DS n°06 - 4h

Ce sujet est composé de deux parties complètement indépendantes.

Commentaires extraits du rapport de l'épreuve :

La concision dans la rédaction est explicitement demandée dans certaines questions. Il convient de la respecter et d'aller à l'essentiel. Il y a un lien direct entre le manque de concision et les erreurs dans les preuves. Le sujet demande d'écrire de nombreuses preuves. Si la plupart des candidats savent écrire des preuves, un certain nombre d'erreurs ont mené des candidats à prétendre prouver des assertions fausses. La preuve est un moyen de vérifier si un résultat est vrai, ou s'il ne l'est pas. L'objectif n'est pas de pouvoir tout prouver, mais bien de ne pouvoir prouver que ce qui est vrai. La preuve est censée présenter une certaine « résistance ». Le jury encourage les candidats à se demander, lorsqu'ils écrivent une preuve « Aurais-je échoué à écrire ma preuve si le résultat que je souhaite prouver était faux ? ».

Il est assez facile d'illustrer tout cela, par des schémas. De trop nombreux candidats ont préféré tout décrire sous forme de texte là où un schéma aurait rendu le discours plus clair. Le jury encourage fortement les candidats à faire un usage plus fréquent des schémas. Les meilleures copies sont celles qui ont utilisé des schémas y compris dans les questions où ce n'était pas obligatoire.

Un conseil qui peut être donné aux candidats est de bien vérifier la spécification des fonctions qu'ils programment. Leur fonction doit la satisfaire entièrement.

I Problème du voyageur de commerce

On considère des graphes non orientés $G(V, E)$ où V est l'ensemble des sommets et E l'ensemble des arêtes. On notera $n = |V|$ le nombre de sommets.

Un *chemin* est une suite de sommets reliés par des arêtes. On dit qu'un chemin *passe* par un sommet si ce sommet appartient au chemin. Un *circuit* est un chemin qui commence et se termine au même sommet. Un *chemin hamiltonien* est un chemin qui passe une et une seule fois par chaque sommet du graphe. Un *circuit hamiltonien* est un circuit qui passe par chaque sommet une et une seule fois.

Le *problème du voyageur de commerce* consiste, étant donnée une liste de villes toutes reliées entre elles, à trouver le circuit le plus court qui passe une et une seule fois par chacune des villes.

Plus formellement, on considère un graphe complet non orienté, dont les arêtes sont étiquetées avec des nombres entiers strictement positifs, appelés *poids*, et on cherche le circuit passant par chacun des sommets du graphe qui minimise la somme des poids des arêtes. On appellera *poids d'un circuit* la somme des poids des arêtes empruntées par ce circuit. Une solution au problème du voyageur de commerce est un circuit hamiltonien de poids minimal.

Dans cette partie, on représente les graphes en C par des matrices d'adjacence. Le poids d'une arête est représenté par un `int`, une arête absente étant représentée par un 0. On représente un graphe par une structure de données avec deux attributs : son nombre de sommets `V` et un pointeur `adj` vers sa matrice d'adjacence de taille $\texttt{V} \times \texttt{V}$. Les sommets sont associés aux entiers de 0 à `V-1`.

```
struct Graphe {
    int V;
    int* adj;
};
```

Pour accéder au poids de l'arête entre le sommet `i` et le sommet `j` du graphe `G`, on pourra utiliser l'expression `G.adj[i * G.V + j]`.

On pourra par la suite utiliser les deux fonctions suivantes, qui sont supposées travailler en temps constant :

```
struct Graphe alloue_graphe(int V) {
    int* adj = malloc(V * V * sizeof(int));
    struct Graphe g = {.V = V, .adj = adj};
    return g;
}

void libere_graphe(struct Graphe g) {
    free(g.adj);
}
```

La fonction `alloue_graphe` prend comme argument un nombre `V` et renvoie un graphe qui possède `V` sommets et dont la matrice d'adjacence est allouée sur le tas, grâce à la fonction `malloc`. La fonction `libere_graphe` libère la mémoire du tas occupée par la matrice d'adjacence d'un graphe dont on n'a plus besoin.

