Épreuve d'informatique Info C — Sujet 0

Ce sujet est fourni comme exemple de sujet de l'épreuve Info C du concours X-ENS. Nous avons choisi d'y faire figurer trois exercices distincts afin de couvrir différentes parties du programme d'informatique de CPGE MP2I/MPI. Les sujets de cette épreuve proposés au concours pourront suivre un format similaire ou une organisation différente avec un unique problème.

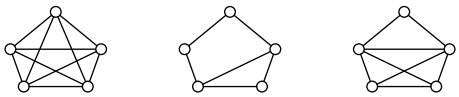
Ce sujet est composé de trois problèmes indépendants. Pour le premier problème, le langage de programmation est C. Pour les deux autres problèmes, le langage de programmation est OCaml.

Partie I. Circuits eulériens

Ce problème utilise le langage C.

Dans ce problème, on s'intéresse uniquement à des graphes *non orientés*, contenant au moins un sommet et *connexes*, c'est-à-dire où il existe toujours un chemin entre deux sommets donnés. On supposera également qu'il n'y a pas de boucle c'est à dire qu'il n'existe pas d'arête ayant pour source et destination un même sommet. Dans ce contexte, un *cycle eulérien* est un chemin qui part d'un sommet et revient à ce dernier et qui emprunte *chaque arête une et une seule fois.*

Question 1. Pour chacun des trois graphes suivants, indiquer s'il possède ou non un cycle eulérien. Dans le cas positif, en exhiber un. On ne demande pas de justification.



Correction : oui / non / non

Question 2. Montrer que si un graphe admet un cycle eulérien alors tous ses sommets sont de degré pair. On rappelle que le *degré* d'un sommet est le nombre de ses voisins.

Correction : Soit un cycle eulérien partant du sommet s. Soit k_u le nombre d'occurence du sommet u dans ce cycle eulérien. Pour $u \neq s$, le degré de u est alors $2k_u$ et donc pair. Pour le sommet s, le degré est $2k_s - 2$ et donc pair également.

Réciproque. Dans la suite de ce problème, on va démontrer que la réciproque est vraie, c'est-à-dire qu'un graphe dont tous les sommets sont de degré pair admet un cycle eulérien. On va le faire en programmant un algorithme qui construit un cycle eulérien. On identifie les sommets avec des entiers.

Pour construire un cycle eulérien, on va représenter toute arête $a-b$ du graphe par deux arcs orientés inverses ($a \to b$ et $b \to a$) en utilisant la structure Edge donnée figure 1. Une valeur de type `Edge` représente un arc du graphe entre les sommets `src` et `dst`. Les deux autres champs `prev` et `next` permettent de chaîner les arcs dans une liste circulaire doublement chaînée, afin de représenter un cycle dans le graphe. Même si le graphe est non orienté, on choisit donc de voir une structure `Edge` comme allant de `src` à `dst`, de telle sorte que si deux valeurs a et b de type `Edge` sont liées, dans cet ordre, on aura

$$\begin{aligned} a\texttt{.next} &= \texttt{\&}b \\ b\texttt{.prev} &= \texttt{\&}a \\ a\texttt{.dst} &= b\texttt{.src} \end{aligned}$$

Une fonction `connect` est fournie pour réaliser une telle connection. Une autre fonction, `join`, est fournie et sera utilisée plus loin.

Le graphe est décrit figure 2. On identifie les sommets avec les entiers $0, 1, \ldots, \texttt{N}-1$. Le graphe est donné dans une variable globale `graph`, sous la forme d'une matrice d'adjacence, dans laquelle on représente toute arête du graphe non orienté par deux arcs orientés inverses. S'il n'y a pas d'arc entre les sommets i et j, alors `graph[`i`][`j`]` et `graph[`j`][`i`]` valent `NULL`. S'il y a un arc entre les sommets i et j, alors `graph[`i`][`j`]` pointe vers une structure e telle que e`.src` $= i$ et e`.dst` $= j$ et, inversement, `graph[`j`][`i`]` pointe vers une structure e telle que e`.src` $= j$ et e`.dst` $= i$. Les champs `next` et `prev` de ces deux structures sont `NULL`. Autrement dit, les arcs ne sont pas connectés entre eux initialement. Au fur et à mesure de la construction du cycle eulérien, ces arcs seront retirés du graphe et connectés entre eux pour former un cycle.

