbole terminal, il est représenté par `Term 'a'`.
- Si X est un symbole non terminal (variable), il est représenté par `Var 'X'`.
- Chacune des variables X_i introduites par l'étape quadratique est représentée par `Dupl ('X',i)`.
- Chacune des variables X_a introduites par l'étape variable est représentée par `X 'a'`.

Initialement, une grammaire ne fait intervenir que des symboles construits avec `Term` ou avec `Var`. Le type `regle` permet de représenter une règle : le champ gauche contient le non terminal à gauche de la règle et le champ droite contient dans l'ordre la liste des symboles à droite de la règle. Enfin, le type `grammaire` permet de représenter une grammaire. Par défaut, l'axiome d'une telle grammaire sera toujours `Var 'S'`. Les terminaux et non terminaux sont implicitement connus via les règles. Par exemple, la grammaire de la question 13 serait définie en OCaml par la liste de règles suivante.

```
[
  {gauche = Var 'S'; droite = [Var 'A'; Var 'S'; Var 'A']};    (* S -> ASA *)
  {gauche = Var 'S'; droite = [Term 'a'; Var 'B']};            (* S -> aB *)
  {gauche = Var 'A'; droite = [Var 'B']};                      (* A -> B *)
  {gauche = Var 'A'; droite = [Term 'a']};                     (* A -> a *)
  {gauche = Var 'B'; droite = [Term 'b']};                     (* B -> b *)
  {gauche = Var 'B'; droite = []}                              (* B -> epsilon *)
]
```

Question 16. Écrire une fonction `est_terminal (s : symbole) : bool` renvoyant `true` si et seulement si le symbole en entrée est un symbole terminal.

Question 17. Écrire une fonction `char_of_symb (s : symbole) : char` qui extrait le caractère sous jacent à un symbole. Par exemple, le caractère sous-jacent à `Dupl('X', 3)` est `'X'` et celui de `Term 'o'` est `'o'`.

Question 18. Écrire une fonction `regle_eligible (r : regle) : bool` qui renvoie `true` si et seulement si la règle en entrée doit être traitée par l'étape quadratique.

Pour implémenter l'étape quadratique, il faut, pour chaque variable initiale de la grammaire X, se rappeler quel est l'entier i tel que la prochaine variable dérivée de X à utiliser soit X_i. Pour ce faire, on se sert d'un dictionnaire de type `(char, int) Hashtbl.t`. Les clés sont des caractères. La valeur associée au caractère `c` est le plus petit entier `i` tel que `Dupl(c,i)` n'ait pas encore été utilisé. L'annexe rappelle quelques fonctions utilisables du module `Hashtbl`.

Question 19. Écrire une fonction `variables_initiales (g : grammaire) : (char, int) Hashtbl.t` qui créé un dictionnaire contenant les associations `(c, 1)` pour tout caractère `c` intervenant dans un symbole de gauche d'une des règles de `g`.

Question 20. Écrire une fonction `incrementer (numerotation : (char, int) Hashtbl.t) (v : char) : unit` qui incrémente de un la valeur associée à une clé supposée présente dans un dictionnaire.

Question 21. Écrire une fonction `eclater_regle (v : symbole) (prod : symbole list) (regles : regle list) (numerotation : (char,int) Hashtbl.t) : regle list`.

- `v` représente une variable X et `prod` un mot $m = m_1 \dots m_k$ tel que $X \to m$ soit une règle d'une grammaire, éligible à être transformée par l'étape quadratique.

- `regles` représente un ensemble de règles quelconque.
- `numerotation` est un dictionnaire de renumérotation de variables tel que précédemment décrit ; on suppose qu'il est à jour au début de l'appel de la fonction.
- Cette fonction doit renvoyer l'ensemble de règles constitué des règles de `regles` et de toutes les règles $X \to m_1X_1, X_1 \to m_2X_2, \dots, X_{k-2} \to m_{k-1}m_k$ créées par l'éclatement de la règle $X \to m$ lors de l'étape quadratique (voir ligne 4 de l'algorithme 3).

Question 22. En déduire une fonction `etape_quadratique (g : grammaire) : grammaire` qui renvoie la grammaire issue de la grammaire `g` après lui avoir appliqué l'étape quadratique.