On définit également une structure `Chemin` qui représente un chemin par un attribut `longueur` et un attribut `l_sommets`, pointeur vers un tableau à `longueur` éléments. Si `C` est un chemin et `k` un entier naturel strictement plus petit que `C.longueur`, alors `C.l_sommets[k]` est le `k`-ième sommet du chemin `C`. De même que pour les graphes, on définit aussi des fonctions permettant d'allouer un chemin et de libérer un chemin dont on n'a plus besoin, et qui sont supposées travailler en temps constant.

```
struct Chemin {
    int longueur;
    int* l_sommets;
};

struct Chemin alloue_chemin(int longueur) {
    int* l_sommets = malloc(longueur * sizeof(int));
    struct Chemin c = {.longueur = longueur, .l_sommets = l_sommets};
    return c;
}

void libere_chemin(struct Chemin c) {
    free(c.l_sommets);
}
```

I.A – Prise en main du problème et appartenance à NP

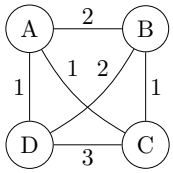

Figure 1 Exemple d'instance du problème du voyageur de commerce

Q 1. Donner une solution du problème du voyageur de commerce sur l'exemple de la figure 1 en précisant le poids du circuit trouvé.

Q 2. Donner le nombre de circuits hamiltoniens sur un graphe complet de n sommets. Donner un exemple de pondération pour que chacun de ces circuits ait un poids minimal pour le problème du voyageur de commerce.

Q 3. Écrire une fonction

```
int poids_chemin(struct Graphe g, struct Chemin c);
```

qui prend en arguments un graphe et un chemin et renvoie le poids de ce chemin. Donner la complexité de cette fonction.

Q 4. Le problème du voyageur de commerce est un problème d'optimisation ; à l'aide d'un seuil, transformer ce problème en un problème de décision. Montrer que ce nouveau problème, que nous appellerons par la suite « problème de décision du voyageur de commerce », appartient à la classe de complexité NP.

I.B – Étude de la complexité

Le *problème du chemin hamiltonien* consiste, étant donné un graphe non orienté et deux sommets a et b, à déterminer s'il existe un chemin hamiltonien commençant en a et finissant en b. Le *problème du chemin hamiltonien orienté* consiste, étant donné un graphe orienté et deux sommets a et b, à déterminer s'il existe un chemin hamiltonien commençant en a et finissant en b. Le *problème du circuit hamiltonien* consiste, étant donné un graphe non orienté, à déterminer s'il existe un circuit hamiltonien dans ce graphe.

I.B.1) NP-complétude

Q 5. Montrer que le problème du chemin hamiltonien se réduit au problème du circuit hamiltonien.

Q 6. Montrer que le problème du circuit hamiltonien se réduit au problème de décision du voyageur de commerce.

Q 7. Montrer que le problème du chemin hamiltonien orienté se réduit au problème du chemin hamiltonien.

Le problème 3-SAT consiste à déterminer la satisfiabilité d'une formule logique sous forme normale conjonctive avec exactement 3 littéraux : pour n clauses C_i et m variables y_k, déterminer s'il existe une valuation des y_k qui permette de rendre vraie la formule $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ avec $C_i = x_{i,1} \vee x_{i,2} \vee x_{i,3}$ sachant que, pour chaque i et chaque p, il existe un k tel que $x_{i,p} = y_k$ ou $x_{i,p} = \neg y_k$. On admet que le problème 3-SAT est NP-complet.

On va maintenant montrer que 3-SAT se réduit au problème du chemin hamiltonien orienté.