Question 3. On commence par un simple test que le critère est vérifié. Écrire une fonction

```
bool is_eulerian()
```

qui détermine si le graphe possède uniquement des sommets de degré pair. Votre fonction devra laisser le graphe inchangé.

Correction :

```
bool is_eulerian() {
  for (int i = 0; i < N; i++) {
    int d = 0;
    for (int j = 0; j < N; j++)
      if (graph[i][j] != NULL)
        d++;
    if (d % 2 != 0) return false;
  }
  return true;
}
```

```
typedef struct Edge  {
  int src, dst;
  struct Edge *next, *prev;
} edge;

// relie l'arc x à l'arc y, en supposant x->dst == y->src
void connect(edge *x, edge *y) {
  assert(x->dst == y->src);
  x->next = y;
  y->prev = x;
}

// joint les deux cycles c1 et c2, en supposant c1->src == c2->src
void join(edge *c1, edge *c2) {
  edge *p = c1->prev;
  assert(c1->src == c2->src);
  edge *q = c2->prev;
  connect(p, c2);
  connect(q, c1);
}
```

FIGURE 1 – Une structure pour représenter les chemins.

Deux fonctions sont fournies. La fonction `has_edges` détermine si un sommet donné possède au moins un arc. Étant donné un sommet `v` qui possède au moins un arc, la fonction `any_edge_from` renvoie un arc `e` tel que `e.src` = `v`, après l'avoir supprimé du graphe (ainsi que l'arc inverse).

Question 4. Soit un graphe possédant au moins deux sommets dont tous les sommets ont un degré pair. Soit un sommet v_0 dans ce graphe. Montrer que le graphe possède un cycle (pas nécessairement eulérien) partant de v_0. On rappelle que le graphe est connexe.

Correction :

Comme le graphe est connexe et $N \geq 2$ tout sommet possède au moins deux arrêtes.

On va construire un chemin $v_0 - v_1 - \cdots - v_k$ tel que chaque arête dans ce chemin n'est prise qu'une seule fois. Notre construction se termine lorsque $v_k = v_0$.

Comme le degré de v_0 est au moins égal à 2 on peut choisir un sommet voisin v_1, ce qui permet d'initialiser notre construction. Supposons que l'on a construit un chemin $v_0 - v_1 - \cdots - v_k$ comme annoncé et que pour tout $i > 0$ on a $v_i \neq v_0$. Comme le degré de v_k est pair et que v_k est impliqué dans un nombre impair d'arêtes du chemin construit jusqu'alors, il existe une arête $v_k - v_{k+1}$ partant de v_k non encore présente dans le chemin. On obtient alors un chemin $v_0 - v_1 - \cdots - v_k - v_{k+1}$ comme annoncé. Si $v_{k+1} = v_0$ on termine notre construction et on a bien exhibé un cycle.

Notre construction termine toujours car si ce n'était pas le cas, on pourrait construire un chemin de longueur arbitrairement grande qui n'utilise jamais deux fois la même arête ce qui est impossible puisque le graphe est fini (et contient donc un nombre fini d'arêtes).

```
#define N ...
  // la taille du graphe

edge *graph[N][N];
  // et sa matrice d'adjacence

bool has_edges(int v) {
  // indique s'il y a au moins un arc sortant de v
  for (int j = 0; j < N; j++)
    if (graph[v][j] != NULL)
      return true;
  return false;
}

edge *any_edge_from(int v) {
  // renvoie un arc sortant de v
  // l'arc est supprimé du graphe (et l'arc inverse également)
  for (int j = 0; j < N; j++) {
    if (graph[v][j] == NULL) continue;
    edge *e = graph[v][j];
    graph[v][j] = NULL;
    graph[j][v] = NULL;
    return e;
  }
  return NULL;
}
```

FIGURE 2 – Une structure pour représenter le graphe.