On admet dans la suite qu'on dispose d'une fonction `liste_terminaux (g : grammaire) : char list` qui calcule la liste sans doublons de tous les terminaux qui interviennent dans la grammaire en entrée.

Question 23. Écrire une fonction `remplacer_terminaux (g : grammaire) : grammaire` qui renvoie la grammaire dont les règles sont les mêmes que celles de `g` sauf que tous les terminaux a dans ses membres droits contenant deux symboles ont été remplacés par la variable X_a (voir lignes 7 et 8 de l'algorithme 3).

Question 24. En déduire une fonction `etape_variable (g : grammaire) : grammaire` qui renvoie la grammaire issue de la grammaire `g` après lui avoir appliqué l'étape variable.

Question 25. En déduire enfin une grammaire `chomsky (g : grammaire) : grammaire` qui renvoie une grammaire sous forme normale de Chomsky équivalente à la grammaire `g` qu'on suppose propre sans le vérifier.

Algorithme de Cocke-Younger-Kasami

Dans cette partie, le langage de programmation est le C. On suppose que les bibliothèques `string.h`, `stdlib.h` et `stdbool.h` ont été chargées.

On considère un langage algébrique $L(G)$ ne contenant pas ε, décrit par une grammaire algébrique $G = (\Sigma, V, S, \mathcal{R})$ sous forme normale de Chomsky. On rappelle que dans ce cas, les règles de G sont nécessairement de la forme $X \to a$ ou $X \to YZ$ avec $X, Y, Z \in V$ et $a \in \Sigma$.
Soit un mot $m = m_1 \dots m_n \in \Sigma^+$. On cherche à savoir s'il appartient à $L(G)$. Pour ce faire, on introduit pour tous $i, j \in [\![1, n]\!]$ tel que $i \leqslant j$ l'ensemble $E_{i,j}$ des non terminaux de G qui engendrent le mot $m_i \dots m_j$. L'objectif est de calculer $E_{1,n}$: l'axiome S en fait partie si et seulement si $m \in L(G)$.

Question 26. Si $i \in [\![1, n]\!]$, expliquer comment déterminer l'ensemble $E_{i,i}$ à partir de G.

Question 27. Si $i, j \in [\![1, n]\!]$ et $i < j$, montrer que :

$$E_{i,j} = \bigcup_{k=i}^{j-1} \{X \in V \mid X \to YZ, Y \in E_{i,k} \text{ et } Z \in E_{k+1,j}\}$$

On considère alors l'algorithme de programmation dynamique suivant dû à *Cocke*, *Younger* et *Kasami*.

Algorithme 4 : Algorithme de Cocke-Younger-Kasami (CYK)

Entrée : une grammaire G sous forme normale de Chomsky et un mot m de taille $n \neq 0$
Sortie : vrai si $m \in L(G)$ et faux sinon

Initialiser une matrice $E = (e_{i,j})_{i,j \in [\![1,n]\!]}$ remplie de $\varnothing$
pour *i allant de 1 à n* **faire**
  pour *toute règle de la forme $X \to a$ de G* **faire**
    si $a = m_i$ **alors**
      $e_{i,i} \leftarrow e_{i,i} \cup \{X\}$
pour *d allant de 1 à $n-1$* **faire**
  pour *$e_{i,j}$ sur la d-ème surdiagonale de E* **faire**
    pour *k allant de i à $j-1$* **faire**
      pour *toute règle de la forme $X \to YZ$ de G* **faire**
        si $Y \in e_{i,k}$ *et* $Z \in e_{k+1,j}$ **alors**
          $e_{i,j} \leftarrow e_{i,j} \cup \{X\}$
renvoyer *le booléen* $(S \in e_{1,n})$

Question 28. On considère la grammaire G_{ex} définie par les règles suivantes.

$$S \to XY \qquad T \to ZT \mid a \qquad X \to TY \qquad Y \to YT \mid b \qquad Z \to TZ \mid b$$

En utilisant l'algorithme précédent, déterminer si $abab$ appartient à $L(G_{ex})$. On dessinera explicitement la matrice E et le contenu de ses cases en fin d'algorithme.

Question 29. Justifier la correction de cet algorithme.

Question 30. Déterminer sa complexité en fonction de $|m|$ et d'une autre grandeur pertinente. Dans quelle classe de complexité se trouve finalement le problème Mot ?

