Informatique - MPI

Exercice 1

On considère la grammaire algébrique G sur l'alphabet $\Sigma = \{a, b\}$, d'axiome S, de règles $S \to SaS \mid b$.

Question 1. Montrer que cette grammaire est ambiguë.

Question 2. Déterminer (sans preuve pour cette question) le langage L engendré par G. Quelle est la plus petite classe de langages à laquelle L appartient ?

Question 3. Prouver que $L = \mathcal{L}(G)$.

Question 4. Décrire une grammaire qui engendre L de manière non ambiguë en justifiant cette non ambiguité.

Question 5. Montrer que tout langage dans la même classe de langages que L peut être engendré par une grammaire algébrique non ambiguë.

Exercice 2

Soit Σ un alphabet et w un mot de Σ^*. On note $E_1(w)$ le langage des mots de Σ^* qui ne contiennent qu'une seule occurrence du facteur w.

Question 1. Montrer que le langage L_1 des mots de Σ^* qui contiennent un facteur du type wuw est rationnel. Construire un automate reconnaissant L_1 si $\Sigma = \{a, b\}$ et $w = aba$.

Question 2. Montrer que le langage L_2 des mots de Σ^* qui possèdent un facteur u contenant au moins deux occurrences de w et tel que $|u| < 2|w|$ est rationnel. Construire un automate reconnaissant L_2 si $\Sigma = \{a, b\}$ et $w = aba$.

Question 3. Montrer que le langage L_3 des mots de Σ^* qui possèdent deux occurrences de w est rationnel. Construire un automate reconnaissant L_3 si $\Sigma = \{a, b\}$ et $w = aba$.

Question 4. En déduire que $E_1(w)$ est rationnel.

Exercice 3

Soit G un graphe orienté. On dit que s est une *racine* de G si pour tout sommet x de G, il existe un chemin de s à x. On note $d(x)$ la distance de s à x dans G, longueur minimum (en nombre d'arcs) d'un chemin de s à x dans G.
Soit $L = (s_1, \dots, s_n)$ un parcours en largeur de G à partir de $s = s_1$. On pose $\lambda(s_1) = 0$ et, pour tout entier i de $\{2, \dots, n\}$, $\lambda(s_i) = 1 + \lambda(\text{par}(s_i))$ où $\text{par}(s_i)$ désigne le sommet *parent* de s_i dans l'arborescence associée au parcours L. On veut montrer que :

$$\forall i \in [\![1, \dots, n]\!] \quad d(s_i) = \lambda(s_i) \quad \text{et} \quad d(s_1) \leqslant d(s_2) \leqslant \cdots \leqslant d(s_n)$$

On note $\mathcal{P}_i$ la propriété suivante :

$$d(s_1) \leqslant d(s_2) \leqslant \cdots \leqslant d(s_i) \qquad \forall j \in [\![1, i]\!] \quad d(s_j) = \lambda(s_j)$$

Question 1. Montrer que $\mathcal{P}_1$ est vraie.

Question 2. On suppose $\mathcal{P}_{i-1}$ vraie pour une valeur de $i \geqslant 2$ et on note $s_h = \text{par}(s_i)$. Montrer que $d(s_i) \leqslant \lambda(s_h) + 1$.

Question 3. Soit γ un plus court chemin de s_1 à s_i. On note $s_p = \text{pred}(s_i)$ le sommet *prédécesseur* de s_i dans γ.
□ **3.1.** Montrer que $h \leqslant p$.
□ **3.2.** En déduire que $d(s_h) \leqslant d(s_p) < d(s_i)$ puis que $d(s_i) = \lambda(s_i)$.

Question 4. On suppose que $i > h + 1$. Soit s_j un sommet tel que $j \in \{h + 1, \dots, i - 1\}$. On note $s_k = \text{par}(s_j)$.
□ **4.1.** Montrer que $k \leqslant h$.
□ **4.2.** En déduire que $d(s_j) \leqslant d(s_i)$.

Question 5. Soit $k \leqslant h$. Montrer que $d(s_k) \leqslant d(s_i)$.

Question 6. Déduire des questions précédentes que $\mathcal{P}_i$ est vraie.

Exercice 4

Les règles de la déduction naturelle sont rappelées en fin d'exercice.

