Problème 1

1. On a $E_0 = \{B\}$, $E_1 = \{B, A\}$, $E_2 = \{B, A, S\}$, $E_3 = \{B, A, S\}$ et donc finalement $E = \{B, A, S\}$. Voici les règles obtenues :

$$S \to AB \,|\, A \,|\, B \,|\, aA \,|\, a \,|\, Cb$$
$$A \to BB \,|\, B \,|\, a,\ B \to b,\ C \to c$$

2. On montre par récurrence que E_n est l'ensemble des variables pouvant se dériver en ε en au plus $(n+1)$ dérivations. L'égalité souhaitée se montre alors immédiatement par double inclusion.

3. La suite $(E_n)_{n\in\mathbb{N}}$ est une suite d'ensembles inclus dans V croissante pour l'inclusion par construction. Comme V est fini, elle stationne donc nécessairement à partir d'un entier $n_0 \leq |V|$ convenable.

   À la place de calculer tous les E_n, il suffit donc de les calculer jusqu'à atteindre le stationnement, qui arrive nécessairement au bout d'un nombre fini d'étapes d'après ce qui précède. Il n'y a pas de problème de terminaison dans la suite de l'algorithme 1.

4. On constate que $\varepsilon \in L(G)$ si et seulement S peut se dériver en ε en un certain nombre d'étapes c'est-à-dire si et seulement si $S \in E$ avec E l'ensemble calculé par les trois premières lignes de l'algorithme 1. Or, déterminer cet ensemble peut se faire de manière naïve comme suit :

   - Calculer E_0 se fait en $O(|\mathcal{R}|)$ en lisant toutes les règles.
   - L'ensemble E_n étant connu, le calcul de E_{n+1} peut se faire en parcourant le membre droit de chaque règle et pour chaque lettre de ce membre on vérifie s'il est dans E_n (de taille inférieure à $|V|$). Le tout peut se faire en $O(|\mathcal{R}| \times |V|)$.

   Comme on calculera au plus $|V|$ ensembles E_n d'après la question 3, la complexité de cet algorithme très naïf est en $O(|\mathcal{R}||V|^2) = O(|G|^3)$ ce qui conclut.

5. On a $E(S) = \{A, B, C\} = E(A) = E(C)$ et $E(B) = \emptyset$. Les règles obtenues sont :

$$S \to aA \,|\, a \,|\, b \,|\, cc \,|\, AB$$
$$A \to aA \,|\, a \,|\, b \,|\, cc$$
$$B \to a \,|\, b$$
$$C \to aA \,|\, a \,|\, b \,|\, cc$$

6. Si on considère le graphe Γ dont les sommets sont les variables et dont les arcs sont les couples (A, B) tels que $A \to B$ est une règle de la grammaire G, alors calculer l'ensemble $E(X)$ associé à une variable X revient à déterminer les sommets accessibles à partir de X dans Γ ce qui se fait via un parcours dans Γ à partir du sommet X.

7. D'après la question 6, pour chaque variable X, le calcul de $E(X)$ peut se faire en $O(|V|^2)$ car il y a au plus $|V|^2$ arcs dans le graphe Γ et la complexité d'un parcours de graphe peut être rendue linéaire en sa taille. Le coût de la boucle de la ligne 1 est donc majoré par un $O(|V|^3)$.

   En majorant largement, les instructions des lignes 3 à 6 peuvent être faites en $O(|V| \times |V| \times |\mathcal{R}|)$ car la taille d'un ensemble $E(X)$ est majoré par $|V|$. On obtient donc bien une complexité polynomiale en la taille de G.

8. D'après la ligne 8 de l'algorithme 1, G' ne contient aucune règle de la forme $X \to \varepsilon$ et l'algorithme 2 n'introduit aucune règle de cette forme donc G'' ne contient pas non plus de règle de la forme $X \to \varepsilon$. Cette dernière ne contient pas de règle de la forme $X \to Y$ car elles sont toutes supprimées à la ligne 7 de l'algorithme 2. Donc G'' est propre.

9. Non, voici un contre-exemple : $S \to BC \,|\, A$, $A \to a$, $B \to \varepsilon$. En appliquant l'algorithme 2 on obtient les règles : $S \to a \,|\, BC$, $A \to a$, $B \to \varepsilon$. En appliquant à ces règles l'algorithme 1 on obtient les règles $S \to a \,|\, BC \,|\, C$, $A \to a$. Or cette grammaire n'est pas propre à cause de la règle $S \to C$.

10. La taille de m' est nécessairement strictement plus grande que celle de m. En effet, elle ne peut pas réduire, sinon on aurait utilisé une règle de la forme $X \to \varepsilon$ mais c'est impossible dans une grammaire propre. Elle ne peut pas rester la même sinon on aurait utilisé une règle $X \to u$ avec $u \in \Sigma$ (contraire à l'hypothèse de la question) ou $u \in V$ (impossible dans une grammaire propre).

11. On déduit de la question 10 que si un mot $m \neq \varepsilon$ est engendré par une grammaire propre, il le sera en moins de $k = 2|m| - 1$ dérivations ($|m| - 1$ au plus pour créer les non terminaux nécessaires à avoir une taille suffisante et $|m|$ au plus pour transformer ces non terminaux en terminaux).

12. Étant donnée une grammaire G et un mot m, l'algorithme suivant décide si $m \in L(G)$:

```
1 si m = ε alors
2 |   Utiliser la question 4 pour conclure
3 sinon
4 |   Appliquer les algorithmes 1 et 2 à G pour produire G' propre et engendrant L(G) \ {ε}
5 |   Créer toutes les dérivations dans G' de taille inférieure à 2|m| − 1
6 |   renvoyer oui si et seulement si l'une d'entre elles à généré m
```

    Tester toutes les dérivations en ligne 5 peut demander un temps exponentiel en $|m|$: par exemple, si à chaque dérivation on a deux règles applicables, on a déjà $\Theta(2^{|m|})$ dérivations à construire. Par ailleurs, il y en à moins que $k^{|m|}$ avec k le nombre maximal de règles ayant un même non terminal comme membre gauche. Donc a priori, `MOT` $\in$ EXP.