Question 5. On suppose que le graphe contient encore des arcs sortant d'un sommet `start` et que tous les sommets sont de degré pair. Écrire une fonction

```
edge *round_trip(int start)
```

qui construit un cycle à partir du sommet `start` en utilisant des arcs encore présents dans le graphe et les retire au fur et à mesure (grâce à la fonction `any_edge_from`). L'arc renvoyé est l'arc de ce cycle dont le champ `src` est `start`. Cette fonction ne pourra pas échouer, en vertu de la question précédente. On prendra soin de bien mettre à jour les champs `next` et `prev` des arcs utilisés, en appelant la fonction `connect` fournie.

Correction :

```
edge *round_trip(int start) {
  edge *path = any_edge_from(start), *e = path;
  for (; e->dst != start; e = e->next) {
    edge *x = any_edge_from(e->dst);
    assert(x != NULL);
    connect(e, x);
  }
  connect(e, path);
  return path;
}
```

Question 6. Écrire une fonction

```
edge *find_vertex_with_edges(edge *c)
```

qui reçoit en argument un cycle et cherche le long de ce cycle un sommet qui possède encore des arcs. S'il existe un tel sommet v, cette fonction renvoie un arc e de ce cycle pour lequel $e.\texttt{src} = v$. Si en revanche il n'existe aucun sommet le long du cycle possédant encore des arcs, cette fonction renvoie `NULL`.

Correction : On se sert avantageusement de `do while` :

```
edge *find_vertex_with_edges(edge *c) {
  edge *e = c;
  do {
    if (has_edges(e->src)) return e;
    e = e->next;
  } while (e != c);
  return NULL;
}
```

Algorithme de Hierholzer. Pour construire un cycle eulérien, on peut procéder ainsi. On commence par construire un cycle arbitraire, avec la fonction `round_trip` et en partant d'un

sommet initial quelconque. Si tous les arcs ont été utilisés, on a terminé. Sinon, on prend un sommet v sur le cycle qui possède encore des arcs. C'est possible, car le graphe est connexe. À partir de v, on construit un nouveau cycle, toujours avec la fonction `round_trip`. Puis on joint les deux cycles pour n'en former qu'un seul. Et ainsi de suite jusqu'à épuisement des arcs. Pour joindre deux cycles, on utilise la fonction `join` fournie dans la figure 1.

Question 7. Écrire une fonction

```
edge *eulerian_cycle(int start)
```

qui reçoit en argument un sommet initial `start` et construit et renvoie un cycle eulérien en suivant l'algorithme de Hierholzer.

Correction :

```
edge *eulerian_cycle(int start) {
  edge *path = round_trip(start);
  while (true) {
    edge *e = find_vertex_with_edges(path);
    if (e == NULL) break;
    edge *cv = round_trip(e->src);
    join(e, cv);
  }
  return path;
}
```

Question 8. Donner la complexité dans le pire des cas de l'algorithme de Hierholzer tel que nous l'avons écrit, en fonction du nombre N de sommets et du nombre E d'arcs du graphe. Est-il possible d'obtenir une meilleure complexité pour la construction d'un cycle eulérien ? (Si oui, on ne demande pas d'écrire le code, mais seulement d'en décrire les idées.)

Correction : Les fonctions `has_edges` et `any_edge_from` ont chacune une complexité en $O(N)$. La fonction `round_trip` a un coût proportionnel au produit de la complexité de la fonction `any_edge_from` et du nombre d'arcs du cycle qu'elle construit, qui sont retirés du graphe. La fonction `find_vertex_with_edges` a, dans le pire des cas, une complexité proportionnelle au produit de la complexité de la fonction `has_edges` et de la taille du cycle dans lequel on fait la recherche. Formellement, soit $C_1, C_2, \ldots, C_K$ les tailles des cycles successivement construits par `round_trip`, avec donc $E = C_1 + C_2 + \cdots + C_K$. Le coût de la i-ème itération (qui consiste à la fois à trouver un sommet avec des arcs restants et à construire un cycle à partir de celui-ci) est donc un

$$O(N \cdot (C_1 + \cdots + C_{i-1}) + C_i)$$

d'où un total

$$O(\sum_i C_i + N \cdot \sum_i (K + 1 - i)C_i) = O(N \cdot E^2)$$

car

$$\sum_i (K+1-i)C_i \leq E \sum_i C_i = E^2$$

La complexité est donc en $O(N \cdot E^2)$. Cette complexité peut être atteinte, par exemple avec une succession de cycles de longueur 3 (mais la question ne demandait pas de le montrer).