Question 1. Pour chacun des séquents suivants, exhiber une preuve ou justifier qu'il n'en existe pas.
□ **1.1.** $\neg p \vdash \neg(p \wedge q)$
□ **1.2.** $\neg p \vdash \neg(p \vee q)$
□ **1.3.** $\neg p, p \to q \vdash \neg q$

Question 2. On s'intéresse à présent au typage du code OCaml suivant.

```
1 let f = fun x -> x 0 + 1
2 let g = fun x -> 1 + x
3 let h = fun x -> f g + g x
```

Les règles de typage de la syntaxe OCaml dont on dispose sont celles données ci-dessous où on utilise le même formalisme que pour la présentation des règles dans un système de déduction logique.

$$\frac{}{\Gamma, t \vdash t}\ ax \qquad \frac{}{\Gamma \vdash 0 : \texttt{int}}\ 0_i \qquad \frac{}{\Gamma \vdash 1 : \texttt{int}}\ 1_i \qquad \frac{\Gamma \vdash \texttt{x} : \texttt{int} \quad \Gamma \vdash \texttt{y} : \texttt{int}}{\Gamma \vdash \texttt{x + y} : \texttt{int}}\ +_i$$

$$\frac{\Gamma, \texttt{x} : \sigma \vdash \texttt{e} : \tau}{\Gamma \vdash \texttt{fun x} \to \texttt{e} : (\sigma\tau)}\ \to_i \qquad \frac{\Gamma \vdash \texttt{f} : \sigma \to \tau \quad \Gamma \vdash \texttt{x} : \sigma}{\Gamma \vdash \texttt{f x} : \tau}\ \to_e$$

Par exemple, la règle $\to_e$ signifie en français que si on peut montrer que `f` est une fonction de type σ `->` τ et que sous les mêmes hypothèses on peut montrer que `x` est de type σ alors l'expression `f x` est de type τ.
□ **2.1.** Montrer que le séquent suivant est déductible : $\vdash$ `g : (int -> int)`.
□ **2.2.** Montrer que le séquent suivant est déductible : $\vdash$ `f : ((int -> int) -> int)`.
□ **2.3.** Quel est le type de la fonction `h` ? Le montrer en prouvant la déductibilité d'un séquent dont les hypothèses sont $\Gamma = \{$`f : ((int -> int) -> int)`, `g : (int -> int)`$\}$.

Annexe : règles de la déduction naturelle

- axiome

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ \text{ax}$$

- introduction et élimination de $\wedge$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ \wedge_i$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ \wedge_e^g$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ \wedge_e^d$$

- introduction et élimination de $\vee$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ \vee_i^g$$

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ \vee_i^d$$

$$\frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \sigma \quad \Gamma, \psi \vdash \sigma}{\Gamma \vdash \sigma}\ \vee_e$$

- introduction et élimination de $\to$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ \to_i$$

$$\frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ \to_e$$

- introduction et élimination de $\neg$

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ \neg_i$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \neg\varphi}{\Gamma \vdash \bot}\ \neg_e$$

- élimination de $\bot$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \bot_e$$

Exercice 5

Un code à compléter est joint en fin d'exercice. Lors de l'oral, il est fourni sous la forme d'un fichier à compléter sur machine. Il contient le type décrit par l'énoncé, certaines fonctions et un jeu de test.

Un *arbre binaire de recherche aléatoire* est une structure de données correspondant à un arbre binaire de recherche dans lequel chaque noeud contient la taille du sous-arbre enraciné en ce noeud. Dans tout l'exercice, on interdit aux arbres de contenir des doublons. On représente un arbre binaire de recherche aléatoire en OCaml à l'aide du type suivant :

```
1 type 'a abralea = Vide | N of int * 'a abralea * 'a * 'a abralea
```

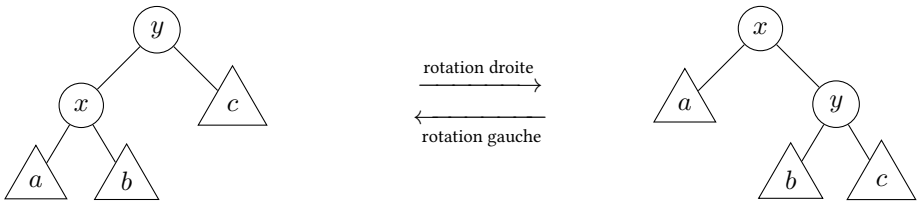
Ainsi, si A est un arbre binaire de recherche aléatoire représenté par `N(n,g,r,d)`, alors `n` $= |A|$, `r` est l'étiquette de sa racine et `g` et `d` sont ses sous-arbres gauche et droit respectivement.