13. Après suppression des règles du type $X \to \varepsilon$ on a : $S \to ASA \,|\, AS \,|\, SA \,|\, S \,|\, aB \,|\, a$, $A \to B \,|\, a$, $B \to b$.
Après suppression des règles du type $X \to Y$ on a : $S \to ASA \,|\, AS \,|\, SA \,|\, aB \,|\, a$, $A \to b \,|\, a$, $B \to b$.
Après l'étape quadratique on obtient : $S \to AS_1 \,|\, AS \,|\, SA \,|\, aB \,|\, a$, $A \to b \,|\, a$, $B \to b$, $S_1 \to SA$.
Enfin, on obtient en fin d'algorithme 3 les règles suivantes :

$$S \to AS_1 \,|\, AS \,|\, SA \,|\, X_aB \,|\, a,\ A \to b \,|\, a,\ B \to b,\ S_1 \to SA,\ X_a \to a$$

14. L'opération de la ligne 1 produit une grammaire qui engendre les mêmes mots que G à ε près. L'étape quadratique ne change pas le langage car on ne fait que décomposer une règle en une suite de règles qu'il suffit d'imbriquer pour continuer à produire les anciens mots. L'étape variable ne modifie pas non plus le langage : cette dernière permet d'obtenir un terminal donné en deux temps.

15. La ligne 1 supprime les règles $X \to m$ avec $|m| = 0$ ou $m \in V$. Les lignes 2 à 4 suppriment les règles $X \to m$ avec $|m| > 2$ sans défaire le travail effectué par la ligne 1. Les lignes 5 à 9 suppriment les règles $X \to u_1u_2$ avec $u_1u_2 \notin V^\star$ sans défaire le travail des étapes précédentes et ajoute des règles $X_a \to a$ avec $a \in \Sigma$ qui sont autorisées dans une grammaire de Chomsky.

    Au final on n'obtient que des règles $X \to m$ avec $|m| = 1$ ou $|m| = 2$. Parmi celles où $|m| = 1$, il n'y en a aucune de la forme $X \to Y$ donc elle sont de la forme $X \to a$. Parmi celles où $|m| = 2$, il n'y en a aucune faisant intervenir un terminal à droite donc elles sont toutes de la forme $X \to YZ$.

16. Rien à signaler :

```
let est_terminal (s:symbole) :bool = match s with
  |Term _ -> true
  |_ -> false
```

17. Il n'est pas obligatoire de factoriser les cas mais on rappelle la syntaxe pour ce faire :

```
let char_of_symb (s:symbole) :char = match s with
  |Term c |Var c |Dupl (c,_) |X c -> c
```

18. Les règles traitées dans l'étape quadratique sont celles qui ont plus de trois symboles à droite :

```
let est_eligible (r:regle) :bool = (List.length (r.droite)) >= 3
```

19. Il suffit pour chaque règle d'extraire le caractère de son membre gauche pour l'associer à l'entier 1 :

```
let variables_initiales (g:grammaire) :(char,int) Hashtbl.t =
  let numerotation = Hashtbl.create 42 in
  List.iter (fun r -> Hashtbl.replace numerotation (char_of_symb r.gauche) 1) g;
  numerotation
```

20. L'énoncé dit que `v` est présente dans `numerotation` donc on peut utiliser `Hashtbl.find` sans crainte :

```
let incrementer (numerotation:(char,int) Hashtbl.t) (v:char) :unit =
  let numero = Hashtbl.find numerotation v in
  Hashtbl.replace numerotation v (numero+1)
```

21. On procède récursivement :

    - Si la liste de symboles `prod` est trop courte, cela entre en contradiction avec le fait que la règle représentée $X \to m$ par `v` et `prod` est éligible à la transformation par l'étape quadratique.
    - Le deuxième cas correspond au cas où on arrive à la fin de la règle $X \to m$ et traduit l'ajout de la règle $X_{\text{quelque chose}} \to m_{k-1}m_k$ aux règles de `regles` qui sert d'accumulateur.
    - Le dernier cas est le cas récursif générique : la règle à ajouter est celle dont le membre de gauche est la variable actuelle et le membre de droite est constitué du symbole lu et de la prochaine duplication de variable disponible, qu'on construit grâce à la renumérotation.

```
let rec eclater_regle (v:symbole) (prod:symbole list) (regles:regle list)
↪  numerotation :regle list =
  match prod with
  |[] |[_] -> failwith "impossible pour une règle éligible"
  |[u;v] -> let r = {gauche = v ; droite = [u;v]} in
            r::regles
  |u::q -> let c = char_of_symb v in
           let numero = Hashtbl.find numerotation c in
           let nouvelle_var = Dupl(c,numero) in
           let r = {gauche = v ;
                    droite = [u;nouvelle_var]} in
           let _ = incrementer numerotation c in
           eclater_regle nouvelle_var q (r::regles) numerotation
```

    Une remarque : techniquement dans le dernier cas, il n'y a pas de règle `nouvelle_var` $\to$ `q` dans la grammaire mais il est néanmoins légitime d'appeler `eclater_regle` avec ces entrées.