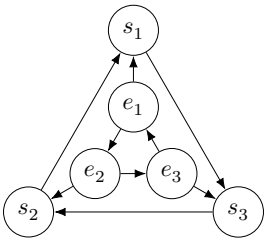

Figure 2 Graphe A

Q 8. On considère le graphe A (figure 2). Montrer qu'il existe un chemin entrant par e_1 et passant par tous les sommets pour ressortir en s_1. Puis qu'il existe un chemin entrant par e_1 et sortant par s_1 et un chemin entrant par e_2 et sortant par s_2, tels que chaque sommet soit visité par un et un seul des deux chemins. Même question, mais avec trois chemins, pour e_1, e_2, e_3 et s_1, s_2, s_3.

On se donne une instance du problème 3-SAT, pour n clauses C_i et m variables y_k : $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ avec $C_i = x_{i,1} \vee x_{i,2} \vee x_{i,3}$ et $x_{i,p} = y_k$ ou $x_{i,p} = \neg y_k$. On veut donc savoir s'il existe une valuation des y_k pour que la formule soit vraie.

On construit alors le graphe orienté G de la manière suivante :

— pour chaque variable y_k on crée un sommet v_k ;

— on ajoute un sommet supplémentaire v_{m+1} ;

— pour chaque clause C_i on ajoute une copie du graphe A, notée A_i ;

— pour chaque variable y_k, on note $C_{k_1}, ..., C_{k_\ell}$ les clauses dans lesquelles y_k apparait en positif. On relie alors v_k à A_{k_1} par un arc allant de v_k vers e_p dans A_{k_1} lorsque y_k est en position p dans C_{k_1} c'est-à-dire lorsque $C_{k_1} = x_{k_1,1} \vee x_{k_1,2} \vee x_{k_1,3}$ et $x_{k_1,p} = y_k$. On relie ensuite la sortie s_p de A_{k_1} à l'entrée de A_{k_2} correspondant à la position de y_k dans C_{k_2} et ainsi de suite jusqu'au dernier dont on relie la sortie à v_{k+1}. On appelle G_k^+ le sous-graphe constitué du sommet v_k, des graphes A_{k_1}, A_{k_2}, ..., A_{k_ℓ} et du sommet v_{k+1}, ainsi que des arcs que l'on vient d'ajouter entre eux en considérant y_k et les clauses dans lesquelles il apparait positivement ;

— on crée de même des arcs pour chaque variable y_k et chaque clause dans laquelle y_k apparait en négatif. On note G_k^-, le sous-graphe correspondant entre v_k et v_{k+1}.

Q 9. Montrer que pour toute valuation de la formule il existe un chemin hamiltonien orienté de v_1 à v_{m+1} dans le graphe G.

Q 10. Montrer, en une dizaine de lignes au maximum, que pour chaque chemin hamiltonien orienté de v_1 à v_{m+1} il existe bien une valuation.

Q 11. En déduire que le problème du circuit hamiltonien et le problème de décision du voyageur de commerce sont NP-complets.

I.B.2) Approximation

Soit $\varepsilon > 0$, on va montrer que, si P $\neq$ NP, il n'existe pas de $1+\varepsilon$ approximation pour le problème du voyageur de commerce. Un algorithme est une $1+\varepsilon$ approximation à un problème d'optimisation d'un poids p lorsque la solution proposée de poids p se compare toujours à la solution optimale p^* par $p < (1+\varepsilon)p^*$.

Soit G un graphe non orienté à n sommets, on considère le graphe complet G', de mêmes sommets que G, avec des poids aux arêtes obtenus en donnant un poids de 1 aux arêtes de G et un poids de $n(1+\varepsilon)+1$ aux arêtes qui ne sont pas dans G.

Q 12. Montrer que si G possède un circuit hamiltonien, alors G' possède un circuit de poids n.

Q 13. Montrer que si G ne possède pas de circuit hamiltonien alors toute solution pour l'instance de voyageur de commerce est de poids au moins $n(2+\varepsilon)$.