La fin de cette partie fait implémenter l'*algorithme de Cocke-Younger-Kasami*. Les terminaux sont un sous-ensemble des caractères ASCII. Les non terminaux sont représentés par des entiers numérotés consécutivement à partir de 0. L'axiome porte systématiquement le numéro 0.

Une règle dans une grammaire sous forme normale de Chomsky est représentée à l'aide de la structure suivante.

```
struct regle {
  int type; // 1 ou 2
  int membre_gauche;
  char lettre;
  int variable1;
  int variable2;
}
typedef struct regle regle;
```

Le type d'une règle est un entier valant 1 si la règle est de la forme $X \to a$ et 2 si elle est de la forme $X \to YZ$. Dans les deux cas, le champ `membre_gauche` contient le numéro du non terminal X. Dans le premier cas, le champ `lettre` contient a et les champs `variable1` et `variable2` contiennent -1. Dans le deuxième cas, les champs `variable1` et `variable2` contiennent les numéros de Y et Z respectivement et le champ `lettre` un caractère ne faisant pas partie de l'ensemble des terminaux de la grammaire considérée.
Une grammaire sous forme normale de Chomsky sera représentée à l'aide de la structure suivante.

```
struct grammaire {
  int nb_variables;
  int nb_regles;
  regle* productions;
};
typedef struct grammaire grammaire;
```

Si `g` est un objet de type `grammaire` représentant $G = (\Sigma, V, S, \mathcal{R})$, `g.nb_variables` correspond à $|V|$, `g.nb_regles` à $|\mathcal{R}|$ et `g.productions` est un tableau de taille `g.nb_regles` contenant les règles de G.

Question 31. Indiquer les valeurs des champs `nb_variables` et `nb_regles` d'un objet de type `grammaire` représentant la grammaire G_{ex} définie en question 28. Après avoir précisé une numérotation des non terminaux, allouer deux objets de type `regle*` sur le tas et les initialiser pour qu'ils représentent les règles $S \to XY$ et $Y \to b$ de G_{ex}.

Pour représenter un ensemble de non terminaux d'une grammaire donnée sur un ensemble de variables V, on utilisera un tableau de taille $|V|$ rempli de booléens. À la case i, ce tableau contiendra true si la variable numéro i est présente dans l'ensemble et false sinon.

Question 32. Écrire une fonction `bool CYK(grammaire* g, char m[])` implémentant l'algorithme de Cocke-Younger-Kasami. On expliquera les idées ne découlant pas directement de la lecture du pseudo-code.

Annexe

- `('a, 'b) Hashtbl.t` est le type des dictionnaires dont les clés sont de type `'a` et les valeurs de type `'b`.
- `let dico = Hashtbl. create 42` permet de créer un dictionnaire vide dico de taille `42`. La taille est automatiquement augmentée si le dictionnaire est plein.
- `Hashtbl.mem dico c` teste l'appartenance de la clé `c` dans `dico`.
- `Hashtbl.find dico c` renvoie la valeur `v` associée à la clé `c` dans `dico` si elle existe. Si elle n'existe pas, l'exception `Not_found` est levée.
- `Hashtbl. replace dico c v` permet de remplacer l'éventuelle association de clé `c` déjà présente dans `dico` par l'association `(c, v)`. S'il n'existe pas d'association dont la clé est `c` dans `dico`, alors cette opération ajoute l'association `(c, v)` dans `dico`.

Problème 2 (MPI*)

Dans ce problème, le langage de programmation est OCaml. Sauf mention contraire explicite, on pourra utiliser toutes les fonctions des modules `List` et `Array`.

Une *grammaire algébrique* est notée $G = (\Sigma, V, S, \mathcal{R})$ où Σ est un alphabet de terminaux appelés *tokens* et V est un alphabet de variables. Pour tout $X \in V$, on note $L_G(X)$ l'ensemble des mots de Σ^* engendrés dans G par X. On pourra noter $L(X)$ plutôt que $L_G(X)$ lorsqu'il n'y a pas de risque d'ambiguité. On ne travaillera qu'avec des grammaires *accessibles*, c'est-à-dire telles que pour tout $X \in V$, il existe $\alpha, \beta \in (\Sigma \cup V)^*$ et une dérivation telle que $S \Rightarrow^* \alpha X \beta$.