22. Le plus gros du travail a été fait par la question 21. La fonction `traiter_regles` a le fonctionnement suivant : pour chaque règle dans la liste en entrée, elle l'éclate selon le principe de l'étape quadratique si c'est nécessaire et la conserve sinon. Attention, il faut que le dictionnaire de renumérotation des

variables soit le même à chaque appel de `traiter_regles`, il ne faut pas le redéfinir à chaque règle (sinon, plusieurs règles $X \to m$ feraient intervenir les mêmes X_i).

```
let etape_quadratique (g:grammaire) :grammaire =
  let numerotation = variables_initiales g in
  let rec traiter_regles (r_liste:regle list) :regle list =
    match r_liste with
    |[] -> []
    |r::q when (est_eligible r) ->
          let fin_regles = traiter_regles q in
          eclater_regle r.gauche r.droite fin_regles numerotation
    |r::q -> r::(traiter_regles q)
  in traiter_regles g
```

23. La fonction `est_quadratique` permet de vérifier si une règle est de la bonne forme pour être modifiée par l'étape variable (c'est le cas si et seulement si il y a deux lettres dans le membre droit) :

```
let est_quadratique (r:regle) :bool =
  List.length r.droite = 2
```

La fonction auxiliaire `supprimer_terminaux` remplace les occurrences de terminaux par leur variable associée dans une liste de symboles. On applique cette fonction à tous les membres droits des règles quadratiques avec `List.map` :

```
let remplacer_terminaux (g:grammaire) :grammaire =
  let rec supprimer_terminaux (d:symbole list) :symbole list = match d with
    |[] -> []
    |(Term a)::q -> (X a)::(supprimer_terminaux q)
    |s::q -> s::(supprimer_terminaux q)
  in List.map (fun r -> if est_quadratique r
                        then {gauche = r.gauche;
                              droite = supprimer_terminaux r.droite}
                        else r) g
```

24. Pas de difficulté : on récupère les terminaux de la grammaire avec `liste_terminaux` puis on construit les règles de la forme $X_a \to a$ dans `regles_variables` et les règles produites par les lignes 7-8 de l'algorithme 3 dans `regles_autres` à l'aide de la fonction précédente :

```
let etape_variable (g:grammaire) :grammaire =
  let terminaux = liste_terminaux g in
  let regles_variables = List.map
        (fun c -> {gauche = X c; droite = [Term c]}) terminaux in
  let regles_autres = remplacer_terminaux g in
  regles_variables@regles_autres
```

25. On utilise les fonctions des questions 22 et 24 :

```
let chomsky (g:grammaire) :grammaire =
  let g1 = etape_quadratique g in etape_variable g1
```

26. Si $i \in [\![1, n]\!]$, l'ensemble $E_{i,i}$ est par définition l'ensemble des non terminaux qui engendrent la lettre m_i. Comme G est sous forme normale de Chomsky, pour calculer $E_{i,i}$, il suffit de parcourir toutes

les règles de G et pour toutes celles qui sont de la forme $X \to m_i$, ajouter X à l'ensemble $E_{i,i}$.

27. Soit $i, j \in [\![1, n]\!]$ tels que $i < j$. Montrons l'égalité d'ensembles demandée par double inclusion.

    Si il existe $k \in [\![i, j-1]\!]$ et $Y, Z \in V$ tels que $X \to YZ$, $Y \in E_{i,k}$ et $Z \in E_{k+1,j}$, alors par définition $Y \Rightarrow^\star m_i...m_k$ et $Z \Rightarrow^\star m_{k+1}...m_j$. Mézalors $X \Rightarrow^\star m_i...m_k m_{k+1}...m_j$ et donc $X \in E_{i,j}$.

    Réciproquement, si $X \in E_{i,j}$ alors $X \Rightarrow^\star m_i...m_j$. Comme $i < j$, le mot $m_i...m_j$ contient au moins deux lettres. Comme G est sous forme normale de Chomsky, on en déduit que la première dérivation de $m_i...m_j$ à partir de X est nécessairement de la forme $X \Rightarrow YZ$ pour $Y, Z \in V$. Grâce au "lemme fondamental", on en déduit l'existence de $k \in [\![i-1, j]\!]$ tel que $Y \Rightarrow^\star m_i...m_k$ et $Z \Rightarrow m_{k+1}...m_j$. Mais on sait d'autre part que la grammaire G ne permet pas de produire le mot ε : l'entier k est donc compris entre i et $j-1$ ce qui conclut.

28. Pour le mot *abab*, on obtient la matrice suivante :

$$\begin{pmatrix} T & X,Z & X,T & S,X,Z \\ \emptyset & Y,Z & Y,T & X,Z \\ \emptyset & \emptyset & T & X,Z \\ \emptyset & \emptyset & \emptyset & Y,Z \end{pmatrix}$$

    On explicite le calcul du coefficient en position $(1,3)$. Pour le déterminer, il faut trouver tous les non terminaux qui engendrent un élément du produit cartésien des ensembles $E_{1,1}$ et $E_{2,3}$ (encadrés sur fond grisé ci-dessus), c'est-à-dire $\{TY, TT\}$. On en trouve un : X. A ceux-ci s'ajoutent tous les non terminaux qui engendrent un élément du produit cartésien de $E_{1,2}$ et $E_{3,3}$ (encadrés sur fond blanc), à savoir $\{XT, ZT\}$. Là encore, on en trouve un : T.

    Comme l'axiome S fait partie de l'ensemble $E_{1,n}$, on en déduit que *abab* est engendré par G_{ex}.