Pour obtenir une meilleure complexité, il y a deux axes d'amélioration possible :

— On peut diminuer le coût de `find_vertex_with_edges`. Il suffit pour cela de maintenir dans un ensemble (par exemple une table de hachage) les sommets présents sur le cycle déjà construit et possédant encore des arcs disponibles. Pendant `round_trip`, on prend soin de mettre à jour cet ensemble (ajouter les nouveaux sommets et supprimer les sommets qui n'ont plus d'arcs) mais cela ne change pas la complexité de `round_trip` car les opérations d'ajout/suppression se font en temps constant. On a alors au final un algorithme en $O(E)$, ce qui est optimal.

— On pourrait également (et indépendamment) diminuer le coût des fonctions `has_edges` et `any_edge_from` en utilisant des structures additionnelles en plus de la structure `graphe`.

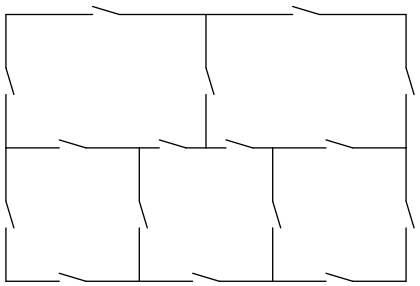
Chemin eulérien. Un *chemin* eulérien dans un graphe non orienté est un chemin qui emprunte chaque arête une et une seule fois, sans nécessairement revenir au point de départ.

Question 9. Montrer qu'un graphe admet un chemin eulérien si et seulement s'il possède 0 ou 2 sommets de degré impair. Pour la réciproque, on s'attachera à décrire un algorithme construisant le chemin (mais on ne demande pas d'en écrire le code).

Correction : Soit un graphe admettant un chemin eulérien du sommet a au sommet b. On ajoute un nouveau sommet c et deux arêtes $a - c - b$. On a alors un cycle eulérien et donc uniquement des sommets de degré pair. En supprimant le sommet c et les deux arêtes $b - c - a$ on a bien au plus deux sommets de degré impair (aucun dans le cas où $a = b$, c'est-à-dire lorsque le chemin était un cycle).

Pour la réciproque, on peut procéder exactement de la même façon. Si le graphe a deux sommets de degré impair, a et b, on ajoute un nouveau sommet c et deux arêtes $b - c - a$. On a alors uniquement des sommets de degré pair et donc un cycle eulérien, que l'on peut construire avec l'algorithme de Hierholzer. En supprimant les arêtes $b - c - a$, on obtient un chemin eulérien de a à b. C'est bien là un algorithme qui construit un chemin eulérien. Et dans le cas où le graphe n'a aucun sommet de degré impair, on a un cycle eulérien par les questions précédentes, donc un chemin eulérien.

Question 10. Est-il possible de tracer un chemin continu qui traverse les cinq pièces suivantes en empruntant chaque porte une et une seule fois ?



Le chemin n'a pas besoin d'être cyclique. Si oui, exhiber un tel chemin ; si non, justifier.

Correction : On construit un graphe ayant un sommet pour chaque pièce, ainsi qu'un sixième sommet pour l'extérieur. On relie ensuite les sommets chaque fois qu'il y a une porte entre les deux. (Il y a plusieurs arête entre deux sommets, ce qui n'est pas un problème en soi. On peut ajouter des sommets intermédiaires à l'extérieur pour que ce ne soit pas le cas.)

On se retrouve alors avec quatre sommets de degrés impairs (trois pièces à 5 portes et l'extérieur à 9 portes). En vertu de la question précédente, il n'existe pas de chemin eulérien pour ce graphe et donc pas de chemin traversant les cinq pièces.

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 :

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

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.