Question 1. Le code compagnon fournit un arbre binaire de recherche aléatoire `a` étiqueté par des lettres (comparées selon l'ordre alphabétique). Le dessiner.

Question 2. Écrire une fonction `taille : 'a abralea -> int` renvoyant la taille d'un arbre binaire de recherche aléatoire. Cette fonction devra avoir une complexité temporelle constante.

Question 3. Écrire une fonction `hauteur : 'a abralea -> int` déterminant la hauteur de son entrée.

Question 4. On rappelle que les opérations de rotation permettent de modifier la racine d'un arbre binaire de recherche tout en préservant une structure d'arbre binaire de recherche. Les fonctions `rotation_gauche` et `rotation_droite` fournies dans le code compagnon permettent de réaliser ces opérations :



Compléter la fonction `insere_racine : 'a -> 'a abralea -> 'a abralea`. Cette dernière doit insérer x dans un arbre binaire de recherche aléatoire aux feuilles puis faire remonter x à la racine à l'aide de rotations. Si x est déjà présent, on lèvera une exception. On s'inspirera du cas déjà traité.

Question 5. On admet que, si A est un arbre binaire de recherche aléatoire et B l'arbre obtenu par un appel à `insere_racine` avec un élément quelconque, alors $h(B) \leq h(A) + 1$. Peut-on avoir $h(B) < h(A)$?

Question 6. L'opération d'insertion d'un élément x dans un arbre binaire de recherche aléatoire A de taille n consiste à tirer aléatoirement un entier k entre 0 et n inclus puis :

- Si $k = 0$, on insère x à la racine avec la fonction précédente.
- Sinon on insère x récursivement dans l'enfant gauche ou droit de A selon l'ordre entre x et la racine.

Là encore, on lève une exception dans le cas où on tente d'insérer un élément déjà présent. On rappelle l'existence de la fonction `Random.int`. Écrire une fonction `insere : 'a -> 'a abralea -> 'a abralea` réalisant cette opération.

Question 7. Pour supprimer un élément x dans un arbre binaire de recherche aléatoire, on commence par le chercher, puis on remplace l'arbre enraciné en x par la fusion de ses deux enfants. La racine de cette fusion est l'une des deux racines des enfants de x avec une probabilité proportionnelle à la taille de l'enfant correspondant. Les éléments restants doivent être récursivement répartis à gauche ou à droite de sorte à respecter la structure d'ABR.

□ **7.1.** Écrire une fonction `fusion : 'a abralea -> 'a abralea -> 'a abralea` respectant ce principe. On suppose sans le vérifier que toutes les étiquettes du premier argument sont strictement inférieures à celles du second.

□ **7.2.** Déterminer la complexité des fonctions `insere` et `supprime` (fournie dans le code compagnon) en fonction de la hauteur de l'arbre en entrée.

□ **7.3.** En utilisant la dernière expression fournie dans le code compagnon, indiquer quelle semble être l'ordre de grandeur de la hauteur d'un arbre binaire de recherche aléatoire obtenu par insertions successives en fonction de sa taille. Pourquoi est-ce intéressant ?