29. La première boucle pour calcule les ensembles $E_{i,i}$ d'après la question 26. La deuxième boucle pour calcule les ensembles $E_{i,j}$ pour $1 \leq i < j \leq n$ d'après la question 27 à condition que tous les $e_{i',j'}$ nécessaires au calcul de $e_{i,j}$ aient déjà été calculés lorsque le calcul de $e_{i,j}$ commence. Mais c'est bien le cas, car la question 27 indique que les ensembles dont on a besoin pour calculer $E_{i,j}$ sont ceux grisés sur le dessin suivant :

$$\begin{pmatrix} E_{1,1} & \cdots & E_{1,i} & \cdots & E_{1,j-1} & E_{1,j} & \cdots & E_{1,n} \\ \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ E_{i,1} & \cdots & E_{i,i} & \cdots & E_{i,j-1} & E_{i,j} & \cdots & E_{i,n} \\ E_{i+1,1} & \cdots & E_{i+1,i} & \cdots & E_{i+1,j-1} & E_{i+1,j} & \cdots & E_{i+1,n} \\ \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ E_{j,1} & \cdots & E_{j,i} & \cdots & E_{j,j-1} & E_{j,j} & \cdots & E_{j,n} \\ \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ E_{n,1} & \cdots & E_{n,i} & \cdots & E_{n,j-1} & E_{n,j} & \cdots & E_{n,n} \end{pmatrix}$$

    Ainsi, il suffit d'avoir calculé tous les coefficients sur les surdigaonales (la surdiagonale numéro 0 correspondant à la diagonale elle-même) précédant la surdiagonale sur laquelle se trouve $e_{i,j}$ pour disposer de tous les ensembles nécessaires à la construction de cet ensemble : l'ordre de calcul des $e_{i,j}$ est adapté à la relation récurrente qui les lie.

    En fin d'algorithme, le coefficient $e_{1,n}$ est égal à l'ensemble $E_{1,n}$. L'axiome S s'y trouve si et seulement

si $S \to^\star m_1...m_n = m$ c'est-à-dire si et seulement si $m \in L(G)$.

30. On considère que déterminer le type d'une règle — soit de la forme $X \to a$, soit de la forme $X \to YZ$ —, extraire son membre droit dans le premier cas, extraire les deux lettres de son membre droit dans le second et extraire son membre gauche dans les deux cas se fait en temps constant.

    L'initialisation de la matrice E se fait en temps $O(|m|^2)$. Le calcul de la diagonale de la matrice E (première boucle pour) nécessite $O(|m||\mathcal{R}|)$ opérations où $|\mathcal{R}|$ est le nombre de règles dans la grammaire G en entrée. Le calcul des surdiagonales est grossièrement majoré par un $O(|m|^3|\mathcal{R}|)$.

    On en déduit que la complexité de l'algorithme de Cocke-Younger-Kasami est en $O(|m|^3|\mathcal{R}|)$.

    Au passage ceci montre que le problème `MOT` est en fait dans P car mettre une grammaire sous forme normale de Chomsky se fait en temps polynomial en $|G|$ vu la complexité des algorithmes 1,2,3 et à partir d'une grammaire sous forme normale de Chomsky, l'algorithme CYK permet de savoir si un mot est engendré en temps polynomial en $|m|$ et $|G|$.

    Remarque : La majoration grossière du calcul des surdiagonales ne l'est en fait pas tellement. Pour calculer la surdiagonale d, il faut calculer tous les $e_{i,i+d}$ pour $i \in [\![0, n-d]\!]$ (connaître i et d permet de déterminer la colonne du coefficient). Pour ce faire on considère tous les $k \in [\![i, d+i-1]\!]$ et pour chacun, on parcourt les règles de G. On obtient donc une complexité de l'ordre de

$$\sum_{d=1}^{n-1}\sum_{i=0}^{n-d}\sum_{k=i}^{d+i-1}|\mathcal{R}| = \sum_{d=1}^{n-1}\sum_{i=0}^{n-d} d|\mathcal{R}| = \sum_{d=1}^{n-1}(n-d+1)d|\mathcal{R}| = \Theta(n^3|\mathcal{R}|) \text{ où } n = |m|.$$

31. Si `gex` représente G_{ex}, `gex.nb_variables` vaut 5 et `gex.nb_regles` vaut 8. On numérote les non terminaux de G_{ex} comme suit (le numéro de S étant forcément 0) :

non terminal	S	T	X	Y	Z
numéro	0	1	2	3	4

    Les règles $S \to XY$ et $Y \to b$ sont alors représentées respectivement par `r0` et `r6` :

```
regle* r0 = (regle*) malloc(sizeof(regle));
r0->type = 2;
r0->membre_gauche = 0;
r0->lettre = '?';
r0->variable1 = 2;
r0->variable2 = 3;

regle* r6 = (regle*) malloc(sizeof(regle));
r0->type = 1;
r0->membre_gauche = 3;
r0->lettre = 'b';
r0->variable1 = -1;
r0->variable2 = -1;
```

32. Je vous encourage à le faire pendant les vacances et à utiliser cet algorithme pour déterminer si $m_1 = bbaabaabbbab$ et $m_2 = abaabaabbbab$ font partie de $L(G_{ex})$ (définie en question 28). Le code source légèrement commenté permettant d'obtenir ces réponses vous sera envoyé par mail pour que vous puissiez vérifier votre travail.

Problème 2

1. Le mot $1 + 1 \times 1$ admet deux arbres syntaxiques différents, que je vous laisse dessiner.

2. Les règles $S \to S + S \,|\, S \times S$ posent problème : si on écrit une fonction récursive `parser_S`, elle devrait pouvoir être immédiatement appelée sans consommer de token du mot à cause de ces règles. Mais cela implique qu'elle ne termine pas.