Correction : Il suffit de considérer le mot

$$\underbrace{\texttt{sym sym ... sym}}_{n}\ \texttt{\#}$$

Sa dérivation à gauche est

$$\begin{array}{rcl} S & \Rightarrow & L\ \texttt{\#} \\ & \Rightarrow & E\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym}\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym}\ E\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym sym}\ L\ \texttt{\#} \\ & \vdots & \\ & \Rightarrow & \texttt{sym ... sym}\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym ... sym \#} \end{array}$$

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`.

Correction :

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

```
| Lpar :: q ->
    (match parseL q with Rpar :: r -> r | _ -> raise SyntaxError)
| [] | (Rpar | Eof) :: _ ->
    raise SyntaxError
```

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

Correction : Il faut rattraper l'exception `SyntaxError`, mais il faut également vérifier que tous les terminaux ont bien été consommés.

```
let accepts l =
  try parseS l = [] with SyntaxError -> false
```

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.

Correction : Le symbole L est trivialement nul, car on a la règle de production $L \to \varepsilon$.

Les symbole S et E ne sont pas nuls, car tout mot dérivé de S contient au moins le terminal `#`, et de même tout mot dérivé de E contient soit le terminal `sym`, soit le terminal `(`.

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.

Correction : La valeur de $\textsc{Nul}(X)$ n'évolue que dans le sens `false` vers `true`. Dès lors, le nombre de valeurs `false` ne fait que diminuer. S'il ne change pas, l'algorithme s'arrête. Sinon, il diminue strictement et constitue donc un variant de l'algorithme.

En particulier, le nombre d'étapes est borné par le nombre de non terminaux de la grammaire.

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

Correction : D'une part, on montre par récurrence sur le nombre d'étapes de l'algorithme que, si on obtient $\textsc{Nul}(X) = \texttt{true}$, alors effectivement $X \Rightarrow^\star \varepsilon$. C'est clair initialement, car $\textsc{Nul}(X) = \texttt{false}$ pour tout X. Et si $\textsc{Nul}(X)$ devient $\texttt{true}$, alors soit $X \to \varepsilon$, soit $X \to X_1 X_2 \dots X_p$ avec $X_i \Rightarrow^\star \varepsilon$ par hypothèse de récurrence.

D'autre part, on montre par récurrence sur le nombre d'étapes de la dérivation $X \Rightarrow^\star \varepsilon$ qu'on obtient bien $\textsc{Nul}(X) = \texttt{true}$ par l'algorithme. Si $X \Rightarrow \varepsilon$, c'est qu'il existe une production $X \to \varepsilon$ et on aura bien $\textsc{Nul}(X) = \texttt{true}$ par l'algorithme. Sinon, $X \Rightarrow X_1 X_2 \dots X_p$ avec $X_i \Rightarrow^\star \varepsilon$ et par hypothèse de récurrence, on aura $\textsc{Nul}(X_i) = \texttt{true}$ par l'algorithme, pour tout i, et donc $\textsc{Nul}(X) = \texttt{true}$.

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

Question 17. Donner les ensembles $\textsc{Premiers}(X)$ et $\textsc{Suivants}(X)$ pour la grammaire prise en exemple plus haut (soit six ensembles au total).

Correction :

$$
\begin{aligned}
\textsc{Premiers}(S) &= \{\texttt{sym}, \texttt{(}, \texttt{\#}\} \\
\textsc{Premiers}(L) &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Premiers}(E) &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Suivants}(S) &= \{\} \\
\textsc{Suivants}(L) &= \{\texttt{)}, \texttt{\#}\} \\
\textsc{Suivants}(E) &= \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\}
\end{aligned}
$$

Construction de la table. On admet que l'on peut calculer les ensembles $\textsc{Premiers}(X)$ et $\textsc{Suivants}(X)$ pour toute grammaire. On étend $\textsc{Nul}$ et $\textsc{Premiers}$ sur tout mot de $(N \cup T)^\star$ de

la manière suivante :

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

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 $t \in \textsc{Premiers}(\gamma)$,
— ou si $\textsc{Nul}(\gamma)$ et $t \in \textsc{Suivants}(X)$.
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.

Correction : On comprend que G reconnaît les listes par la gauche et G' par la droite, le reste étant identique. On va montrer la double inclusion des langages.