Le principe d'une *analyse syntaxique descendante* consiste à écrire des fonctions mutuellement récursives permettant de construire l'arbre syntaxique d'un mot m à partir de la racine en lisant et consommant les tokens de m au fur et à mesure.

On considère la grammaire G_0 sur $\Sigma = \{1, +, \times, (,)\}$ définie par les règles suivantes.

$$S \to S + S \mid S \times S \mid (S) \mid 1$$

Question 1. Montrer que cette grammaire est ambiguë.

Question 2. Quel autre défaut présente-t-elle si on souhaite écrire un analyseur syntaxique descendant ?

Tables d'analyse syntaxique

Une *table d'analyse syntaxique* d'une grammaire $G = (\Sigma, V, S, \mathcal{R})$ est un tableau destiné à en faciliter l'analyse syntaxique descendante.

- Chaque ligne correspond à une variable $X \in V$.
- Chaque colonne correspond à un token $a \in \Sigma$.
- La case en ligne X et colonne a contient les règles qui peuvent être appliquées lorsque le prochain token lu est a et qu'on essaie de construire le mot commençant par ce token par une dérivation gauche de X.

Considérons par exemple une version simplifiée G_L de la grammaire du langage de programmation LISP. L'ensemble de ses terminaux est $\Sigma = \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\}$, l'ensemble de ses non terminaux est $V = \{S, L, E\}$, son symbole initial est S et ses règles sont :

$$S \to L\texttt{\#} \qquad L \to \varepsilon \mid EL \qquad E \to \texttt{sym} \mid \texttt{(}L\texttt{)}$$

La table d'analyse syntaxique de G_L est alors représentée par le tableau suivant.

	`sym`	`(`	`)`	`#`
S	$S \to L\texttt{\#}$	$S \to L\texttt{\#}$		$S \to L\texttt{\#}$
L	$L \to EL$	$L \to EL$	$L \to \varepsilon$	$L \to \varepsilon$
E	$E \to \texttt{sym}$	$E \to \texttt{(}L\texttt{)}$		

Par exemple le contenu de la case $(L, \texttt{sym})$ est $L \to EL$ car il est possible d'obtenir un mot commençant par `sym` dérivant de L via la dérivation $L \Rightarrow EL \Rightarrow \texttt{sym}L$. Les cases vides représentent des impossibilités. Ainsi la case vide en ligne S et colonne `)` indique qu'il est impossible de dériver un mot qui commence par `)` à partir de S selon les règles de G_L.

Pour réaliser l'analyse syntaxique d'un mot engendré par G_L, on définit le type suivant.

```
type token = Sym | Lpar | Rpar | Eof
```

Le constructeur `Sym` représente le token `sym`, `Lpar` le token `(`, `Rpar` le token `)` et `Eof` le terminal `#`. Un mot est représenté par une liste d'éléments de type `token`.

Question 3. Écrire les trois fonctions suivantes telles `parser_X` consomme un préfixe maximal de la liste en entrée pouvant être généré à partir de `X` et renvoie la liste résultant de la suppression des tokens utilisés pour former ce préfixe. Si un tel préfixe n'existe pas, l'exception `SyntaxError` est levée. Expliquer en quoi la table d'analyse syntaxique de G_L pour écrire ces fonctions.

```
parser_S : token list -> token list
parser_L : token list -> token list
parser_E : token list -> token list
```

Question 4. En déduire une fonction `engendrer : token list -> bool` qui renvoie `true` si et seulement si le mot en entrée est engendré par la grammaire G_L.

Question 5. Expliquer brièvement et sans l'implémenter ce qu'il faudrait modifier aux fonctions précédentes pour obtenir à la place d'un résultat booléen un arbre syntaxique pour un mot selon la grammaire G_L quand il existe.

À l'issue de cette partie se posent (au moins) deux questions.

- Comment construire une table d'analyse syntaxique pour une grammaire ?
- Une table d'analyse syntaxique permet-elle toujours une analyse descendante aisée ?

Les parties suivantes apportent quelques réponses à ces questions.

Calcul de symboles nuls

Soit $G = (\Sigma, V, S, \mathcal{R})$ une grammaire. Si $X \in V$, on dit que X est *nul* si $\varepsilon \in L_G(X)$. On définit le prédicat *Nul*(X) comme valant vrai si X est nul et faux sinon.