3. On propose :

```
let rec parser_S t = match t with
  |(Sym|Lpar|Eof)::_ -> (match parser_L t with
                          |Eof::q -> q
                          |_ -> raise SyntaxError)
  |[] |Rpar::_ -> raise SyntaxError
and parser_L t = match t with
  |(Sym|Lpar)::_ -> parser_L (parser_E t)
  |(Rpar|Eof)::_ -> t
  |[] -> raise SyntaxError
and parser_E t = match t with
  |Sym::q -> q
  |Lpar::q -> (match parser_L q with
               |Rpar::r -> r
               |_ -> raise SyntaxError)
  |_ -> raise SyntaxError
```

   Donnons quelques explications pour `parser_E`, par exemple :

   - Si on doit faire correspondre le non terminal E avec une liste commençant par `sym`, la table d'analyse nous indique que la seule règle qu'il est possible d'appliquer est $E \to$ `sym` ce qui permet de faire directement coïncider E et la tête de la liste : on en renvoie donc la queue.
   - Si on doit faire correspondre le non terminal E avec une liste commençant par `(`, la table d'analyse indique qu'il faut utiliser la règle $E \to$ `(`L`)`. Cette dernière introduit une parenthèse ouvrante qui coïncide avec la parenthèse ouvrante de début de liste : on les supprime toutes les deux et il reste à faire coïncider L`)` avec la queue de la liste. On appelle donc `parser_L` et après avoir dérivé L il faudra nécessairement avoir une liste commençant par `)` pour avoir correspondance.
   - Dans les autres cas, la table indique qu'on ne peut pas dériver de mots commençant par `)` ou `#` de E et on ne peut pas non plus en dériver le mot vide donc on lève donc une erreur.

   Autrement dit, la table indique pour chaque symbole en début de la liste `t` quel est l'appel récursif qu'il est obligatoire de faire pour pouvoir continuer l'analyse.

4. Il suffit de vérifier que tous les tokens de la liste en entrée ont bien été consommés par l'analyse :

```
let engendrer (mot:token list) :bool =
  try parser_S mot = [] with SyntaxError -> false
```

5. Il faudrait d'abord se doter d'un type permettant de représenter un tel arbre syntaxique puis modifier les fonctions de la question 3 de sorte à renvoyer en plus de la liste de tokens restant à analyser, l'arbre syntaxique du préfixe qui vient d'être analysé. La fonction `engendrer` renverrait alors cet arbre quand il existe et lèverait une exception (`SyntaxError` tant qu'à faire) sinon.

6. Les symboles nuls de la grammaire G_{ex} sont :
   - S via la dérivation $S \Rightarrow A \Rightarrow CC \Rightarrow C \Rightarrow \varepsilon$.
   - A via la dérivation $A \Rightarrow CC \Rightarrow C \Rightarrow \varepsilon$.
   - C via la dérivation immédiate $C \Rightarrow \varepsilon$.

   Le symbole B n'est pas nul car $L(B) = \{b\}$, qui ne contient pas ε.

7. La valeur de $\text{NUL}(X)$ ne peut évoluer que de `false` vers `true`. Par conséquent, le nombre de `false` parmi les $|V|$ valeurs de $\text{NUL}(X)$ à calculer, qui est un entier naturel, décroît au fil des itérations. S'il ne change pas d'une étape à la suivante, l'algorithme termine immédiatement en vertu de (3). Sinon, il décroît strictement et ceci ne peut arriver que $|V| < \infty$ fois.

8. On montre par récurrence forte sur $n \in \mathbb{N}^\star$ que, si $X \Rightarrow^n \varepsilon$, alors $\text{NUL}(X) =$ `true` en fin d'algorithme. Si $X \Rightarrow \varepsilon$ alors $X \to \varepsilon$ est une règle et l'étape (2) garantit que $\text{NUL}(X) =$ `true`. Si $X \Rightarrow^{n+1} \varepsilon$ alors $X \Rightarrow X_1 ... X_p$ avec pour tout i, $X_i \Rightarrow^k \varepsilon$ avec $k \leq n$. Par hypothèse de récurrence on aura $\text{NUL}(X_i)$ = `true` et là encore l'étape (2) garantit le calcul correct de $\text{NUL}(X)$.

   Réciproquement, on montre que la propriété : "si $\text{NUL}(X) =$ `true` alors $X \Rightarrow^\star \varepsilon$" est un invariant pour la boucle tant que implicite à l'étape (2) de l'algorithme. Cette propriété est vraie avant d'entrer dans cette boucle pour la première fois puisqu'à cet instant tous les $\text{NUL}(X)$ sont égaux à `false` et la propagation est immédiate. On conclut alors par double inclusion.

9. On peut décomposer comme suit. D'abord, on écrit une fonction `produit_epsilon (m:mot) (nuls:bool array) :bool` qui détermine si on peut dériver ε de `m` en sachant que les symboles nuls connus sont fournis par `nuls`. Pour ce faire, il suffit de vérifier que tous les symboles du mot sont nuls :

```
let produit_epsilon (nuls:bool array) (m:mot) :bool =
  let est_nul (s:symbole) = match s with
    |T _ -> false
    |V x -> nuls.(x)
  in List.for_all est_nul m
```

   Puis, on itère l'étape (2) tant qu'on trouve un nouveau symbole annulable :

```
let calculer_nuls () :bool array =
  let nb_var = Array.length g in
  let nuls = Array.make nb_var false in
  let stabilisation = ref false in
  while not !stabilisation do
    stabilisation := true;
    for x = 0 to nb_var -1 do
      if not nuls.(x) && List.exists (produit_epsilon nuls) g.(x) then
        (nuls.(x) <- true ; stabilisation := false)
    done
  done;
  nuls
```

10. La fonction `produit_epsilon` s'exécute linéairement en la taille du mot en entrée. La boucle `while` de `calculer_nuls` s'exécute au plus $|V|$ fois comme vu en question 7. À chaque itération, pour chaque règle, on fait un appel à `produit_epsilon` sur un mot de taille au plus n_{max}. Par conséquent on obtient une complexité en $O(|V| \times |\mathcal{R}| \times n_{max})$.

11. On obtient :

m	S	A	B	C	SA	AC	CC
$\textsc{Premier}(m)$	$\{a,b,c\}$	$\{a,c\}$	$\{b\}$	$\{c\}$	$\{a,b,c\}$	$\{a,c\}$	$\{c\}$

12. On a $\textsc{Premier}(\varepsilon) = \emptyset$ car il est impossible de dériver un mot ayant une première lettre à partir de ε. On a pour tout token $a \in \Sigma$, $\textsc{Premier}(a) = \{a\}$.