On commence par un lemme : si $L \Rightarrow^\star w \in T^\star \backslash \{\varepsilon\}$ alors $w = w'u$ avec $L \Rightarrow^\star w'$ et $E \Rightarrow^\star u$. On le prouve par récurrence sur la longueur de la dérivation. La dérivation commence nécessairement par $L \Rightarrow EL$ car $w \neq \varepsilon$. Ensuite, elle se poursuit avec $E \Rightarrow^\star w_1$ et $L \Rightarrow^\star w_2$ et $w = w_1 w_2$. Par hypothèse de récurrence, $w_2 = w_3 u$ avec $L \Rightarrow^\star w_3$ et $E \Rightarrow^\star u$. Il vient donc $L \Rightarrow EL \Rightarrow^\star w_1 w_3$ et on conclut en posant $w = w_1 w_3$.

Montrons maintenant que $X \Rightarrow^\star w$ implique $X' \Rightarrow^\star w$ pour chaque des trois non terminaux. On procède par récurrence sur la longueur de la dérivation $X \Rightarrow^\star w$.
— pour $X = S$: la dérivation est $S \Rightarrow L\texttt{\#}$ avec $L \Rightarrow^\star w'$ et $w = w'\texttt{\#}$. Par HR, $L' \Rightarrow^\star w'$ et donc $S' \Rightarrow L'\texttt{\#} \Rightarrow^\star w'\texttt{\#}$.

— pour $X = L$: Si $L \Rightarrow^\star \varepsilon$ alors $L' \Rightarrow^\star \varepsilon$. Sinon, on peut appliquer le lemme et $w = w'u$ avec $L \Rightarrow^\star w'$ et $E \Rightarrow^\star u$. Par HR, on a $L' \Rightarrow^\star w'$ et $E' \Rightarrow^\star u$, et donc $L' \Rightarrow L'E' \Rightarrow^\star w'u = w$.
— pour $X = E$, on a deux cas. Si $E \Rightarrow^\star \texttt{sym}$ alors $E' \Rightarrow \texttt{sym}$ également. Si $E \Rightarrow \texttt{(}L\texttt{)} \Rightarrow^\star \texttt{(}w'\texttt{)}$ alors par HR $L' \Rightarrow^\star w'$ et donc $E' \Rightarrow \texttt{(}L'\texttt{)} \Rightarrow^\star \texttt{(}w'\texttt{)} = w$.

L'autre inclusion se prouve de façon tout à fait similaire (avec un lemme similaire).

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 ?

Correction : Pour cette grammaire, on trouve

$$\begin{aligned}
\textsc{Premiers}(S') &= \{\texttt{sym}, \texttt{(}, \texttt{\#}\} \\
\textsc{Premiers}(L') &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Premiers}(E') &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Suivants}(S') &= \{\} \\
\textsc{Suivants}(L') &= \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\} \\
\textsc{Suivants}(E') &= \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\}
\end{aligned}$$

et on obtient du coup la table suivante :

	sym	(	)	#
S'	$L'\texttt{\#}$	$L'\texttt{\#}$		$L'\texttt{\#}$
L'	$\varepsilon / L'E'$	$\varepsilon / L'E'$	ε	ε
E'	sym	(L)		

Il y a deux cases dans la table où deux choix apparaissent, ce qui ne permet pas l'analyse.

Partie III. Logique propositionnelle

Ce problème utilise le langage OCaml.

Dans ce problème, on considère des formules de logique propositionnelle sur n variables notées X_i avec $0 \leq i < n$. Une valeur de vérité est un élément de $\mathbb{B} = \{V, F\}$. Une valuation est une fonction v de $\{0, 1, \ldots, n-1\}$ dans $\mathbb{B}$, qui assigne une valeur de vérité à chacune des variables. On note $\mathcal{V}_v(f)$ la valeur de vérité de la formule f pour la valuation v.