Q 14. En déduire que, si P $\neq$ NP, il n'existe pas de $1+\varepsilon$ approximation au problème du voyageur de commerce.

I.C – Algorithme de Christofides

On va proposer une heuristique pour le problème du voyageur de commerce, l'algorithme de Christofides, et on va montrer que, sous certaines conditions sur le graphe en entrée, cette heuristique constitue un algorithme d'approximation. L'algorithme prend en argument un graphe G et procède comme suit :

— calculer un arbre couvrant de poids minimal T de G ;

— en notant I l'ensemble des sommets de degré impair dans T, calculer un couplage parfait M de poids minimum dans le sous-graphe de G induit par les sommets de I, $G_{|I}$;

— construire H le multigraphe ayant pour sommet les sommets de G et comme arêtes les arêtes de M et celles de T ;

— trouver un cycle eulérien dans H ;

— transformer le cycle eulérien en circuit hamiltonien en supprimant les éventuels sommets vus plusieurs fois.

Dans la suite, on étudie plus précisément certaines étapes de cet algorithme, avant de proposer une implémentation de cet algorithme.

I.C.1) Arbre couvrant

Un arbre couvrant est un sous-graphe connexe sans cycle d'un graphe avec les mêmes sommets. On appelle poids de l'arbre la somme des poids des arêtes de cet arbre. On rappelle que l'algorithme de Kruskal est un algorithme glouton qui vise à construire un arbre couvrant de poids minimal en considérant les arêtes par poids croissant et en ajoutant chaque arête si elle ne crée pas de cycle.

Pour cela, on va représenter une arête par une structure de données avec trois attributs : `s1` et `s2` donnent les sommets reliés par l'arête et `p` son poids :

```
struct Arete {
    int s1;
    int s2;
    int p;
};
```

Q 15. Écrire une fonction

```
struct Arete* liste_aretes(struct Graphe g);
```

qui prend en argument un graphe complet `g` et alloue et renvoie un tableau contenant les $\frac{\texttt{g.v}\times(\texttt{g.v}-1)}{2}$ arêtes du graphe.

On dispose d'une fonction

```
void tri_aretes(struct Arete a[], int k);
```

qui prend en arguments un tableau d'arêtes et sa longueur et trie le tableau par ordre croissant de poids. La complexité de cette fonction est en $\mathcal{O}(\texttt{k}\ln(\texttt{k}))$.

Q 16. Écrire une fonction

```
struct Graphe kruskal(struct Graphe g);
```

implémentant l'algorithme de Kruskal qui renvoie un graphe représentant l'arbre couvrant de poids minimal du graphe donné en argument. On mettra des 0 lorsque l'arête est absente et des 1 lorsqu'elle est présente. Donner la complexité de la fonction `kruskal`.

Q 17. Montrer la correction de cet algorithme, c'est-à-dire l'optimalité de la solution proposée pour le problème d'arbre couvrant de poids minimal.

I.C.2) Couplage

On appelle couplage d'un graphe un ensemble d'arêtes qui n'ont pas de sommets en commun. Un couplage est parfait si tous les sommets du graphe appartiennent à une arête du couplage.

Q 18. Écrire une fonction

```
int degre(struct Graphe g, int i);
```

qui prend en arguments un graphe et l'indice d'un sommet et renvoie le degré de ce sommet.

Q 19. Écrire une fonction

```
int* sommets_impairs(struct Graphe g, int* nb_sommets);
```

qui prend en arguments un graphe `g` et un pointeur vers un entier. Cette fonction alloue sur le tas un tableau, le remplit avec les numéros des sommets de degré impair et le renvoie. Par ailleurs, elle renseigne l'entier pointé par `nb_sommets` avec le nombre des sommets de degré impair.

Q 20. Montrer l'existence d'un couplage parfait de poids minimal dans $G_{|I}$.