13. On distingue les cas selon que x soit nul ou pas :

$$\textsc{Premier}(x\alpha) = \begin{cases} \textsc{Premier}(x) \cup \textsc{Premier}(\alpha) \text{ si } \textsc{Nul}(x) \\ \textsc{Premier}(x) \text{ sinon} \end{cases}$$

14. On propose l'algorithme suivant :

```
1  pour X ∈ V
2  |  Premier(X) ← ∅
3  stabilisation ← faux
4  tant que stabilisation = faux
5  |  stabilisation ← vrai
6  |  pour toute règle X → m_1...m_n
7  |  |  i ← 1
8  |  |  tant que i ≤ n
9  |  |  |  si Premier(m_i) ⊄ Premier(X) alors
10 |  |  |  |  Premier(X) ← Premier(X) ∪ Premier(m_i)
11 |  |  |  |  stabilisation ← faux
12 |  |  |  si Nul(m_i) alors
13 |  |  |  |  i → i + 1
14 |  |  |  sinon
15 |  |  |  |  i ← n + 1
```

En ligne 9, $\textsc{Premier}(m_i)$ est bien sûr $\{m_i\}$ si m_i est un token. Le principe est de mettre à jour les ensembles $\textsc{Premier}(X)$ jusqu'à obtenir stabilisation (testée en ligne 9 : l'inclusion est fausse si et seulement si l'un des ensembles de premiers a été modifié). L'indice i en ligne 7 permet de progresser dans la production tant que le symbole le plus à gauche de la production est nul. Dès qu'on rencontre un symbole non nul, on arrête le traitement de la règle via la ligne 15.

15. On traduit l'idée de la question 15 en OCaml. On commence par écrire une fonction `traiter_regle` telle que `traiter_regle p v m` modifie la case `v` du tableau `p` destiné à être le tableau des premiers en cours de construction pour y ajouter les éléments issus du traitement de la règle `v` $\to$ `m` selon les directives de la boucle pour à la ligne 6 du pseudo-code précédent.

```
let rec traiter_regle (premiers:TokenSet.t array) (v:int) (m:mot) =
  match m with
  |[] -> ()
  |(T a)::_ -> premiers.(v) <- ajouter a premiers.(v)
  |(V x)::q -> premiers.(v) <- unir premiers.(v) premiers.(x);
               if nuls.(x) then traiter_regle premiers v q
```

On peut ensuite écrire `calculer_premiers` en testant la stabilisation via le calcul des cardinaux des ensembles de premiers : si aucun ne bouge pour aucune règle, c'est que la stabilisation est atteinte.

```
let calculer_premiers () :TokenSet.t array =
  let nb_var = Array.length g in
  let premiers = Array.make nb_var vide in
  let stabilisation = ref false in
  while not !stabilisation do
    stabilisation := true;
    for x = 0 to nb_var -1 do
      let cardinal_avant = cardinal premiers.(x) in
      List.iter (traiter_regle premiers x) g.(x);
      if cardinal_avant <> cardinal premiers.(x) then
        stabilisation := false
    done;
  done;
  premiers
```

16. On traduit cette fois en OCaml la question 14 :

```
let rec premiers_mot (m:mot) :token list = match m with
  |[] -> vide
  |(T a)::q -> ajouter a vide
  |(V x)::q -> if nuls.(x) then unir premiers.(x) (premiers_mot q)
               else premiers.(x)
```

17. Avec la grammaire G_{ex}, on a en fait pour tout $X \in V$, $\textsc{Suivant}(X) = \{\texttt{EOF}, a, c\}$.

18. Montrons que $\textsc{Suivant}(X) = \bigcup_i \textsc{Premier}(\beta_i) \cup \bigcup_{i,\textsc{Nul}(\beta_i)=\text{vrai}} \textsc{Suivant}(X_i)$ par double inclusion.

    Soit $a \in \textsc{Suivant}(X)$. Alors il existe $\alpha, \beta \in (\Sigma \cup V)^\star$ tels que $S \Rightarrow^\star \alpha X a \beta$ par définition. Quitte à réordonner les dérivations immédiates, on a en fait $S \Rightarrow^\star \gamma X_i \eta \Rightarrow \gamma \alpha_i X \beta_i \eta$ pour une certaine variable X_i avec $\gamma\alpha_i \Rightarrow^\star \alpha$ et $\beta_i \eta \Rightarrow^\star a\beta$. Alors :

    - Si dans cette dernière dérivation $\beta_i \Rightarrow^\star \varepsilon$ on a d'une part $\textsc{Nul}(\beta_i) = \text{vrai}$ et d'autre part $\eta \Rightarrow^\star a\beta$ donc $S \Rightarrow^\star \gamma X_i a \beta$ ce qui montre que $a \in \textsc{Suivant}(X_i)$.
    - Sinon, $\beta_i \Rightarrow^\star a\beta_i'$ et donc $a \in \textsc{Premier}(\beta_i)$.

    Réciproquement on montre que tous les éléments de chaque ensemble des deux paquets d'unions sont dans $\textsc{Suivant}(X)$:

    - Si $a \in \textsc{Premier}(\beta_i)$ alors il existe une dérivation $\beta_i \Rightarrow^\star a\beta_i'$. De plus, toutes les variables de G sont accessibles par hypothèse donc en particulier X_i ce qui implique l'existence d'une dérivation $S \Rightarrow^\star \alpha X_i \beta$. En appliquant la règle $X_i \to \alpha_i X \beta_i$ on a donc :

      $$S \Rightarrow^\star \alpha X_i \beta \Rightarrow \alpha\alpha_i X \beta_i \beta \Rightarrow \alpha\alpha_i X a \beta_i' \beta$$

      ce qui montre que a peut suivre X dans une dérivation.