On choisit de représenter nos formules en OCaml avec le type suivant.

```
type formula =
  | True
  | False
  | If of int * formula * formula
```

La valeur de vérité d'une telle formule est définie de la manière suivante :

$$\begin{aligned} \mathcal{V}_v(\texttt{True}) &= V \\ \mathcal{V}_v(\texttt{False}) &= F \\ \mathcal{V}_v(\texttt{If}(i, f, g)) &= \text{si } v(i) = V \text{ alors } \mathcal{V}_v(f) \text{ sinon } \mathcal{V}_v(g) \end{aligned}$$

On ajoute par ailleurs la contrainte que toute formule est *ordonnée*, au sens où si elle est de la forme $\texttt{If}(i, f, g)$, alors les formules f et g ne font intervenir que des variables *strictement plus grandes que* i. Ainsi,

$$\texttt{If}(0, \texttt{If}(1, \texttt{False}, \texttt{True}), \texttt{If}(2, \texttt{True}, \texttt{False}))$$

est une formule ordonnée, mais

$$\texttt{If}(0, \texttt{If}(1, \texttt{False}, \texttt{True}), \texttt{If}(0, \texttt{True}, \texttt{False}))$$

n'en est pas une.

Question 20. Écrire une fonction `check: int -> formula -> bool` qui prend en arguments un entier n et une formule f et qui détermine si d'une part f est bien une formule ordonnée et si d'autre part f est limitée aux variables X_i avec $0 \leq i < n$. La complexité doit être linéaire en la taille de la formule. On ne demande pas de justifier la complexité.

Correction : On commence par une fonction qui vérifie le caractère ordonné et que les variables sont dans $[n, m[$:

```
let rec check n m = function
  | True | False -> true
  | If (i, l, r) -> n <= i && i < m && check (i+1) m l && check (i+1) m r
```

La fonction demandée s'en déduit trivialement :

```
let check n f = check 0 n f
```

Question 21. Donner une formule ordonnée pour $n = 3$ variables qui est vraie si et seulement si les trois variables ont la même valeur de vérité.

Correction :

```
If (0,
    If (1, If (2, True, False), False),
    If (1, False, If (2, False, True)))
```

Tautologies. Afin de décider si une formule est une tautologie, on se donne le type OCaml `result` suivant :

```
type assignment = bool array
type result = Tautology | Refutation of assignment
```

Le type `assignment` correspond à une valuation. Une valeur de type `assignment` est un tableau `a` de taille n, où `a.(`i`)` donne la valeur de la variable X_i.

Question 22. Écrire une fonction `decide: int -> formula -> result` qui prend en arguments un entier n et une formule f, supposée ordonnée et sur n variables, et qui renvoie

— `Tautology` si f est une tautologie ;

— `Refutation` v sinon, avec $\mathcal{V}_v(f) = F$.

On s'efforcera de proposer quelque chose de plus efficace que le test systématique de toutes les valuations possibles, en faisant intervenir le caractère ordonné de la formule.

Correction : On exploite ici le fait que la formule est ordonnée : la réfutation trouvée pour `l` ou `r` n'inclut pas de valeur pour la variable i, que l'on peut donc fixer ensuite à loisir.

```
let rec decide n = function
  | False -> Refutation (Array.make n false)
  | True -> Tautology
  | If (i, l, r) ->
      (match decide n l with
       | Tautology -> (match decide n r with
                       | Tautology    -> Tautology
                       | Refutation a -> a.(i) <- false; Refutation a)
       | Refutation a -> a.(i) <- true; Refutation a)
```

Construire des formules arbitraires. Pour montrer que toute formule booléenne classique admet une formule ordonnée équivalente, il suffit de se donner des fonctions sur les formules ordonnées qui correspondent aux connecteurs logiques usuels, telles que la négation, la conjonction, la disjonction, etc.

Question 23. Écrire une fonction `mk_not: formula -> formula` qui reçoit en argument une formule ordonnée f et qui renvoie une formule ordonnée correspondant à la négation $\neg f$, c'est-à-dire une formule ordonnée g telle que, pour toute valuation v, on a $\mathcal{V}_v(g) = \neg\mathcal{V}_v(f)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille de f.

Correction : Pas de difficulté. On fait un simple parcours de la formule :

```
let rec mk_not = function
  | True -> False
  | False -> True
  | If (i, l, r) -> If (i, mk_not l, mk_not r)
```

Il est clair que la formule renvoyée est bien ordonnée et que la complexité est linéaire en la taille de f.