Code à compléter

```
type 'a abralea = Vide | N of int * 'a abralea * 'a * 'a abralea

(* Question 1 *)
let a = N(6, N(2, Vide, 'a', N(1, Vide, 'g', Vide)), 'h',
             N(3, N(1, Vide, 'm', Vide), 'r',
                  N(1, Vide, 'z', Vide)))

(* Question 2 *)
let taille (a:'a abralea) :int =
  failwith "À compléter"

(* Question 3 *)
let hauteur (a:'a abralea) :int =
  failwith "A compléter"

(* Fonctions de rotation, à utiliser en boite noire *)
let rotation_droite = function
  | N(n, N(_, a, x, b), y, c) -> N(n, a, x, N(taille b + taille c + 1, b, y, c))
  | _                         -> failwith "Rotation impossible"

let rotation_gauche = function
  | N(n, a, x, N(_, b, y, c)) -> N(n, N(taille a + taille b + 1, a, x, b), y, c)
  | _                         -> failwith "Rotation impossible"

(* Question 4 *)
let rec insere_racine (x: 'a) (a:'a abralea) :'a abralea =
  match a with
  | N(n,g,y,d) when x < y -> let t = insere_racine x g
                             in rotation_droite (N(n+1,t,y,d))
  | _                     -> failwith "À compléter"

(* Question 6 *)
let insere (x:'a) (a:'a abralea) :'a abralea =
  failwith "À compléter"

(* Question 7 *)
let fusion (a:'a abralea) (b:'a abralea) :'a abralea =
  (* On suppose que toutes les étiquettes de a sont strictement inférieures à
   celles de b. *)
  failwith "À compléter"

(* Fonction fournie *)
let rec supprime (x:'a) (a:'a abralea) :'a abralea =
  match a with
  |Vide -> failwith "Élément non trouvé"
  |N(n, g, y, d) -> if x = y then fusion g d
                    else if x < y then N(n - 1, supprime x g, y, d)
                    else N(n - 1, g, y, supprime x d)

(* Question 9 *)
let _ =
  Random.self_init ();
  let b = ref Vide in
  for i = 0 to 999 do
    b := insere i !b
  done;
  Printf.printf "hauteur : %d, taille : %d\n" (hauteur !b) (taille !b)
```

Exercice 6

Un code à compléter est joint en fin d'exercice. Lors de l'oral, il est fourni sous la forme d'un fichier à compléter sur machine.

L'objectif de cet exercice est de programmer une fonction générant la liste des chemins simples sans issue d'un graphe. On rappelle les définitions d'un graphe, d'un chemin et on en donne une représentation en OCaml.

Un *graphe orienté* est un couple (V, E) où V est un ensemble fini (ensemble des sommets), E est un sous-ensemble de $V \times V$ où tout élément $(v_1, v_2) \in E$ vérifie $v_1 \neq v_2$ (ensemble des arcs). Étant donné un graphe $G = (V, E)$ un *chemin non vide* de G est une suite finie $s_0, \cdots, s_n$ de sommets de V avec $n \geqslant 0$ et vérifiant $\forall i \in \{0, \cdots, n-1\}, (s_i, s_{i+1}) \in E$. On dit que ce chemin est *simple* si $s_0, \cdots, s_n$ sont distincts deux à deux. On dit qu'il est *sans issue* si pour tout s_{n+1} sommet tel que $(s_n, s_{n+1}) \in E$, s_{n+1} appartient à $\{s_0, \cdots, s_n\}$.

Dans la suite, les graphes considérés sont définis sur un ensemble de sommets de la forme $\{0, 1, \cdots, n-1\}$. Pour représenter un graphe en OCaml, on adopte le type suivant :

```
1 type graphe = int list array
```

qui correspond à un encodage par un tableau de listes d'adjacence. Par exemple, le graphe

$$G_1 = (\{0,1,2,3\}, \{(0,1),(0,3),(2,0),(2,1),(2,3),(3,1)\})$$

est représenté par le tableau `[|[1;3];[];[0;1;3];[1]|]`. L'ordre dans lequel sont écrits les éléments dans les listes importe peu. Par contre, l'emplacement des listes dans le tableau est important. Par exemple, `[|[];[0];[0;3;1];[1]|]` représente le graphe :

$$G_2 = (\{0,1,2,3\}, \{(1,0),(2,0),(2,1),(2,3),(3,1)\})$$

On rappelle différentes fonctions susceptibles d'être utilisées.

- `List.filter : ('a -> bool) -> 'a list -> 'a list` où l'expression `List.filter f l` est la liste obtenue en gardant uniquement les éléments `x` de `l` vérifiant `f`.
- `List.iter :('a -> unit) -> 'a list -> unit` où `List.iter f l` correspond à `(f a0);(f a1); ...;(f an)` dans le cas où on a `l = a0::a1::...::an::[]`.
- `List.rev : 'a list -> 'a list` est une fonction qui renvoie le retourné d'une liste. Par exemple, `List.rev [3;1;2;2;4]` est égal à `[4;2;2;1;3]`.
- `Array.length : 'a array -> int` est une fonction qui renvoie la longueur d'un tableau.