On dispose des deux fonctions suivantes :

```
struct Graphe graphe_induit(struct Graphe g, int nb_sommets, int* liste_sommets);
struct Graphe couplage(struct Graphe g);
```

La fonction `graphe_induit` renvoie le graphe induit dans un graphe `g` donné par un nombre et un tableau de sommets comme ceux de la question 19.

La fonction `couplage` renvoie un couplage parfait de poids minimal s'il existe, sous la forme d'un graphe représentant ce couplage, avec des 0 lorsque l'arête est absente et 1 lorsque l'arête est présente.

On supposera ces deux fonctions de complexité polynomiale.

I.C.3) Cycle eulérien

On appelle cycle eulérien un circuit qui passe par chaque arête une et une unique fois. On admet qu'un graphe possède un cycle eulérien si et seulement les degrés des sommets de ce graphe sont pairs et que le graphe est connexe. Un multigraphe est un graphe dans lequel il peut exister plusieurs arêtes reliant un même couple de sommets.

Q 21. Montrer que le multigraphe H, défini dans l'introduction de la sous-partie I.C, possède un cycle eulérien.

On dispose des trois fonctions suivantes :

```
struct Multigraphe multigraphe(struct Graphe g1, struct Graphe g2);
struct Chemin eulerien(struct Multigraphe h);
void libere_multigraphe(struct Multigraphe h);
```

La fonction `multigraphe` prend en arguments deux graphes sur les mêmes sommets et renvoie le multigraphe obtenu en considérant les arêtes des deux graphes. La fonction `eulerien` renvoie un circuit eulérien d'un multigraphe sous la forme d'un chemin. La fonction `libere_multigraphe` libère la mémoire du tas utilisée par un multigraphe. Toutes trois sont supposées de complexité polynomiale.

Q 22. Écrire une fonction

```
struct Chemin euler_to_hamilton(struct Chemin c);
```

qui transforme un cycle eulérien `c` du multigraphe H en un circuit hamiltonien du graphe G en supprimant les doublons, et renvoie le chemin représentant la suite des sommets.

I.C.4) Implémentation

Q 23. Sur l'exemple de la figure 1, réaliser les différentes étapes de l'algorithme. On ne demande pas de détailler les étapes pour trouver un couplage et un cycle eulérien.

Q 24. Écrire une fonction

```
struct Chemin christofides(struct Graphe g);
```

qui implémente l'algorithme de Christofides, en prenant soin de libérer la mémoire allouée sur le tas qui n'est plus utilisée.

Q 25. Justifier que la fonction `christophides` renvoie bien un circuit hamiltonien.

Q 26. Montrer que la fonction `christophides` est de complexité polynomiale.

I.C.5) Preuve de l'approximation

On va maintenant montrer que cet algorithme est une 3/2-approximation pour le problème du voyageur de commerce, dans le cas où les poids des arêtes vérifient l'inégalité triangulaire, c'est-à-dire que pour tout sommet u, v, w les poids des arêtes vérifient, en notant $c_{i,j}$ le poids de l'arête entre des sommets i et j : $c_{u,v} \leqslant c_{u,w} + c_{w,v}$.

On note U une solution optimale et $c(U)$ son poids.

Q 27. Montrer que $c(T) \leqslant c(U)$, où $c(T)$ est le coût de l'arbre couvrant minimal.

Q 28. Montrer que $c(M) \leqslant 0.5c(U)$ où $c(M)$ est le coût du couplage parfait minimal.

Q 29. Montrer que la solution construite par l'algorithme est une 3/2-approximation de U.

Q 30. Sans la supposition de l'inégalité triangulaire, cette solution est-elle toujours une 3/2-approximation ? Proposer un contre-exemple ou une justification.

Partie II. Analyse descendante

Ce problème utilise le langage OCaml.