    - Si $a \in \textsc{Suivant}(X_i)$ avec $\textsc{Nul}(X_i) = \text{vrai}$ alors il existe une dérivation $S \Rightarrow^\star \alpha X_i a \beta$ et une dérivation $\beta_i \Rightarrow^\star \varepsilon$ En combinant les deux et en utilisant la règle $X_i \to \alpha_i X \beta_i$ on a donc :

      $$S \Rightarrow^\star \alpha X_i a \beta \Rightarrow \alpha\alpha_i X \beta_i a \beta \Rightarrow^\star \alpha\alpha_i X a \beta$$

      ce qui montre dans ce cas également que $a \in \textsc{Suivant}(X)$.

19. On a montré à la question 11 que $\textsc{Premier}(SA) = \{a, b, c\}$ et $\textsc{Premier}(A) = \{a, c\}$. Comme ces deux ensembles ne sont pas disjoints et que $S \to SA \,|\, A$ est un couple de règles dans G_{ex}, cette grammaire n'est pas LL(1).

20. Il y a quatre couples à examiner, dérivés des règles $L \to \varepsilon \,|\, EL$ et $E \to \texttt{sym} \,|\, (L)$. Or on a :

$\textsc{Premier}(\varepsilon)$	$\textsc{Premier}(EL)$	$\textsc{Premier}(\texttt{sym})$	$\textsc{Premier}((L))$
$\emptyset$	{sym, (}	{sym}	{(}

$\textsc{Nul}(\varepsilon)$	$\textsc{Nul}(EL)$	$\textsc{Nul}(\texttt{sym})$	$\textsc{Nul}((L))$
vrai	faux	faux	faux

$\textsc{Suivant}(E)$	$\textsc{Suivant}(L)$
{#, sym, (}	{#,)}

On constate que les deux propriétés sont vérifiées pour chacun des quatre cas à considérer. Par exemple pour $L \to EL \,|\, \varepsilon$, $\textsc{Premier}(\varepsilon) \cap \textsc{Premier}(EL)$, $\textsc{Nul}(\varepsilon)$ est vrai mais $\textsc{Nul}(EL)$ est bien faux et $\textsc{Premier}(EL) \cap \textsc{Suivant}(L) = \emptyset$. Donc G_L est LL(1).

21. On commence par écrire une fonction `tester_regles : int -> mot -> mot -> bool` qui renvoie `true` si et seulement si les couples de règles $X \to \alpha \,|\, \beta$ et $X \to \beta \,|\, \alpha$ n'empêchent pas la grammaire considérée d'être LL(1) (c'est-à-dire vérifient les deux propriétés souhaitées) :

```
let tester_regles (x:int) (alpha:mot) (beta:mot) :bool =
  let nul_alpha = produit_epsilon nuls alpha in
  let nul_beta = produit_epsilon nuls beta in
  let prems_alpha = premiers_mot alpha in
  let prems_beta = premiers_mot beta in
  let condition_1 = (intersecter prems_alpha prems_beta = vide) in
  let condition_2 = not nul_beta ||
      (not nul_alpha && (intersecter prems_alpha suivants.(x)) = vide) in
  let condition_3 = not nul_alpha ||
      (not nul_beta && (intersecter prems_beta suivants.(x)) = vide) in
  condition_1 && condition_2 && condition_3
```

Dans la suite, la fonction `collision_1 (x:int) (m:mot) :(l:mot list)` renvoie `true` si et seulement la production `x` $\to$ `m` entre en collision (donc viole l'une des deux règles de l'énoncé) avec une des productions `x` $\to$ `m'` pour `m'` dans `l`. On en déduit une fonction `collision_toutes (x:int) (l:mot list)` qui détermine si l'un des couples de règles ayant `x` pour membre gauche viole les propriétés de l'énoncé (la symétrie introduite par `tester_regles` permet d'appliquer `collision_1` à chaque élément `m` de `l` et les suivants de `m` dans `l`). La grammaire est LL(1) si et seulement si il n'y a aucune collision donc aucune case `true` dans `collisions` :

```
let est_ll1 () :bool =
    let rec collision_1 (x:int) (m:mot) (l:mot list) :bool =
      match l with
      |[] -> false
      |t::q -> not (tester_regles x m t) || (collision_1 x m q)
    in
    let rec collision_toutes (x:int) (l:mot list) :bool = match l with
      |[] -> false
      |t::q -> (collision_1 x t q) || (collision_toutes x q)
    in
    let collisions = Array.mapi (fun x l -> collision_toutes x l) g in
    not (Array.mem true collisions)
```

22. On obtient la table suivante à l'aide des questions 11 et 17 :

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

On constate que contrairement à la table pour G_L certaines cases contiennent plusieurs règles et on devine que ce sera un problème pour effectuer une analyse descendante : si on lit a en voulant dériver X, quelle règle de la case (X, a) choisir ?

23. Supposons par l'absurde qu'il existe $X \in V$ et $a \in \Sigma$ tels que la case (X, a) de la table d'analyse syntaxique de G contiennent les deux règles $X \to \alpha \mid \beta$. Alors on est dans l'un des cas suivants :

- $a \in \textsc{Premier}(\alpha)$ et $a \in \textsc{Premier}(\beta)$ mézalors l'intersection de ces ensembles est non vide et G n'est pas LL(1) car la première propriété est brisée.
- $a \in \textsc{Premier}(\alpha)$, $\textsc{Nul}(\beta)$ et $a \in \textsc{Suivant}(X)$. Comme $\textsc{Nul}(\beta)$, on devrait avoir $\textsc{Premier}(\alpha) \cap \textsc{Suivant}(X) = \emptyset$ d'après le caractère LL(1) de G mais a appartient à cet ensemble.
- $\textsc{Nul}(\alpha)$, $a \in \textsc{Suivant}(X)$ et $a \in \textsc{Premier}(\beta)$ se traite de la même manière que le point précédent par symétrie.
- $\textsc{Nul}(\alpha)$, $\textsc{Nul}(\beta)$ et $a \in \textsc{Suivant}(X)$ mézalors $\textsc{Nul}(\alpha)$ et $\textsc{Nul}(\beta)$ sont vrais en même temps et cela contredit la deuxième propriété pour que G soit LL(1) vis-à-vis de la règle $X \to \alpha \mid \beta$.