Les questions de programmation sont à traiter dans le fichier joint. L'utilisation d'autres fonctions de la bibliothèque que celles mentionnées sont à reprogrammer.

Question 1. Écrire une fonction `est_sommet : graphe -> int -> bool` où `est_sommet g a` est égal à `true` si `a` est un sommet du graphe `g` et `false` sinon.

Question 2. Écrire une fonction `appartient : 'a list -> 'a -> bool` où `appartient l x` est égal à `true` si `x` est un élément de `l` et `false` sinon.

Question 3. Écrire une fonction `est_chemin : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin de `g` et `false` sinon. On suppose que la liste vide représente le chemin vide, qui est bien un chemin et que les éléments de `l` sont bien des sommets du graphe `g`.

Question 4. Compléter la fonction `est_chemin_simple_sans_issue : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin simple sans issue de `g` et `false` sinon. On supposera que les éléments de `l` sont des sommets du graphe `g` et que le chemin vide n'est pas simple sans issue.

Question 5. On cherche à écrire une fonction qui construit la liste des chemins simples sans issue d'un graphe. Pour cela, on procède à l'aide de parcours en profondeur et d'un algorithme de retour sur trace. Compléter le code de la fonction `genere_chemins_simples_sans_issue` présent dans le fichier chemins_simples.ml et qui permet de générer la liste des chemins simples sans issue d'un graphe.

Question 6. Écrire des expressions donnant les listes des chemins simples pour les deux graphes G_1 et G_2.

Question 7. Expliciter la complexité des fonctions `appartient` et `est_chemin_simple_sans_issue`.

Code à compléter

```
type graphe = int list array
(* graphes exemples *)
let g1 = [|[1;3];[];[0;1;3];[1]|]
let g2 = [|[];[0];[0;3;1];[1]|]

(* Question 1 *)
(* décommenter pour tester
est_sommet g1 0
est_sommet g2 4
 *)
(* Question 2 *)
(* Question 3 *)
(* Question 4 *)
let est_chemin_simple_sans_issue g liste =
  let n = Array.length g in
  let visites = Array.make false (n-1) (* tableau des sommets déjà vus *)
  in
  let rec test_aux liste =
    (* parcourt la liste en vérifiant que tous les critères sont satisfaits *)
    match liste with
      | [] -> (* à completer : cas du chemin vide *)
      | [a] -> (* à compléter : vrai si le dernier sommet n'a jamais été visité et que tout ses voisins l'on été *)
      | a::b::suite -> begin (*  à compléter  *) end
    in
    test_aux liste
(* Question 5 *)
let genere_chemins_simples_sans_issue (g:graphe) =
  let taille = Array.length g in
  let liste_chemins = ref [] in (* garde en mémoire les chemins déjà trouvés, en sens inverse *)
  let visites = Array.make taille false in (* garde en mémoire les sommets en cours de visite *)
  let chemin_courant_envers = ref [] in (* garde en mémoire le début d'un chemin *)
  let rec profondeur s = (* trouve tous les chemins simples sans issue commençant par s *)
    if not visites.(s) then
    begin
      visites.(s) <- true ;
      chemin_courant_envers := s::(!chemin_courant_envers);
      let voisins_libres = (* à compléter : calcule la liste des voisins de s non encore visités *)
      in
      if voisins_libres = [] then
        begin (*  à compléter : chemin simple sans issue trouvé *) end
      else
        begin (* à compléter : parcours pour trouver d'autres chemins *) end;
      (* fin de l'instruction conditionnelle en fonction de voisins_libres *)
      visites.(s) <- false ; (* pour revenir en arrière *)
      chemin_courant_envers := List.tl !chemin_courant_envers; (* pour revenir en arrière, List.tl permet de priver une liste de son premier element *)
    end
  in
  for i = 0 to (taille-1) do profondeur i done;
  !liste_chemins
(* Question 6 *)
```