Question 24. Écrire une fonction `mk_or: formula -> formula -> formula` qui reçoit en arguments deux formules ordonnées f et g et qui renvoie une formule ordonnée correspondant à la disjonction $f \vee g$, c'est-à-dire une formule ordonnée h telle que, pour toute valuation v, on a $\mathcal{V}_v(h) = \mathcal{V}_v(f) \vee \mathcal{V}_v(g)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille des formules f et g.

Correction : On fait les simplifications lorsque l'une ou l'autre des formules est `False` ou `True`. Sinon, on compare les deux indices, pour décider en premier lieu sur la variable de plus petit indice, garantissant ainsi le caractère ordonné.

```
let rec mk_or a b = match a, b with
  | True, _ | _, True -> True
  | False, c | c, False -> c
  | If (ia, la, ra), If (ib, lb, rb) ->
      if ia = ib then If (ia, mk_or la lb, mk_or ra rb)
      else if ia < ib then If (ia, mk_or la b, mk_or ra b)
      else If (ib, mk_or a lb, mk_or a rb)
```

La complexité est proportionnelle au produit des tailles de f et de g.

Poisson d'avril. Le poisson d'avril est une espèce extrêmement rare qui n'a jamais été permis d'observer dans la nature. Néanmoins, les ichtyologues ont permis de déterminer les faits suivants :

- (F_1) tout poisson d'avril qui ne nage pas en mer chaude a des rayures rouges ;
- (F_2) tout poisson d'avril a des nageoires bleues ou n'a pas de rayures rouges ;
- (F_3) les poissons d'avril qui vivent dans le corail ne mangent pas de crevettes ;
- (F_4) un poisson d'avril mange des crevettes si et seulement s'il nage en mer chaude ;
- (F_5) tout poisson d'avril qui a des nageoires bleues nage en mer chaude et vit dans le corail ;
- (F_6) tout poisson d'avril qui nage en mer chaude a des nageoires bleues.

On souhaite mettre en application le code OCaml développé plus haut pour démontrer qu'il n'existe pas de poisson d'avril.

Question 25. Expliquer comment construire une formule ordonnée de type `formula` qui est une tautologie si et seulement si les poissons d'avril n'existent pas.

Correction : L'idée est de construire la formule

$$F_1 \Rightarrow (F_2 \Rightarrow (F_3 \Rightarrow (F_4 \Rightarrow (F_5 \Rightarrow (F_6 \Rightarrow \texttt{False})))))$$

Il y a six variables propositionnelles, correspondant aux six critères. On commence par les introduire :

```
let var i = If (i, True, False)
let mc = var 0 (* nage en mer chaude *)
let rr = var 1 (* a des rayures rouges *)
let nb = var 2 (* a des nageoires bleues *)
let vc = var 3 (* vit dans le corail *)
let  c = var 4 (* mange des crevettes *)
```

On se donne ensuite deux fonctions pour construire une implication et une conjonction :

```
let mk_imp a b = mk_or (mk_not a) b
let mk_and a b = mk_not (mk_or (mk_not a) (mk_not b))
```

On peut alors enfin construire la formule :

```
let f1 = mk_imp (mk_not mc) rr
and f2 = mk_or nb (mk_not rr)
and f3 = mk_imp vc (mk_not c)
and f4 = mk_and (mk_imp c mc) (mk_imp mc c)
and f5 = mk_imp nb (mk_and mc vc)
and f6 = mk_imp mc nb
let pas_de_poisson =
  mk_imp f1 (mk_imp f2 (mk_imp f3 (mk_imp f4 (mk_imp f5 (mk_imp f6 False)))))
```

et vérifier sa tautologie avec

```
let () = assert (decide 6 pas_de_poisson = Tautology)
```

Remarque. On aurait pu ici travailler avec d'autres formules équivalentes comme $(F_1 \wedge F2 \wedge F3 \wedge F4 \wedge F5) \Rightarrow \texttt{False}$ ou encore $\neg F_1 \vee \neg F2 \vee \neg F3 \vee \neg F4 \vee \neg F_5$

---

* *
*