Question 6. En justifiant votre réponse, indiquer quels sont les symboles nuls de la grammaire G_{ex} décrite par les règles suivantes.

$$S \to SA \mid A \mid B \qquad A \to AC \mid CC \mid a \qquad B \to b \qquad C \to c \mid \varepsilon$$

Pour déterminer *Nul*(X) pour toute variable X, on propose l'algorithme suivant.

(1) Initialement, on fixe *Nul*(X) à `false` pour tout non terminal X.

(2) Pour chaque non terminal X, on affecte la valeur `true` à *Nul*(X) s'il existe une production $X \to \varepsilon$ ou une production $X \to X_1 \dots X_p$ avec les X_i des non terminaux tels que pour tout i on ait *Nul*(X_i).

(3) Si l'étape (2) a modifié au moins une valeur *Nul*(X), on recommence l'étape (2).

Question 7. Montrer que cet algorithme termine.

Question 8. Montrer que cet algorithme détermine bien les valeurs de *Nul*(X).

Sans la suite de cette partie, on cherche à déterminer les symboles nuls d'une grammaire $G = (\Sigma, V, S, \mathcal{R})$. Pour ce faire, on représente les éléments de Σ via un type `token` prédéfini indépendant du type introduit à la partie précédente (il n'y a pas besoin d'en connaitre la définition précise). Les éléments de V sont représentés par $[\![0, |V|-1]\!]$ avec comme convention que S est représenté par 0. Un élément de $\Sigma \cup V$, qu'on appelle un symbole, est ainsi représenté par le type `symbole` ci-dessous. Un mot de $(\Sigma \cup V)^*$ est représenté par une liste de symboles. Enfin, une grammaire est représentée par un tableau de taille $|V|$ tels que sa case i contienne la liste des mots pouvant être dérivés depuis la variable numéro i, notée X_i, dans la grammaire.

```
type symbole = T of token | V of int
type mot = symbole list
type grammaire = mot list array
```

Par exemple, en supposant que `a`, `b` et `c` sont de type `token`, si on choisit d'attribuer le numéro 1 à A, 2 à B et 3 à C, alors la grammaire G_{ex} est représentée par :

```
let g1 = [|
  [[V 0; V 1]; [V 1]; [V 2]];
  [[V 1; V 3]; [V 3; V 3]; [T a]];
  [[T b]];
  [[T c]; []]
|]
```

Pour alléger la suite, on définit au fil du problème un certain nombre de variables de manière globale plutôt que de les passer en argument à chaque fonction. En particulier, à partir de la question suivante, on suppose que `g` est une variable globale représentant une certaine grammaire G dont on suppose que toutes les variables sont accessibles.

Question 9. Écrire une fonction `calculer_nuls : unit -> bool array` qui renvoie un tableau de booléens de taille $|V|$ contenant `true` en case i si et seulement si *Nul*(X_i) selon la grammaire `g`.

Question 10. Déterminer sa complexité temporelle pire cas en fonction de $|\mathcal{R}|$, $|V|$ et $n_{\max}$, taille maximale d'un mot α tel qu'il existe une règle $X \to \alpha$ dans la grammaire G représentée par `g`.

On suppose à présent défini de manière globale le tableau des variables nulles de G via :

```
let nuls = calculer_nuls ()
```

On étend la définition de *Nul* à l'ensemble des mots de $(\Sigma \cup V)^*$: un mot α est dit *nul* si $\alpha \Rightarrow^* \varepsilon$.

Premiers et suivants

Pour $\alpha \in (\Sigma \cup V)^*$ on définit *Premier*(α) par :

$$\mathit{Premier}(\alpha) = \{a \in \Sigma \mid \exists \beta \in (\Sigma \cup V)^*, \alpha \Rightarrow^* a\beta\}$$

C'est l'ensemble des tokens qui peuvent apparaître en début d'un mot obtenu par dérivation à partir du mot α.

Question 11. Pour la grammaire G_{ex} définie en question 6, déterminer sans justifier les ensembles *Premier*(X) pour tout $X \in V$ ainsi que les ensembles *Premier*(SA), *Premier*(AC) et *Premier*(CC).

Question 12. De manière générale, que vaut *Premier*(ε) ? Et *Premier*(a) quand $a \in \Sigma$?