Dans ce problème, on étudie un algorithme d'analyse syntaxique, appelé analyse descendante, qui s'applique à certaines grammaires non contextuelles. Pour une grammaire donnée, on note N l'ensemble de ses non terminaux (notés avec des majuscules) et T l'ensemble de ses terminaux (notés avec des minuscules). Les lettres grecques (α, β, γ, etc.) désignent des mots de $(N \cup T)^\star$. Le mot vide est noté ε. Une règle de production de la grammaire est notée $X \to \gamma$. Une dérivation immédiate est notée $\alpha \Rightarrow \beta$, ce qui signifie qu'il existe une règle de production $X \to \gamma$ avec $\alpha = \alpha_1 X \alpha_2$ et $\beta = \alpha_1 \gamma \alpha_2$. On note $\Rightarrow^\star$ la clôture réflexive transitive de la relation $\Rightarrow$, c'est-à-dire une dérivation en un nombre quelconque d'étapes, y compris zéro.

On prend en exemple une version simplifiée de la grammaire du langage de programmation LISP, avec un ensemble de quatre terminaux $T = \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\}$, un ensemble de trois non terminaux $N = \{S, L, E\}$, dont le symbole initial S, et les cinq règles de production suivantes :

$$\begin{array}{lcl} S & \to & L\ \texttt{\#} \\ L & \to & \varepsilon \\ & \mid & E\ L \\ E & \to & \texttt{sym} \\ & \mid & \texttt{(}\ L\ \texttt{)} \end{array}$$

Appelons G cette grammaire.

Question 11. Pour un entier $n \in \mathbb{N}$ arbitraire, donner un mot de longueur au moins n engendré par cette grammaire et la dérivation à gauche correspondante.

Pour réaliser l'analyse syntaxique de ce petit langage avec OCaml, on se donne un type `token` pour les symboles terminaux `sym`, `(`, `)` et `#` :

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

Ainsi `Sym` représente le terminal `sym`, `Lpar` le terminal `(`, `Rpar` le terminal `)` et `Eof` le terminal `#`.

Notre objectif est d'écrire une fonction `accepts: token list -> bool` qui détermine si un mot, donné comme une liste de terminaux, appartient ou non au langage de cette grammaire. Pour cela, on va commencer par construire trois fonctions, une pour chaque non terminal de la grammaire, avec les types suivants :

```
val parseS: token list -> token list
val parseL: token list -> token list
val parseE: token list -> token list
```

La fonction `parse`X reconnaît un préfixe maximal de la liste passée en argument comme étant un mot dérivé de X et renvoie le reste de la liste. S'il n'y a pas de tel préfixe, alors la fonction lève l'exception `SyntaxError`, que l'on suppose définie.

Pour définir ces trois fonctions, on se donne une table à deux entrées appelée *table LL*. Cette table indique, pour un non terminal que l'on cherche à reconnaître et pour un terminal au début de la liste, la règle de production à utiliser. Voici cette table pour notre grammaire :

	`sym`	`(`	`)`	`#`
S	L`#`	L`#`		L`#`
L	EL	EL	ε	ε
E	`sym`	`(`L`)`		

La fonction `parseS` est donc définie en suivant la première ligne de cette table et voici son code :

```
let rec parseS l = match l with
  | (Sym | Lpar | Eof) :: _ ->
      (match parseL l with Eof :: q -> q | _ -> raise SyntaxError)
  | [] | Rpar :: _ -> raise SyntaxError
```

En particulier, une case vide dans la table est interprétée comme un échec de l'analyse.

Question 12. Donner le code des fonctions `parseL` et `parseE`.

Question 13. Donner le code de la fonction `accepts`.

Construction de la table. On va maintenant chercher à construire une telle table pour une grammaire arbitraire. On introduit pour cela une première notion : On dit qu'un non terminal X est *nul*, et on note $\textsc{Nul}(X)$, si le mot vide peut être dérivé de X, c'est-à-dire $X \Rightarrow^\star \varepsilon$.

Question 14. Indiquer quels sont les symboles nuls de la grammaire G prise en exemple plus haut.