On en déduit que pour une grammaire LL(1) chaque case de la table d'analyse syntaxique contient au plus une règle et donc le résultat à la question 22 confirme le résultat à la question 19.

24. Supposons par l'absurde que G est ambiguë et soit $m = m_1...m_n$ un mot de $L(G)$ ayant deux arbres syntaxiques donc deux dérivations gauches. Notons X la première variable de gauche à laquelle on applique une règle différente. Alors ces deux dérivations gauches commencent par $S \Rightarrow^\star m_1...m_iX\gamma$ et l'une se poursuit via $X\gamma \Rightarrow^\star \alpha\gamma \Rightarrow^\star m_{i+1}...m_n$ tandis que l'autre se poursuit via $X\gamma \Rightarrow^\star \beta\gamma \Rightarrow^\star m_{i+1}...m_n$ avec $\alpha \neq \beta$. Alors :

- Si $i = n$ alors nécessairement $\textsc{Nul}(\alpha)$ et $\textsc{Nul}(\beta)$ ce qui contredit le caractère LL(1) de G.
- Sinon, on montre que $X \to \alpha$ et $X \to \beta$ sont toutes les deux dans la case (X, m_{i+1}) ce qui contredit le caractère LL(1) de G d'après la question 23. En effet, si $\alpha \Rightarrow m_{i+1}\alpha'$ alors $m_{i+1} \in \textsc{Premiers}(\alpha)$ donc $X \to \alpha$ est dans la case (X, a). Si ce n'est pas le cas alors nécessairement $\textsc{Nul}(\alpha)$ et alors $\gamma \Rightarrow^\star m_{i+1}...m_n$. Mézalors $S \Rightarrow^\star m_1...m_iXm_{i+1}...m_n$ et donc $m_{i+1} \in \textsc{Suivant}(X)$ puis $X \to \alpha$ est dans la case (X, a). Le même raisonnement tient pour β.

On en déduit qu'une grammaire LL(1) n'est pas ambiguë.

25. La fonction `ajouter_regle` ajoute une association $((X, a), \alpha)$ dans la table si le mot α en entrée vérifie les propriétés impliquant son ajout dans la case (X, a). Comme on sait qu'au plus une règle apparaît dans chaque case de la table puisque `g` est LL(1), on ajoutera au plus une association de clé (X, a) et ce pour tout $X \in V$ et tout $a \in \Sigma$. On réutilise la fonction `produit_epsilon` introduite en question 9 pour tester si un mot est nul.

```
let table_analyse () :(int*token, mot) Hashtbl.t =
  let sigma = calculer_tokens () in
  let nb_tokens = Array.length sigma in
```

```
  let nb_var = Array.length g in
  let tas = Hashtbl.create 1 in
  for x = 0 to nb_var -1 do
    for a = 0 to nb_tokens -1 do
      let ajouter_regle (alpha:mot) :unit =
        if appartient sigma.(a) (premiers_mot alpha)
           || (produit_epsilon nuls alpha && appartient sigma.(a) suivants.(x))
        then Hashtbl.replace tas (x,sigma.(a)) alpha
      in List.iter ajouter_regle g.(x)
    done;
  done;
  tas
```

26. Une façon de faire est de s'appuyer sur une fonction `deriver` qui construit l'arbre dans l'ordre préfixe à partir d'un symbole donné selon ce principe :

    - Si $m = av$ et que le symbole est a, on créé une feuille a et on tente de dériver la fin du mot.
    - Si $m = av$ et que le symbole est X, on récupère si elle existe le (il n'y en a bien qu'un) mot α tel que $X \to \alpha$ est dans la case (X, a) de la table d'analyse puis :
      - Si $\alpha = \varepsilon$, on créé une feuille ε.
      - Sinon on créé récursivement les arbres découlant de l'analyse des symboles de α dont on fait les fils d'un noeud étiqueté par X.

    On tente alors de dériver le mot en entrée à partir de l'axiome (représenté par `V 0`) : l'analyse réussit si le mot peut être lu en entier, c'est-à-dire si la liste le représentant est réduite à `[EOF]` à la fin.

```
let analyse_syntaxe (m:mot) :arbre_syntaxe =
  let reste = ref m in
  let rec deriver (s:symbole) :arbre_syntaxe =
    match !reste, s with
    |(T a)::v, T b when a = b -> reste := v; F a
    |(T a)::v, V x when Hashtbl.mem tas (x,a) ->
      let alpha = Hashtbl.find tas (x,a) in
      if alpha = [] then N(x, [Epsilon])
      else N(x, List.map deriver alpha)
    |_ -> raise SyntaxError
  in let arbre = deriver (V 0) in
     match !reste with
     |[EOF] -> arbre
     |_ -> raise SyntaxError
```

Culture générale : L'algorithme de Cocke-Younger-Kasami étudié dans le problème 1 permet de résoudre le problème du mot en temps polynomial pour toute grammaire mais le polynôme impliqué dans la complexité est de gros degré et il n'est donc pas vraiment utilisé. L'algorithme du problème 2 est plus efficace pour résoudre ce problème mais nécessite de bonnes propriétés sur la grammaire considérée : elle doit être LL(1). On peut en fait étendre son principe aux grammaires dites LL(k) pour lesquelles on observe k tokens pour choisir la prochaine règle à utiliser plutôt qu'un seul.