Question 13. Pour tout $x \in \Sigma \cup V$ et tout $\alpha \in (\Sigma \cup V)^*$, exprimer *Premier*$(x\alpha)$ en fonction de Premier (x), *Premier*(α) et *Nul*(x).

Question 14. Décrire en langage naturel ou en pseudo-code un algorithme permettant de déterminer les *Premier*(X) pour tout $X \in V$.

On suppose disposer d'une structure de données persistante de type `TokenSet.t` correspondant à des ensembles sans doublons de tokens. Certaines de ses primitives sont les suivantes.

- `vide` est une constante de type `TokenSet.t` qui correspond à un ensemble vide de tokens.
- `appartient : token -> TokenSet.t -> bool` détermine si un token est présent dans un ensemble.
- `ajouter : token -> TokenSet.t -> TokenSet.t` ajoute un token à un ensemble.
- `unir : TokenSet.t -> TokenSet.t -> TokenSet.t` calcule l'union de deux ensembles.
- `intersecter : TokenSet.t -> TokenSet.t -> TokenSet.t` en calcule l'intersection.
- `cardinal : TokenSet.t -> int` calcule le cardinal d'un ensemble de tokens.
- `to_list : TokenSet.t -> token list` renvoie une liste contenant les éléments de l'ensemble.
- `est_inclus : TokenSet.t -> TokenSet.t -> bool` teste si le premier ensemble est inclus dans le second.

Question 15. Écrire une fonction `calculer_premiers : unit -> TokenSet.t array` qui renvoie un tableau de taille $|V|$ contenant en case i l'ensemble correspondant à *Premier*(X_i) (ces variables étant celles de `g`).

Question 16. On suppose à présent défini de manière globale le tableau des premiers pour les variables de G via :

```
let premiers = calculer_premiers ()
```

Déduire des questions précédentes une fonction `premiers_mot : mot -> TokenSet.t` qui détermine l'ensemble de tokens correspondant à *Premier*(m) où m est le mot en entrée.

Pour une variable $X \in V$, on définit *Suivant*(X) par :

$$\textit{Suivant}(X) = \{a \in \Sigma \mid \exists \alpha, \beta \in (\Sigma \cup V)^*, S \Rightarrow^* \alpha X a \beta\}$$

C'est l'ensemble des tokens qui peuvent immédiatement suivre la variable X dans un mot obtenu par une dérivation depuis l'axiome S. On suppose dans la suite l'existence d'un token particulier `EOF` (*End Of File*). S'il existe $\alpha \in (\Sigma \cup V)^*$ tel que $S \Rightarrow^* \alpha X$, alors on impose à *Suivant*(X) de contenir `EOF`. En particulier, on aura toujours `EOF` $\in$ *Suivant*(S).

Question 17. Déterminer, sans justifier, les ensembles *Suivant*(X) pour toutes les variables X de G_{ex}.

Question 18. Soit $X \in V$. On décompose toute règle $X_i \to m$ faisant intervenir X dans la production m sous la forme $X_i \to \alpha_i X \beta_i$. Notons qu'une règle comme $Y \to aXbXa$ sera décomposée deux fois (une fois avec $\alpha = a$ et $\beta = bXa$ et une autre fois avec $\alpha = aXb$ et $\beta = a$). En justifiant votre réponse, établir une formule permettant de calculer *Suivant*(X) en fonctions des X_i, α_i et β_i.

Dans la suite, on suppose disposer d'une fonction `calculer_suivants` qui renvoie un tableau de taille $|V|$ contenant en case i l'ensemble *Suivant*(X_i) et d'une variable globale stockant les suivants de G via :

```
let suivants = calculer_suivants ()
```

Tables syntaxiques pour les grammaires LL(1)

On dit qu'une grammaire $G = (\Sigma, V, S, \mathcal{R})$ est LL(1) (pour *Left to right*, *Leftmost derivation*, *1 token*) si et seulement si pour tout couple de règles $X \to \alpha \mid \beta$ on a :

- *Premier*$(\alpha) \cap$ *Premier*$(\beta) = \varnothing$.
- si *Nul*(β) alors $(\neg$*Nul*(α) et *Premier*$(\alpha) \cap$ *Suivant*$(X) = \varnothing)$.

Question 19. Montrer que la grammaire G_{ex} définie à la question 6 n'est pas LL(1).

Question 20. La grammaire G_L définie en première partie est-elle LL(1) ?