Calcul des symboles nuls. Pour déterminer $\textsc{Nul}(X)$, on propose l'algorithme suivant.

1. Initialement, on fixe $\textsc{Nul}(X)$ à `false` pour tout $X \in N$.
2. Pour chaque non terminal X, on affecte la valeur `true` à $\textsc{Nul}(X)$ s'il existe une production $X \to \varepsilon$ ou une production $X \to X_1 X_2 \dots X_p$ avec X_i des non terminaux et $\textsc{Nul}(X_i)$ pour tout i.
3. Si l'étape 2 a modifié au moins une valeur $\textsc{Nul}(X)$, alors on recommence l'étape 2.

Question 15. Montrer que cet algorithme termine.

Question 16. Montrer que cet algorithme détermine bien la valeur de $\textsc{Nul}(X)$.

Les premiers et les suivants. On introduit deux autres notions sur la grammaire. Pour un non terminal $X \in N$, on définit deux ensembles de terminaux :

— $\textsc{Premiers}(X)$ est l'ensemble des terminaux qui peuvent apparaître au début des mots dérivés depuis X, c'est-à-dire $\textsc{Premiers}(X) = \{t \in T \mid \exists \alpha \in (N \cup T)^\star.\, X \Rightarrow^\star t\alpha\}$;

— SUIVANTS(X) est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation, c'est-à-dire SUIVANTS(X) = $\{t \in T \mid \exists \alpha, \beta \in (N \cup T)^\star.\, S \Rightarrow^\star \alpha X t \beta\}$.

Question 17. Donner les ensembles PREMIERS(X) et SUIVANTS(X) pour la grammaire prise en exemple plus haut (soit six ensembles au total).

Construction de la table. On admet que l'on peut calculer les ensembles PREMIERS(X) et SUIVANTS(X) pour toute grammaire. On étend NUL et PREMIERS sur tout mot de $(N \cup T)^\star$ de la manière suivante :

$$\begin{aligned}
\text{NUL}(\varepsilon) &= \texttt{true} \\
\text{NUL}(X_1 X_2 \dots X_p) &= \texttt{true} \quad \text{si NUL}(X_i) = \texttt{true pour tout } i \\
\\
\text{PREMIERS}(\varepsilon) &= \emptyset \\
\text{PREMIERS}(t\alpha) &= \{t\} \\
\text{PREMIERS}(X\alpha) &= \text{PREMIERS}(X) \quad \text{si NUL}(X) = \texttt{false} \\
\text{PREMIERS}(X\alpha) &= \text{PREMIERS}(X) \cup \text{PREMIERS}(\alpha) \quad \text{si NUL}(X) = \texttt{false}
\end{aligned}$$

où t est un terminal et X un non terminal.

On construit alors la table LL de la manière suivante : pour un non terminal $X \in N$ et un terminal $t \in T$, on indique la règle de production $X \to \gamma$ dans la case (X, t) de la table
— si $t \in$ PREMIERS(γ),
— ou si NUL(γ) et $t \in$ SUIVANTS(X).
On peut alors se servir de cette table pour réaliser une analyse syntaxique dès lors que chaque case ne comporte qu'au plus une règle de production. *On pourra vérifier que l'on obtient bien la table donnée au début de cette partie avec les résultats obtenus aux questions 14 et 17.*

Un autre exemple. On considère une seconde grammaire G', sur les mêmes symboles terminaux, avec des non terminaux $\{S', L', E'\}$ et un symbole initial S' :

$$\begin{aligned}
S' &\to L'\ \texttt{\#} \\
L' &\to \varepsilon \\
&\mid L'\ E' \\
E' &\to \texttt{sym} \\
&\mid \texttt{(}\ L'\ \texttt{)}
\end{aligned}$$

Question 18. Montrer que G' reconnaît le même langage que G.

Question 19. Construire la table LL pour cette seconde grammaire. Permet-elle de coder un algorithme pour l'analyse syntaxique des mots générés par cette grammaire ?