Question 21. Écrire une fonction `est_ll1 : unit -> bool` qui détermine si la grammaire `g` est LL(1).

Les ensembles calculés dans les deuxième et troisième parties permettent de rendre plus formelle la définition d'une table d'analyse syntaxique (notion introduite en première partie). Si $X \in V$ et $a \in \Sigma$, la règle $X \to \alpha$ est dans la case (X, a) de cette table si et seulement si :

$$a \in \textit{Premier}(\alpha) \vee (\textit{Nul}(\alpha) \wedge a \in \textit{Suivant}(X))$$

Question 22. Construire la table d'analyse syntaxique de la grammaire G_{ex}. Que constate-t-on ?

Question 23. Montrer que si G est une grammaire LL(1), alors chaque case de sa table d'analyse syntaxique contient au plus une règle de production.

Question 24. En déduire qu'une grammaire LL(1) n'est pas ambiguë.

Les résultats précédents montrent qu'il est toujours possible de déterminer algorithmiquement la table d'analyse syntaxique d'une grammaire et que dans le cas d'une grammaire LL(1), cette table permet par ailleurs de construire un analyseur syntaxique descendant à l'instar de ce qui a été fait dans le cas particulier de la grammaire G_L en première partie. Dans la fin de ce problème, on s'attache à construire un analyseur syntaxique descendant générique pour la grammaire `g` qu'on suppose à présent **LL**$(\mathbf{1})$.

On représente une table d'analyse syntaxique LL(1) par un dictionnaire dont les clés sont les couples `(variable, token)`. La valeur associée à une clé (X, a) est le mot α tel que $X \to \alpha$ est l'unique règle apparaissant dans la case (X, a) de la table d'analyse syntaxique. Les fonctions usuelles sur les tables de hachage sont rappelées en annexe.

Question 25. On suppose disposer d'une fonction `calculer_tokens` telle que `calculer_tokens ()` renvoie le tableau des tokens utilisés dans `g`, y compris le token `EOF`. Écrire une fonction `table_analyse : unit -> (int * token, mot) Hashtbl.t` qui renvoie la table d'analyse syntaxique de `g`.

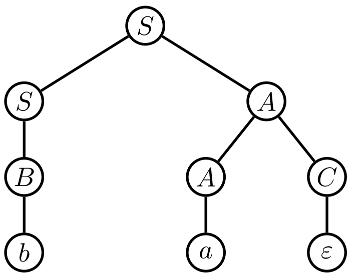
Par la suite on suppose définie de manière globale cette table d'analyse syntaxique via :

```
let tas = analyse_syntaxique ()
```

On représente un arbre de dérivation par le type suivant.

```
type arbre_syntaxe = Epsilon | F of token | N of int * arbre_syntaxe list
```

Par exemple, la représentation d'un arbre syntaxique pour le mot ba dans la grammaire G_{ex}, en reprenant la numérotation des variables utilisée peu avant la question 9, est donnée ci-dessous.




```
N(0, [N(0, [N(2, [F b])]);
      N(1, [N(1, [F a]);
            N(3, [Epsilon])])])
```

Question 26. Écrire une fonction `analyse_syntaxique : mot -> arbre_syntaxe` qui renvoie un arbre de dérivation pour `m` selon la grammaire `g`. On suppose que `m` finit par le token spécial `EOF` et que `EOF` n'apparaît nulle part ailleurs. La fonction lèvera `SyntaxError` si `m` n'est pas engendré par `g`.

Annexe

- `('a, 'b) Hashtbl.t` est le type des dictionnaires dont les clés sont de type `'a` et les valeurs de type `'b`.
- `let dico = Hashtbl. create 42` permet de créer un dictionnaire vide dico de taille `42`. La taille est automatiquement augmentée si le dictionnaire est plein.
- `Hashtbl.mem dico c` teste l'appartenance de la clé `c` dans `dico`.
- `Hashtbl.find dico c` renvoie la valeur `v` associée à la clé `c` dans `dico` si elle existe. Si elle n'existe pas, l'exception `Not_found` est levée.
- `Hashtbl. replace dico c v` permet de remplacer l'éventuelle association de clé `c` déjà présente dans `dico` par l'association `(c, v)`. S'il n'existe pas d'association dont la clé est `c` dans `dico`, alors cette opération ajoute l'association `(c, v)` dans `dico`.